Scheme and Functional Programming

Technical Report CPSLO-CSC-09-03

Preface

This volume contains the papers of the tenth annual Workshop on Scheme and Functional Programming, held August 22nd at Northeastern University in close proximity to the Symposium in honor of Mitchell Wand.

The Workshop recevied eighteen submissions this year, and accepted fifteen of these. In addition, we're pleased to include in the workshop an invited talk by Emmanuel Schanzer, on the Bootstrap program, and a talk by the newly elected Scheme Language Steering committee on the future directions of Scheme.

Many people worked hard to make the Scheme Workshop happen. I would like to thank the Program Committee, along with two external reviewers, Christopher Dutchyn and Daniel King, for their thoughtful, detailed, and well-received reviews. The Scheme Workshop would also *never have taken place* without the marvelous and timely work done by the Northeastern University development office staff headed by Jenn Wong.

We used the Continue2 submission server to handle workshop submissions and found it effective and robust. Our thanks go to Shriram Krishnamurthi and Arjun Guha for designing and maintaining it, along with the many that have worked on it in the last seven years.

I found the advice of the Steering Committee invaluable in running the workshop, particularly the written summaries provided by Olin Shivers and Mike Sperber. In addition, the phrasing of the web pages and of this very note draws heavily on the words of Will Clinger and Robby Findler.

John Clements
Cal Poly State University
Organizer and Program Chair
on behalf of the program committee

Program Committee

Dominique Boucher (Nu Echo)
John Clements (Cal Poly)
Abdulaziz Ghuloum (American University of Kuwait)
David Herman (Northeastern University)
Shriram Krishnamurthi (Brown University)
Matthew Might (University of Utah)
David Van Horn (Northeastern University)

Steering Committee

William D. Clinger (Northeastern University)
Marc Feeley (Université de Montréal)
Robby Findler (University of Chicago)
Dan Friedman (Indiana University)
Christian Queinnec (University Paris 6)
Manuel Serrano (INRIA Sophia Antipolis)
Olin Shivers (Georgia Tech)
Mitchell Wand (Northeastern University)

Schedule & Table of Contents

8:45am **Invited Talk: If programming is like math, why don't math teachers teach programming?**
Emmanuel Schanzer

9:30am Break

9:55am **Sequence Traces for Object-Oriented Executions** 7
Carl Eastlund, Matthias Felleisen

Scalable Garbage Collection with Guaranteed MMU 14
William D Clinger, Felix S. Klock II

Randomized Testing in PLT Redex 26
Casey Klein, Robert Bruce Findler

11:10am Break

11:30am **A pattern-matcher for miniKanren -or- How to get into trouble with CPS macros** 37
Andrew W. Keep, Michael D. Adams, Lindsey Kuper, William E. Byrd, Daniel P. Friedman

Higher-Order Aspects in Order 46
Eric Tanter

Fixing Letrec (reloaded) 57
Abdulaziz Ghuloum, R. Kent Dybvig

12:45pm Lunch

1:45pm **The Scribble Reader: An Alternative to S-expressions for Textual Content** 66
Eli Barzilay

Interprocedural Dependence Analysis of Higher-Order Programs via Stack Reachability 75
Matthew Might, Tarun Prabhu

Descot: Distributed Code Repository Framework 86
Aaron W. Hsu

Keyword and Optional Arguments in PLT Scheme 66
Matthew Flatt, Eli Barzilay

Screen-Replay: A Session Recording and Analysis Tool for DrScheme 103
Mehmet Fatih Köksal, Remzi Emre Başar, Suzan Üsküdarlı

3:00pm Break

3:20pm **Get stuffed: Tightly packed abstract protocols in Scheme** 111
John Moore

Distributed Software Transactional Memory 116
Anthony Cowley

World With Web: A compiler from world applications to JavaScript 121
Remzi Emre Başar, Caner Derici, Çağdaş Şenol

4:05pm Break

4:25pm **Peter J Landin (1930-2009)** 126
Olivier Danvy

Invited Talk: Future Directions for the Scheme Language
The Newly Elected Scheme Language Steering Committee

Sequence Traces for Object-Oriented Executions

Carl Eastlund   Matthias Felleisen

Northeastern University

{cce,matthias}@ccs.neu.edu

Abstract

Researchers have developed a large variety of semantic models of object-oriented computations. These include object calculi as well as denotational, small-step operational, big-step operational, and reduction semantics. Some focus on pure object-oriented computation in small calculi; many others mingle the object-oriented and the procedural aspects of programming languages.

In this paper, we present a novel, two-level framework of object-oriented computation. The upper level of the framework borrows elements from UML's sequence diagrams to express the message exchanges among objects. The lower level is a parameter of the upper level; it represents all those elements of a programming language that are not object-oriented. We show that the framework is a good foundation for both generic theoretical results and practical tools, such as object-oriented tracing debuggers.

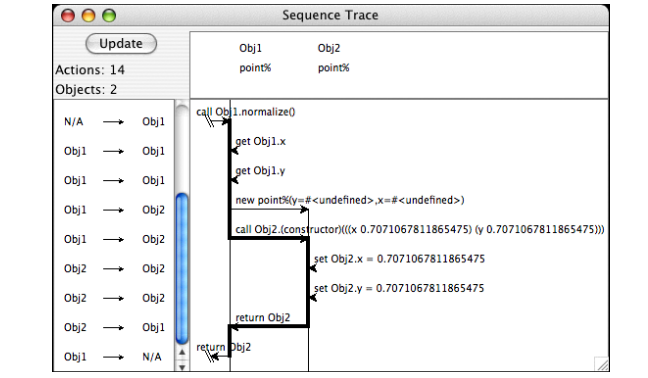


Figure 1. Graphical sequence trace.

1. Models of Execution

Some 30 years ago, Hewitt [22, 23] introduced the ACTOR model of computation, which is arguably the first model of object-oriented computation. Since then, people have explored a range of mathematical models of object-oriented program execution: denotational semantics of objects and classes [7, 8, 25, 33], object calculi [1], small step and big step operational semantics [10], reduction semantics [16], formal variants of ACTOR [2], and others [4, 20].

While all of these semantic models have made significant contributions to the community's understanding of object-oriented languages, they share two flaws. First, consider theoretical results such as type soundness. For ClassicJava, the type soundness proof uses Wright and Felleisen's standard technique of ensuring that type information is preserved while the computation makes progress. If someone extends ClassicJava with constructs such as while loops or switch statements, it is necessary to re-prove everything even though the extension did not affect the object-oriented aspects of the model. Second, none of these models are good starting points for creating practical tools. Some models focus on pure core object-oriented languages; others are models of real-world languages but mingle the semantics of object-oriented constructs (e.g., method invocations) with those of procedural or applicative nature (internal blocks or `while` loops). If a programmer wishes to debug the object-oriented actions in a Java program, a tracer based on any of these semantics would display too much procedural information. In short, a typical realistic model is to object-oriented debugging as a bit-level representation is to symbolic data structure exploration.

In this paper, we introduce a two-level [32] semantic framework for modeling object-oriented programming languages that overcomes these shortcomings. The *upper level* represents all object-oriented actions of a program execution. It tracks six kinds of actions via a rewriting system on object-configurations [26]: object creation, class inspection, field inspection, field mutation, method calls, and method return; we do not consider any other action an object-oriented computation. The computations at this upper level have a graphical equivalent that roughly corresponds to UML sequence diagrams [17]. Indeed, each configuration in the semantics corresponds to a diagram, and each transition between two configurations is an extension of the diagram for the first configuration.

The upper level of the framework is parameterized over the internal semantics of method bodies, dubbed the *lower level*. To instantiate the framework for a specific language, a semanticist must map the object-oriented part of a language to the object-oriented level of the framework and must express the remaining actions as the lower level. The sets and functions defining the lower level may be represented many ways, including state machines, mathematical functions, or whatever else a semanticist finds appropriate. We demonstrate how to instantiate the framework with a Java subset.

In addition to developing a precise mathematical meaning for the framework, we have also implemented a prototype of the framework. The prototype traces a program's object-oriented actions and allows programmers to inspect the state of objects. It is a component of the DrScheme programming environment [13] and covers the kernel of PLT Scheme's class system [15].

The next section presents a high-level overview. Section 3 introduces the framework and establishes a generalized soundness theorem. Section 4 demonstrates how to instantiate the framework for a subset of Java and extends the soundness theorem to that instantiation. Section 5 presents our tool prototype. The last two sections are about related and future work.

Proceedings of the 2009 Scheme and Functional Programming Workshop
California Polytechnic State University Technical Report CPSLO-CSC-09-03

$\overrightarrow{t}$	Any number of elements of the form t.
$c[e]$	Expression e in evaluation context c.
$e[x := v]$	Substitution of v for free variable x in expression e.
$d \xrightarrow{\mathrm{p}} r$	The set of partial functions of domain d and range r.
$d \xrightarrow{\mathrm{f}} r$	The set of finite mappings of domain d and range r.
$[\overrightarrow{a \mapsto b}]$	The finite mapping of each a to the corresponding b.
$f[\overrightarrow{a \mapsto b}]$	Extension of finite mapping f by each mapping of a to b (overriding any existing mappings).

Figure 2. Notational conventions.

2. Sequence Traces

Sequence traces borrow visual elements from UML sequence diagrams, but they represent concrete execution traces rather than specifications. A sequence trace depicts vertical object lifelines and horizontal message arrows with class and method labels, just as in sequence diagrams. The pool of objects extends horizontally; execution of message passing over time extends vertically downward. There are six kinds of messages in sequence traces: **new** messages construct objects, **get** and **set** messages access fields, **call** and **return** messages mark flow control into and out of methods, and **inspect** messages extract an object's tag.

Figure 1 shows a sample sequence trace. This trace shows the execution of the method `normalize` on an object representing the cartesian point $(1, 1)$. The method constructs and returns a new object representing $(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2})$. The first object is labeled `Obj1` and belongs to class `point%`. Its lifeline spans the entire trace and gains control when an external agent calls `Obj1.normalize()`. The first two actions access its `x` and `y` fields (self-directed messages, represented by lone arrowheads). `Obj1` constructs the second `point%` object, `Obj2`, and passes control to its constructor method. `Obj2` initializes its `x` and `y` fields and returns control to `Obj1`. Finally, `Obj1` returns a reference to `Obj2` and yields control.

Sequence traces suggest a model of computation as communication similar to π-calculus models [35]. In this model, an execution for an object-oriented program is represented as a collection of object lifelines and the messages passed between them. The model "hides" computations that take place inside of methods and that don't require any externally visible communication. This is the core of any object-oriented programming language and deserves a formal exploration.

3. The Framework

Our framework assigns semantics to object-oriented languages at two levels. The upper level describes objects, their creation, their lifelines, and their exchanges of messages. The lower level concerns all those aspects of a language's semantics that are unrelated to its object-oriented nature, e.g., static methods, blocks, decision constructs, looping constructs, etc. In this section we provide syntax, semantics, a type system, and a soundness theorem for the upper level.

3.1 The Upper Level

For the remainder of the paper we use the notational conventions shown in Figure 2. Figure 3 gives the full syntax of the upper level using this notation and specifies the language-specific sets over which it is parameterized. A sequence trace is a series of states each containing a pool of objects, a stack of active methods, a reference to a controlling object, and a current action. Objects consist of a static record (their unchanging properties, such as their class) and a dynamic record (their mutable fields). Actions may be one of six message types (**new**, **inspect**, **get**, **set**, **call**, or **return**) or an execution error.

Syntax:

$T = \overrightarrow{S}$	Sequence trace
$S = \langle P, K, r, A \rangle$	Execution state
$P : r \xrightarrow{\mathrm{f}} O$	Object pool
$K = \epsilon \mid \langle r, k \rangle\ K$	Method stack
$O = \langle s, D \rangle$	Object record
$D : f \xrightarrow{\mathrm{f}} V$	Dynamic record
$V = v \mid r \mid s$	Value
$A = M \mid ERR$	Action
$M = \mathbf{new}\ O;\ k \mid \mathbf{inspect}\ r;\ k$ $\mid \mathbf{get}\ r.f;\ k \mid \mathbf{set}\ r.f := V;\ k$ $\mid \mathbf{call}\ r.m(\overrightarrow{V});\ k \mid \mathbf{return}\ V$	Message
$R = \langle P, \epsilon, r, \mathbf{return}\ V \rangle$ $\mid \langle P, K, r, ERR \rangle$	Result
$ERR = err \mid \mathbf{error{:}ref} \mid \mathbf{error{:}field}$	Execution error

Where:

p	lower-level parameter	Program
k	lower-level parameter	Method-local continuation
s	lower-level parameter	Static record
f	lower-level parameter	Field name
m	lower-level parameter	Method name
v	lower-level parameter	Primitive value
err	lower-level parameter	Language-specific error
r	countable set	Object reference

Figure 3. Sequence trace syntax.

Figure 4 gives the upper-level operational semantics of sequence traces along with descriptions and signatures for its lower-level parameters. The parameter *init* is a function mapping a program to its initial state. A trace is the result of rewriting the initial state, step by step, into a final state. Each subsequent state depends on the previous state and action, as follows:

object creation A **new** action adds a reference and an object to the pool. The initiating object retains control.

object inspection An **inspect** action retrieves the static record of an object.

field lookup A **get** action retrieves the value of a field from an object.

field update A **set** action changes the value of a field in an object.

method call A **call** action invokes a method in an object, supplies a number of arguments, and transfers control.

method return A **return** action completes the current method call.

All of these transitions have a natural graphical equivalent (see Section 2).

At each step, the rewriting system uses either the (partial) function *invoke* or *resume* to compute the next action. These functions, like the step relation $\rightarrow$ and several others described below, are indexed by the source program p. Both functions are parameters of the rewriting system. The former begins executing a method; the latter continues one in progress using a method-local continuation. Both functions are partial, admitting the possibility of nontermination at the method-internal level. Also, both functions may map their inputs to a language-specific error.

3.2 Soundness

Our two-level semantic framework comes with a two-level type system. The purpose of this type system is to eliminate *all* upper-level type errors (reference error, field error) and to allow only those language-specific errors on which the lower-level insists. For

Evaluation:

$\langle P, K, r, \textbf{new}\ O;\ k\rangle$	$\rightarrow_p \langle P[r' \mapsto O], K, r, resume_p(k, r')\rangle$	where $r' \notin dom(P)$
$\langle P, K, r, \textbf{inspect}\ r';\ k\rangle$	$\rightarrow_p \langle P, K, r, resume_p(k, s)\rangle$	where $P(r') = \langle s, D\rangle$
$\langle P, K, r, \textbf{get}\ r'.f;\ k\rangle$	$\rightarrow_p \langle P, K, r, resume_p(k, V)\rangle$	where $P(r') = \langle s, D\rangle$ and $D(f) = V$
$\langle P, K, r, \textbf{set}\ r'.f := V;\ k\rangle$	$\rightarrow_p \langle P[r' \mapsto \langle s, D[f \mapsto V]\rangle], K, r, resume_p(k, V)\rangle$	where $P(r') = \langle s, D\rangle$ and $f \in dom(D)$
$\langle P, K, r, \textbf{call}\ r'.m(\overrightarrow{V});\ k\rangle$	$\rightarrow_p \langle P, \langle r, k\rangle\ K, r', invoke_p(r', P(r'), m, \overrightarrow{V})\rangle$	where $r' \in dom(P)$
$\langle P, \langle r', k\rangle\ K, r, \textbf{return}\ V\rangle$	$\rightarrow_p \langle P, K, r', resume_p(k, V)\rangle$	
$\langle P, K, r, \textbf{inspect}\ r';\ k\rangle$ $\langle P, K, r, \textbf{get}\ r'.f;\ k\rangle$ $\langle P, K, r, \textbf{set}\ r'.f := V;\ k\rangle$ $\langle P, K, r, \textbf{call}\ r'.m(\overrightarrow{V});\ k\rangle$	$\rightarrow_p \langle P, K, r, \textbf{error:ref}\rangle$	where $r' \notin dom(P)$
$\langle P, K, r, \textbf{get}\ r'.f;\ k\rangle$ $\langle P, K, r, \textbf{set}\ r'.f := V;\ k\rangle$	$\rightarrow_p \langle P, K, r, \textbf{error:field}\rangle$	where $P(r') = \langle s, D\rangle$ and $f \notin dom(D)$

Where:

$init : p \longrightarrow S$	Constructs the initial program state.
$invoke_p : \langle r, O, m, \overrightarrow{V}\rangle \xrightarrow{\text{p}} A$	Invokes a method.
$resume_p : \langle k, V\rangle \xrightarrow{\text{p}} A$	Resumes a suspended computation.

Figure 4. Sequence trace semantics.

Upper level:

$p \vdash^u S : t$	State S has type t.
$p \vdash^u P$	Object pool P is well-formed.
$p, P \vdash^u K : t_1 \xrightarrow{\text{s}} t_2$	Stack K produces type t_2 if the current method produces type t_1.
$p, P \vdash^u r : o$	Reference r has type o.
$p, P \vdash^u s : t$	Static record s has type t as a value.
$p, P \vdash^u O$ OK in o	Object record O is an object of type o.
$p, P \vdash^u D$ OK in o	Dynamic record D stores fields for an object of type o.
$p, P \vdash^u A : t$	Action A's method returns type t.

Lower level:

$\vdash^\ell p : t$	Program p has type t.
$p, P \vdash^\ell k : t_1 \xrightarrow{\text{c}} t_2$	Continuation k produces an action of type t_2 when given input of type t_1.
$p, P \vdash^\ell s$ OK in o	Static record s is well-formed in an object of type o.
$p, P \vdash^\ell v : t$	Primitive value v has type t.

Figure 5. Type judgments.

t	any set	Value types
$o \subseteq t$		Object types
$exn \subseteq err$		Allowable exceptions
$\sqsubseteq_p$	partial order on t	Subtype relation
$fields_p$	$: o \longrightarrow (f \xrightarrow{\text{f}} t)$	Produce an object's field, method, or static record types.
$methods_p$	$: o \longrightarrow (m \xrightarrow{\text{f}} \langle \overrightarrow{t}, t\rangle)$	
$metatype_p$	$: o \longrightarrow t$	

Figure 6. Sets, functions, and relations used by the type system.

example, in the case of Java, the lower level cannot rule out `null` pointer errors and must therefore raise the relevant exceptions.

Type judgments in this system are split between those defined at the upper level and those defined at the lower level, as shown in Figure 5. The upper level relies on the lower-level judgments and possibly vice versa. The lower-level type system must provide type judgments for programs, continuations, the static records of objects, and primitive values. The upper-level type system defines type judgments for everything else: program states, object pools, stacks, references, static records when used as values, object records, dynamic records, and actions of both the message and error variety.

$$\textsc{Init}\ \frac{\vdash^\ell p : t}{p \vdash^u init(p) : t}$$

$$\textsc{Resume}\ \frac{p, P \vdash^{u\ell} V : t_1 \quad p, P \vdash^\ell k : t_2 \xrightarrow{\text{c}} t_3 \quad t_1 \sqsubseteq_p t_2 \quad t_4 \sqsubseteq_p t_3}{p, P \vdash^u resume_p(k, V) : t_4}$$

$$\textsc{Invoke}\ \frac{p, P \vdash^u r : o \quad \overrightarrow{p, P \vdash^{u\ell} V : t_1} \quad methods_p(o)(m) = \langle \overrightarrow{t_2}, t_3\rangle \quad \overrightarrow{t_1 \sqsubseteq_p t_2} \quad t_4 \sqsubseteq_p t_3}{p, P \vdash^u invoke(r, P(r), m, \overrightarrow{V}) : t_4}$$

Figure 7. Constraints on the lower-level type system.

The lower level must also define several sets, functions, and type judgments, shown in Figure 6. The set t defines types for the language's values; o defines the subset of t representing the types of objects. The subset exn of err distinguishes the runtime exceptions that well-typed programs may throw.

The subtype relation $\sqsubseteq$ induces a partial order on types. The total functions $fields$ and $methods$ define the field and method signatures of object types. The total function $metatype$ determines the type of a static record from the type of its container object; it is needed to type **inspect** messages.

The INIT, RESUME, and INVOKE typing rules, shown in Figure 7, constrain the lower-level framework functions of the same names. The INIT rule states that a program must have the same type as its initial state. The RESUME rule states that a continuation's argument object and result action must match its input type and output type, respectively. The INVOKE rule states that when an object's method is invoked and given appropriately-typed arguments, it must produce an appropriately-typed action. In addition, a sound system requires all three to be total functions, whereas the untyped operational semantics allows $resume$ and $invoke$ to be partial. The

Syntax:

p	$=$	$\overrightarrow{\Delta}$
s	$=$	c
f	$=$	$\langle c, f^{\text{cj}} \rangle$
m	$=$	$m^{\text{cj}} \mid \langle c, m^{\text{cj}} \rangle$
v	$=$	`null`
err	$=$	**error:method** | **error:null**
	|	**error:typecast** | **error:var**
k	$=$	$\{\ \tau\ x\texttt{=}k;\ \overline{\tau\ x\texttt{=}e};\ e\ \}$
	|	$(\tau)k \mid (k \sqsubseteq \tau)r \mid k\texttt{:}c\texttt{.}f^{\text{cj}}$
	|	$k\texttt{:}c\texttt{.}f^{\text{cj}}\texttt{=}e \mid V\texttt{:}c\texttt{.}f^{\text{cj}}\texttt{=}k$
	|	$k\texttt{.}m^{\text{cj}}\texttt{(}\overrightarrow{e}\texttt{)} \mid V\texttt{.}m^{\text{cj}}\texttt{(}\overrightarrow{V}\ k\ \overrightarrow{e}\texttt{)}$
	|	$\texttt{super}\equiv r\texttt{:}c\texttt{.}m^{\text{cj}}\texttt{(}\overrightarrow{V}\ k\ \overrightarrow{e}\texttt{)} \mid []$

Where:

i		countable set	Interface name
c		countable set	Class name
m^{cj}		countable set	Method label
f^{cj}		countable set	Field label
Δ	$=$	$\texttt{interface}\ i\ \texttt{extends}\ \overrightarrow{i}\ \{\ \overrightarrow{\sigma}\ \}$	Definition
	|	$\texttt{class}\ c\ \texttt{extends}\ c\ \texttt{implements}\ \overrightarrow{i}\ \{\ \overrightarrow{\phi}\ \overrightarrow{\delta}\ \}$	
σ	$=$	$\tau\ m^{\text{cj}}\texttt{(}\overrightarrow{\tau}\texttt{);}$	Method signature
δ	$=$	$\tau\ m^{\text{cj}}\texttt{(}\overline{\tau\ x}\texttt{)}\ \{\ e\ \}$	Method definition
ϕ	$=$	$\tau\ f^{\text{cj}}\texttt{=}e\texttt{;}$	Field definition
e	$=$	$V \mid x \mid \texttt{this} \mid \{\ \overline{\tau\ x\texttt{=}e};\ e\ \} \mid \texttt{new}\ c$	Expression
	|	$(\tau)e \mid (c \sqsubseteq \tau)e \mid e\texttt{:}c\texttt{.}f^{\text{cj}} \mid e\texttt{:}c\texttt{.}f^{\text{cj}}\texttt{=}e$	
	|	$e\texttt{.}m^{\text{cj}}\texttt{(}\overrightarrow{e}\texttt{)} \mid \texttt{super}\equiv e\texttt{:}c\texttt{.}m^{\text{cj}}\texttt{(}\overrightarrow{e}\texttt{)}$	

Figure 8. Java core syntax.

$\mathit{field}_p : \langle c, f^{\text{cj}} \rangle \longrightarrow \phi$	Looks up field definitions.
$\mathit{method}_p : \langle c, m^{\text{cj}} \rangle \longrightarrow \delta$	Looks up method definitions.
$\mathit{object}_p : c \longrightarrow O$	Constructs new objects.
$\mathit{call}_p : \langle r, c, m^{\text{cj}}, \overrightarrow{V} \rangle \longrightarrow A$	Picks a method's first action.
$\mathit{eval}_p : e \longrightarrow A$	Chooses the next action.
$\rightarrow^{\text{cj}}_p : e \overset{\text{p}}{\longrightarrow} e$	Computes a single step.

Figure 9. Java core relations and functions.

lower level type system must guarantee these rules, while the upper level relies on them for a parametric soundness proof.

THEOREM 1 (Soundness). *If the functions init, resume, and invoke are total and satisfy constraints* INIT, RESUME, *and* INVOKE *respectively, then if* $\vdash^{\ell} p : t$, *then either* p *diverges or* $\mathit{init}(p) \twoheadrightarrow_p R$ *and* $p \vdash^{u} R : t$.

The type system satisfies a conventional type soundness theorem. Its statement assumes that lower-level exceptions are typed; however, they can only appear in the final state of a trace. Due to space limitations, the remaining details of the type system and soundness proof have been relegated to our technical report [12].

4. Framework Instantiations

The framework is only useful if we can instantiate its lower level for a useful object-oriented language. In this section we model a subset of Java in our framework, establishes its soundness, and consider an alternate interpretation of Java that strikes at the heart of the question of which language features are truly object-oriented. We also discuss a few other framework instantiations.

4.1 Java via Sequence Traces

Our framework can accomodate the sequential core of Java, based on ClassicJava [16], including classes, subclasses, interfaces, method overriding, and typecasts. Figure 8 shows the syntax of the Java core. Our set of expressions includes lexically scoped blocks, object creation, typecasts, field access, method calls, and superclass method calls. Field access and superclass method calls have class annotations on their receiver to aid the type soundness lemma in Section 4.3. Typecast expressions have an intermediate form used in our evaluation semantics. We leave out many other Java constructs such as conditionals, loops, etc.

Programs in this language are a sequence of class and interface definitions. An object's static record is the name of its class. Field names include a field label and a class name. Method names include a label and optionally a class name. The sole primitive value is `null`. We define errors for method invocation, null dereference, failed typecasts, and free variables. Last but not least, local continuations are evaluation contexts over expressions.

Figure 10 defines the semantics of our Java core using the relations and functions described in Figure 9. We omit the definitions of $\sqsubseteq$, *field*, and *method*, which simply inspect the sequence of class and interface definitions. The *init* function constructs an object of class `Program` and invokes its `main` method. The *resume* function constructs a new expression from the given value and the local continuation (a context), then passes it to *eval*; *invoke* simply uses *call*.

Method invocation uses *call* for dispatch. This function looks up the appropriate method in the program's class definitions. It substitutes the method's receiver and parameters, then calls *eval* to evaluate the expression.

The *eval* function is defined via a reduction relation $\rightarrow^{\text{cj}}$. That is, its results are determined by the canonical forms of expression with respect to $\twoheadrightarrow^{\text{cj}}$, the reflexive transitive closure. Object creation, field lookup, field mutation, method calls, and method returns all generate corresponding framework actions. Unelaborated typecast expressions produce inspection actions, adding an elaborated typecast context to their continuation. The *eval* function signals an error for all null dereferences and typecast failures.

Calls to an object's superclass generate method call actions; that is, an externally visible message. The method name includes the superclass name for method dispatch, which distinguishes it from the current definition of the method.

The step relation ($\rightarrow^{\text{cj}}$) performs all purely object-internal computations. It reduces block expressions by substitution and completes successful typecasts by replacing the elaborated expression with its argument.

LEMMA 1. *For any expression* e, *there is some* e' *such that* $e \twoheadrightarrow^{\text{cj}}_p e'$ *and* e' *is of canonical form.*

Together, the sets of canonical expressions and of expressions on which $\rightarrow^{\text{cj}}$ is defined are exhaustive. Furthermore, each step of $\rightarrow^{\text{cj}}$ strictly reduces the size of the expression. The expression must reduce in a finite number of steps to a canonical form for which *eval* produces an action. Therefore *eval* is total.

COROLLARY 1. *The functions invoke and resume are total.*

Because these functions are total, evaluation in the sequential core of Java cannot get stuck; each state must either have a successor or be a final result.

4.2 Alternate Interpretation of the Java Core

Our parameterization of the sequence trace framework for Java answers the question: "what parts of the Java core are object-

$$
\begin{array}{l}
\mathit{init}(p) = \langle [r_0 \mapsto \mathit{object}_p(\texttt{Program})], \epsilon, r_0, \textbf{call}\ r_0.\texttt{main}();\ [] \rangle \\
\mathit{resume}_p(k, V) = \mathit{eval}_p(k[V]) \\
\mathit{invoke}_p(r, \langle c, D\rangle, m^{\mathrm{cj}}, \overrightarrow{V}) = \mathit{call}_p(r, c, m^{\mathrm{cj}}, \overrightarrow{V}) \\
\mathit{invoke}_p(r, \langle c, D\rangle, \langle c', m^{\mathrm{cj}}\rangle, \overrightarrow{V}) = \mathit{call}_p(r, c', m^{\mathrm{cj}}, \overrightarrow{V})
\end{array}
$$

$$
\begin{array}{l}
\mathit{object}(c) = \langle c, [\overrightarrow{\langle c', f^{\mathrm{cj}}\rangle \mapsto \texttt{null}}]\rangle \\
\quad \text{where } \overrightarrow{\mathit{field}_p(c, f^{\mathrm{cj}}) = \tau\ f^{\mathrm{cj}}\texttt{=}c'\texttt{;}}
\end{array}
$$

$$
\mathit{call}_p(r, c, m^{\mathrm{cj}}, \overrightarrow{V}) =
\begin{cases}
\mathit{eval}_p(e[\texttt{this} := r][\overrightarrow{x := V}]) \\
\quad \text{if } \mathit{method}_p(c, m^{\mathrm{cj}}) = \tau\ m^{\mathrm{cj}}\texttt{(}\overrightarrow{\tau\ x}\texttt{)}\ \{\ e\ \} \\
\textbf{error:method} \text{ otherwise}
\end{cases}
$$

$$
\mathit{eval}_p(e) =
\begin{cases}
\textbf{return}\ V & \text{if } e \twoheadrightarrow_p^{\mathrm{cj}} V \\
\textbf{new}\ \mathit{object}(c);\ k & \text{if } e \twoheadrightarrow_p^{\mathrm{cj}} k[\texttt{new}\ c] \\
\textbf{get}\ r.\langle c, f\rangle;\ k & \text{if } e \twoheadrightarrow_p^{\mathrm{cj}} k[r\texttt{:}c\texttt{.}f] \\
\textbf{set}\ r.\langle c, f\rangle := V;\ k & \text{if } e \twoheadrightarrow_p^{\mathrm{cj}} k[r\texttt{:}c\texttt{.}f\texttt{=}V] \\
\textbf{call}\ r.m(\overrightarrow{V});\ k & \text{if } e \twoheadrightarrow_p^{\mathrm{cj}} k[r\texttt{.}m\texttt{(}\overrightarrow{V}\texttt{)}] \\
\textbf{call}\ r.\langle c, m\rangle(\overrightarrow{V});\ k & \text{if } e \twoheadrightarrow_p^{\mathrm{cj}} k[\texttt{super}\equiv r\texttt{:}c\texttt{.}m^{\mathrm{cj}}\texttt{(}\overrightarrow{V}\texttt{)}] \\
\textbf{inspect}\ r;\ k[\texttt{(}[] \sqsubseteq \tau\texttt{)}r] & \text{if } e \twoheadrightarrow_p^{\mathrm{cj}} k[\texttt{(}\tau\texttt{)}r] \\
\textbf{error:typecast} & \text{if } e \twoheadrightarrow_p^{\mathrm{cj}} k[\texttt{(}\tau\texttt{)null}] \text{ or } e \twoheadrightarrow_p^{\mathrm{cj}} k[\texttt{(}c \sqsubseteq \tau\texttt{)}V] \text{ and } c \not\sqsubseteq_p \tau \\
\textbf{error:null} & \text{if } e \twoheadrightarrow_p^{\mathrm{cj}} k[\texttt{null:}c\texttt{.}f] \text{ or } e \twoheadrightarrow_p^{\mathrm{cj}} k[\texttt{null:}c\texttt{.}f\texttt{=}V] \text{ or } e \twoheadrightarrow_p^{\mathrm{cj}} k[\texttt{null.}m\texttt{(}\overrightarrow{V}\texttt{)}] \\
\textbf{error:var} & \text{if } e \twoheadrightarrow_p^{\mathrm{cj}} k[x] \text{ or } e \twoheadrightarrow_p^{\mathrm{cj}} k[\texttt{this}]
\end{cases}
$$

$$
\begin{array}{lcl}
k[\{\ \tau'\ x'\texttt{=}V'\texttt{;}\ \overrightarrow{\tau\ x\texttt{=}e\texttt{;}}\ e'\ \}] & \rightarrow_p^{\mathrm{cj}} & k[\{\ \overrightarrow{\tau\ x\texttt{=}e[x' := V']\texttt{;}}\ e'[x' := V']\ \}] \\
k[\{\ e\ \}] & \rightarrow_p^{\mathrm{cj}} & k[e] \\
k[\texttt{(}c \sqsubseteq \tau\texttt{)}V] & \rightarrow_p^{\mathrm{cj}} & k[V] \text{ if } c \sqsubseteq_p \tau
\end{array}
$$

Figure 10. Java core semantics and auxiliary definitions.

oriented?" In the semantics above, the answer is clear: object creation, field lookup and mutation, method calls, method returns, superclass method calls, and typecasts.

Let us reconsider this interpretation. The most debatable aspect of our model concerns superclass method calls. They take place entirely inside one object and cannot be invoked by outside objects, yet we have formalized them as messages. An alternate perspective might formulate superclass method calls as object-internal computation for comparison.

Our framework is flexible enough to allow this reinterpretation of Java. In our semantics above, as in other models of Java [3, 10, 16, 24], **super** expressions evaluate to method calls. Method calls use *invoke* which uses *call*. We can change *eval* to use *call* directly in the **super** rule, i.e. no object-oriented action is created. The extra clauses for method names and *call* that were used for superclass calls can be removed. These modifications are shown in Figure 11.[1]

Now that we have two different semantics for Java, it is possible to compare them and to study the tradeoffs; implementors and semanticists can use either interpretation as appropriate.

4.3 Soundness of the Java Core

We have interpreted the type system for the Java core in our framework and established its soundness. Again, the details of the type system and soundness proof can be found in our technical report.

LEMMA 2. *The functions init, resume, and invoke are total and satisfy constraints* INIT, RESUME, *and* INVOKE.

According to Corollary 1, these functions are total. Since INIT, RESUME, and INVOKE hold, type soundness is just a corollary of Theorem 1.

COROLLARY 2 (Java Core Soundness). *In the Java core, if* $\vdash^{\ell} p : t$, *then either* p *diverges or* $\mathit{init}(p) \twoheadrightarrow_p R$ *and* $p \vdash^{u} R : t$.

[1] Note that *invoke* and *resume* are no longer total for cyclic class graphs. A soundness proof for this formulation must account for this exception, or *call* must be further refined to reject looping **super** calls.

$$
m = m^{\mathrm{cj}}\ \cancel{\mid \langle c, m^{\mathrm{cj}}\rangle}
$$

$$
\begin{array}{l}
\mathit{invoke}_p(r, \langle c, D\rangle, m^{\mathrm{cj}}, \overrightarrow{V}) = \mathit{call}_p(r, c, m^{\mathrm{cj}}, \overrightarrow{V}) \\
\cancel{\mathit{invoke}_p(r, \langle c, D\rangle, \langle c', m^{\mathrm{cj}}\rangle, \overrightarrow{V}) = \mathit{call}_p(r, c', m^{\mathrm{cj}}, \overrightarrow{V})}
\end{array}
$$

$$
\mathit{eval}_p(e) =
\begin{cases}
\ldots \\
\cancel{\textbf{call}\ r.\langle c, m\rangle(\overrightarrow{V});\ k \text{ if } e \twoheadrightarrow_p^{\mathrm{cj}} k[\texttt{super}\equiv r\texttt{:}c\texttt{.}m^{\mathrm{cj}}\texttt{(}\overrightarrow{V}\texttt{)}]} \\
\mathit{call}_p(r, c, m^{\mathrm{cj}}, \overrightarrow{V}) \text{ if } e \twoheadrightarrow_p^{\mathrm{cj}} k[\texttt{super}\equiv r\texttt{:}c\texttt{.}m^{\mathrm{cj}}\texttt{(}\overrightarrow{V}\texttt{)}]
\end{cases}
$$

Figure 11. Changes for an alternate interpretation of Java.

4.4 Other Languages

The expressiveness of formal sequence traces is not limited to just one model. In addition to ClassicJava, we have modeled Abadi and Cardelli's object calculus [1], the λ-calculus, and the $\lambda\&$-calculus [5] in our framework. The λ-calculus is the canonical model of functional computation, and the $\lambda\&$-calculus is a model of dispatch on multiple arguments. These instantiations demonstrate that sequence traces can model diverse (even non-object-oriented) languages and complex runtime behavior. Our technical report contains the full embeddings.

5. Practical Experience

To demonstrate the practicality of our semantics, we have implemented a Sequence Trace tool for the PLT Scheme class system [15]. As a program runs, the tool displays messages passed between objects. Users can inspect data associated with objects and messages at each step of execution. Method-internal function calls or other applicative computations remain hidden.

PLT Scheme classes are implemented via macros [9, 14] in a library, but are indistinguishable from a built-in construct. Traced programs link to an instrumented version of the library. The instrumentation records object creation and inspection, method entry and exit, and field access, exactly like the framework. Both instru-

```
(define point%
  (class object%
    ...
    (define (translate dx dy) ...)))

(define polygon%
  (class object%
    ...
    (define (add-vertex v) ...)
    (define (translate dx dy) ...)))

(send* (new polygon%)
  (add-vertex ...)
  (add-vertex ...)
  (add-vertex ...)
  (translate 5 5))
```

Figure 12. Excerpt of an object-oriented PLT Scheme program.

mented and non-instrumented versions of the library use the same implementation of objects, so traced objects may interact with untraced objects; however, untraced objects do not pay for the instrumentation overhead.

Figure 13 shows a sample sequence trace generated by our tool. This trace represents a program fragment, shown in Figure 12, using a class-based geometry library. The primary object is a `polygon%` containing three `point%` objects. The trace begins with a call to the polygon's `translate` method. The polygon must in turn translate each point, so it iterates over its vertices invoking their `translate` methods. Each original point constructs, initializes, and returns a new translated point.

The graphical layout allows easy inspection and navigation of a program. The left edge of the display allows access to the sender and receiver objects of each message. Each object lifeline provides access to field values and their history. Each message exposes the data and objects passed as its parameters. Highlighted sections of lifelines and message arrows emphasize flow control. Structured algorithms form recognizable patterns, such as the three iterations of the method `translate` on class `point%` shown in Figure 13, aiding in navigating the diagram, tracking down logic errors, and comparing executions to specifications.

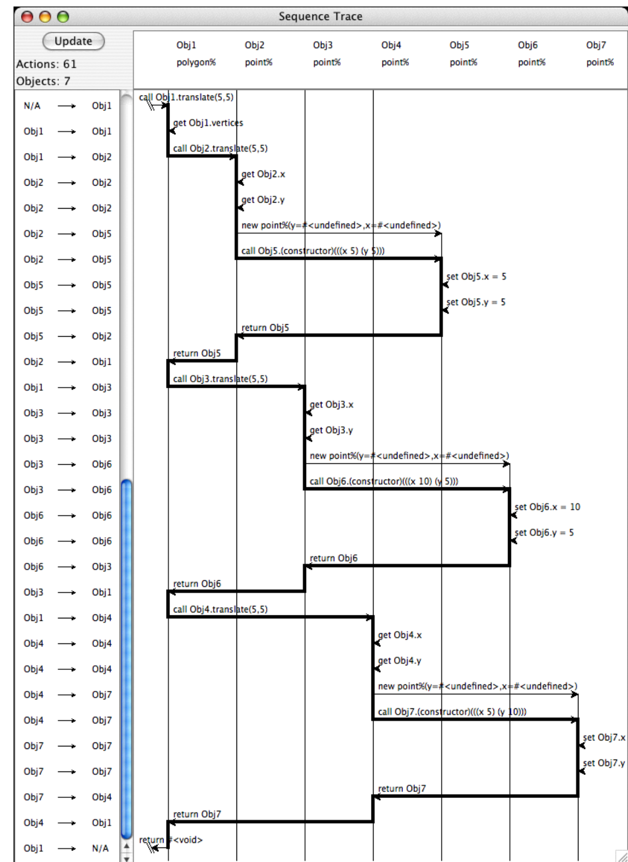


Figure 13. Sample output of the PLT Scheme Sequence Trace tool.

6. Related Work

Our work has two inspirational sources. Calculi for communicating processes often model just those actions that relate to process creation, communication, etc. This corresponds to our isolation of object-oriented actions in the upper level of the framework. Of course, our framework also specifies a precise interface between the two levels and, with the specification of a lower level, has the potential to model entire languages. Starting from this insight, Graunke et al. [18, 19, 27] have recently created a trace calculus for a sequential client-server setting. This calculus models a web client (browser) and web server with the goal of understanding systemic flaws in interactive web programs. Roughly speaking, our paper generalizes Graunke et al.'s research to an arbitrarily large and growing pool of objects with a general set of actions and a well-defined interface to the object-internal computational language.

Other tools for inspecting and debugging program traces exist, tackling the problem from many different perspectives. Lewis [28] presents a so-called omniscient debugger, which records every change in program state and reconstructs the execution after the fact. Intermediate steps in the program's execution can thus be debugged even after program completion. This approach is similar to our own, but with emphasis on the pragmatics of debugging rather than presenting an intuitive model of computation. Lewis does not present a theoretical framework and does not abstract his work from Java.

Execution traces are used in many tools for program analysis. Walker et al.'s tool [36] allows users to group program elements into abstract categories, then coalesces program traces accordingly and presents the resulting abstract trace. Richner and Ducasse [34] demonstrate automated recovery of class collaborations from traces. Ducasse et al. [11] provide a regression test framework in which successful logical queries over existing execution traces become specifications for future versions. Our tool is similar to these in that it uses execution traces; however, we do not generate abstract specifications. Instead we allow detailed inspection of the original trace itself.

Even though our work does not attempt to assign semantics to UML's sequence diagrams, many pieces of research in this direction exist and share some similarities with our own work. We therefore describe the most relevant work here. Many semantics for UML provide a definition for sequence diagrams as program specifications. Xia and Kane [37] and Li et al. [29] both develop paired static and dynamic semantics for sequence diagrams. The static semantics validate classes, objects, and operations referenced by methods; the dynamic semantics validate the execution of individual operations. Nantajeewarawat and Sombatsrisomboon [31] define a model-theoretic framework that can infer class diagrams from sequence diagrams. Cho et al. [6] provide a semantics in a new temporal logic called HDTL. These semantics are all concerned with specifications; unlike our work, they do not address object-oriented computation itself.

Lund and Stølen [30] and Hausmann et al. [21] both provide an operational semantics for UML itself, making specifications executable. Their work is dual to ours: we give a graphical, UML-inspired semantics to traditional object-oriented languages, while they give traditional operational semantics to UML diagrams.

7. Conclusions and Future Work

This paper presents a two-level semantics framework for object-oriented programming. The framework carefully distinguishes actions on objects from internal computations of objects. The two levels are separated via a collection of sets and partial functions. At this point the framework can easily handle models such as the core features of Java, as demonstrated in section 4, and languages such as PLT Scheme, as demonstrated in section 5.

Sequence traces still present several opportunities for elaboration at the object-oriented level. Most importantly, the object-oriented level currently assumes a functional creation mechanism for objects. While we can simulate the complex object construction of Java or PLT Scheme with method calls, we cannot model them directly. Conversely, the framework does not support a destroy action. This feature would require the extension of sequence traces with an explicit memory model, possibly parameterized over lower level details.

References

[1] Abadi, M. and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[2] Agha, G., I. A. Mason, S. F. Smith and C. L. Talcott. A foundation for actor computation. *J. Functional Programming*, 7(1):1–72, 1997.

[3] Bierman, G. M., M. J. Parkinson and A. M. Pitts. MJ: an imperative core calculus for Java and Java with effects. Technical report, Cambridge University, 2003.

[4] Bruce, K. B. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.

[5] Castagna, G., G. Ghelli and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.

[6] Cho, S. M., H. H. Kim, S. D. Cha and D. H. Bae. A semantics of sequence diagrams. *Information Processing Letters*, 84(3):125–130, 2002.

[7] Cook, W. R. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[8] Cook, W. R. and J. Palsberg. A denotational semantics of inheritance and its correctness. In *Proc. 1989 Conference on Object-Oriented Programming: Systems, Languages, and Applications*, p. 433–443. ACM Press, 1989.

[9] Culpepper, R., S. Tobin-Hochstadt and M. Flatt. Advanced macrology and the implementation of Typed Scheme. In *Proc. 8th Workshop on Scheme and Functional Programming*, p. 1–14. ACM Press, 2007.

[10] Drossopoulou, S. and S. Eisenbach. Java is type safe—probably. In *Proc. 11th European Conference on Object-Oriented Programming*, p. 389–418. Springer, 1997.

[11] Ducasse, S., T. Gîrba and R. Wuyts. Object-oriented legacy system trace-based logic testing. In *Proc. 10th European Conference on Software Maintenance and Reengineering*, p. 37–46, 2006.

[12] Eastlund, C. and M. Felleisen. Sequence traces for object-oriented executions. Technical report, Northeastern University, 2006.

[13] Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen. DrScheme: a programming environment for Scheme. *J. Functional Programming*, 12(2):159–182, 2002.

[14] Flatt, M. Composable and compilable macros: you want it *when?* In *Proc. 7th ACM SIGPLAN International Conference on Functional Programming*, p. 72–83. ACM Press, 2002.

[15] Flatt, M., R. B. Findler and M. Felleisen. Scheme with classes, mixins, and traits. In *Proc. 4th Asian Symposium on Programming Languages and Systems*, p. 270–289. Springer, 2006.

[16] Flatt, M., S. Krishnamurthi and M. Felleisen. Classes and mixins. In *Proc. 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 171–183. ACM Press, 1998.

[17] Fowler, M. and K. Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.

[18] Graunke, P., R. Findler, S. Krishnamurthi and M. Felleisen. Modeling web interactions. In *Proc. 15th European Symposium on Programming*, p. 238–252. Springer, 2003.

[19] Graunke, P. T. *Web Interactions*. PhD thesis, Northeastern University, 2003.

[20] Gunter, C. A. and J. C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. MIT Press, 1994.

[21] Hausmann, J. H., R. Heckel and S. Sauer. Towards dynamic meta modeling of UML extensions: an extensible semantics for UML sequence diagrams. In *Proc. IEEE 2001 Symposia on Human Centric Computing Languages and Environments*, p. 80–87. IEEE Press, 2001.

[22] Hewitt, C. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

[23] Hewitt, C., P. Bishop and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proc. 3rd International Joint Conference on Artificial Intelligence*, p. 235–245. Morgan Kaufmann, 1973.

[24] Igarashi, A., B. Pierce and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. In *Proc. 1999 Conference on Object-Oriented Programming: Systems, Languages, and Applications*, p. 132–146. ACM Press, 1999.

[25] Kamin, S. N. Inheritance in SMALLTALK-80: a denotational definition. In *Proc. 15th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 80–87. ACM Press, 1988.

[26] Klop, J. W. Term rewriting systems: a tutorial. *Bulletin of the EATCS*, 32:143–182, 1987.

[27] Krishnamurthi, S., R. B. Findler, P. Graunke and M. Felleisen. Modeling web interactions and errors. In *Interactive Computation: the New Paradigm*, p. 255–275. Springer, 2006.

[28] Lewis, B. Debugging backwards in time. In *Proc. 5th International Workshop on Automated Debugging*, 2003. `http://www.lambdacs.com/debugger/AADEBUG_Mar_03.pdf`.

[29] Li, X., Z. Liu and J. He. A formal semantics of UML sequence diagrams. In *Proc. 15th Australian Software Engineering Conference*, p. 168–177. IEEE Press, 2004.

[30] Lund, M. S. and K. Stølen. Extendable and modifiable operational semantics for UML 2.0 sequence diagrams. In *Proc. 17th Nordic Workshop on Programming Theory*, p. 86–88. DIKU, 2005.

[31] Nantajeewarawat, E. and R. Sombatsrisomboon. On the semantics of Unified Modeling Language diagrams using Z notation. *Int. J. Intelligent Systems*, 19(1–2):79–88, 2004.

[32] Nielson, F. and H. R. Nielson. *Two-level functional languages*. Cambridge University Press, 1992.

[33] Reddy, U. S. Objects as closures: abstract semantics of object-oriented languages. In *Proc. 1988 ACM Conference on LISP and Functional Programming*, p. 289–297. ACM Press, 1988.

[34] Richner, T. and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proc. International Conference on Software Maintenance*, p. 34–43. IEEE Press, 2002.

[35] Sangiorgi, D. and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2003.

[36] Walker, R. J., G. C. Murphy, J. Steinbok and M. P. Robillard. Efficient mapping of software system traces to architectural views. In *Proc. 2000 Conference of the Centre for Advanced Studies on Collaborative Research*, p. 12. IBM Press, 2000.

[37] Xia, F. and G. S. Kane. Defining the semantics of UML class and sequence diagrams for ensuring the consistency and executability of OO software specification. In *Proc. 1st International Workshop on Automated Technology for Verification and Analysis*, 2003. `http://cc.ee.ntu.edu.tw/~atva03/papers/16.pdf`.

Scalable Garbage Collection with Guaranteed MMU

William D. Clinger
Northeastern University
will@ccs.neu.edu

Felix S. Klock II
Northeastern University
pnkfelix@ccs.neu.edu

Abstract

Regional garbage collection offers a useful compromise between real-time and generational collection. Regional collectors resemble generational collectors, but are scalable: our main theorem guarantees a positive lower bound, independent of mutator and live storage, for the theoretical worst-case minimum mutator utilization (MMU). The theorem also establishes upper bounds for worst-case space usage and collection pauses.

Standard generational collectors are not scalable. Some real-time collectors are scalable, while others assume a well-behaved mutator or provide no worst-case guarantees at all.

Regional collectors cannot compete with hard real-time collectors at millisecond resolutions, but offer efficiency comparable to contemporary generational collectors combined with improved latency and MMU at resolutions on the order of hundreds of milliseconds to a few seconds.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection)

General Terms Algorithms, Design, Performance

Keywords scalable, real-time, regional garbage collection

1. Introduction

We have designed and prototyped a new kind of scalable garbage collector that delivers a provable fixed upper bound for the duration of collection pauses. This theoretical worst-case bound is completely independent of the mutator (defined as the non-gc portion of an application) and the size of its data.

The collector also delivers a provable fixed lower bound for worst-case minimum mutator utilization (MMU, expressed as the smallest percentage of the machine cycles that are available to the mutator during any sufficiently long interval of time) and a simultaneous worst-case upper bound for space, expressed as a fixed multiple of the mutator's peak storage requirement.

These guarantees are achieved by sacrificing throughput on unusually gc-intensive programs. For most programs, however, the loss of throughput is small. Indeed, our prototype's overall throughput remains competitive with several generational collectors that are currently deployed in popular systems.

Section 5 discusses one near-worst-case benchmark. To reduce this paper to an acceptable length, we defer most discussion of more typical programs, and of throughput generally, to another paper that will also describe the engineering of our prototype in greater detail.

Worst-case performance, both theoretical and observed, is the focus of this paper. Many garbage collectors have been designed to exploit common cases, with little or no concern for the worst case. As illustrated by section 5, their worst-case performance can be quite poor. When designing our new *regional* collector, our main goal was to guarantee a minimal level of performance, independent of problem size and mutator behavior. We exploit common cases only when we can do so without compromising latency or asymptotic performance for the worst case.

1.1 Bounded Latency

Generational collectors that rarely stop the mutator while they collect the entire heap have worked well enough for many applications, but that paradigm breaks down for truly large heaps: even an occasional full collection can produce alarming or annoying delays (Nettles and O'Toole 1993). This problem is evident on 32-bit machines, and will only get worse as 64-bit machines become the norm.

Real-time, incremental, or concurrent collectors can eliminate those delays, but at significant cost. On stock hardware, most bounded-latency collectors depend upon a read barrier, which reduces throughput (average mutator utilization) even for programs that create little garbage. Read barriers and other invariants also increase the complexity of compilers and run-time infrastructure, while impeding use of libraries that were written and compiled without knowledge of the garbage collector's invariants.

Our regional collector is a novel bounded-latency collector whose invariants resemble the invariants of standard generational garbage collectors. In particular, our regional collector does not require a read barrier.

1.2 Scalability

Unlike standard generational collectors, the regional collector is *scalable*: Theorem 1 below establishes that the regional collector's theoretical worst-case collection latency and MMU are bounded by nontrivial constants that are independent of the volume of reachable storage and are also independent of mutator behavior. The theorem also states that these fixed bounds are achieved in space bounded by a fixed multiple of the volume of reachable storage.

Although most real-time, incremental, or concurrent collectors appear to be designed for embedded systems in which they can be tuned for a particular mutator, some (though not all) hard real-time collectors are scalable in the same sense as the regional collector. Even so, we are not aware of any published proofs that establish all three scalability properties of our main theorem for a hard real-time collector.

The following theorem characterizes the regional collector's worst-case performance.

Proceedings of the 2009 Scheme and Functional Programming Workshop
California Polytechnic State University Technical Report CPSLO-CSC-09-03

Theorem 1. *There exist positive constants c_0, c_1, c_2, and c_3 such that, for every mutator, no matter what the mutator does:*

1. *GC pauses are independent of heap size: c_0 is larger than the worst-case time between mutator actions.*
2. *Minimum mutator utilization is bounded below by constants that are independent of heap size: within every interval of time longer than $3c_0$, the MMU is greater than c_1.*
3. *Memory usage is $O(P)$, where P is the peak volume of reachable objects: the total memory used by the mutator and collector is less than $c_2P + c_3$.*

We must emphasize that the constants c_0, c_1, c_2, and c_3 are completely independent of the mutator. Their values do depend upon several parameters of the regional collector, upon details of how the collector is implemented in software, and upon the hardware used to execute the mutator and collector. Later sections will discuss the worst-case constants and report on the performance actually observed for one near-worst-case benchmark.

Major contributions of this paper include:

- a new algorithm for scalable garbage collection
- a proof of its scalability, independent of mutator behavior
- a novel solution to the problem of popular objects
- formulas that describe how theoretical worst-case performance varies as a function of collector parameters
- empirical measurements of actual performance for one near-worst-case benchmark

The remainder of this paper describes the processes, data structures, and algorithms of the regional collector, provides a proof of our main theorem above, estimates worst-case bounds, and summarizes related and future work.

2. Regional Collection

The regional collector resembles a stop-the-world generational collector with several additional data structures, processes, and invariants.

In place of generations that segregate objects by age, the regional collector maintains a set of relatively small regions, all of the same size R. There is no strict correlation between an object's region and the object's age. Only one region is collected at a time. (In most generational collectors, collecting a generation implies the simultaneous collection of all younger generations.)

The regional collector assumes every object is small enough to fit within a region. For justification, see sections 3.4 and section 7.

The regional collector maintains a remembered set, a collection of summary sets, and a snapshot structure. Each component is described in detail below, after an overview of the memory management processes. In short, the remembered set tracks region-crossing references, the summary sets summarize portions of the remembered set that will be relevant to upcoming collections, and the snapshot structure gathers past reachability information to refine the remembered set.

The interplay between regions, the remembered set and the summary sets is an important and novel aspect of our design.

2.1 Processes

The regional collector adds three distinct computational processes to those of the mutator:

- a collection process uses the Cheney (1970) algorithm to move a region's reachable storage into some other region(s),
- a summarization process computes summary sets from the remembered set, and
- a snapshot-at-the-beginning marking process marks every object reachable in a snapshot of the object graph.

The summarization and marking processes run concurrently or interleaved with the mutator processes. When the collection process is executing, all other processes are suspended.

The collection and marking processes serve distinct purposes. The collection process moves objects to prevent fragmentation, and updates pointers from outside the collected region to point to the newly relocated objects; it also reclaims unreachable storage.[1]

The pointers that must be updated during a relocating collection reside in uncollected regions, in the marking process's snapshot structure, and in the mutator stack(s); the latter are discussed in sections 2.6 and 2.8 respectively.

The summarization process constructs *summary sets* in preparation for collections, and is the subject of section 2.3.

The regional collector imposes a fixed constant bound on the duration of each collection. That means that a *popular* region, whose summary set is larger than a fixed threshold, would take too long to collect. Section 3.3 proves that, with appropriate values for the collector's parameters, the percentage of popular regions is so well bounded that the regional collector can afford to leave popular regions uncollected. That is one of the critical lemmas that establish the scalability of regional garbage collection.

The main purpose of the marking process is to limit unreachable storage to a bounded fraction of peak live storage; it accomplishes that by removing unreachable references from the remembered set. The marking process also calculates the volume of reachable storage at the time of its initiation; without that information, the collector might not be able to guarantee worst-case bounds for its storage requirements.

2.2 Remembered Set

We bound the pause time by collecting one region independently of all others. To enable this, the mutator and collector collaboratively maintain a *remembered set*, which contains every location (or object) that points from one region to a different region. A similar structure is a standard component of generational collectors.

The mutator can create such region-crossing pointers by allocation or assignment. The collector can create region-crossing pointers by relocating an object from one region to another.

The remembered set is affected by two distinct kinds of imprecision:

- The remembered set may contain entries for locations or objects that are no longer reachable by the mutator.
- The remembered set may contain entries for locations or objects that are still reachable, but no longer contain a pointer that points from one region to a different region.

The regional collector represents its remembered set using a data structure that records at most one entry for each location in the heap (e.g. a hash table or fine-grain card table suffices). The size of the remembered set's representation is therefore bounded by the size of the heap, even though the remembered set is imprecise.

2.3 Summary Sets

A typical generational collector will scan most (or all) of the remembered set during collections of the younger portions of the heap. In the worst case the remembered set can grow proportional to the heap; hence this technique would not satisfy our pause time bounds, and is not an option for the regional collector.

[1] The collection process is the *only* process permitted to move objects. The summarization and marking processes do not change the correspondence between addresses and objects; hence neither interferes with the other's view of the heap (nor the mutator's view), even if run concurrently.

To collect a region independently of other regions, the collector must know all locations in uncollected regions that may hold pointers into the collected region. This set of locations is the *summary set* for the collected region.

If an imprecise remembered set were organized as a set of summary sets, one for each region, then the collector would not be scalable: in the worst case, the storage occupied by those summary sets would be proportional to the number of regions times the size of the heap. Since regions are of fixed constant size, the summary sets could occupy storage proportional to the square of the heap size. That is why the regional collector uses a remembered set representation that records pointers that come out of a region instead of pointers that go into the region.

There are two distinct issues to address regarding the use and construction of summary sets.

First, the regional collector *must* compute a region's summary set before it can collect the region. But a naïve construction could take both time and space proportional to the size of the heap, which would violate our bounds.

Second, in the worst case, a summary set for a region may consist of all locations in the heap. That means that a *popular* region, defined as a region whose summary set is larger than a fixed threshold, would take too long to collect.

To address these two issues, and thus keep time and space under control, the summarization process

- amortizes the cost in time by incrementally computing multiple summary sets for a fixed fraction $1/F_1$ of the heap's regions, but
- abandons the computation of any summary set whose size exceeds a fixed wave-off threshold (expressed as a multiple S of the region size R).

Waving off summarization raises the question: when do popular regions get collected? Our answer, inspired by Detlefs et al. (2004), is simple: such regions are not collected.[2] Instead we bound the percentage of popular regions to ensure that the regional collector can afford to leave popular regions uncollected. See sections 3.2 and 3.3.

2.4 Nursery

Like most generational collectors, the regional collector allocates all objects within a relatively small *nursery*. The nursery has little impact on worst-case performance, so our proofs ignore it. For most programs, however, the nursery greatly improves the observed MMU and overall efficiency of the regional collector.

Since the nursery is collected as part of every collection, locations within the nursery that point outside the nursery do not need to be added to the remembered set.

Pointers from a region into the nursery can be created only by assignments. Those pointers are recorded in a special summary set, which is updated by processing of write barrier logs. If the size of that summary set exceeds a fixed threshold, then the regional collector forces a minor collection that empties the nursery, promoting survivors into a region.

2.5 Grouping Regions

Figure 1 depicts how regions are partitioned into five groups: { **ready**, **unfilled**, **filled**, **popular**, **summarizing** }. In the figure, each small rectangle is a fixed-size region, the tiny ovals are objects allocated within a region, and the triangular "hats" atop some of the regions are summary sets. The dotted hats are under construction, while the filled hats are completely constructed.

[2] Our strategy is subtly different from Detlefs et al. (2004); Garbage-First migrates popular *objects* to a dedicated space; that still requires time proportional to the heap size in the worst case. We do not migrate the popular objects at all.

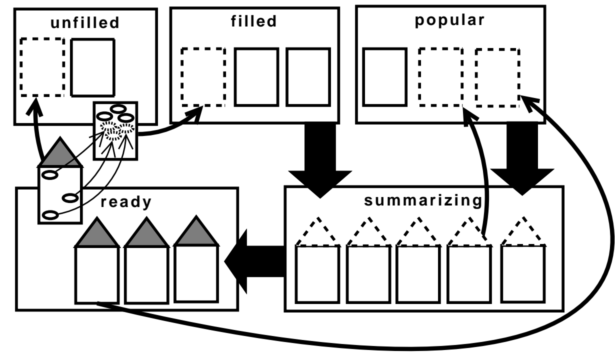


Figure 1. Grouping and transition of regions

The thinnest arcs in the figure, connecting small ovals, represent migration of individual objects during a major collection; that is the *only* time at which objects move from one region to another. Arcs of medium thickness represent transitions of a single region from one group to another, and the thickest arcs represent transitions of many regions at once.

At all times, one of the unfilled regions is the current *to-space*; it may contain some objects, but all other regions in the unfilled group are empty.

Four of the arcs form a cycle that describes the usual transitions of a region:

(ready, unfilled) On each major collection, one region (the *from-space*) is selected from the **ready** group. All of its reachable objects are forwarded to unfilled region(s) via Cheney's algorithm (the thinnest arcs). After object forwarding is complete, the now empty region is reclassified as **unfilled**.

(unfilled, filled) When the collector fills the current to-space region to capacity, it is reclassified as **filled**, and another unfilled region is picked to be the new to-space.

(filled, summarizing) The summarization process starts its cycle by reclassifying a subset of regions *en masse* as **summarizing**, preparing them for future collection.

(summarizing, ready) At the end of a summarization cycle the summarized regions become **ready** for collection.

The remaining three arcs in the diagram describe transitions for **popular** regions:

(summarizing, popular) As the summarization process passes over the remembered set, it may discover that a summary set for a particular region is too large: i.e., the region has too many incoming references to be updated within the pause time bound. The sumarization process will then remove that region from the summarizing group, and deem that region **popular**.

(ready, popular) Mutator activity can increase the number of incoming references to a **ready** region, to the point where it has too many incoming references to be updated within the pause time bound. Such regions are likewise removed from the **ready** group and become **popular**.

(popular, summarizing) Our collector does *not* assume that popular regions will remain popular forever. At the start of a summarization cycle, **popular** regions can be shifted into the **summarizing** group, where their fitness for collection will be re-evaluated by the summarization process.

2.6 Snapshots

The remembered set is imprecise. To bound its imprecision, a periodic snapshot-at-the-beginning (Yuasa 1990) marking process incrementally constructs a snapshot of the heap at a particular point in time. The resulting snapshot classifies every object as either unreachable or live/unallocated at the time of the snapshot.

The marking process incrementally traces the snapshot's object graph; objects allocated after the instant the snapshot was initiated are considered live by the snapshot and are not traced by the marking process. Objects relocated by the Cheney algorithm retain their current unreachable/live classification in the snapshot.

When the marking process completes snapshot construction, it removes dead locations from the remembered set. This increases remembered set precision, reducing the amount of floating garbage; in particular, it ensures that cyclic garbage across different regions is eventually removed from the remembered set.

The developing snapshot has a frontier of objects remaining to be processed, called the *mark stack*. The regional collector treats the portion of the mark stack holding objects in the collected region as an additional source of roots. In order to ensure that collection pauses only take time proportional to the size of a region, each regions' substacks are threaded through the single mark stack, and the collector scans *only* the portion of the stack relevant to a particular region.

2.7 Write Barrier

Assignments and other mutations that store into pointer fields of objects must go through a *write barrier* that updates the remembered set to account for the assignment.

The regional collector uses a variant of a Yuasa-style logging write barrier (Yuasa 1990). Our write barrier logs three things: (1) the location on the left hand side of the assignment, (2) its previous contents, and (3) its new contents.

The first is for remembered set and summary set maintenance. The second is for snapshot maintenance (the marker). The third identifies which summary set (if any) needs maintenance for the log entry.

2.8 Mutator Stacks

The regional collector assumes mutator stacks are constructed from heap-allocated objects of bounded size, as though all stack frames were allocated on the heap (Appel 1992). Although mixed stack/heap, incremental stack/heap, Hieb-Dybvig-Bruggeman, and Cheney-on-the-MTA strategies are often used (Clinger et al. 1999; Hieb et al. 1990), their bounded stack caches can be regarded as special parts of the nursery. That allows a regional collector to deal with them as though the mutator uses a pure heap strategy.

3. Collection Policies

This section describes the policies the collector follows to achieve scalability, even in the worst case.

Some of the policies are parameterized by numerical parameters: F_1 (described in Section 2.3), F_2 (3.2), F_3 (3.2), R (3.3), S (3.3), L_{soft} and L_{hard} (3.6). See section 5 for typical values. These parameters provide implementors with valuable flexibility, but we assume that the values of these parameters will be fixed by the implementors of a regional collector, and will not be tailored for particular mutators.

3.1 Minor, Major, Full, and Mark Cycles

The nursery is collected every time a region is collected, but the nursery may also be collected without collecting a region. A collection that collects only the nursery is a *minor collection*. A collection that collects both the nursery and a region is a *major collection*.

The interval between successive collections, whether minor or major, is a *minor cycle*. The interval between major collections is a *major cycle*.

The interval between successive initiations of the summarization process is a *summarization cycle*.

Regions are ordered arbitrarily, and collected in roughly round-robin fashion (see Figure 1), skipping popular and empty (unfilled) regions. When all non-popular, non-empty regions have been collected, a new *full cycle* begins.

The snapshot-at-the-beginning marking process is initiated at the start of a new full cycle. The interval between successive initiations of the marking process is a *mark cycle*.

Our proofs assume that mark and full cycles coincide, because worst-case mutators require relatively frequent marking (to limit the size of the remembered set and to reduce floating garbage). On normal programs, however, the mark cycle may safely be several times as long as a full cycle.

Usually there are F_1 summarization cycles per full cycle, but that can drop to F_1/F_3; see Section 3.3.

The number of major collections per full cycle is bounded by the number of regions N/R, where N is the total size of all regions.

The number of minor collections per major cycle is mostly determined by the promotion rate and by two parameters that express the desired (soft) ratio and a mandatory hard bound on N divided by the peak live storage.

3.2 Summarization Details

If the number of summary sets computed exceeds a fixed fraction $1/(F_1F_2)$ of the heap's regions, then the summarization process can be suspended until one of the regions associated with the newly computed summary sets is scheduled for the next collection.

If on the other hand the summarization process has to wave off the construction of too many summary sets, then the summarization process makes another pass over the remembered set, computing summary sets for a different group of regions. The maximum number of passes that might be needed before $1/(F_1F_2)$ of the heap's regions have been summarized is a parameter F_3 whose value depends upon parameters S, F_1, and F_2; see section 3.3.

Mutator actions can change which regions are classified as popular; popular regions can become unpopular, and vice versa. To prevent this from happening at a faster rate than the collection and summarization processes can handle, the mutator's allocation and assignment activity must be linked to collection and summarization progress (measured by the number of regions collected and progress made toward computation of summary sets).[3] As explained in 4.2, this extremely rare contention between the summarization process and the mutator determines the theoretical worst-case MMU of the collector.

When a region is collected, its surviving objects move and its other objects disappear. Entries for reclaimed objects must be removed from all existing summary sets, and entries for surviving objects must be updated to reflect the new addresses. A good representation for summary sets allows this updating to be done in time proportional to the size of the collected region.

3.3 Popular Regions

Suppose there are N/R regions, each of size R, so the total storage occupied by all regions is N.

Definition 2. *A region is* popular *if its summary set would exceed S times the size of the region itself, where S is the collector's wave-off threshold.*

[3] This leads to a curious property: in a regional collector, allocation-free code fragments containing assignment operations can cause a collection (and thus object relocation).

It is impossible for all regions to be more popular than average. That observation generalizes to the following lemma.

Lemma 3. *If $S > 1$, then the fraction of regions that are popular is no greater than $1/S$.*

Proof. If there were more than $1/S$ popular regions, then the total size of the summary sets for all popular regions would be greater than

$$\frac{1}{S}\frac{N}{R}SR = N$$

That is impossible: there are only N words in all regions combined, so how could more than N words be pointing into the popular regions? □

Example: If $S = 3$, then at most $1/3$ of the regions are popular, and not collecting those popular regions will add at most 50% to the size of the heap.

Corollary 4. *Suppose marking cycles coincide with full cycles, and a new full cycle is about to start. Let P_{old} be the volume of reachable storage, as computed by the marking process, at the start of the previous full cycle, and let A be an upper bound on the storage allocated during the previous full cycle. If $S > 1$, then the fraction of regions that are popular is no greater than*

$$\frac{P_{old} + A}{S}$$

Mutator activity can make previously popular regions unpopular, and can make previously unpopular regions popular, but the number of new pointers into a region is bounded by the number of words allocated plus the number of distinct locations assigned. Furthermore the fraction of popular regions can approach $1/S$ only if there are very few pointers into the unpopular regions. That means the mutator would have to do a lot of work before it could prevent a second or third pass of the summarization process from succeeding, provided of course that the collector's parameters are well-chosen.

Recall that the summarization process attempts to create summary sets for $1/F_1$ of the regions in each pass, and that it keeps making those passes until it has created summary sets for $1/(F_1F_2)$ of the regions.

Lemma 5. *Suppose S, F_1, and F_2 are greater than 1, and F_3 is a positive integer. Suppose also that*

$$c = \frac{F_2F_3 - 1}{F_1F_2}S - 1 > 0$$

and the mutator is limited to cN words allocated plus distinct locations assigned while the summarization process is performing up to F_3 passes. Then F_3 passes suffice.

Proof. We calculate the smallest number of allocations and assignments cN that would be required to leave at least i regions popular at the end of the summarization cycle. If i is less than or equal to the bound given by lemma 3, then no allocations/assignments are needed. Otherwise the smallest number of allocations/assignments occurs when the bound given by lemma 3 is met at both the beginning and end of the summarization cycle.[4] If that bound is met at the beginning of the cycle, then all non-popular regions have no pointers into them, and it takes SR allocations/assignments to create another popular region.

The summarization process will compute usable summaries for at least $1/(F_1F_2)$ of all N/R regions if

$$\frac{1}{F_1F_2}\frac{N}{R} \quad \leq \quad \frac{F_3}{F_1}\frac{N}{R} - \frac{1}{S}\frac{N}{R} - \frac{cN}{SR}$$

Equivalently

$$\begin{aligned} c &\leq \left(\frac{F_3}{F_1} - \frac{1}{S} - \frac{1}{F_1F_2}\right)S \\ &= \frac{F_2F_3 - 1}{F_1F_2}S - 1 \end{aligned}$$

□

That lemma, when combined with an upper bound for the duration of a collection, basically determines the theoretical worst-case MMU. See section 4.2.

For simplicity, we will henceforth assume that F_1/F_3 is an integer.

The following lemma bounds the number of regions that will not be collected during a full cycle.

Lemma 6. *Within any full cycle, the fraction of regions whose summary sets are not computed by the summarization process is no greater than*

$$1 - \frac{1}{F_2F_3}$$

Proof. Each summarization cycle makes up to F_3 passes, summarizing $1/F_1$ of the regions in each pass over the remembered set, to obtain at least $1/(F_1F_2)$ usable summary sets. In the worst case, there are F_1/F_3 summarization cycles in a full cycle. The largest possible fraction of unusable summary sets is therefore

$$\frac{F_1}{F_3}\left(\frac{F_3}{F_1} - \frac{1}{F_1F_2}\right) = 1 - \frac{1}{F_2F_3}$$ □

Each major collection consumes one summary set. The worst-case MMU is calculated by assuming each summary cycle yields only

$$\frac{1}{F_1F_2} \cdot \frac{N}{R}$$

usable summaries. The worst-case MMU is therefore unaffected by starting each summarization cycle when the number of summary sets has been reduced to the value used to calculate the worst-case MMU.

Corollary 7. *The space occupied by summary sets is never more than*

$$\frac{SF_3}{F_1}N$$

Proof. During any summarization cycle, the space occupied by the summary sets being computed is bounded by $N + cN$. Hence the total space occupied by all summary sets is bounded by

$$\begin{aligned} &\left(\frac{1}{F_1F_2} \cdot \frac{N}{R}\right)SR + N + cN \\ = &\frac{SN}{F_1F_2} + N + \left(\frac{F_2F_3 - 1}{F_1F_2}S - 1\right)N \\ = &\frac{SF_3}{F_1}N \end{aligned}$$

□

[4] In other words, starting with fewer popular regions *increases* the mutator activity required to end the cycle with large i; we are deriving the *minimum* number of actions required.

3.4 Fragmentation

As was mentioned in section 2 and justified in section 7, the regional collector assumes objects are limited to some size $m < R$. The Cheney algorithm ensures that worst-case fragmentation in collected regions is less than m/R. Our calculations assume that ratio is negligible.

3.5 Work-Based Accounting

The regional collector performs work in proportion to a slightly peculiar accounting of mutator work. The peculiarities reflect our focus on worst cases, which occur when the rate of promotion out of the nursery is nearly 100% and the mutator spends almost all of its time allocating storage and performing assignments.

The mutator's work is measured by the volume of storage that survives to be promoted out of the nursery and the number of assignments that go through the write barrier. If we ignore the nursery (which has little effect on the worst case) then promoted objects are, in effect, newly allocated within some region.

The collector's work is measured by the number of regions collected. A full cycle concludes when all nonempty, non-popular regions have been collected, so the number of regions collected also measures time relative to the current full cycle. That notion of time drives the scheduling of marking and summarization processes.

The marking and summarization processes are counted as overhead, not work. Our calculations assume their cost is evenly distributed (at the fairly coarse resolution of one major cycle) over the interval they are active, using mutator work as the measure of time. That makes sense for worst cases, and overstates the collector's relative overhead when the mutator does things besides allocation and assignments (because counting those other things as work would increase the mutator utilization).

3.6 Matching Collection Work to Allocation

At the beginning of a full cycle, the regional collector calculates the amount of storage the mutator will allocate (that is, promote into regions) during the full cycle.

Almost any policy that makes the mutator's work proportional to the collector's work would suffice for the proof of our main theorem, but the specific values of worst-case constants are sensitive to details of the policy. Furthermore, several different policies may have essentially the same worst-case performance but radically different overall performance on normal programs.

We are still experimenting with different policies. The policy stated below is overly conservative, but allows simple proofs of this section's lemmas because A is a monotonically increasing function of the peak live storage, and does not otherwise depend upon the current state of the collector.

Outside of this section, nothing depends upon the specific policy stated below. The proof of our main theorem relies only upon its properties as encapsulated by lemmas 9 and 10.

The following policy computes a hard lower bound for the amount of free space that will become available as regions are collected during this full cycle, and divides that free space equally between this full cycle and the next. If promoting that volume of storage might exceed the desired bound on heap size, then the promotion budget for this full cycle is reduced accordingly.

Policy 8. *The promotion to be performed during the coming full cycle is*

$$A = \min\left(\frac{1}{2}((1-k)L_{hard} - 1)P_{old}, (L_{soft} - 1)P_{old}\right)$$

where

- *k is any fixed upper bound for the fraction of nonempty regions that go uncollected within a full cycle. (Lemma 6 calculates a specific value for k.)*
- *P_{old} is the peak live storage, computed as the maximum value of N_{old} (see below).*
- *N_{old} is the volume of reachable storage at the beginning of the previous full cycle, as measured by the marking process during that cycle; if this is the first full cycle, then N_{old} is the size of the initial heap plus some headroom.*
- *L_{soft} is the desired ratio of N to peak live storage.*
- *$L_{hard} > 1/(1-k)$ is a fixed hard bound on the ratio of N to peak live storage at the beginning of a full cycle.*

The two lemmas below express the only properties that A must have.

Lemma 9. *If the collector parameters are consistent, then A is in $\Theta(P_{old})$.*

The following lemma states the regional collector's most critical invariant, and establishes that this invariant is preserved by every full cycle.

The critical insight of its proof is that the Cheney collection process reclaims all storage that was unreachable as of the beginning of the previous full cycle, except for the bounded fraction of objects that lie in uncollected regions. Furthermore there is no fragmentation among the survivors of collected regions, so the total storage in all regions at the end of a full cycle, excluding free space recovered by the cycle, is the sum of the total storage occupied by the survivors, the regions that aren't collected, and the storage that was promoted into regions during the cycle.

Lemma 10. *Let N_0 be the volume of storage in all regions, including live storage and garbage but not free space, at the beginning of a full cycle. Then $N_0 \leq N \leq L_{hard}P_{old}$.*

Proof. The lemma is true at the beginning of the first full cycle.

At the beginning of the second full cycle, N_0 consists of

- storage that was reachable at the beginning of the first full cycle (bounded by N_{old})
- storage in uncollected regions (bounded by kN)
- storage promoted into regions during the previous full cycle (bounded by A)

At the beginning of subsequent full cycles, N_0 consists of

- storage that was reachable at the beginning of the full cycle before the previous full cycle and is still reachable (bounded by N_{old})
- storage in uncollected regions (bounded by kN)
- storage promoted into regions during the previous full cycle (bounded by A)
- storage promoted into regions during the cycle before the previous full cycle (bounded by A, because A is nondecreasing)

Therefore

$$\begin{aligned} N_0 &\leq N_{old} + kN + A + A \\ &= N_{old} + kN + ((1-k)L_{hard} - 1)P_{old} \\ &\leq P_{old} + kL_{hard}P_{old} + ((1-k)L_{hard} - 1)P_{old} \\ &= L_{hard}P_{old} \end{aligned}$$

□

4. Worst-case Bounds

The subsections below sketch proofs for the three parts of our main theorem, which was stated in section 1.2.

We use asymptotic calculations because we cannot know the hardware- and software-dependent relative cost of basic operations such as allocations, write barriers, marking or tracing a word of memory, and so on. Constant factors are important, however, so we make a weak attempt to estimate some constants by assuming that all basic operations have the same cost per word. That is roughly true, but only for appropriate values of "roughly". The constant factors calculated for space may be more trustworthy than those calculated for time.

4.1 GC Pauses

It's easy to calculate an upper bound for the duration of major collections. The size of the region to be collected is a constant R. The size of its summary set is bounded by SR. The summary and mark-stack state to be updated is bounded by $O(R)$. A Cheney collection of the region therefore takes time $O(R + SR) = O(R)$.

4.2 Worst-case MMU

For any resolution Δt, the minimum mutator utilization is the infimum, over some set of intervals of length Δt, of the mutator's CPU time during that interval divided by Δt (Cheng and Blelloch 2001). The MMU is therefore a function from resolutions to the interval $[0, 1]$.

The obvious question is: What set of intervals are we talking about? In most cases, an MMU is defined over the intervals recorded during some specific execution of some specific benchmark on some specific machine. We'll call that an *observed MMU*.

Our main theorem uses a very different notion of MMU, which can be regarded as the infimum of observed MMUs over all possible executions of all possible benchmarks. We have been referring to that notion as the *theoretical worst-case MMU*.

The theoretical worst-case MMU is the notion that matters when we talk about worst-case guarantees or scalable algorithms.

The theoretical worst-case MMU is easily bounded above using observed MMUs; for example, an observed MMU of zero implies a theoretical worst-case MMU of zero. On the other hand, we cannot use observed MMUs to prove that a regional collector's theoretical worst-case MMU is bounded below by a non-zero constant. Our only hope is to prove something like our main theorem.

Some programs reach a storage equilibrium, which allows us to define the inverse load factor L as the ratio of heap size to reachable heap storage. Although some collectors can do better on some programs, it appears that, for any garbage collector, the theoretical worst-case ratio of allocation to marking is less than or equal to $L - 1$, from which it follows that there must be resolutions at which the worst-case MMU is less than or equal to

$$\frac{L-1}{(L-1)+1} = \frac{L-1}{L}$$

For a stop-and-collect collector, the worst-case MMU is zero for intervals shorter than the duration of the worst-case collection. For collectors that occasionally perform a full collection, taking time proportional to the reachable storage, the theoretical worst-case MMU is therefore zero at all resolutions. If there is some finite bound on the worst-case gc pause, however, then the theoretical worst-case MMU may be positive for sufficiently large resolutions.

Our main theorem claims this is true for a regional collector at resolutions greater than $3c_0$, where c_0 is a bound on the worst-case duration of a gc pause. At that resolution and above, the worst case occurs when two worst-case gc pauses surround a mutator interval in which the mutator performs a worst-case (small) amount of work. The two gc pauses take $O(R)$ time, so we need to show that the mutator will perform $\Omega(R)$ work between every two major collections.

The regional collector performs $\Theta(N/R)$ major collections per full cycle, and the scheduling of those collections is driven by mutator work. Between two successive major collections, the mutator performs $\Omega(AR/N)$ work, where A, the promotion per full cycle as defined in section 3.6, is in $\Theta(P_{old})$ and therefore in $\Omega(N)$.

If the regional collector had no overhead outside of major collections, the paragraph above would establish that the theoretical worst-case MMU at that resolution is bounded below by a constant. Since the regional collector does have overhead from the marking and summarization processes, we have yet to establish that (1) the overhead per major cycle of those processes is $O(R)$ and (2) their overhead is distributed fairly evenly within the interval; that is, there are no subintervals of duration $3c_0$ or longer that have an overly high concentration of overhead or overly low fraction of mutator work.

The marking process's overhead per full cycle is $O(N)$, and standard scheduling algorithms suffice to ensure that its overhead per major cycle is $O(R)$, with that overhead being quite evenly distributed when observed at the coarse resolution of $3c_0$.

The summarization process, as described in sections 2.3 and 3.3, is more complicated. The summarization process performs up to F_3 passes over the remembered set per summarization cycle. Each pass takes $O(N)$ time to scan the remembered set, while creating

$$O\left(\frac{1}{F_1}\frac{N}{R}SR\right)$$

entries in the summary sets. There are between F_1 and F_1/F_3 summarization cycles per full cycle, distributed as evenly as those tight constant bounds allow. In conclusion, the summarization process has $O(N)$ overhead per full cycle and $O(R)$ overhead per major cycle.

That would complete the proof of part 2, except for one nasty detail mentioned in section 2.3 and lemma 5: The mutator's work during summarization is limited to cN, where c is the constant defined in lemma 5.

That doesn't interfere with the proof of part 2, because the mutator is still performing $\Theta(N)$ work per summarization cycle, but it does lower mutator utilization. If we assume that all basic operations have about the same cost per word, then the theoretical worst-case MMU at sufficiently large resolutions is a constant of which we have some actual knowledge.

Lemma 11. *When regarded as a function of the collector's parameters, the regional collector's theoretical worst-case MMU is roughly proportional to*

$$\frac{SF_2F_3 - S - F_1F_2}{(S+1)(F_2F_3+2) + F_1F_2F_3}$$

Proof. The worst-case MMU is proportional to the worst-case mutator work accomplished during a major cycle, divided by the worst-case cost of the marking and summarization processes during a major cycle plus the worst-case cost of the two major collections that surround the mutator work. We assume that work and costs are spread evenly across the relevant cycles; any bounded degree of unevenness can be absorbed by the constant of proportionality.

The number of regions collected during a worst-case summarization cycle is

$$d = \frac{1}{F_1F_2}\frac{N}{R}$$

- The worst-case mutator work per major cycle is cN/d.
- The worst-case cost of summarization per major cycle is

$$F_3N + \frac{F_3}{F_1}\frac{N}{R}SR = (F_3 + \frac{F_3}{F_1}S)N$$

divided by d.

- The worst-case cost of the marking process during a major cycle is F_2F_3R, which is N divided by the worst-case number of major collections during a full cycle (as given by lemma 6).
- The worst-case cost of a major collection is $R + SR$.

The theoretical worst-case MMU is therefore roughly proportional to

$$\frac{F_1F_2cR}{2(1+S)R + F_1F_2(F_3 + SF_3/F_1)R + F_2F_3R}$$
$$= \frac{SF_2F_3 - S - F_1F_2}{(S+1)(F_2F_3+2) + F_1F_2F_3}$$

□

That calculation was pretty silly, but gives us quantitative insight into how much we can improve the theoretical worst-case MMU by choosing good values for the collector's parameters or by designing a more efficient summarization process.

4.3 Worst-case Space

The regional collector allocates a new region only when the current set of regions does not have enough free space to accomodate all of the objects that need to be promoted out of the nursery. Lemmas 9 and 10 therefore establish that N, the total storage occupied by all regions, is in $\Theta(P_{old})$ (where P_{old} is a lower bound for the peak live storage).

The remembered set is $O(N)$. The set of previously computed summary sets that have not yet been consumed by a major collection is $O(N)$. The set of summary sets currently under construction is $O(N)$. The mark bitmap is $O(N)$. Each mark stack (one per region) is $O(R)$, so the total size for all mark stacks is $O(N)$.

The total space required by the regional collector is therefore $\Theta(P_{old})$. The specific constants of proportionality depend upon collector parameters L_{hard}, S, F_1, and F_2 as well as details of the collector's data structures; for example, the size of the mark bitmap might be N, $N/2$, $N/4$, $N/8$, $N/32$, or $N/64$ depending on object alignment, granularity of marking, and number of bits per mark. With plausible assumptions about data structures, the theoretical worst-case space is about

$$\left(\left(\frac{5}{4} + \frac{SF_3}{F_1}\right) L_{hard} + \frac{1}{2}\right) P$$

where P is the peak reachable storage.

No program can reach theoretical worst-case bounds for all of the collector's data structures simultaneously. For example, the mark stack's worst case is achieved when the heap is filled by a single linked structure of objects with only two fields. That means half the pointers are perfectly distributed among regions, which halves the worst-case number of popular regions; it also removes the factor of L_{hard}, because all objects that get pushed onto the mark stack are reachable. On gc-intensive benchmarks, our prototype uses about the same amount of storage as stop-and-copy or generational collectors.

4.4 Floating Garbage

Floating garbage is storage that is reachable from the remembered set but is not reachable from mutator structures (and will not be marked by the next snapshot-at-the-beginning marking process).

In the calculations above, the peak reachable storage P does not include floating garbage, but the theoretical worst-case bounds do include floating garbage. In this section, we calculate a bound for how much of the worst-case space can be occupied by floating garbage.

When bounding the space used by collectors that never perform a full collection, the hard part is to find an upper bound for floating garbage. The regional collector is especially interesting because

- When a region is collected, its objects that were unreachable as of the beginning of the most recently completed marking cycle will be reclaimed.
- The regional collector does not guarantee that all unreachable objects will eventually be collected.
- The regional collector does guarantee that the total volume of unreachable objects is always bounded by a small constant times the total volume of reachable objects.

Suppose some object x, residing in some region r, becomes unreachable. If there are no references to x from outside r, then x will be reclaimed the next time r is collected.

If there are references to x from outside r, then those references will be removed from the remembered set at the end of the first marking cycle that begins after x becomes unreachable (because all references to an unreachable object are from unreachable objects). Then x will be reclaimed by the first collection of r that follows the completion of that marking cycle.

On the other hand, there is no guarantee that r will ever be collected. r will remain forever uncollected if and only if the summarization process deems r popular on every attempt to construct r's summary set.

Lemma 3 proves that the total volume of popular regions is no greater than N/S. Lemma 10 proves that $N \leq L_{hard}P$, where P is the peak live storage. Hence the total volume of perpetually uncollected garbage is no greater than L_{hard}/S times the peak live storage.

4.5 Collector Parameters

Most of the collector's parameters can be changed at the beginning of any full cycle. If the parameters change at the beginning of a full cycle, then it will take at most two more full cycles for the collector to perform within the theoretical worst-case bounds for the new parameters.

5. Near-Worst-Case Benchmarks

We have implemented a prototype of the regional collector, and will provide a more detailed report on its engineering and performance in some other paper. For this paper, we compare its performance to that of several other collectors on a very simple but extremely gc-intensive benchmark (Clinger 2009).

The benchmark repeatedly allocates a list of one million elements, and then stores the list into a circular buffer of size k. The number of popular objects (used as list elements) is a separate parameter p; with $p = 0$, the list elements are small integers, which are usually represented by non-pointers that the garbage collector does not have to trace.

To illustrate scalability and the effect of popular objects, we ran three versions of the benchmark:

- with $k = 10$ and $p = 0$
- with $k = 50$ and $p = 0$
- with $k = 50$ and $p = 50$

All three versions allocate exactly the same amount of storage, but the peak storage with $k = 10$ is about one fifth of the peak storage with $k = 50$. The third version, with popular objects, is the most challenging benchmark we have been able to devise for the regional collector. The queue-like object lifetimes of all three versions make them near-worst-case benchmarks for generational

system	version	technology	elapsed (sec)	gc time (sec)	max gc pause (sec)	max variation (sec)	max RSIZE (MB)
Larceny	prototype	regional	192	170	.07	.60	386
Gambit	v4.4.3	stop©	63	44		.52	493
Ypsilon	0.9.6-update3	mostly concurrent	265	$\geq$ 53	.64	?	711
Sun JVM	1.5.0	generational	175	?		.78	333
Larceny	prototype	generational	109	88	.80	.88	555
Sun JVM	1.5.0	parallel	275	?		.91	511
Larceny	prototype	stop©	76	55	.90	.94	518
Chicken	4.0.0	Cheney-on-the-MTA	87	36		1.	490
PLT	v4.1.4	generational	227	211		1.	617
Ikarus	0.0.3	generational	264	242		2.25	1055
Sun JVM	1.5.0	incremental mark/sweep	409	?		3.41	530

Figure 2. GC-intensive performance with about 160 MB of live storage.

system	version	technology	elapsed (sec)	gc time (sec)	max gc pause (sec)	max variation (sec)	max RSIZE (MB)
Larceny	prototype	regional	212	187	.11	.7	1808
Ypsilon	0.9.6-update3	mostly concurrent	24971	$\geq$ 24818	2.4	?	2067
Gambit	v4.4.3	stop©	68	47		2.5	2363
Chicken	4.0.0	Cheney-on-the-MTA	118	62		4.	1955
Sun JVM	1.5.0	parallel	311	?		4.2	1973
Larceny	prototype	generational	149	128	4.2	4.3	2073
Larceny	prototype	stop©	119	95	4.5	4.5	2058
Sun JVM	1.5.0	generational	212	?		4.9	1497
PLT	v4.1.4	generational	286	273		5.	2109
Ikarus	0.0.3	generational	419	371		11.6	2575
Sun JVM	1.5.0	incremental mark/sweep	457	?		15.8	2083

Figure 3. GC-intensive performance with about 800 MB of live storage.

system	version	technology	elapsed (sec)	gc time (sec)	max gc pause (sec)	max variation (sec)	max RSIZE (MB)
Larceny	prototype	regional	618	592	.35	2.9	1865
Gambit	v4.4.3	stop©	72	51		2.7	2363
Ypsilon	0.9.6-update3	mostly concurrent	28366	$\geq$ 28212	2.89	?	1772
Sun JVM	1.5.0	parallel	314	?		4.1	1918
Larceny	prototype	generational	162	141	4.5	4.6	2064
Larceny	prototype	stop©	120	96	4.8	4.8	2060
Chicken	4.0.0	Cheney-on-the-MTA	127	69		5.	1955
Sun JVM	1.5.0	generational	216	?		5.0	1497
PLT	v4.1.4	generational	339	320		5.	2089
Ikarus	0.0.3	generational	427	409		10.7	2588
Sun JVM	1.5.0	incremental mark/sweep	479	?		18.1	2083

Figure 4. GC-intensive performance with 800 MB live storage and 50 popular objects.

collectors in general, and their simplicity and regularity make the results easy to interpret.

To eliminate pair-specific optimizations that might give Larceny (and some other systems) an unfair advantage, the lists are constructed from two-element vectors. Hence the representation of each list in Scheme is likely to resemble the representation used by Java and similar languages. In Larceny and in Sun's JVM, each element of the list occupies four 32-bit words (16 bytes), and each list occupies 16 megabytes.

The benchmarks allocate one thousand of those lists, which is enough for the timing to be dominated by the steady state but small enough for convenient benchmarking.

We benchmarked a prototype fork of Larceny with three different collectors. The regional collector was configured with a 1-megabyte nursery, 8-megabyte regions (R), a waveoff threshold of $S = 8$, and parameters $F_1 = 2$, $F_2 = 2$, and $F_3 = 1$; these parameters have worked well for a wide range of benchmarks, and were not optimized for the particular benchmarks reported here. To make the generational collector more comparable to the regional collector, it was benchmarked with a nursery size of 1 MB instead of the usual 4 MB.

For perspective, we benchmarked several other systems as well. We ran all benchmarks on a MacBook Pro equipped with a 2.4 GHz Intel Core 2 Duo (with two processor cores) and 4 GB of 667 MHz DDR2 SDRAM. Only three of the collectors made use of the second processor core: Ypsilon, Sun's JVM with the parallel collector, and Sun's JVM with the incremental mark/sweep collector. For those three systems, the total cpu time was greater than the elapsed times reported in this paper.

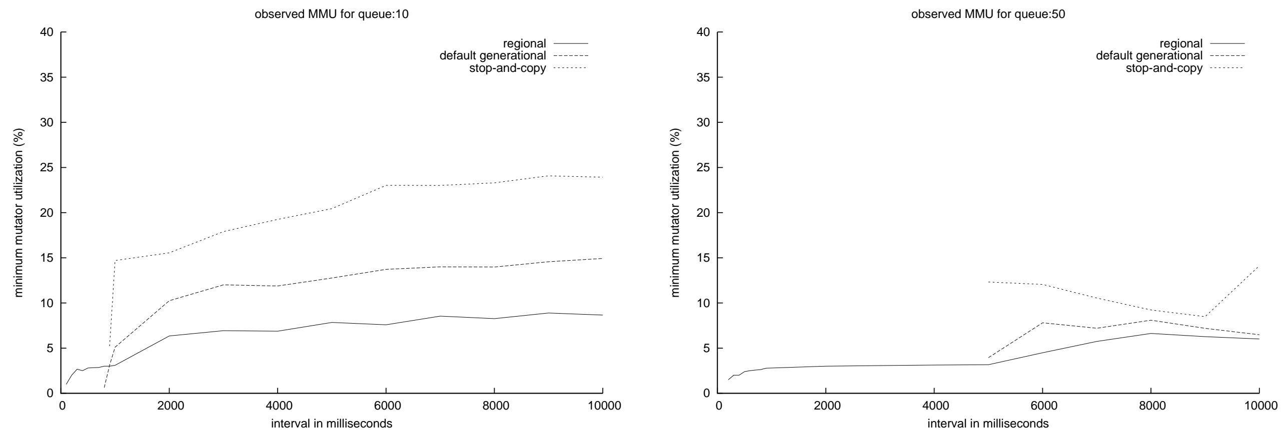


Figure 5. Observed MMU for $k = 10$ and $k = 50$.

Figures 2, 3, and 4 report the elapsed time (in seconds), the total gc time (in seconds), the duration of the longest pause to collect garbage (in seconds), the maximum variation (calculated by subtracting the average time to create a million-element list from the longest time to create one of those lists), and the maximum RSIZE (in megabytes) reported by `top`.

For most collectors, the maximum variation provides a good estimate of the longest pause for garbage collection. For the regional collector, however, most of the maximum variation is caused by uneven scheduling of the marking and summarization processes. With no popular objects, the regional collector's total gc time includes 51 to 54 seconds of marking and about 1 second of summarization. With 50 popular objects, the marking time increased to 104 seconds and the summarization time to 152 seconds. It should be possible to decrease the maximum variation of the regional collector by improving the efficiency of its marking and summarization processes and/or the regularity of their scheduling.

Figure 5 shows the MMU (minimum mutator utilization as a function of time resolution) for the three collectors implemented by our prototype fork of Larceny.

Although none of the other collectors were instrumented for MMU, their MMU would be zero at resolutions up to the longest gc pause, and their MMU at every resolution would be less than their average mutator utilization (which can be estimated by subtracting the total gc time from the elapsed time and dividing by the elapsed time).

As can be seen from figures 2 and 3, simple garbage collectors often have good worst-case performance. Gambit's non-generational stop© collector has the best throughput on this particular benchmark, followed by Larceny's stop© collector and Chicken's Cheney-on-the-MTA (which is a relatively simple generational collector).

Of the benchmarked collectors, Sun's incremental mark/sweep collector most resembles a soft real-time collector; it combines low throughput with inconsistent mutator utilization. Ypsilon performs poorly on the larger benchmarks, apparently because it needs more than 2067 megabytes of RAM, which is the largest heap it supports; Ypsilon's representation of a Scheme vector may also consume more space than in other systems.

The regional collector's throughput and gc pause times are degraded by popular objects, but its gc pause times remain the best of any collector tested, while using less memory than any system except for Sun's default generational collector.

The regional collector's scalability can be seen by comparing its pause times and MMU for $k = 10$ and $k = 50$. The maximum pause time increases only slightly, from .07 to .11 seconds. For all other systems whose pause times were measured with sub-second precision, the pause time increased by a factor of about 5 (because multiplying the peak live storage by 5 also multiplies the time for a full collection by 5). The regional collector's MMU is almost the same for $k = 10$ as for $k = 50$; for all other collectors, the MMU degrades substantially as the peak live storage increases.

6. Related Work

6.1 Generational garbage collection

Generational collection was introduced by (Lieberman and Hewitt 1983). A simplification of that design was first implemented by (Ungar 1984). Most modern generational collectors are modeled after Ungar's, but our regional collector's design is more similar to that of Lieberman and Hewitt.

6.2 Heap partitioning

Our regional collector is centered around the idea of partitioning the heap and collecting the parts independently. (Bishop 1977) allows single areas to be collected independently; his work targets Lisp machines and requires hardware support.

The *Garbage-First* collector of (Detlefs et al. 2004) inspired many aspects of our regional collector. Unlike the garbage-first collector, which uses a points-into remembered set representation with no size bound, we use a points-outof remembered set representation and points-into summaries which are bounded in size. The garbage-first collector does not have worst-case bounds on space usage, pause times, or MMU. According to Sun, the garbage-first collector's gc pause times are "sometimes better and sometimes worse than" the incremental mark/sweep collector's (Sun Microsystems 2009).

The *Mature Object Space* (a.k.a. *Train*) algorithm of (Hudson and Moss 1992) uses a fixed policy for choosing which regions to collect. To ensure completeness, their policy migrates objects across regions until a complete cycle is isolated to its own train and then collected. This gradual migration can lead to significant problems with floating garbage. Our marking process eliminates floating garbage in collected regions, while our handling of popular regions provides an elegant and novel solution that bounds the worst-case storage requirements.

The *Beltway* collector of (Blackburn et al. 2002) uses heap partitioning and clever infrastructure to enable flexible selection of collection policies via command line options. Their policy selection is expressive enough to emulate the behavior of semi-space, genera-

tional, renewal-older-first, and deferred-older-first collectors. They demonstrate that having a more flexible policy parameterization can introduce improvements of 5%, 10%, and up to 35% over a fixed generational collection policy. Unfortunately, in the Beltway system one must choose between incremental or complete collection. The Beltway collector does not provide worst-case guarantees independent of mutator behavior.

The *MarkCopy* collector of (Sachindran and Moss 2003) breaks the heap down into fixed sized *windows*. During a collection pause, it builds up a remembered set for each window and then collects each window in turn. An extension interleaves the mutator process with individual window copy collection; one could see our design as taking the next step of moving the marking process and remembered set construction off of the critical path of the collector.

The Parallel Incremental Compaction algorithm of (Ben-Yitzhak et al. 2002) also has similarities to our approach. They select an area of the heap to collect, and then concurrently build a summary for that area. However, they construct their points-into set by tracing the whole heap, rather than maintaining points-outof remembered sets. Their goals are also different from ours; their technique adds incremental compaction to a mark-sweep collector, while we provide utilization and space guarantees in a copying collector.

6.3 Older-first garbage collection

Our design employs a round-robin policy for selecting the region to collect next, focusing the collector on regions that have been left alone the longest. Thus our regional collector, like older-first collectors (Stefanović et al. 2002; Hansen and Clinger 2002), tends to give objects more time to die before attempting to collect them.

6.4 Bounding collection pauses

There is a broad body of research on bounding the pause times introduced by garbage collection, including (Baker 1978; Brooks 1984; Appel et al. 1988; Yuasa 1990; Boehm et al. 1991; Baker 1992; Nettles and O'Toole 1993; Henriksson 1998; Larose and Feeley 1998). In particular, (Blelloch and Cheng 1999) provides proven bounds on pause-times and space-usage.

Several attempts to bring the pause-times down to precisions suitable for real-time applications run afoul of the problem that bounding an individual pause is not enough; one must also ensure that the mutator can accomplish an appropriate amount of work in between the pauses, keeping the processor utilization high. (Cheng and Blelloch 2001) introduces the MMU metric to address this issue. That paper presents an *observed* MMU for a parallel real-time collector, not a theoretical worst-case MMU.

6.5 Collection scheduling

Metronome (Bacon et al. 2003a) is a hard real-time collector. It can use either time- or work-based collection scheduling, and is mostly non-moving, but will copy objects to reduce fragmentation. Metronome also requires a read barrier, although the average overhead of the read barrier is only 4%. More significantly, Metronome's guaranteed bounds on utilization and space usage depend upon the accuracy of application-specific parameters; (Bacon et al. 2003b) extends this set of parameters to provide tighter bounds on collection time and space overhead.

Similarly, (Robertz and Henriksson 2003) depends on a supplied schedule to provide real-time collector performance. Unlike Metronome, it schedules work according to collection cycle times rather than finer grained quanta; like Metronome, it provides a proven bound on space usage (that depends on the accurary of application-specific parameters).

In contrast to those designs, our regional collector provides worst-case guarantees independent of mutator behavior, but cannot provide millisecond-resolution guarantees. Our regional collector is mostly copying, has no read barrier, and uses work-based accounting to drive the collection policy.

6.6 Incremental and concurrent collection

There are many treatments of concurrent collectors dating back to (Dijkstra et al. 1978). In our collector, reclamation of dead object state is not performed concurrently with the mutator, but the activity of the summarization and marking processes could be.

Our summarization process was inspired by the performance of Detlefs' implementation of a concurrent thread that refines data within the remembered set to reduce the effort spent towards scanning older objects for roots during a collection pause (Detlefs et al. 2002).

The summarization and marking processes require a write barrier, which we piggy-back onto the barrier in place to support generational collection. This is similar to how (Printezis and Detlefs 2000), building on the work of (Boehm et al. 1991), merges the overhead of maintaining concurrency related invariants with the overhead of maintaining generational invariants.

7. Future Work

Our current prototype interleaves the marking and summarization processes with the mutator, scheduling at the granularity of minor cycles and the processing of write barrier logs. Both the marking and summarization processes could be concurrent with the mutator, which would improve throughput on programs that do not fully utilize all processor cores. The marking process was actually implemented as a concurrent thread by one of our earlier prototypes, but the current single-threaded prototype makes it easier to measure every process's effect on throughput.

The collections performed by the regional collector can themselves be parallelized, but that is essentially independent of the design.

We assume that object sizes are bounded, so every object will fit into a region. Because we have implemented our prototype in Larceny, we can change both the compiler and the run-time representations of objects, choosing representations that break extremely large objects into pieces of bounded size.

The regional collector's nursery provides most of the benefits associated with generational garbage collection. Although the regional collector sacrifices some throughput on extremely gc-intensive programs, its performance on more normal programs can and does approach that of contemporary generational collectors. We will offer a more complete report on our prototype's observed performance in a separate paper.

8. Conclusions

We have described and prototyped a regional collector, which is a new kind of generational garbage collector.

We have proved that the regional collector is scalable: It guarantees worst-case bounds for gc latency, minimum mutator utilization, and space usage, independent of the peak live storage and mutator behavior.

Such guarantees remain rare. Although our proof is not the first of its kind, it may be the first to guarantee worst-case bounds for MMU as well as latency and space.[5]

The regional collector incorporates novel and elegant solutions to the problems presented by popular objects and floating garbage.

[5] For example, Cheng and Blelloch proved that a certain hard real-time collector has nontrivial worst-case bounds for both gc latency and space, but they had not yet invented the concept of MMU (Blelloch and Cheng 1999).

We have prototyped the regional collector, using a near-worst-case benchmark to illustrate its performance.

References

Andrew W. Appel. *Compiling with Continuations*, chapter 16, pages 205–214. Cambridge University Press, 1992.

Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collecor with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, New Orleans, LA, January 2003a. ACM Press.

David F. Bacon, Perry Cheng, and V.T. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'2003)*, pages 81–92, San Diego, CA, June 2003b. ACM Press.

Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

Henry G. Baker. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992.

Ori Ben-Yitzhak, Irit Goft, Elliot Kolodner, Kean Kuiper, and Victor Leikehman. An algorithm for parallel incremental compaction. In David Detlefs, editor, *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 100–105, Berlin, June 2002. ACM Press.

Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, MIT Laboratory for Computer Science, May 1977. Technical report MIT/LCS/TR–178.

Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 153–164, Berlin, June 2002. ACM Press. ISBN 1-58113-463-0.

Guy E. Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In *Proceedings of SIGPLAN 1999 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 104–117, Atlanta, May 1999. ACM Press.

Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In Guy L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 256–262, Austin, TX, August 1984. ACM Press.

C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

Perry Cheng and Guy Blelloch. A parallel, real-time garbage collector. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 125–136, Snowbird, Utah, June 2001. ACM Press.

William D. Clinger. Queue benchmark for estimating worst-case gc pause times. Website, 2009. `http://www.ccs.neu.edu/home/will/Research/SW2009/`.

William D. Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, April 1999.

David Detlefs, William D. Clinger, Matthias Jacob, and Ross Knippel. Concurrent remembered set refinement in generational garbage collection. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '02)*, San Francisco, CA, August 2002.

David Detlefs, Christine Flood, Steven Heller, and Tony Printezis. Garbage-first garbage collection. In Amer Diwan, editor, *ISMM'04 Proceedings of the Fourth International Symposium on Memory Management*, Vancouver, October 2004. ACM Press.

Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

Lars Thomas Hansen and William D. Clinger. An experimental study of renewal-older-first garbage collection. In *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming (ICFP02)*, volume 37(9) of *ACM SIGPLAN Notices*, pages 247–258, Pittsburgh, PA, 2002. ACM Press.

Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.

R. Hieb, R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. *ACM SIGPLAN Notices*, 25(6):66–77, 1990.

Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Massachusetts, USA, 16–18 September 1992. Springer-Verlag.

Martin Larose and Marc Feeley. A compacting incremental collector and its performance in a production quality compiler. In Richard Jones, editor, *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 1–9, Vancouver, October 1998. ACM Press. ISBN 1-58113-114-3.

Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, 1983. ISSN 0001-0782.

Scott M. Nettles and James W. O'Toole. Real-time replication-based garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, Carnegie Mellon University, USA, June 1993. ACM Press.

Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In Tony Hosking, editor, *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, Minneapolis, MN, October 2000. ACM Press. ISBN 1-58113-263-8.

Sven Gestegard Robertz and Roger Henriksson. Time-triggered garbage collection: robust and adaptive real-time gc scheduling for embedded systems. In *ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'2003)*, pages 93–102, San Diego, CA, June 2003. ACM Press.

Narendran Sachindran and Eliot Moss. MarkCopy: Fast copying GC with less space overhead. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.

Darko Stefanović, Matthew Hertz, Stephen M. Blackburn, Kathryn S. Mckinley, J. Eliot, and B. Moss. Older-first garbage collection in practice: Evaluation in a java virtual machine. In *In Memory System Performance*, pages 25–36. ACM Press, 2002.

Sun Microsystems. Java HotSpot garbage collection. Website, 2009. `http://java.sun.com/javase/technologies/hotspot/gc/g1_intro.jsp`.

David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990.

Randomized Testing in PLT Redex

Casey Klein
University of Chicago
clklein@cs.uchicago.edu

Robert Bruce Findler
Northwestern University
robby@eecs.northwestern.edu

Abstract

This paper presents new support for randomized testing in PLT Redex, a domain-specific language for formalizing operational semantics. In keeping with the overall spirit of Redex, the testing support is as lightweight as possible—Redex programmers simply write down predicates that correspond to facts about their calculus and the tool randomly generates program expressions in an attempt to falsify the predicates. Redex's automatic test case generation begins with simple expressions, but as time passes, it broadens its search to include increasingly complex expressions. To improve test coverage, test generation exploits the structure of the model's metafunction and reduction relation definitions.

The paper also reports on a case-study applying Redex's testing support to the latest revision of the Scheme standard. Despite a community review period, as well as a comprehensive, manually-constructed test suite, Redex's random test case generation was able to identify several bugs in the semantics.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—testing tools; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—assertions, invariants, mechanical verification; D.2.4 [*Software Engineering*]: Software / Program Verification—assertion checkers; D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Design

Keywords Randomized test case generation, lightweight formal models, operational semantics

1. Introduction

Much like software engineers have to cope with maintaining a program over time with changing requirements, semantics engineers have to maintain formal systems as they evolve over time. In order to help maintain such formal systems, a number of tools that focus on providing support for either proving or checking proofs of such systems have been built (Hol [13], Isabelle [15], Twelf [16], and Coq [22] being some of the most prominent).

In this same spirit, we have built PLT Redex [8, 12]. Unlike other tools, however, Redex's goal is to be as lightweight as possible. In particular, our goal is that Redex programmers should write down little more than they would write in a formal model of their system in a paper and to still provide them with a suite of tools for working with their semantics. Specifically, Redex programmers write down the language, reduction rules, and any relevant metafunctions for their calculi, and Redex provides a stepper, hand-written unit test suite support, automatic typesetting support, and a number of other tools.

To date, Redex has been used with dozens of small, paper-size models and a few large models, the most notable of which is the formal semantics in the current standard of Scheme [21]. Redex is also the subject of a book on operational semantics [7].

Inspired by QuickCheck [5], we recently added a random test case generator to Redex and this paper reports on our experience with it. The test case generator has found bugs in every model we have tested with it, even the most well-tested and widely used models (as discussed in section 4).

The rest of the paper is organized as follows. Section 2 introduces Redex by presenting the formalization of a toy programming language. Section 3 demonstrates the application of Redex's randomized testing facilities. Section 4 presents our experience applying randomized testing to a formal model of R^{6}RS Scheme. Section 5 describes the general process and specific tricks that Redex uses to generate random terms. Finally, section 6 discusses related work, and section 7 concludes.

2. Redex by Example

Redex is a domain-specific language, embedded in PLT Scheme. It inherits the syntactic and lexical structure from PLT Scheme and allows Redex programmers to embed full-fledged Scheme code into a model, where appropriate. It also inherits DrScheme, the program development environment, as well as a large standard library. This section introduces Redex and context-sensitive reduction semantics through a series of examples, and makes only minimal assumptions about the reader's knowledge of operational semantics. In an attempt to give a feel for how programming in Redex works, this section is peppered with code fragments; each of these expressions runs exactly as given (assuming that earlier definitions have been evaluated) and the results of evaluation are also as shown (although we are using a printer that uses a notation that matches the input notation for values, instead of the standard Scheme printer).

Our goal with this section is to turn the formal model specified in figure 1 into a running Redex program; in section 3, we will test the model. The language in the figure 1 is expression-based, containing application expressions (to invoke functions), conditional expressions, values (i.e., fully simplified expressions), and variables. Values include functions, the plus operator, and numbers.

The `eval` function gives the meaning of each program (either a number or the special token `proc`), and it is defined via a binary relation $\longrightarrow$ on the syntax of programs. This relation, commonly referred to as a standard reduction, gives the behavior of programs in a machine-like way, showing the ways in which an expression can fruitfully take a step towards a value.

Proceedings of the 2009 Scheme and Functional Programming Workshop
California Polytechnic State University Technical Report CPSLO-CSC-09-03

Language

$$e ::= (e\ e \cdots) \mid (\text{if0}\ e\ e\ e) \mid v \mid x$$
$$v ::= \lambda(x \cdots).e \mid + \mid \mathbb{N}$$
$$E ::= [] \mid (v \cdots\ E\ e \cdots) \mid (\text{if0}\ E\ e\ e)$$

Evaluator

$$\text{eval} : e \rightarrow \mathbb{N} \cup \{\texttt{proc}\}$$
$$\text{eval}(e) = n, \text{ if } e \longrightarrow^* \lceil n \rceil \text{ for some } n \in \mathbb{N}$$
$$\text{eval}(e) = \texttt{proc}, \text{ if } \begin{cases} e \longrightarrow^* \lambda(x \cdots).e', \text{ or} \\ e \longrightarrow^* + \end{cases}$$

Reduction relation

$$\begin{array}{lll} E[(\text{if0}\ \lceil 0 \rceil\ e_1\ e_2)] & \longrightarrow & E[e_1] \\ E[(\text{if0}\ v\ e_1\ e_2)] & \longrightarrow & E[e_2] \qquad v \neq \lceil 0 \rceil \\ E[((\lambda(x \cdots).e)\ v \cdots)] & \longrightarrow & E[e\{x \leftarrow v, \cdots\}] \\ E[(+\ \lceil n \rceil \cdots)] & \longrightarrow & E[\lceil \sum(n \cdots) \rceil] \end{array}$$

Figure 1. Mathematical Model of Core Scheme

The non-terminal E defines evaluation contexts. It gives the order in which expressions are evaluated by providing a rule for decomposing a program into a context—an expression containing a "hole"—and the sub-expression to reduce. The context's hole, written [], may appear either inside an application expression, when all the expressions to the left are already values, or inside the test position of an if0 expression.

The first two reduction rules dictate that an if0 expression can be reduced to either its "then" or its "else" subexpression, based on the value of the test. The third rule says that function applications can be simplified by substitution, and the final rule says that fully simplified addition expressions can be replaced with their sums.

We use various features of Redex (as below) to illuminate the behavior of the model as it is translated to Redex, but just to give a feel for the calculus, here is a sample reduction sequence illustrating how the rules and the evaluation contexts work together.

$$\begin{array}{l} (+\ (\text{if0}\ 0\ 1\ 2)\ (\text{if0}\ 2\ 1\ 0)) \\ \longrightarrow (+\ 1\ (\text{if0}\ 2\ 1\ 0)) \\ \longrightarrow (+\ 1\ 0) \\ \longrightarrow 1 \end{array}$$

Consider the step between the first and second term. Both of the if0 expressions are candidates for reduction, but the evaluation contexts only allow the first to be reduced. Since the rules for if0 expressions are written with $E[]$ outside of the if0 expression, the expression must decompose into some E with the if0 expression in the place where the hole appears. This decomposition is what fails when attempting to reduce the second if0 expression. Specifically, the case for application expressions requires values to the left of the hole, but this is not the case for the second if0 expression.

Like a Scheme program, a Redex program consists of a series of definitions. Redex programmers have all of the ordinary Scheme definition forms (variable, function, structure, etc.) available, as well as a few new definition forms that are specific to operational semantics. For clarity, when we show code fragments, we italicize Redex keywords, to make clear where Redex extends Scheme.

Redex's first definition form is *define-language*. It uses a parenthesized version of BNF notation to define a tree grammar,[1] consisting of non-terminals and their productions. The following defines the same grammar as in figure 1, binding it to the Scheme-level variable L.

```
(define-language L
  (e (e e ...)
     (if0 e e e)
     v
     x)
  (v +
     n
     (λ (x ...) e))
  (E hole
     (v ... E e ...)
     (if0 E e e))
  (n number)
  (x variable-not-otherwise-mentioned))
```

In addition to the non-terminals `e`, `v`, and `E` from the figure, this grammar also provides definitions for numbers `n` and variables `x`. Unlike the traditional notation for BNF grammars, Redex encloses a non-terminal and its productions in a pair of parentheses and does not use vertical bars to separate productions, simply juxtaposing them instead.

Following the mathematical model, the first non-terminal in `L` is `e`, and it has four productions: application expressions, `if0` expressions, values, and variables. The ellipsis is a form of Kleene-star; i.e., it admits repetitions of the pattern preceding it (possibly zero). In this case, this means that application expressions must have at least one sub-expression, corresponding to the function position of the application, but may have arbitrarily many more, corresponding to the function's arguments.

The `v` non-terminal specifies the language's values; it has three productions—one each for the addition operator, numeric literals, and functions. As with application expressions, function parameter lists use an ellipsis, this time indicating that a function can have zero or more parameters.

The `E` non-terminal defines the contexts in which evaluation can occur. The *hole* production gives a place where evaluation can occur, in this case, the top-level of the term. The second production allows evaluation to occur anywhere in an application expression, as long as all of the terms to the left of the have been fully evaluated. In other words, this indicates a left-to-right order of evaluation. The third production dictates that evaluation is allowed only in the test position of an `if0` expression.

The `n` non-terminal generates numbers using the built-in Redex pattern *number*. Redex exploits Scheme's underlying support for numbers, allowing arbitrary Scheme numbers to be embedded in Redex terms.

Finally, the `x` generates all variables except λ, `+`, and `if0`, using *variable-not-otherwise-mentioned*. In general, the pattern *variable-not-otherwise-mentioned* matches all variables except those that are used as literals elsewhere in the grammar.

Once a grammar has been defined, a Redex programmer can use *redex-match* to test whether a term matches a given pattern. It accepts three arguments—a language, a pattern, and an expression—and returns `#f` (Scheme's false), if the pattern does not match, or bindings for the pattern variables, if the term does match. For example, consider the following interaction:

```
> (redex-match L e (term (if0 (+ 1 2) 0)))
#f
```

This expression tests whether `(if0 (+ 1 2) 0)` is an expression according to `L`. It is not, because `if0` must have three subexpressions.

When *redex-match* succeeds, it returns a list of match structures, as in this example.

```
> (redex-match
```

[1] See *Tree Automata Techniques and Applications* [6] for an excellent summary of the properties of tree grammars.

```
  L
  (if0 v e_1 e_2)
  (term (if0 3 0 (λ (x) x))))
(list (make-match
       (list (make-bind 'v 3)
             (make-bind 'e_1 0)
             (make-bind 'e_2 (term (λ (x) x))))))
```

Each element in the list corresponds to a distinct way to match the pattern against the expression. In this case, there is only one way to match it, and so there is only one element in the list. Each match structure gives the bindings for the pattern's variables. In this case, `v` matched `3`, `e_1` matched `0`, and `e_2` matched `(λ (x) x)`. The *`term`* constructor is absent from the `v` and `e_1` matches because numbers are simultaneously Redex terms and ordinary Scheme values (and this will come in handy when we define the reduction relation for this language).

Of course, since Redex patterns can be ambiguous, there might be multiple ways for the pattern to match the expression. This can arise in two ways: an ambiguous grammar, or repeated ellipses. Consider the following use of repeated ellipses.

```
> (redex-match L
               (n_1 ... n_2 n_3 ...)
               (term (1 2 3)))
(list (make-match
       (list (make-bind 'n_1 (list))
             (make-bind 'n_2 1)
             (make-bind 'n_3 (list 2 3))))
      (make-match
       (list (make-bind 'n_1 (list 1))
             (make-bind 'n_2 2)
             (make-bind 'n_3 (list 3))))
      (make-match
       (list (make-bind 'n_1 (list 1 2))
             (make-bind 'n_2 3)
             (make-bind 'n_3 (list)))))
```

The pattern matches any sequence of numbers that has at least a single element, and it matches such sequences as many times as there are elements in the sequence, each time binding `n_2` to a distinct element of the sequence.

Now that we have defined a language, we can define the reduction relation for that language. The *`reduction-relation`* form accepts a language and a series of rules that define the relation case-wise. For example, here is a reduction relation for `L`. In preparation for Redex's automatic test case generation, we have intentionally introduced a few errors into this definition. The explanatory text does not contain any errors;[2] it simply avoids mention of the mistakes.

```
(define eval-step
  (reduction-relation
   L
   (--> (in-hole E (if0 0 e_1 e_2))
        (in-hole E e_1)
        "if0 true")
   (--> (in-hole E (if0 v e_1 e_2))
        (in-hole E e_2)
        "if0 false")
   (--> (in-hole E ((λ (x ...) e) v ...))
        (in-hole E (subst (x v) ... e))
        "beta value")
   (--> (in-hole E (+ n_1 n_2))
        (in-hole E ,(+ (term n_1) (term n_2)))
        "+")))
```

[2] We hope.

Each case begins with the arrow `-->` and includes a pattern, a term template, and a name for the case. The pattern indicates when the rule will fire and the term indicates what it should be replaced with.

Each rule begins with an *`in-hole`* pattern that decomposes a term into an evaluation context `E` and some instruction. For example, consider the first rule. We can use *`redex-match`* to test its pattern against a sample expression.

```
> (redex-match L
               (in-hole E (if0 0 e_1 e_2))
               (term (+ 1 (if0 0 2 3))))
(list (make-match
       (list (make-bind 'E (term (+ 1 hole)))
             (make-bind 'e_1 2)
             (make-bind 'e_2 3))))
```

Since the match succeeded, the rule applies to the term, with the substitutions for the pattern variables shown. Thus, this term will reduce to `(+ 1 2)`, since the rule replaces the `if0` expression with `e_1`, the "then" branch, inside the context `(+ 1 hole)`. Similarly, the second reduction rule replaces an `if0` expression with its "else" branch.

The third rule defines function application in terms of a meta-function `subst` that performs capture-avoiding substitution; its definition is not shown, but standard.

The relation's final rule is for addition. It exploits Redex's embedding in Scheme to use the Scheme-level `+` operator to perform the Redex-level addition. Specifically, the comma operator is an escape to Scheme and its result is replaced into the term at the appropriate point. The *`term`* constructor does the reverse, going from Scheme back to a Redex term. In this case, we use it to pick up the bindings for the pattern variables `n_1` and `n_2`.

This "escape" from the object language that we are modeling in Redex to the meta-language (Scheme) mirrors a subtle detail from the mathematical model in figure 1, specifically the use of the $\lceil \cdot \rceil$ operator. In the model that operator translates a number into its textual representation. Consider its use in the addition rule; it defers the definition of addition to the summation operator, much like we defer the definition to Scheme's `+` operator.

Once a Redex programmer has defined a reduction relation, Redex can build reduction graphs, via `traces`. The `traces` function takes a reduction relation and a term and opens a GUI window showing the reduction graph rooted at the given term. Figure 2 shows such a graph, generated from `eval-step` and an `if0` expression. As the screenshot shows, the `traces` window also lets the user adjust the font size and connects to dot [9] to lay out the graphs. Redex can also detect cycles in the reduction graph, for example when running an infinite loop, as shown in figure 3.

In addition to `traces`, Redex provides a lower-level interface to the reduction semantics via the *`apply-reduction-relation`* function. It accepts a reduction relation and a term and returns a list of the next states, as in the following example.

```
> (apply-reduction-relation eval-step
                            (term (if0 1 2 3)))
(list 3)
```

For the `eval-step` reduction relation, this should always be a singleton list but, in general, multiple rules may apply to the same term, or a single rule may even apply in multiple different ways.

3. Random Testing in Redex

If we intend `eval-step` to model the deterministic evaluation of expressions in our toy language, we might expect `eval-step` to define exactly one reduction for any expression that is not already a value. This is certainly the case for the expressions in figures 2 and 3.

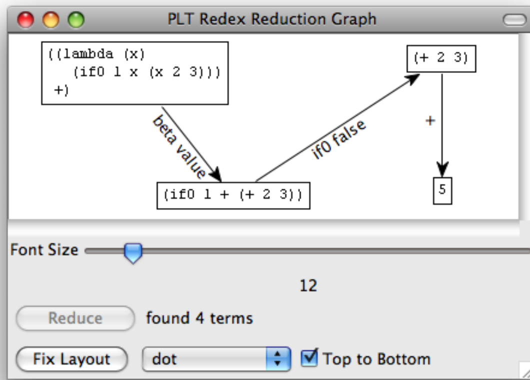


Figure 2. A reduction graph with four expressions

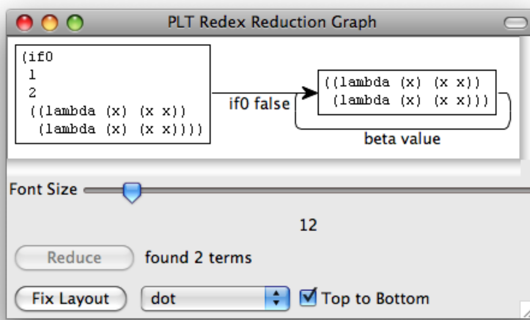


Figure 3. A reduction graph with an infinite loop

To test this, we first formulate a Scheme function that checks this property on one example. It accepts a term and returns true when the term is a value, or when the term reduces just one way, using *redex-match* and *apply-reduction-relation*.

```
;; value-or-unique-step? : term → boolean
(define (value-or-unique-step? e)
  (or (redex-match L v e)
      (= 1 (length (apply-reduction-relation
                     eval-step e)))))
```

Once we have a predicate that should hold for every term, we can supply it to *redex-check*, Redex's random test case generation tool. It accepts a language, in this case L, a pattern to generate terms from, in this case just e, and a boolean expression, in this case, an invocation of the value-or-unique-step? function with the randomly generated term.

```
> (redex-check
   L e
   (value-or-unique-step? (term e)))
counterexample found after 1 attempt:
q
```

Immediately, we see that the property does not hold for open terms. Of course, this means that the property does not even hold for our mathematical model! Often, such terms are referred to as "stuck" states and are ruled out by either a type-checker (in a typed language) or are left implicit by the designer of the model. In this case, however, since we want to uncover all of the mistakes in the model, we instead choose to add explicit error transitions, following how most Scheme implementations actually behave. These rules generally reduces to something of the form (error *description*). For unbound variables, this is the rule:

```
(--> (in-hole E x)
     (error "unbound-id"))
```

It says that when the next term to reduce is a variable (i.e., the term in the hole of the evaluation context is x), then instead reduce to an error. Note that on the right-hand side of the rule, the evaluation context E is omitted. This means that the entire context of the term is simply erased and (error "unbound-id") becomes the complete state of the computation, thus aborting the computation.

With the improved relation in hand, we can try again to uncover bugs in the definition.

```
> (redex-check
   L e
   (value-or-unique-step? (term e)))
counterexample found after 6 attempts:
(+)
```

This result represents a true bug. While the language's grammar allows addition expressions to have an arbitrary number of arguments, our reduction rule only covers the case of two arguments. Redex reports this failure via the simplest expression possible: an application of the plus operator to no arguments at all.

There are several ways to fix this rule. We could add a few rules that would reduce n-ary addition expressions to binary ones and then add special cases for unary and zero-ary addition expressions. Alternatively, we can exploit the fact that Redex is embedded in Scheme to make a rule that is very close in spirit to the rule given in figure 1.

```
(--> (in-hole E (+ n ...))
     (in-hole E ,(apply + (term (n ...))))
     "+")
```

But there still may be errors to discover, and so with this fix in place, we return to *redex-check*.

```
> (redex-check L
               e
               (value-or-unique-step? (term e)))
checking ((λ (i) 0)) raises an exception
syntax: incompatible ellipsis match counts
 for template in: ...
```

This time, *redex-check* is not reporting a failure of the predicate but instead that the input example ((λ (i) 0)) causes the model to raise a Scheme-level runtime error. The precise text of this error is a bit inscrutable, but it also comes with source location highlighting that pinpoints the relation's application case. Translated into English, the error message says that the this rule is ill-defined in the case when the number of formal and actual parameters do not match. The ellipsis in the error message indicates that it is the ellipsis operator on the right-hand side of the rule that is signaling the error, since it does not know how to construct a term unless there are the same number of xs and vs.

To fix this rule, we can add subscripts to the ellipses in the application rule

```
(--> (in-hole E ((λ (x ..._1) e) v ..._1))
     (in-hole E (subst (x v) ... e))
     "beta value")
```

Duplicating the subscript on the ellipses indicates to Redex that it must match the corresponding sequences with the same length.

Again with the fix in hand, we return to *redex-check*:

```
> (redex-check L
               e
```

```
                (value-or-unique-step? (term e)))
counterexample found after 196 attempts:
(if0 0 m +)
```

This time, Redex reports that the expression `(if0 0 m +)` fails, but we clearly have a rule for that case, namely the first `if0` rule. To see what is happening, we apply `eval-step` to the term directly, using *`apply-reduction-relation`*, which shows that the term reduces two different ways.

```
> (apply-reduction-relation eval-step
                            (term (if0 0 m +)))
(list (term +)
      (term m))
```

Of course, we should only expect the second result, not the first. A closer look reveals that, unlike the definition in figure 1, the second `eval-step` rule applies regardless of the particular `v` in the conditional. We fix this oversight by adding a *`side-condition`* clause to the earlier definition.

```
(--> (in-hole E (if0 v e_1 e_2))
     (in-hole E e_2)
     (side-condition (not (equal? (term v) 0)))
     "if0 false")
```

Side-conditions are written as ordinary Scheme code, following the keyword *`side-condition`*, as a new clause in the rule's definition. If the side-condition expression evaluates to `#f`, then the rule is considered not to match.

At this point, *`redex-check`* fails to discover any new errors in the semantics. The complete, corrected reduction relation is shown in figure 4.

In general, after this process fails to uncover (additional) counterexamples, the task becomes assessing *`redex-check`*'s success in generating well-distributed test cases. Redex has some introspective facilities, including the ability to count the number of reductions that fire. With this reduction system, we discover that nearly 60% of the time, the random term exercises the free variable rule. To get better coverage, Redex can take into account the structure of the reduction relation. Specifically, providing the `#:source` keyword tells Redex to use the left-hand sides of the rules in `eval-step` as sources of expressions.

```
> (redex-check L
               e
               (value-or-unique-step? (term e))
               #:source eval-step)
```

With this invocation, Redex distributes its effort across the relation's rules by first generating terms matching the first rule's left-hand side, then terms matching the second term's left-hand side, etc. Note that this also gives Redex a bit more information; namely that all of the left-hand sides of the `eval-step` relation should match the non-terminal `e`, and thus Redex also reports such violations. In this case, however, Redex discovers no new errors, but it does get an even distribution of the uses of the various rewriting rules.

```
(define complete-eval-step
  (reduction-relation
   L

   ;; corrected rules
   (--> (in-hole E (if0 0 e_1 e_2))
        (in-hole E e_1)
        "if0 true")
   (--> (in-hole E (if0 v e_1 e_2))
        (in-hole E e_2)
        (side-condition (not (equal? (term v) 0)))
        "if0 false")
   (--> (in-hole E ((λ (x ..._1) e) v ..._1))
        (in-hole E (subst (x v) ... e))
        "beta value")
   (--> (in-hole E (+ n ...))
        (in-hole E ,(apply + (term (n ...))))
        "+")

   ;; error rules
   (--> (in-hole E x)
        (error "unbound-id"))
   (--> (in-hole E ((λ (x ...) e) v ...))
        (error "arity")
        (side-condition
         (not (= (length (term (x ...)))
                 (length (term (v ...)))))))
   (--> (in-hole E (+ n ... v_1 v_2 ...))
        (error "+")
        (side-condition (not (number? (term v_1)))))
   (--> (in-hole E (v_1 v_2 ...))
        (error "app")
        (side-condition
         (and (not (redex-match L + (term v_1)))
              (not (redex-match L
                                (λ (x ...) e)
                                (term v_1)))))))))
```

Figure 4. The complete, corrected reduction relation

4. Case Study: R⁶RS Formal Semantics

The most recent revision of the specification for the Scheme programming language (R⁶RS) [21] includes a formal, operational semantics defined in PLT Redex. The semantics was vetted by the editors of the R⁶RS and was available for review by the Scheme community at large for several months before it was finalized.

In an attempt to avoid errors in the semantics, it came with a hand-crafted test suite of 333 test expressions. Together these tests explore 6,930 distinct program states; the largest test case explores 307 states. The semantics is non-deterministic in order to avoid over-constraining implementations. That is, an implementation conforms to the semantics if it produces any one of the possible results given by the semantics. Accordingly the test suite contains terms that explore multiple reduction sequence paths. There are 58 test cases that contain at least some non-determinism and, the test case with the most non-determinism visits 17 states that each have multiple subsequent states.

Despite all of the careful scrutiny, Redex's randomized testing found four errors in the semantics, described below. The remainder of this section introduces the semantics itself (section 4.1), describes our experience applying Redex's randomized testing framework to the semantics (sections 4.2 and 4.3), discusses the current state of the fixes to the semantics (section 4.4), and quantifies the size of the bug search space (section 4.5).

4.1 The R⁶RS Formal Semantics

In addition to the features modeled in Section 2, the formal semantics includes: mutable variables, mutable and immutable pairs, variable-arity functions, object identity-based equivalence, quoted expressions, multiple return values, exceptions, mutually recursive bindings, first-class continuations, and `dynamic-wind`. The formal semantics's grammar has 41 non-terminals, with a total of 144 productions, and its reduction relation has 105 rules.

The core of the formal semantics is a relation on program states that, in a manner similar to `eval-step` in Section 2, gives the

behavior of a Scheme abstract machine. For example, here are two of the key rules that govern function application.

```
(--> (in-hole P_1 ((λ (x_1 x_2 ..._1) e_1 e_2 ...)
                  v_1 v_2 ..._1))
     (in-hole P_1 ((r6rs-subst-one
                   (x_1 v_1
                    (λ (x_2 ...) e_1 e_2 ...)))
                  v_2 ...))
     "6appN"
     (side-condition
       (not (term (Var-set!d?
                   (x_1
                    (λ (x_2 ...) e_1 e_2 ...)))))))

(--> (in-hole P_1 ((λ () e_1 e_2 ...)))
     (in-hole P_1 (begin e_1 e_2 ...))
     "6app0")
```

These rules apply only to applications that appear in an evaluation context `P_1`. The first rule turns the application of an n-ary function into the application of an $n-1$-ary function by substituting the first actual argument for the first formal parameter, using the metafunction `r6rs-subst-one`. The side-condition ensures that this rule does not apply when the function's body uses the primitive `set!` to mutate the first parameter's binding; instead, another rule (not shown) handles such applications by allocating a fresh location in the store and replacing each occurrence of the parameter with a reference to the fresh location. Once the first rule has substituted all of the actual parameters for the formal parameters, we are left with a nullary function in an empty application, which is covered by the second rule above. This rule removes both the function and the application, leaving behind the body of the function in a `begin` expression.

The R[6]RS does not fully specify many aspects of evaluation. For example, the order of evaluation of function application expressions is left up to the implementation, as long as the arguments are evaluated in a manner that is consistent with some sequential ordering (i.e., evaluating one argument halfway and then switching to another argument is disallowed). To cope with this in the formal semantics, the evaluation contexts for application expressions are not like those in section 2, which force left to right evaluation, nor do they have the form `(e_1 ... E e_2 ...)`, which would allow non-sequential evaluation; instead, the contexts that extend into application expressions take the form `(v_1 ... E v_2 ...)` and thus only allow evaluation when there is exactly one argument expression to evaluate. To allow evaluation in other application contexts, the reduction relation includes the following rule.

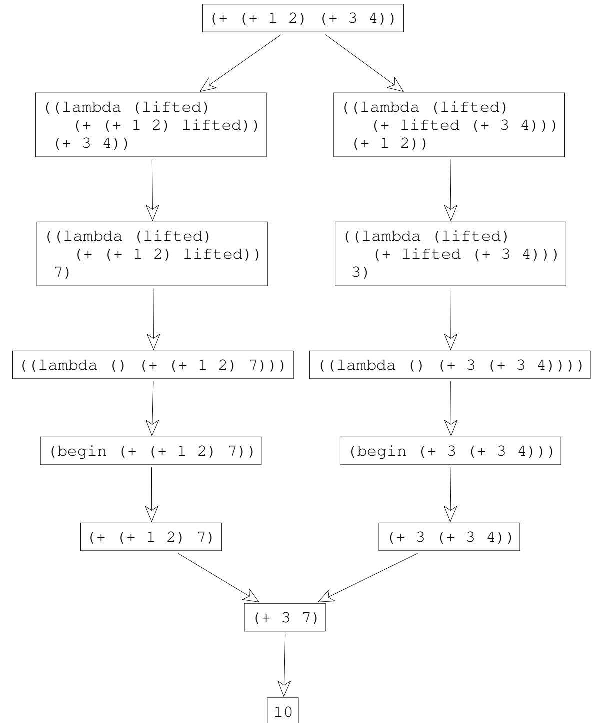
```
(--> (in-hole P_1 (e_1 ... e_i e_i+1 ...))
     (in-hole P_1
       ((λ (x) (e_1 ... x e_i+1 ...)) e_i))
     "6mark"
     (fresh x)
     (side-condition (not (v? (term e_i))))
     (side-condition
      (ormap (λ (e) (not (v? e)))
             (term (e_1 ... e_i+1 ...)))))
```

This rule non-deterministically lifts one subexpression out of the application, placing it in an evaluation context where it will be immediately evaluated then substituted back into the original expression, by the rule `"6appN"`. The *fresh* clause binds `x` such that it does not capture any of the free variables in the original application. The first side-condition ensures that the lifted term is not yet a value, and the second ensures that there is at least one other non-value in the application expression (otherwise the evaluation contexts could just allow evaluation there, without any lifting).

As an example, consider this expression:

```
(+ (+ 1 2)
   (+ 3 4))
```

It contains two nested addition expressions. The `"6mark"` rule applies to both of them, generating two lifted expressions, which then reduce in parallel and eventually merge, as shown in this reduction graph (generated and rendered by Redex).



4.2 Testing the Formal Semantics, a First Attempt

In general, a reduction relation like $\rightarrow$ satisfies the following two properties, commonly known as progress and preservation:

progress If p is a closed program state, consisting of a store and a program expression, then either p is either a final result (i.e., a value or an uncaught exception) or p reduces (i.e., there exists a p' such that $p \rightarrow p'$).

preservation If p is a closed program state and $p \rightarrow p'$, then p' is also a closed program state.

Together these properties ensure that the semantics covers all of the cases and thus an implementation that matches the semantics always produces a result (for every terminating program).

4.2.1 Progress

These properties can be formulated directly as predicates on terms. Progress is a simple boolean combination of a `result?` predicate (defined via a *redex-match* that determines if a term is a final result), an `open?` predicate, and a test to make sure that *apply-reduction-relation* finds at least one possible step. The `open?` predicate uses a `free-vars` function (not shown, but 29 lines of Redex code) that computes the free variables of an R[6]RS expression.

```
;; progress? : program → boolean
(define (progress? p)
  (or (open? p)
      (result? p)
      (not (= 0 (length
                 (apply-reduction-relation
```

```
                reductions
                p))))))

;; open? : program → boolean
(define (open? p)
  (not (= 0 (length (free-vars p)))))
```

Given that predicate, we can use *redex-check* to test it on the R6RS semantics, using the top-level non-terminal (p∗).

```
(redex-check r6rs p∗ (progress? (term p∗)))
```

Bug one This test reveals one bug, a problem in the interaction between `letrec∗` and `set!`. Here is a small example that illustrates the bug.

```
(store ()
      (letrec∗ ([y 1]
                [x (set! y 1)])
        y))
```

All R6RS terms begin with a store. In general, the store binds variable to values representing the current mutable state in a program. In this example, however, the store is empty, and so `()` follows the keyword `store`.

After the store is an expression. In this case, it is a `letrec∗` expression that binds `y` to `1` then binds `x` to the result of the assignment expression `(set! y 1)`. The informal report does not specify the value produced by an assignment expression, and the formal semantics models this under-specification by rewriting these expressions to an explicit `unspecified` term, intended to represent any Scheme value. The bug in the formal semantics is that it neglects to provide a rule that covers the case where an `unspecified` value is used as the initial value of a `letrec∗` binding.

Although the above expression triggers the bug, it does so only after taking several reduction steps. The `progress?` property, however, checks only for a first reduction step, and so Redex can only report a program state like the following, which uses some internal constructs in the R6RS semantics.

```
(store ((lx-x bh))
  (l! lx-x unspecified))
```

Here (and in the presentation of subsequent bugs) the actual program state that Redex identifies is typically somewhat larger than the example we show. Manual simplification to simpler states is straightforward, albeit tedious.

4.2.2 Preservation

The `preservation?` property is a bit more complex. It holds if the expression has free variables or if each each expression it reduces to is both well-formed according to the grammar of the R6RS programs and has no free variables.

```
;; preservation? : program → boolean
(define (preservation? p)
  (or (open? p)
      (andmap (λ (q)
                (and (well-formed? q)
                     (not (open? q))))
              (apply-reduction-relation
               reductions p)))))

(redex-check r6rs p∗ (preservation? (term p∗)))
```

Running this test fails to discover any bugs, even after tens of thousands of random tests. Manual inspection of just a few random program states reveals why: with high probability, a random program state has a free variable and therefore satisfies the property vacuously.

4.3 Testing the Formal Semantics, Take 2

A closer look at the semantics reveals that we can usually perform at least one evaluation step on an open term, since a free variable is only a problem when the reduction system immediately requires its value. This observation suggests testing the following property, which subsumes both progress and preservation: for any program state, either

- it is a final result (either a value or an uncaught exception),
- it does not reduce and it is open, or
- it does reduce, all of the terms it reduces to have the same (or fewer) free variables, and the terms it reduces to are also well-formed R6RS expressions.

The Scheme translation mirrors the English text, using the helper functions `result?` and `well-formed?`, both defined using *redex-match* and the corresponding non-terminal in the R6RS grammar, and `subset?`, a simple Scheme function that compares two lists to see if the elements of the first list are all in the second.

```
(define (safety? p)
  (define fvs (free-vars p))
  (define nexts (apply-reduction-relation
                 reductions p))
  (or (result? p)
      (and (= 0 (length nexts))
           (open? p))
      (and (not (= 0 (length nexts)))
           (andmap (λ (p2)
                     (and (well-formed? p2)
                          (subset? (free-vars p2)
                                   fvs)))
                   nexts))))

(redex-check r6rs p∗ (safety? (term p∗)))
```

The remainder of this subsection details our use of the `safety?` predicate to uncover three additional bugs in the semantics, all failures of the preservation property.

Bug two The second bug is an omission in the formal grammar that leads to a bad interaction with substitution. Specifically, the keyword `make-cond` was allowed to be a variable. This, by itself, would not lead directly to a violation of our safety property, but it causes an error in combination with a special property of `make-cond`—namely that `make-cond` is the only construct in the model that uses strings. It is used to construct values that represent error conditions. Its argument is a string describing the error condition.

Here is an example term that illustrates the bug.

```
(store () ((λ (make-cond) (make-cond ""))
           null)))
```

According to the grammar of R6RS, this is a legal expression because the `make-cond` in the parameter list of the λ expression is treated as a variable, but the `make-cond` in the body of the λ expression is treated as the keyword, and thus the string is in an illegal position. After a single step, however, we are left with this term `(store () (null ""))` and now the string no longer follows `make-cond`, which is illegal.

The fix is simply to disallow `make-cond` as a variable, making the original expression illegal.

Bug three The next bug triggers a Scheme-level error when using the substitution metafunction. When a substitution encounters a λ expression with a repeated parameter, it fails. For example, supplying this expression

```
(store () ((λ (x) (λ (x x) x))
```

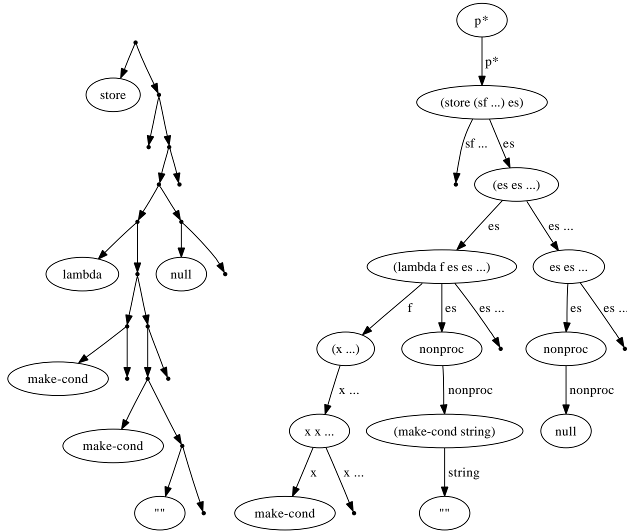


Figure 5. Smallest example of bug two, as a binary tree (left) and as an R^6RS expression (right)

```
            1))
```

to the `safety?` predicate results in this error:

```
r6rs-subst-one: clause 3 matched
 (r6rs-subst-one (x 1 (lambda (x x) x)))
2 different ways
```

The error indicates that the metafunction `r6rs-subst-one`, one of the substitution helper functions from the semantics, is not well-defined for this input.

According to the grammar given in the informal portion of the R^6RS, this program state is not well-formed, since the names bound by the inner λ expression are not distinct. Thus, the fix is not to the metafunction, but to the grammar of the language, restricting the parameter lists of λ expressions to variables that are all distinct.

One could also find this bug by testing the metafunction `r6rs-subst-one` directly. Specifically, testing that the metafunction is well-defined on its input domain also reveals this bug.

Bug four The final bug actually is an error in the definition of the substitution function. The expression

```
(store () ((λ (x) (letrec ([x 1]) 1))
            1))
```

reduces to this (bogus) expression:

```
(store () ((λ () (letrec ((3 1)) 2))))
```

That is, the substitution function replaced the `x` in the binding position of the `letrec` as if the `letrec`-binder was actually a reference to the variable. Ultimately the problem is that `r6rs-subst-one` lacked the cases that handle substitution into `letrec` and `letrec*` expressions.

Redex did not discover this bug until we supplied the `#:source` keyword, which prompted it to generate many expressions matching the left-hand side of the `"6appN"` rule described in section 4.1, on page 31.

4.4 Status of fixes

The version of the R^6RS semantics used in this exploration does not match the official version at `http://www.r6rs.org`, due to version skew of Redex. Specifically, the semantics was written for an older version of Redex and *redex-check* was not present in

Bug #	Uniform, S-expression grammar	R^6RS one var, no dups	R^6RS one var, with dups	R^6RS keywords as vars
1	$D_1(6) > 2^{2^8}$	$p^*(3) > 2^{11}$		
2	$D_0(9) > 2^{2^{11}}$			$p_k^*(6) \approx 2^{556}$
3	$D_1(11) > 2^{2^{13}}$		$p_d^*(8) > 2^{2,969}$ $mf(5) > 2^{501}$	
4	$D_1(12) > 2^{2^{14}}$	$p^*(5) > 2^{110}$		

Figure 6. Exhaustive search space sizes for the four bugs

that version. Thus, in order to test the model, we first ported it to the latest version of Redex. We have verified that all four of the bugs are present in the original model, and we used *redex-check* to be sure that every concrete term in the ported model is also in the original model (the reverse is not true; see the discussion of bug three).

Finally, the R^6RS is going to appear as book published by Cambridge Press [20] and the fixes listed here will be included.

4.5 Search space sizes

Although all four of the bugs in section 4.3 can be discovered with fairly small examples, the search space corresponding to the bug can still be fairly large. In this section we attempt to quantify the size of that search space.

The simplest way to measure the search space is to consider the terms as if they were drawn from an uniform, s-expression representation, i.e., each term is either a pair of terms or a symbol, using repeated pairs to form lists. As an example, consider the left-hand side of figure 5. It shows the parse tree for the smallest expression that discovers bug two, where the dots with children are the pair nodes and the dots without children are the list terminators.

The D_x function computes the number of such trees at a given depth (or smaller), where there are x variables in the expression.

$$D_x(0) = 61 + 1 + x$$
$$D_x(n) = 61 + 1 + x + D_x(n-1)^2$$

The 61 in the definition is the number of keywords in the R^6RS grammar, which just count as leaf nodes for this function; the 1 accounts for the list terminator. For example, the parse tree for bug two has depth 9, and there are more than $2^{2^{11}}$ other trees with that depth (or smaller).

Of course, using that grammar can lead to a much larger state space than necessary, since it contains nonsense expressions like `((λ) (λ) (λ))`. To do a more accurate count, we should determine the depth of each of these terms when viewed by the actual R^6RS grammar. The right-hand side of figure 5 shows the parse tree for bug two, but where the internal nodes represent expansions of the non-terminals from the R^6RS semantics's grammar. In this case, each arrow is labeled with the non-terminal being expanded, the contents of the nodes show what the non-terminal was expanded into, and the dot nodes correspond to expansions of ellipses that terminate the sequence being expanded.

We have computed the size of the search space needed for each of the bugs, as shown in figure 6. The first column shows the size of the search space under the uniform grammar. The second column shows the search space for the first and fourth bugs, using a variant of the R^6RS grammar that contains only a single variable and does not allow duplicate variables, i.e., it assumes that bug three has already been fixed, which makes the search space smaller. Still, the search space is fairly large and the function governing its size is complex, just like the R^6RS grammar itself. The function is shown in figure 7, along with the helper functions it uses. Each

function computes the size of the search space for one of the non-terminals in the grammar. Because `p*` is the top-level non-terminal, the function p^* computes the total size.

Of course it does not make sense to use that grammar to measure the search space for bug three, since it required duplicate variables. Accordingly we used a slightly different grammar to account for it, as shown in the third column in figure 6. The size function we used, p^*_d, has a subscript d to indicate that it allows duplicate variables and otherwise has a similar structure to the one given in figure 7.

Bug three is also possible to discover by testing the metafunction directly, as discussed in section 4.3. In that case, the search space is given by the mf function which computes the size of the patterns used for `r6rs-subst-one`'s domain. Under that metric, the height of the smallest example that exposes the bug is 5. This corresponds to testing a different property, but would still find the bug, in a much smaller search space.

Finally, our approximation to the search space size for bug two is shown in the rightmost column. The k subscript indicates that variables are drawn from the entire set of keywords. Counting this space precisely is more complex than the other functions, because of the restriction that variables appearing in a parameter list must be distinct. Indeed, our p^*_k function over-counts the number of terms in that search space for that reason.[3]

5. Effective Random Term Generation

At a high level, Redex's procedure for generating a random term matching a given pattern is simple: for each non-terminal in the pattern, choose one of its productions and proceed recursively on that pattern. Of course, picking naively has a number of obvious shortcomings. This sections describes how we made the randomized test generation effective in practice.

5.1 Choosing Productions

As sketched above, this procedure has a serious limitation: with non-negligible probability, it produces enormous terms for many inductively defined non-terminals. For example, consider the following language of binary trees:

```
(define-language binary-trees
  (t nil
     (t t)))
```

Each failure to choose the production `nil` expands the problem to the production of two binary trees. If productions are chosen uniformly at random, this procedure will easily construct a tree that exhausts available memory. Accordingly, we impose a size bound on the trees as we generate them. Each time Redex chooses a production that requires further expansion of non-terminals, it decrements the bound. When the bound reaches zero, Redex's restricts its choice to those productions that generate minimum height expressions.

For example, consider generating a term from the `e` non-terminal in the grammar `L` from section 2, on page 27. If the bound is non-zero, Redex freely chooses from all of the productions. Once it reaches zero, Redex no longer chooses the first two productions because those require further expansion of the `e` non-terminal; instead it chooses between the `v` and `x` productions. It is easy to see why `x` is okay; it only generates variables. The `v` non-terminal is also okay, however, because it contains the atomic production `+`.

In general, Redex classifies each production of each non-terminal with a number indicating the minimum number of non-terminal expansion required to generate an expression from the production. Then, when the bound reaches zero, it chooses from one of the productions that have the smallest such number.

$$
\begin{array}{rrl}
p^*(0) = 1 & p^*(n+1) = & (\mathit{es}(n) * \mathit{sfs}(n)) + v(n) + 1 \\
\hat{\mathit{es}}(0) = 1 & \hat{\mathit{es}}(n+1) = & (\hat{\mathit{es}}(n) * \mathit{es}(n)) + 1 \\
\hat{\lambda}(0) = 1 & \hat{\lambda}(n+1) = & (\hat{\lambda}(n) * \lambda(n)) + 1 \\
\mathit{Qs}(0) = 1 & \mathit{Qs}(n+1) = & (\mathit{Qs}(n) * s(n)) + 1 \\
\hat{e}(0) = 1 & \hat{e}(n+1) = & (\hat{e}(n) * e(n)) + 1 \\
\hat{v}(0) = 1 & \hat{v}(n+1) = & (\hat{v}(n) * v(n)) + 1 \\
\mathcal{E}(0) = 1 & \mathcal{E}(n+1) = & (\mathcal{E}(n) * \mathcal{E}^*(n)) \\
 & & + (\mathcal{E}(n) * \mathcal{F}o(n)) + 1 \\
\mathcal{E}^*(0) = 0 & \mathcal{E}^*(n+1) = & \hat{\lambda}(n) + (e(n)^2 * \chi(n)) + \mathcal{F}^*(n) \\
\mathcal{F}^*(0) = 0 & \mathcal{F}^*(n+1) = & \hat{e}(n) + (\hat{e}(n) * \hat{v}(n)) \\
 & & + (\hat{e}(n) * v(n)) + (\hat{e}(n) * e(n) * 2) \\
\mathcal{F}o(0) = 0 & \mathcal{F}o(n+1) = & (\chi(n) * 2) + \hat{v}(n)^2 + e(n)^2 \\
b(0) = 1 & b(n+1) = & v(n) + 1 \\
e(0) = 1 & e(n+1) = & (\hat{\lambda}(n) * e(n)) \\
 & & + (\hat{e}(n) * e(n) * \mathit{lb}(n) * 2) \\
 & & + (\hat{e}(n) * e(n) * 3) + (e(n) * \chi(n) * 2) \\
 & & + (e(n)^3 * \chi(n)) + (\chi(n) * 2) + e(n)^3 \\
 & & + \mathit{non}\lambda(n) + \lambda(n) + 1 \\
\mathit{es}(0) = 2 & \mathit{es}(n+1) = & (\hat{\mathit{es}}(n) * \mathit{es}(n) * f(n)) \\
 & & + (\hat{\lambda}(n) * e(n)) \\
 & & + (\hat{\mathit{es}}(n) * \mathit{es}(n) * \mathit{lbs}(n) * 2) \\
 & & + (\hat{\mathit{es}}(n) * \mathit{es}(n) * 3) \\
 & & + (\mathit{es}(n) * \chi(n) * 2) + (\mathcal{E}(n) * \chi(n)^2) \\
 & & + (e(n)^3 * \chi(n)) + (\chi(n) * 2) + \mathit{es}(n)^3 \\
 & & + \mathit{non}\lambda(n) + p\lambda(n) + \mathit{seq}(n) + \mathit{sqv}(n) \\
 & & + 2 \\
f(0) = 1 & f(n+1) = & (\chi(n) * 2) + 1 \\
\mathit{lb}(0) = 1 & \mathit{lb}(n+1) = & (e(n) * \chi(n)) + 1 \\
\mathit{lbs}(0) = 1 & \mathit{lbs}(n+1) = & (\mathit{es}(n) * \chi(n)) + 1 \\
\mathit{non}\lambda(0) = 2 & \mathit{non}\lambda(n+1) = & \mathit{pp}(n) + \mathit{sqv}(n) + \chi(n) + 2 \\
\mathit{pp}(0) = 0 & \mathit{pp}(n+1) = & \chi(n) * 2 \\
p\lambda(0) = 4 & p\lambda(n+1) = & \mathit{proc1}(n) + 15 \\
\lambda(0) = 0 & \lambda(n+1) = & (\hat{e}(n) * e(n) * f(n)) \\
 & & + (\mathcal{E}(n) * \chi(n)^2) + p\lambda(n) \\
\mathit{proc1}(0) = 7 & \mathit{proc1}(n+1) = & 9 \\
s(0) = 1 & s(n+1) = & \mathit{seq}(n) + \mathit{sqv}(n) + \chi(n) + 1 \\
\mathit{seq}(0) = 0 & \mathit{seq}(n+1) = & (\mathit{Qs}(n) * s(n) * \mathit{sqv}(n)) \\
 & & + (\mathit{Qs}(n) * s(n) * \chi(n)) \\
 & & + (\mathit{Qs}(n) * s(n)) \\
\mathit{sf}(0) = 0 & \mathit{sf}(n+1) = & (b(n) * \chi(n)) + (v(n)^2 * \mathit{pp}(n)) \\
\mathit{sfs}(0) = 1 & \mathit{sfs}(n+1) = & \mathit{sf}(n) + 1 \\
\mathit{sqv}(0) = 2 & \mathit{sqv}(n+1) = & 3 \\
v(0) = 0 & v(n+1) = & \mathit{non}\lambda(n) + \lambda(n) \\
\chi(0) = 0 & \chi(n+1) = & 1
\end{array}
$$

Figure 7. Size of the search space for R^{6}RS expressions

Although this generation technique does limit the expressions Redex generates to be at most a constant taller than the bound, it also results in a poor distribution of the leaf nodes. Specifically, when Redex hits the size bound for the `e` non-terminal, it will never generate a number, preferring to generate `+` from `v`. Although Redex will generate some expressions that contain numbers, the vast majority of leaf nodes will be either `+` or a variable.

In general, the factoring of the grammar's productions into non-terminals can have a tremendous effect on the distribution of randomly generated terms because the collection of several productions behind a new non-terminal focuses probability on the original non-terminal's other productions. We have not, however, been able to detect a case where Redex's poor distribution of leaf nodes impedes its ability to find bugs, despite several attempts. Nevertheless, such situations probably do exist, and so we are investigating a technique that produces better distributed leaves.

[3] Amusingly, if we had not found bug three, this would have been an accurate count.

5.2 Non-linear patterns

Redex supports patterns that only match when two parts of the term are syntactically identical. For example, this revision of the binary tree grammar only matches perfect binary trees

```
(define-language perfect-binary-trees
  (t nil
     (t_1 t_1)))
```

because the subscripts in the second production insists that the two sub-trees are identical. Additionally, Redex allows subscripts on the ellipses (as we used in section 3 on page 29) indicating that the length of the matches must be the same.

These two features can interact in subtle ways that affect term generation. For example, consider the following pattern:

```
(x_1 ... y ..._2 x_1 ..._2)
```

This matches a sequence of xs, followed by a sequence of ys followed by a second sequence of xs. The _1 subscripts dictate that the xs must be the same (when viewed as a complete sequence—the individual members of each sequence may be distinct) and the _2 subscripts dictate that the number of ys must be the same as the number of xs. Taken together, this means that the length of the first sequence of x's must be the same as the length of the sequence of ys, but an left-to-right generation of the term will not discover this constraint until after it has already finished generating the ys.

Even worse, Redex supports subscripts with exclamation marks which insist same-named subscripts match different terms; e.g. (x_!_1 x_!_1) matches sequences of length two where the elements are different.

To support this in the random test case generator, Redex preprocesses the term to normalize the underscores. In the pattern above, Redex rewrites the pattern to this one

```
(x_1 ..._2 y ..._2 x_1 ..._2)
```

simply changing the first ellipsis to ..._2.

5.3 Generation Heuristics

Typically, random test case generators can produce very large test inputs for bugs that could also have been discovered with small inputs.[4] To help mitigate this problem, the term generator employs several heuristics to gradually increase the size and complexity of the terms it produces (this is why the generator generally found small examples for the bugs in section 3).

- The term-height bound increases with the logarithm of the number of terms generated.
- The generator chooses the lengths of ellipsis-produced sequences and the lengths of variable names using a geometric distribution, increasing the distribution's expected value with the logarithm of the number of attempts.
- The alphabet from which the generator constructs variable names gradually grows from the English alphabet to the ASCII set and then to the entire unicode character set. Eventually the generator explicitly considers choosing the names of the language's terminals as variables, in hopes of catching rules which confuse the two. The R[6]RS semantics makes such a mistake, as discussed in section 4.3 (page 4.3), but discovering it is difficult with this heuristic.
- When generating a number, the generator chooses first from the naturals, then from the integers, the reals, and finally the complex numbers, while also increasing the expected magnitude of the chosen number. The complex numbers tend to be especially interesting because comparison operators such as <= are not defined on complex numbers.
- Eventually, the generator biases its production choices by randomly selecting a preferred production for each non-terminal. Once the generator decides to bias itself towards a particular production, it generates terms with more deeply nested version of that production, in hope of catching a bug with deeply nested occurrences of some construct.

[4] Indeed, for this reason, QuickCheck supports a form of automatic test case simplification that tries to shrink a failing test case.

6. Related Work

Our work was inspired by QuickCheck [5], a tool for doing random test case generation in Haskell. Unlike QuickCheck, however, Redex's test case generation goes to some pains to generate tests automatically, rather than asking the user to specify test case generators. This choice reduces the overhead in using Redex's test case generation, but generators for tests cases with a particular property (e.g., closed expressions) still requires user intervention. QuickCheck also supports automatic test case simplification, a feature not yet provided in Redex. Our work is not the only follow-up to QuickCheck; there are several systems in Haskell [3, 19], Clean [11], and even one for the ACL2 integration with PLT Scheme [14].

There are a number of other tools that test formal semantics. Berghofer and Nipkow [1] have applied random testing to semantics written in Isabelle, with the goal of discovering shallow errors in the language's semantics before embarking on a time-consuming proof attempt. αProlog [2] and Twelf [16] both support Prolog-like search for counterexamples to claims. Most recently, Roberson et al. [17] developed a series of techniques to shrink the search space when searching for counterexamples to type soundness results, with impressive results. Rosu et al. [18] use a rewriting logic semantics for C to test memory safety of individual programs.

There is an ongoing debate in the testing community as to the relative merits of randomized testing and bounded exhaustive testing, with the a priori conclusion that randomized testing requires less work to apply, but that bounded exhaustive testing is otherwise superior. Indeed, while most papers on bounded exhaustive testing include a nominal section on the relative merits of randomized testing (typically showing it to be far inferior), there are also few, more careful, studies that do show the virtues of randomized testing. Visser et al. [23] conducted a case study that concludes (among other things) that randomized testing generally does well, but falls down when testing complex data structures like Fibonacci heaps. Randomized testing in Redex mitigates this somewhat, due to the way programs are written in Redex. Specifically, if such heaps were coded up in Redex, there would be one rule for each different configuration of the heap, enabling Redex to easily generate test cases that would cover all of the interesting configurations. Of course, this does not work in general, due to side-conditions on rules. For example, we were unable to automatically generate many tests for the rule [6applyce][5] in the R[6]RS formal semantics, due to its side-condition. Ciupa et al. [4] conducted another study that finds randomized testing to be reasonably effective, and Groce et al. [10] conducted a study finding that random test case generation is especially effective early in the software's lifecycle.

7. Conclusion and Future Work

Randomized test generation has proven to be a cheap and effective way to improve models of programming languages in Redex. With only a 13-line predicate (plus a 29-line free variables function), we were able to find bugs in one of the biggest, most well-tested (even

[5] The is the third rule in figure 11: http://www.r6rs.org/final/html/r6rs/r6rs-Z-H-15.html#node_sec_A.9

community-reviewed), mechanized models of a programming language in existence.

Still, we realize that there are some models for which these simple techniques are insufficient, so we don't expect this to be the last word on testing such models. We have begun work to extend Redex's testing support to allow the user to have enough control over the generation of random expressions to ensure minimal properties, e.g. the absence of free variables.

Our plan is to continue to explore how to generate programs that have interesting structural properties, especially well-typed programs. Generating well-typed programs that have interesting distributions is particularly challenging. While it is not too difficult to generate well-typed terms, generating interesting sets of well-typed terms is tricky since there is a lot of freedom in the choice of the generation of types for intermediate program variables, and using those variables in interesting ways is non-trivial.

Acknowledgments Thanks to Matthias Felleisen for his comments on an earlier draft of this paper and to Sam Tobin-Hochstadt for feedback on `redex-check`.

References

[1] S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In *Proceedings of the International Conference on Software Engineering and Formal Methods*, pages 230–239, 2004.

[2] J. Cheney and A. Momigliano. Mechanized metatheory model-checking. In *Proceedings of the ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 75–86, 2007.

[3] J. Christiansen and S. Fischer. Easycheck – test data for free. In *Proceedings of the International Symposium on Functional and Logic Programming*, pages 322–336, 2008.

[4] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Experimental assessment of random testing for object-oriented software. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 84–94, 2007.

[5] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, 2000.

[6] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. Release October, 12th 2007.

[7] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[8] R. B. Findler. Redex: Debugging operational semantics. Reference Manual PLT-TR2009-redex-v4.2, PLT Scheme Inc., June 2009. `http://plt-scheme.org/techreports/`.

[9] E. R. Gansner and S. C. North. An open graph visualization system and its applications. *Software Practice and Experience*, 30:1203–1233, 1999.

[10] A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 621–631, 2007.

[11] P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In *Proceedings of the International Workshop on the Implementation of Functional Languages*, pages 84–100, 2003.

[12] J. Matthews, R. B. Findler, M. Flatt, and M. Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *International Conference on Rewriting Techniques and Applications*, pages 301–312, 2004.

[13] M. Norrish and K. Slind. Hol4, 2007. `http://hol.sourceforge.net/`.

[14] R. Page, C. Eastlund, and M. Felleisen. Functional programming and theorem proving for undergraduates: a progress report. In *Proceedings of the International Workshop on Functional and Declarative Programming in Education*, pages 21–30, 2008.

[15] L. C. Paulson and T. Nipkow. Isabelle. `http://isabelle.in.tum.de/`, 2005.

[16] F. Pfenning and C. Schürmann. Twelf user's guide. Technical Report CMU-CS-98-173, Carnegie Mellon University, 1998.

[17] M. Roberson, M. Harries, P. T. Darga, and C. Boyapati. Efficient software model checking of soundness of type systems. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications*, pages 493–504, 2008.

[18] G. Rosu, W. Schulte, and T. F. Serbanuta. Runtime verification of c memory safety. In *Proceedings of the International Workshop on Runtime Verification*, 2009. to appear.

[19] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proceedings of the ACM SIGPLAN Symposium on Haskell*, pages 37–48, 2008.

[20] M. Sperber, editor. *Revised^6 report on the algorithmic language Scheme*. Cambridge University Press, 2009. to appear.

[21] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten (editors). The Revised^6 Report on the Algorithmic Language Scheme. `http://www.r6rs.org/`, 2007.

[22] The Coq Development Team. The Coq proof assistant reference manual, version 8.0. `http://coq.inria.fr/`, 2004–2006.

[23] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for java containers using state matching. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 37–48, 2006.

A pattern matcher for miniKanren
or
How to get into trouble with CPS macros

Andrew W. Keep Michael D. Adams Lindsey Kuper William E. Byrd Daniel P. Friedman
Indiana University, Bloomington, IN 47405
{akeep,adamsmd,lkuper,webyrd,dfried}@cs.indiana.edu

Abstract

CPS macros written using Scheme's **syntax-rules** macro system allow for guaranteed composition of macros and control over the order of macro expansion. We identify a limitation of CPS macros when used to generate bindings from a non-unique list of user-specified identifiers. Implementing a pattern matcher for the miniKanren relational programming language revealed this limitation. Identifiers come from the pattern, and repetition indicates that the same variable binding should be used. Using a CPS macro, binding is delayed until after the comparisons are performed. This may cause free identifiers that are symbolically equal to be conflated, even when they are introduced by different parts of the source program. After expansion, this leaves some identifiers unbound that should be bound. In our first solution, we use **syntax-case** with *bound-identifier=?* to correctly compare the delayed bindings. Our second solution uses eager binding with **syntax-rules**. This requires abandoning the CPS approach when discovering new identifiers.

1. Introduction

Macros written in continuation-passing style (CPS) [4, 6] give the programmer control over the order of macro expansion. We chose the CPS approach for implementing a pattern matcher for miniKanren, a declarative logic programming language implemented in a pure functional subset of Scheme [1, 3]. This approach allows us to generate clean miniKanren code, keeping bindings for logic variables in as narrow a scope as possible without generating additional binding forms. During the expansion process, the pattern matcher maintains a list of user-specified identifiers we have encountered, along with the locations in which bindings should be created for them. We accomplish this by using a macro to compare an identifier with the elements of one or more lists of identifiers. Each clause in the macro contains an associated continuation that is expanded if a match is found. The macro can then determine when a unification is unnecessary, when an identifier is already bound, or when an identifier requires a new binding.

While CPS and conditional expansion seemed, at first, to be an effective technique for implementing the pattern matcher, we discovered that the combination of delayed binding of identifiers and conditional expansion based on these identifiers could cause free variables that are symbolically equal to be conflated, even when they are generated from different positions in the source code. The result of conflating two or more identifiers is that only the first will receive a binding. This leaves the remaining identifiers unbound in the final expression, resulting in unbound variable errors.

This issue with delaying identifier binding while the CPS macros expand suggests that some care must be taken when writing macros in CPS. In particular, CPS macros written using Scheme's **syntax-rules** macro system are limited in their ability to compare two identifiers and conditionally expand based on the result of the comparison. The only comparison available to us under **syntax-rules** is an auxiliary keyword check that is the operational equivalent of **syntax-case**'s *free-identifier=?* predicate. Unfortunately, when we use such a comparison, identifiers that are free and symbolically equal may be incorrectly understood as being lexically the same.

In our implementation, the pattern matcher exposes its functionality to the programmer through the $\boldsymbol{\lambda}^e$ and **match**^{e} forms. We begin by describing the semantics of $\boldsymbol{\lambda}^e$ and **match**^{e} and giving examples of their use in miniKanren programs in section 2. In section 3, we present our original implementation of the pattern matcher, and in section 4 we demonstrate how the issue regarding variable binding can be exposed. We follow up in section 5 by presenting two solutions to the variable-binding issue, the first using **syntax-case** and the second using eager binding with **syntax-rules**.

2. Using $\boldsymbol{\lambda}^e$ and match^e

Our aim in implementing a pattern matcher was to allow automatic variable creation similar to that found in the Prolog family of logic programming languages. In Prolog, the first appearance of a variable in the definition of a logic rule leads to a new logic variable being created in the global environment. The $\boldsymbol{\lambda}^e$ and **match**^{e} macros described below allow the miniKanren programmer to take advantage of the power and concision of Prolog-style pattern matching with automatic variable creation, without changing the semantics of the language.

2.1 Writing the *append* relation with $\boldsymbol{\lambda}^e$

Before describing $\boldsymbol{\lambda}^e$ and **match**^{e} in detail, we motivate our discussion of pattern matching by looking at a common operation in logical and functional programming languages—appending two lists. In Prolog, the definition of *append* is very concise:

Proceedings of the 2009 Scheme and Functional Programming Workshop
California Polytechnic State University Technical Report CPSLO-CSC-09-03

```
append([],Y,Y).
append([A|D],Y2,[A|R]) :- append(D,Y2,R).
```

We first present a version of *append* in miniKanren without using λ^e or **match**^{e}. Without pattern matching, the *append* relation in miniKanren is surprisingly verbose when compared with the Prolog equivalent:

```
(define append
  (λ (x y z)
    (conde
      ((≡ '() x) (≡ y z))
      ((exist (a d r)
         (≡ '(,a . ,d) x)
         (≡ '(,a . ,r) z)
         (append d y r))))))
```

Using λ^e, the miniKanren version can be expressed almost as succinctly as the Prolog equivalent:

```
(define append
  (λe (x y z)
    ((() __ ,y))
    (((,a . ,d) __ (,a . ,r)) (append d y r))))
```

The two match clauses of the λ^e version of *append* correspond to the two rules in the Prolog version. In the first match clause, *x* is unified with () and *z* with *y*. In the second clause, *x* is unified with a pair that has *a* as its *car* and *d* as its *cdr*, and *z* is unified with a pair that has the same *a* as its *car* and a fresh *r* as its *cdr*. The *append* relation is then called recursively to finish the work.

No new variables need be created in the first clause, since the only variable referenced, *y*, is already in the λ^e formals list. In the second clause, λ^e is responsible for creating bindings for *a*, *d*, and *r*. In both clauses, the double underscore __ indicates a position in the match that has a value we do not care about. No unification is needed here, since no matter what value *y* has, it will always succeed and need not extend the variable environment. We also have the option of using ,*y* instead of __ because λ^e recognizes a variable being matched against itself and avoids generating the unnecessary unification.

With the *append* relation defined we can now use miniKanren's **run** interface to test the relation.

```
(run 1 (t) (append '(a b c) '(d e f) t)) ⇒
((a b c d e f))
```

where 1 indicates only one answer is desired and *t* is the logic variable bound to the result. Because *append* is a relation we can also use it to generate the input lists that would give us (a b c d e f).

```
(run 5 (t)
  (exist (x y)
    (append x y '(a b c d e f))
    (≡ '(,x ,y) t))) ⇒
((() (a b c d e f))
 ((a) (b c d e f))
 ((a b) (c d e f))
 ((a b c) (d e f))
 ((a b c d) (e f)))
```

where 5 indicates five answers are desired and *x* and *y* are uninstantiated variables used to represent the first and second lists. *append* then returns the first five possible input list pairs that when appended yield (a b c d e f).

2.2 Syntax and semantics of λ^e

Having seen λ^e in action, we now formally describe its syntax and semantics. The syntax of a λ^e expression is:

```
(λe formals
  (pattern1 goal1 ...)
  (pattern2 goal2 ...)
  ...)
```

where *formals* may be any valid λ formal arguments expression, including those for variable-length argument lists. *formals* is the expression to be matched against in the match clauses that follow. Each match clause begins with a pattern followed by zero or more user-supplied goals. The pattern and user-supplied goals represent a conjunction of goals that must all be met for the clause to succeed. Taken together, the clauses represent a disjunction and expand into the clauses of a miniKanren **cond**^{e} (disjunction) expression [3], hence the name λ^e. The pattern within each clause is then further expanded into a set of variable bindings using miniKanren's **exist** and unification operators as necessary.

If no additional goals are supplied by the programmer, then the unifications generated by the pattern will comprise the body of the generated **cond**^{e} clause. Otherwise, the user-supplied goals will be evaluated in the scope of the variables created by the pattern. The first match clause of *append* requires no user-supplied goal, while the second clause uses a user-supplied goal to provide the recursion. It is important to note that λ^e does not attempt to identify unbound identifiers in user-supplied goals, only those in the pattern. Any variables needed in the user-supplied goals not named in the formals list or pattern will need to be bound with an **exist** explicitly by the user.

The pattern matcher recognizes the following forms:

() The null list.

__ Similar to Scheme's _, the double underscore __ represents a position where an expression is expected, but its value can be ignored.

,*x* A logic variable *x*. If this is the first appearance of *x* in the pattern and it does not appear in the formals list of λ^e, a new logic variable will be created.

'e Preserves the expression *e*. This is provided as an escape for special forms where the exact contents should be preserved. For example, if we wish to match the symbol __ rather than having it be treated as an ignored position, we could use '__ in our pattern. λ^e would then know to override the special meaning of __.

sym Where *sym* is any Scheme symbol, other then those assigned special meaning, such as __. These will be preserved in the unification as Scheme symbols.

(*a* . *d*) Arbitrarily nested pairs and lists are also allowed, where *a* and *d* are stand-ins for the *car* and *cdr* positions of the pair. This also allows us to create arbitrary list structures, as is normally the case with pairs in Scheme.

When processing the pattern for each clause, λ^e breaks the pattern down into parts which correspond to the members of the *formals* list. The list of parts is then processed from left to right, with *formals* as the initial list of known variables. As λ^e encounters fresh variable references in each part, it adds them to the known-variables list. If a part is __, or if it is the variable appearing in the corresponding position in *formals*, no unification is necessary. Otherwise, a unification between the processed pattern and the appropriate *formals* variable will be generated.

2.3 Syntax and semantics of match^e

match^e is similar to λ^e in syntax, and it recognizes the same patterns. Unlike λ^e, however, there is no *formals* list, so the list of known variables starts out effectively empty. Strictly speaking, the known-variables list contains the temporary variable introduced to bind the expression in **match**^{e}, which simplifies the implementation of **match**^{e} by making it possible to use the same helper macros

as λ^e. However, since this temporary variable is introduced by a **let** expression generated by **match**^{e}, hygiene ensures that it will never inadvertently match a variable named in the pattern.

match^e has the following syntax:

```
(matche expr
  (pattern1 goal1 ...)
  (pattern2 goal2 ...)
  ...)
```

where *expr* is any Scheme expression. Similar to other pattern matchers, **match**^{e} **let**-binds *expr* to a temporary variable to ensure it is only computed once. Unlike λ^e, which may generate multiple unifications for each clause, **match**^{e} only generates one unification per clause, since it matches each pattern with the variable bound to *expr* as a whole.

Since **match**^{e} can be used on arbitrary expressions, it provides more flexibility then λ^e in defining the matches. For instance, we may want to define the *append* relation using only one of the formal arguments in the match. Consider the following definition of *append*.

```
(define append
  (λ (x y z)
    (matche x
      (() (≡ y z))
      ((,a . ,d)
       (exist (r)
         (≡ `(,a . ,r) z)
         (append d y r))))))
```

Here we have chosen to match against only the first list in the relation, supplying the unifications necessary for the other formal variables. The first clause matches *x* to () and unifies *y* and *z*. The second clause decomposes the list in *x* into *a* and *d*, then uses **exist** to bind *r* and unifies `(,*a* . ,*r*) with *z*. Finally it recurs on the *append* relation to finish calculating the appended lists. This clause requires an explicit **exist** be used to bind *r* since it is not a formal or pattern variable.

The implementations of λ^e and **match**^{e} were designed for use in R^{5}RS, but can be ported to an R^{6}RS library with relative ease, as long as care is taken to ensure that the __ auxiliary keyword is exported with the library.

3. Implementation

Our primary objective in adding pattern-matching capability to miniKanren is to provide convenience to the programmer, but we would prefer that convenience not come at the expense of efficiency. Indeed, we would like to generate the cleanest correct programs possible, so that we can get good performance from the results of our macros.

Since relational programming languages like miniKanren return all possible results from a relation, we would like goals that will eventually reach failure to do so as quickly as possible. In keeping with this "fail fast" principle, we follow two guidelines. First, we limit the scope of logic variables as much as possible. While introducing new logic variables is not an especially time-consuming process, we would still prefer to avoid creating logic variables we will not be using. Second, we generate as few **exist** forms as possible. Minimizing the number of **exist** forms in the code generated by λ^e and **match**^{e} aids efficiency. **exist** wraps its body in two functions. The first is a monadic transform to thread miniKanren's substitution through the goals in its body. The second generates a thunk to allow miniKanren's interleaving search to work through the goals appropriately. This means that each **exist** may cause multiple closures to be generated, and we would like to keep these to a minimum.

To illustrate the benefit of keeping the scope of logic variables as tight as possible, consider the following example:

```
(exist (x y z) (≡ `(,x . ,y) `(a . b)) (≡ x y) (≡ z `c))
```

Here, we create bindings for *x*, *y*, and *z*, even though *z* will never be used. (≡ *x y*) will fail since (≡ `(,*x* . ,*y*) `(a . b)) binds *x* to a and *y* to b, so *z* is never encountered. However, we can tighten the lexical scope for *z* as follows:

```
(exist (x y) (≡ `(,x . ,y) `(a . b)) (≡ x y) (exist (z) (≡ z `c)))
```

The narrower scope around *z* helps the **exist** clauses to fail more quickly, cutting off miniKanren's search for solutions. This example illustrates the trade-off inherent in our twin goals of keeping each variable's scope as narrow as possible and minimizing the overall number of **exist** clauses. Our policy has been to allow more **exist** clauses to be generated when it will tighten the scope of variables. As we continue to explore various performance optimizations in miniKanren, the pattern matcher could benefit from more detailed investigation to determine if the narrowest-scope-possible policy wins more often then it loses.

3.1 λ^e and match^e

All three of our implementations for the pattern matcher expose their functionality to the programmer via the λ^e and **match**^{e} macros. λ^e and **match**^{e} are implemented as follows:

```
(define-syntax λe
  (syntax-rules ()
    ((_ args c c* ...)
     (λ args (handle-clauses args (c c* ...))))))
```

```
(define-syntax matche
  (syntax-rules ()
    ((_ e c c* ...)
     (let ((t e)) (handle-clauses t (c c* ...))))))
```

The interface to these two macros is shared by all three implementations. In all three cases, λ^e and **match**^{e} use the same set of macros to implement their functionality.

In general, the CPS macro approach [4, 6] seems well-suited for our purposes in implementing a pattern matcher in that parts of the pattern must be reconstructed for use during unification and bindings for variables must be generated outside these unifications. Since the CPS macro approach gives us the ability to control the order of expansion, we decided to take an "inside-out" approach: clauses are processed first, and the **cond**^{e} form is then generated around all processed clauses, rather than first expanding the **cond**^{e} and then expanding clauses within it. This inside-out expansion allows us to process patterns from left to right without needing to worry about nesting later unifications and user-supplied goals into the **exist** clauses as we go. Patterns must be processed from left to right to ensure we are always generating an **exist** binding form for the outermost occurrence of an identifier. The entire pattern of a clause is processed, with each part of the pattern being transformed into a unification; any variables that require bindings to be generated for them are put into a flat list of unifications in the order they occur.

As an example, consider the λ^e version of the *append* relation from the previous section. At expansion time, the pattern in the second clause is processed into the following flat list of unifications (with embedded indicators of where new variables need to be bound):

```
((ex a d) (≡ (cons a d) x) (ex r) (≡ (cons a r) z))
```

Here (**ex** *a d*) and (**ex** *r*) indicate the places where new variables need to be bound with an **exist** clause. The **build-clause** macro, described below, then takes this list, along with user-specified goals (if any) and a continuation, and calls the continuation on the completed clause, which looks like this after expansion:

```
(exist (a d)
  (≡ (cons a d) x)
  (exist (r)
    (≡ (cons a r) z)
    (append d y r)))
```

where The **exist** forms and unifications were generated as a result of matching the pattern with the λ^e formals list, and (*append d y r*) was the user-specified goal. When both clauses of the *append* relation have been processed and wrapped in a single **cond**^{e}, *append* expands to

```
(define append
  (λ (x y z)
    (cond^e
      ((≡ '() x) (≡ y z))
      ((exist (a d)
         (≡ (cons a d) x)
         (exist (r)
           (≡ (cons a r) z)
           (append d y r)))))))
```

In this example, the first clause does not require any **exist** clauses, since it does not introduce any new bindings.

3.2 CPS macro implementation

Aside from the user-interfacing λ^e and **match**^{e}, the CPS macro implementation of the pattern matcher comprises ten macros: two macros for decomposing clauses and patterns; two helper macros for constructing continuation expressions; five macros for building up clauses, unifications, and expressions; and one macro for matching identifiers to determine when bindings have been seen before. As a guide to the reader, the macros used to decompose clauses and patterns have names starting with **handle**; the helper macros for constructing continuations have names starting with **make**; and the macros used to build up discovered parts of clauses, unifications, and expressions have names starting with **build**. Finally, the **case-id** macro is used to match identifiers in much the same way Scheme's **case** is used to match symbols. We have also endeavoured to use consistent naming conventions for the variables used in the **handle**, **make**, and **build** macros, as follows:

***a*, *a*∗** indicate an argument (*a*) or list of arguments (*a*∗).

***p*, *p*∗, *pr*∗** indicate a part (*p*), parts (*p*∗), or the patterns remaining to be processed (*pr*∗) from the initial pattern.

***u*∗, *g*∗, *g*∗∗** indicate user-supplied goals (*u*∗), goals from a clause (*g*∗), or the remaining clauses (*g*∗∗).

***pc*∗, *pp*∗, *pg*∗** indicate a list of processed clauses (*pc*∗), processed pattern parts (*pp*∗), and processed goals (*pg*∗).

***k*∗** indicates the continuation for the macro.

***svar*∗** indicates a list of variables we have already seen in processing the pattern.

***evar*∗** indicates a list of variables that need to be bound with **exist** for the unification currently being worked on.

pa*, *pd indicate the *car* (*pa*) and *cdr* (*pd*) positions of a pattern pair.

3.2.1 The *handle* macros

The **handle-clauses** and **handle-pattern** macros implement the forward phase of pattern processing and are responsible for breaking the λ^e and **match**^{e} clauses and patterns down into parts for the **build** macros to reconstruct. The **handle-clauses** macro is implemented as follows:

```
(define-syntax handle-clauses
  (syntax-rules ()
    ((_ a* () . pc*) (cond^e . pc*))
    ((_ (a . a*) (((p . p*) . g*) (pr* . g**) ...) . pc*)
     (make-clauses-cont
       (a . a*) a a* p p* g* ((pr* . g**) ...) . pc*))
    ((_ a ((p . g*) (pr* . g**) ...) . pc*)
     (make-clauses-cont a a () p () g* ((pr* . g**) ...) . pc*))))
```

handle-clauses transforms the list of λ^e and **match**^{e} clauses into a list of **cond**^{e} clauses. The first rule recognizes when the list of λ^e clauses to be processed is empty and generates a **cond**^{e} to wrap the processed clauses *pc*∗. The second and third rules both serve to decompose the clauses, processing each one in order using the **make-clauses-cont** macro described below. The second rule processes clauses of λ^e expressions where the *formals* start with a pair. The third rule handles **match**^{e} clauses where the expression to be matched is **let**-bound to a temporary and λ^e clauses where the formal is a single identifier rather than a list.

handle-pattern is where the main work of the pattern matcher takes place. It is responsible for deciding when new logic variables need to be introduced and generating the expressions to be unified against in the final output.

```
(define-syntax handle-pattern
  (syntax-rules (quote unquote top __)
    ((_ top a __ (k* ...) svar* evar* pp* ...)
     (k* ... svar* evar* pp* ...))
    ((_ tag a __ (k* ...) svar* evar* pp* ...)
     (k* ... (t . svar*) (t . evar*) pp* ... t))
    ((_ tag a () (k* ...) svar* evar* pp* ...)
     (k* ... svar* evar* pp* ... ()))
    ((_ tag a (quote p) (k* ...) svar* evar* pp* ...)
     (k* ... svar* evar* pp* ... (quote p)))
    ((_ tag a (unquote p) (k* ...) svar* evar* pp* ...)
     (case-id p
       ((a) (k* ... svar* evar* pp* ...))
       (svar* (k* ... svar* evar* pp* ... p))
       (else (k* ... (p . svar*) (p . evar*) pp* ... p))))
    ((_ tag a (pa . pd) k* svar* evar* pp* ...)
     (handle-pattern inner t1 pa
       (handle-pattern inner t2 pd
         (build-cons k*)) svar* evar* pp* ...))
    ((_ tag a p (k* ...) svar* evar* pp* ...)
     (k* ... svar* evar* pp* ... 'p))))
```

The first two rules both match the __ "ignore" pattern. However, the first rule is distinguished by its use of the **top** auxiliary keyword indicating that it is at the top level of the pattern, i.e., it will be matched directly with an input variable, either a λ^e formal or **let**-bound temporary variable for the **match**^{e} expression. In either case, no unification is needed, so we do not extend the list of processed pattern parts *pp*∗. In the second rule, we know that __ must be nested within a pair, so a new logic variable is generated to indicate that an expression is expected here, even though we do not care what the value of the expression is. Since the logic variable is generated as a temporary, it will not clash with any other variable already bound, thanks to hygienic macro expansion.

The remaining rules do not require this special handling around the top element, and so they ignore the "tag" supplied as the first part of the pattern. The third, fourth, and seventh rules handle the null, quoted expression, and bare symbol cases, respectively. In all of these cases, the continuation is invoked with either a null list or a quoted expression. If we are at the top level of the

pattern, the continuation builds a unification directly using **build-goal**; otherwise, it builds a more complex expression using **build-cons**. The CPS nature of the macro, however, frees us from having to concern ourselves with the kind of expression generated.

The sixth rule handles pairs. Here, **handle-pattern** is called on the *car* of the pair, *pa*, with a continuation that processes the *cdr* of the pair, *pd*, which in turn calls the **build-cons** continuation to build the *cons* pair. The continuations are each created with the part of the current state required for them to finish their jobs, relying on the application sites for the continuation to fill in any extra arguments. This is why expressions of the form (*k*∗ … *args* …) are so prevalent in our macros. Note that in both calls to **handle-pattern**, *inner* is specified to ensure that if __ is encountered, it will recognize that it is no longer at the top level of the pattern.

Finally, the fifth rule in **handle-pattern** determines if a unification can be skipped, because it is unifying a variable with itself; if the identifier is already a bound variable; or if a new binding is needed for this variable. **case-id** provides the functionality for this conditional expansion. The first case of the **case-id** expression checks to see if the formal argument *a* matches the pattern variable just discovered, and skips the unification if they do. Note that if this is not a top-level match, then *a* will be a temporary variable generated by the calls to **handle-pattern** in rule six. The second case checks if *p* occurs in the list of encountered variables *svar*∗; if so, it simply extends the list of pattern parts with *p*. If the **else** case is triggered, it means that we need both a new binding for the logic variable and a pattern for the unification. In this case *p* is added to the overall list of encountered variables *svar*∗ as well as the list of variables to be bound for this unification *evar*∗. The list of pattern parts is also extended with *p*. Here *svar*∗ and *evar*∗ are kept distinct, because *svar*∗ records all of the variables we have encountered in processing this clause, while *evar*∗ records only those needed for the current unification, so that the **exist** clause can bind the logic variables close to their first use.

3.2.2 The *make* helper macros

One of our design principles in implementing the pattern matcher was to write several smaller macros, each with one relatively simple task to accomplish, rather than writing a few monolithic ones. This "small pieces" approach relies on the ability to compose macros as continuations to accomplish more complex actions. Therefore, we often find ourselves needing to construct continuations within continuations. The **make-clauses-cont** and **make-pattern-cont** macros help streamline the code by factoring this continuation-building behavior out into its own macros.

```
(define-syntax make-clauses-cont
  (syntax-rules ()
    ((_ args a a* p p* g* ((pr* . g**) ...) . pc*)
     (handle-pattern top a p
       (make-pattern-cont a a* p* ()
         (build-clause g*
           (handle-clauses args ((pr* . g**) ...) . pc*)))
       (a . a*) ()))))
```

The **make-clauses-cont** macro is used by **handle-clauses** when we begin processing a pattern. The continuation uses **handle-pattern** to match the first part of a pattern to the first part of the formals list. In addition to handing **handle-pattern** the list of items to work on, a fairly deeply nested chain of continuations is passed along. The outermost continuation, **make-pattern-cont**, is used to construct the continuations that build up a unification goal from the results of **handle-pattern**. Once the unification goal is built, the **build-clause** continuation is used to build the completed clause, and finally the **handle-clauses** continuation is used to begin working on the remaining clauses. This computation is responsible for driving the recursion for **handle-clauses**, the macro that both initiates the computation and finally generates the **cond**^{e} expression.

```
(define-syntax make-pattern-cont
  (syntax-rules ()
    ((_ a a* p* u* k* svar* evar* . pp*)
     (build-goal a
       (build-var evar*
         (build-clause-part a* p* u* svar* k*)) . pp*))))
```

Similar to **make-clauses-cont**, **make-pattern-cont** builds a list of nested continuations. Both **make-clauses-cont** and **build-clause-part** use **make-pattern-cont** to provide a continuation for turning the pattern built during **handle-pattern** into a proper unification goal. The outermost continuation, **build-goal**, wraps the result in a unification with its matching formal argument, *a*. It is passed a continuation of **build-var** that is responsible for turning the *evar*∗ list into an (**ex** *evar*∗ …) part in the flattened list of unifications. Finally, the innermost continuation, **build-clause-part**, drives the recursion through **handle-pattern** so that the entire pattern will be processed into unifications and **ex** indicators before the completed list is passed off to **build-clause** to build the final clause.

3.2.3 The *build* macros

At various stages in the expansion, the **build** macros serve both to build some part of the final expression from parts processed through the **handle** macros, and to drive the recursion through the **handle** macros to bring the expansion to completion. The **build-goal** macro is responsible for generating unifications, when necessary. **build-var** generates **ex** indicators from the *evar*∗ list of variables if the list is not null. The **build-clause-part** macro drives the recursion around **handle-pattern** to finish processing the pattern into a set of unification goals and **ex** indicators. **handle-pattern** uses the **build-cons** macro to rebuild pairs discovered in the pattern. Finally, the **build-clause** macro combines the flattened list of unifications and **ex** indicators with the user-supplied goals and creates a clause for use in the final **cond**^{e} expression.

```
(define-syntax build-goal
  (syntax-rules ()
    ((_ a (k* ...)) (k* ...))
    ((_ a (k* ...) p) (k* ... (≡ p a)))))
```

The two rules in **build-goal** correspond to whether **handle-pattern** has supplied a pattern to it. If __ occurs at the top level, or a formal matches a variable referenced in the same position in the match pattern, **handle-pattern** does not create a pattern; therefore, **build-goal** simply calls its continuation. Otherwise, **build-goal** calls its continuation with the unification of the provided pattern and the argument.

```
(define-syntax build-var
  (syntax-rules ()
    ((_ () (k* ...) . g*) (k* ... . g*))
    ((_ evar* (k* ...) . g*) (k* ... (ex . evar*) . g*))))
```

Likewise, **handle-pattern** may provide an empty list of discovered variables to **build-var**. Therefore, **build-var** need only create a new **ex** indicator if it receives a non-null list. Otherwise, **build-var** simply calls its continuation on the list of goals *g*∗.

```
(define-syntax build-clause-part
  (syntax-rules ()
    ((_ () () (u* ...) svar* (k* ...) . g*) (k* ... (u* ... . g*)))
    ((_ (a . a*) (p . p*) (u* ...) svar* k* . g*)
     (handle-pattern top a p
       (make-pattern-cont a a* p* (u* ... . g*) k*) svar* ()))
    ((_ a p (u* ...) svar* k* . g*)
     (handle-pattern top a p
       (make-pattern-cont a () () (u* ... . g*) k*) svar* ())))))
```

build-clause-part receives the results of **build-goal** and **build-var**. If there are no more pattern parts to process, **build-clause-part** calls its continuation. Otherwise, **build-clause-part** calls **handle-pattern** on the next matching pattern part and argument from the *formals* list.

```
(define-syntax build-cons
  (syntax-rules ()
    ((_ (k* ...) t* p* ... pa pd)
     (k* ... t* p* ... (cons pa pd)))))
```

handle-pattern uses **build-cons** to rebuild pairs that it previously decomposed. **build-cons** simply calls its continuation, adding a *cons* expression in the final pattern to be sent to the miniKanren unifier.

```
(define-syntax build-clause
  (syntax-rules (ex)
    ((_ () (k* ...) ()) (k* ... (succeed)))
    ((_ (pg* ...) (k* ...) ()) (k* ... (pg* ...)))
    ((_ (pg* ...) k* (g* ... (ex . v*)))
     (build-clause ((exist v* pg* ...)) k* (g* ...)))
    ((_ (pg* ...) k* (g* ... g))
     (build-clause (g pg* ...) k* (g* ...)))))
```

Finally, the **build-clause** macro constructs a clause of one or more goals for use in the final **cond**^{e} expression. It processes the flattened list of unifications and **ex** indicators, along with the user-supplied goals, into a finished clause. The first rule in **build-clause** handles the case where no goals are supplied and the pattern produced no unifications. In that case, **build-clause** simply generates a miniKanren *succeed* expression, a goal that always succeeds. The second rule terminates by calling its continuation once all of the goals have been processed. The third rule recognizes an **ex** indicator and generates a new **exist** expression wrapping all of the already processed goals into a new goal. The **exist** expressions must be created in a list, since **build-clause** expects a list of clauses rather than just a single clause. Finally, in the fourth rule, any remaining goals should be unifications or user-supplied goals requiring no further processing and so are simply added to the list of processed goals for the clause.

3.2.4 The *case-id* macro

While the macros described thus far are written for the specific purpose of implementing the pattern matcher, the **case-id** macro is a more general-purpose helper macro. **case-id** determines which list of identifiers contains a match for a supplied identifier. Its syntax is like that of Scheme's standard **case** form, except that an **else** clause is required. The idea is similar to *syn-eq* [4] in that the identifier to be matched is treated as an auxiliary keyword in a generated **let-syntax**-bound macro. However, rather than taking a single list of identifiers to search for a match along with success and failure continuation macros, **case-id** takes a list of clauses, where each clause has a list of identifiers, and a result continuation macro, and requires an **else** clause for when none of the other cases succeed.

```
(define-syntax case-id
  (syntax-rules (else)
    ((_ x ((x** ...) act*) ... (else e-act))
     (letrec-syntax
       ((helper (syntax-rules (x else)
                  ((_ (else a)) a)
                  ((_ (() a) c . c*) (helper c . c*))
                  ((_ ((x . z*) a) c . c*) a)
                  ((_ ((y z* (... ...)) a) c . c*)
                   (helper ((z* (... ...)) a) c . c*)))))
       (helper ((x** ...) act*) ... (else e-act))))))
```

The macro generated by **case-id** determines when identifiers match, which allows us to avoid generating a unification, or when an identifier appears in a list of known identifiers, which indicates that no binding needs to be created for it. The real trick here is that the generated macro exploits the auxiliary keyword support of **syntax-rules** to match an element from the list of identifiers with the identifier to be matched. The auxiliary keyword is the identifier in question.

The **letrec-syntax**-bound macro **helper** searches for a case matching the original identifier *x*, expanding the **else** clause if no match is found. The first rule matches **else** and expands the associated continuation macro. The second rule identifies when the end of a list of identifiers has been encountered, and begins processing the next clause. The third rule matches the originally passed identifier and terminates by expanding the associated continuation macro. Finally, the fourth rule strips off the first identifier from the list, which failed to match in the previous clause, and recurs on the remainder of the list.

4. The variable-binding problem

We have presented a CPS macro implementation of our pattern matcher that delays creation of binding forms, allowing us to generate concise code. Unfortunately, implementing the pattern matcher with CPS macros led us to discover a subtle issue with how **case-id** determines when a variable needs to be created.

4.1 Identifier equality and binding

We can demonstrate the problem by writing a macro that expands into a $\boldsymbol{\lambda}^e$ or **match**^{e} expression. Consider the following macro, which expands into a $\boldsymbol{\lambda}^e$:

```
(define-syntax break-λe
  (syntax-rules ()
    ((_ v) (λe (x y) (((,w . ,v) ,v))))))
```

Here **break-**$\boldsymbol{\lambda}^e$ expects a user-supplied identifier for use in the generated $\boldsymbol{\lambda}^e$ expression. This simple, though admittedly contrived, example demonstrates how a CPS macro implementation of $\boldsymbol{\lambda}^e$ and **match**^{e} that uses **case-id** will not create bindings for identifiers that are free when they are symbolically equal.

To further illustrate the problem, consider some example uses of **break-**$\boldsymbol{\lambda}^e$. First, if we supply *z* as an argument to **break-**$\boldsymbol{\lambda}^e$, it expands as follows:

```
(break-λe z) ⇒
(λe (x y) (((,w . ,z) ,z))) ⇒
(λ (x y)
  (conde
    ((exist (z w)
       (≡ (cons w z) x)
       (≡ z y)))))
```

Here, $\boldsymbol{\lambda}^e$ behaves as expected; it sees both *z* and *w* as new variables that must be bound by the generated **exist** expression.

Instead, a user of **break-λ^e** may decide to use x, which coincidentally happens to be one of the variables bound by the λ^e generated by **break-λ^e**. In the expansion below, x_1 and x_2 are both symbolically x, but represent the x supplied to **break-λ^e** and the generated formal parameter x in the λ^e expression, respectively.

```
(break-λ^e x1) ⇒
(λ^e (x2 y) (((,w . ,x1) ,x1))) ⇒
(λ (x2 y)
 (cond^e
   ((exist (w x1)
     (≡ (cons w x1) x2)
     (≡ x1 y)))))
```

Here, too, identifiers are understood as unique, as we expected, and λ^e creates a binding for x_1.

Finally, the programmer may choose w as the variable to supply to **break-λ^e**. In the example below, w_1 represents the w introduced by the programmer, and w_2 the one introduced by the **break-λ^e** macro. This time, the expansion does not seem to work out so well:

```
(break-λ^e w1) ⇒
(λ^e (x y) (((,w2 . ,w1) ,w1))) ⇒
(λ (x y)
 (cond^e
   ((exist (w2)
     (≡ (cons w2 w1) x)
     (≡ w1 y)))))
```

We would have liked bindings to be created for both w_1 and w_2, but since both are free and symbolically equal when **case-id** compares them, they are incorrectly understood as being equal. Although no variable capture occurred and hence hygiene is preserved, the λ^e macro does not work properly in this case, because it leaves unbound a pattern variable that should have been bound.

The issue arises as the confluence of two events. First, as we process the pattern, we delay the creation of bindings until the whole pattern has been processed, leaving free variables free. Second, **case-id** lifts the variable we are testing into an auxiliary keyword in the helper macro to compare it with the list of identifiers. The comparison between x and the identifiers from each list will succeed when both have the same binding or when both are free and they are symbolically equal [5, 7].

In the first example, both w and z are free, but are not symbolically equal. In the second example, x_1 and x_2 are symbolically equal, but one is bound while the other is free, so the comparison fails, as we would expect. It is only in the final case, where both w identifiers are free and symbolically equal, that the problem exhibits itself.

4.2 With great power comes great responsibility

As we have seen, CPS macros provide a powerful mechanism for controlling the order of macro expansion. However, the variable-binding problem limits our ability to use CPS macros to generate bindings selectively based on a running list of identifiers. In order to avoid unintentionally conflating variables, we must bind identifiers as soon as we encounter them, rather than delaying binding until the invocation of a final continuation.

This limitation suggests that CPS macro writers must take particular care to avoid the accidental conflation of free, symbolically-equal identifiers that are introduced from different places in the source. Hygienic macro expansion does not help us here, since the problem is not inappropriate variable capture; rather, it is that variables that should be bound are left unbound. Avoiding accidental conflation of pattern variables therefore becomes the programmer's responsibility.

5. Workarounds

In this section, we present two solutions to the variable-binding issue demonstrated in the previous section. Our first solution uses the **syntax-case** macro system and the *bound-identifier=?* predicate to perform the comparison we actually intend. Second, we present a **syntax-rules**-based solution using eager binding by foregoing certain uses of CPS in favor of a more traditional approach.

5.1 *case-id* with *syntax-case* and *bound-identifier=?*

If we restrict ourselves to CPS macros written using the **syntax-rules** macro system, there is, unfortunately, no easy change we can make that will resolve the variable-binding issue. Fundamentally, **syntax-rules** only provides us with a way to perform what is essentially a *free-identifier=?* check, by generating a macro that has the identifier we wish to match as an auxiliary keyword.

However, the **syntax-case** macro system gives us the ability to compare identifiers according to their *intended use* by employing the *bound-identifier=?* predicate. *bound-identifier=?* takes two identifier arguments and returns #t only if a binding for one identifier would capture the other. Effectively, two identifiers will be *bound-identifier=?* only if they were introduced by the same transformer or within the same macro [7, 2]. In fact, this is the very comparison we would prefer for **case-id**.

We can implement **case-id** straightforwardly with **syntax-case** by using *bound-identifier=?* in a fender, as follows:

```
(define-syntax case-id
  (λ (exp)
    (syntax-case exp (else)
      ((_ x (else e-act)) #'e-act)
      ((_ x ((y x* ...) act) ((x** ...) act*) ... (else e-act))
       (bound-identifier=? #'x #'y)
       #'act)
      ((_ x ((y x* ...) act) ((x** ...) act*) ... (else e-act))
       #'(case-id x
              ((x* ...) act) ((x** ...) act*) ... (else e-act)))
      ((_ x (() act) ((x** ...) act*) ... (else e-act))
       #'(case-id x ((x** ...) act*) ... (else e-act))))))
```

The interface to **case-id** remains the same, and the rest of the pattern matcher implementation need not be changed. In this version of **case-id**, the first clause matches when only the **else** case is left. The second clause extracts an identifier from the list and uses the *bound-identifier=?* check to compare the identifiers. If the comparison succeeds, that case's action is used. The third clause extracts the identifier and throws it away to continue processing the current list, since we have already verified in the previous clause that x and y are not *bound-identifier=?*. The final clause matches when we have exhausted the list of identifiers to be matched for the current case, and so we proceed to the next case from the call to **case-id**.

Using this implementation of **case-id**, when we expand the third **break-λ^e** expression from section 2.1, we get

```
(break-λ^e w) ⇒
(λ^e (x y) (((,w2 . ,w1) ,w1))) ⇒
(λ (x y)
 (cond^e
   ((exist (w2 w1)
     (≡ (cons w2 w1) x)
     (≡ w1 y)))))
```

with both w_1 and w_2 being bound by the surrounding **exist** expression. This workaround has the advantages of producing very clean miniKanren source and allowing us to keep most of our implementation unchanged, but it does force us to use **syntax-case**.

5.2 Using eager binding with *syntax-rules*

While we can fix the variable-binding issue in our pattern matcher by implementing **case-id** with **syntax-case**, we may prefer to stick with a **syntax-rules**-based implementation. **syntax-rules** offers us the simplicity of pattern matching and rewriting without having to worry about the potentially more complex **syntax-case** macro system or the details of how *bound-identifier=?* works. Here we present a **syntax-rules** solution to the variable-binding issue that works by eagerly binding new identifiers as they are encountered.

Unlike the **syntax-case** solution, which resolved the issue by performing a different kind of comparison in **case-id**, the eager binding approach ensures that our list of seen variables never contains free identifiers. Since we never compare two free identifiers, we no longer need to worry that two symbolically equal identifiers will be conflated, and the **syntax-rules** version of **case-id** can remain unchanged.

This approach is not without complications of its own, since $\boldsymbol{\lambda}^e$ and **match**^{e} must expand into **cond**^{e} and **exist**, which impose their own limitations on the expressions in their clauses. The challenge arises because **cond**^{e} expects a set of clauses in which each clause is a list of one or more goals and **exist** expects a list of bindings followed by one or more goals. Since the helpers for $\boldsymbol{\lambda}^e$ and **match**^{e} will expand within the context of **cond**^{e} and **exist**, they must expand into valid goals. Part of the difficulty arises from the fact that **cond**^{e} and **exist** perform a monadic transform, which $\boldsymbol{\lambda}^e$ and **match**^{e} must be careful not to interfere with.

Unfortunately, these restrictions mean that the eager-binding versions of $\boldsymbol{\lambda}^e$ and **match**^{e} cannot generate quite as clean miniKanren code as the original CPS macro implementation. Returning to our *append* example, the fixed version of $\boldsymbol{\lambda}^e$ expands to the slightly more verbose:

```
(λ (x y z)
  (cond^e
    ((exist () (≡ () x) (≡ y z)))
    ((exist (a)
       (exist (d)
         (exist ()
           (≡ (cons a d) x)
           (exist (r)
             (exist ()
               (≡ (cons a r) z)
               (append d y r)))))))))
```

Here, the **exist** expressions that bind no variables each enclose more than one goal. Grouping multiple goals inside an **exist** allows them to appear in a position where only one goal is allowed, in much the same way Scheme's **begin** can group multiple expressions into a single expression.

The **break-**$\boldsymbol{\lambda}^e$ macro now works correctly in all of the previously shown examples. In particular, (**break-**$\boldsymbol{\lambda}^e$ *w*) now expands as follows:

```
(break-λ^e w) ⇒
(λ^e (x y) (((,w2 . ,w1) ,w1))) ⇒
(λ (x y)
  (cond^e
    ((exist (w2)
       (exist (w1)
         (exist ()
           (≡ (cons w2 w1) x)
           (≡ w1 y)))))))
```

We have taken some liberties in this example, since the **exist** macro would need to be in scope in order for it to work properly, but the full expansion of **exist** would needlessly complicate the example.

Even though this version of the pattern matcher uses the original version of **case-id**, it now correctly identifies the two *w* variables as distinct, since the binding for w_2 is created by **exist** before the next section of the pattern is expanded.

In order to implement the eager binding approach, we must alter the part of our pattern matcher that identifies variables to be bound. We accomplish this by refactoring **handle-pattern** into two macros: **do-pattern**, which binds any necessary variables, and a simplified **handle-pattern**, which builds the processed version of the pattern for use in a unification. **handle-pattern** remains a CPS macro and has been simplified in accordance with its reduced mission. As before, the **do-pattern-opt** macro prevents recognizably unnecessary unifications from being generated.

The interface to $\boldsymbol{\lambda}^e$ and **match**^{e} does not change, and **handle-clauses** has been rewritten to support expanding into the **cond**^{e} in place.

```
(define-syntax handle-clauses
  (syntax-rules ()
    ((_ (a* ...) (c c* ...))
     (cond^e ((do-clause (a* ...) (a* ...) c))
             ((do-clause (a* ...) (a* ...) c*)) ...))
    ((_ (a* ... . r) (c c* ...))
     (cond^e ((do-clause (a* ... r) (a* ... . r) c))
             ((do-clause (a* ... r) (a* ... . r) c*)) ...))
    ((_ a (c c* ...))
     (cond^e ((do-clause (a) a c))
             ((do-clause (a) a c*)) ...))))
```

handle-clauses is responsible for generating the **cond**^{e} expression, passing first a list of named variables, then the original argument list, and finally the clause to be processed to **do-clause**.

```
(define-syntax do-clause
  (syntax-rules ()
    ((_ svar* () (() . g*)) (exist-helper () . g*))
    ((_ svar* (a . a*) ((p . p*) . g*))
     (do-pattern-opt svar* a p a* p* . g*))
    ((_ svar* a (p . g*))
     (do-pattern-opt svar* a p () () . g*))))
```

The **do-clause** macro processes each formal from the argument list with the corresponding part of the pattern, relying on **do-pattern-opt** to generate the variable bindings and unifications for each clause. Finally, it expands into the list of user-supplied goals.

```
(define-syntax do-pattern-opt
  (syntax-rules (unquote __)
    ((_ svar* a (unquote p) a* p* . g*)
     (case-id p
       ((a) (do-clause svar* a* (p* . g*)))
       (else (do-pattern svar* a ,p () ,p a* p* . g*))))
    ((_ svar* a __ a* p* . g*) (do-clause svar* a* (p* . g*)))
    ((_ svar* a p a* p* . g*)
     (do-pattern svar* a p () p a* p* . g*))))
```

While we could generate unifications for each part of the pattern, we would prefer to recognize unnecessary unifications and not generate them, as in the original implementation. **do-pattern-opt** ensures that unifications are not generated when a __ is encountered at the top level or when a logic variable is being matched with itself. In all other cases, **do-pattern-opt** calls **do-pattern**, which generates the necessary **exist** bindings or unifications.

```
(define-syntax do-pattern
  (syntax-rules (quote unquote)
    ((_ svar* a () () op () ())
     (do-clause svar* ()
       (() (handle-pattern op (handle-pattern-cont a) ()))))
    ((_ svar* a () () op a* p* . g*)
     (exist ()
       (handle-pattern op (handle-pattern-cont a) ())
       (do-clause svar* a* (p* . g*))))
    ((_ svar* a (unquote p) r op a* p* . g*)
     (case-id p
       (svar* (do-pattern svar* a r () op a* p* . g*))
       (else (exist (p)
               (do-pattern (p . svar*) a r () op a* p* . g*)))))
    ((_ svar* a (quote p) r op a* p* . g*)
     (do-pattern svar* a r () op a* p* . g*))
    ((_ svar* a (pa . pd) () op a* p* . g*)
     (do-pattern svar* a pa pd op a* p* . g*))
    ((_ svar* a (pa . pd) r op a* p* . g*)
     (do-pattern svar* a pa (pd . r) op a* p* . g*))
    ((_ svar* a p r op a* p* . g*)
     (do-pattern svar* a r () op a* p* . g*))))
```

In **do-pattern**, the **unquote** rule uses **case-id** to determine if the logic variable *p* has been encountered. If not, a binding is generated and *p* is added to the list of known variables. The other rules are responsible for traversing the full pattern. Some optimization is also performed by **do-pattern**: it avoids generating unnecessary *succeed* goals by recognizing when it has reached the end of the pattern and there are no user-supplied goals, in which case it treats the final pattern unification as if it were a user-supplied goal.

Once bindings for all new variables have been created, the original pattern is passed off to **handle-pattern**, and the rest of the pattern and formal parameters are passed back to the **do-clause** macro to continue processing.

```
(define-syntax handle-pattern
  (syntax-rules (quote unquote __)
    ((_ () (k* ...) t* p* ...) (k* ... t* p* ... '()))
    ((_ __ (k* ...) t* p* ...) (k* ... (t . t*) p* ... t))
    ((_ (unquote p) (k* ...) t* p* ...) (k* ... t* p* ... p))
    ((_ (quote p) (k* ...) t* p* ...) (k* ... t* p* ... 'p))
    ((_ (pa . pd) k t* p* ...)
     (handle-pattern pa
       (handle-pattern pd (build-cons k)) t* p* ...))
    ((_ p (k* ...) t* p* ...) (k* ... t* p* ... 'p))))
```

The revised **handle-pattern** macro no longer needs to know about the argument being processed, nor does it need to know whether it is called at the top level of a pattern or within a pattern, so the first two arguments have been removed, simplifying the macro quite a bit. However, **handle-pattern** still needs a continuation, since it proceeds recursively through the pattern.

handle-pattern also adds new bindings for the temporary variables needed by the __ matches. This is safe because we will always need these temporary variables and because we no longer use **case-id** to guide our decisions about which variables need to be bound. The sixth rule of **handle-pattern**, previously the seventh rule, still uses **build-cons** in order to reconstruct a matched pair.

In addition to the updated **handle-pattern**, we need a new continuation for it to call, **handle-pattern-cont**.

```
(define-syntax handle-pattern-cont
  (syntax-rules ()
    ((_ v t* p) (exist-helper t* (≡ p v)))))
```

handle-pattern-cont simply generates a unification, wrapping it with an **exist** for any temporaries that need to be bound. Rather than a standard **exist** expression, we use the following **exist-helper** in order to avoid generating unnecessary **exist** expressions when possible:

```
(define-syntax exist-helper
  (syntax-rules ()
    ((_ ()) succeed)
    ((_ () g) g)
    ((_ t* g* ...) (exist t* g* ...))))
```

exist-helper generates the *succeed* goal when supplied an empty bindings list and no goals, or the provided goal when supplied an empty bindings list and a single goal. Otherwise, it generates a normal **exist** expression.

6. Conclusion

CPS macros provide a powerful mechanism for controlling the order of macro expansion, but care must be taken when using this technique with conditional expansion. In particular, we must use caution when using **syntax-rules** with the auxiliary keyword trick to perform variable comparisons, or we may end up treating two free identifiers that are symbolically equal as the same, even if they will not be equal when they are bound. However, we can work around this limitation either by using **syntax-case** for performing the comparisons with *bound-identifier=?*, or by using eager binding to ensure that no two free variables will ever be compared. We hope that these techniques will prove useful for macro implementors who find themselves faced with a similar issue in using CPS macros. An interesting area of further investigation in this regard would be to look at ways to bring the ability to perform *bound-identifier=?* comparisons to **syntax-rules**. Already some implementations of **syntax-rules**, such as the one included with Chez Scheme [2], provide a fender syntax similar to that of **syntax-case** which allows the use of such techniques, although this has not yet found its way into the standard.

Acknowledgments

The authors wish to express their thanks to the anonymous reviewers, whose thoughtful comments and suggestions have improved this paper. We thank Dorai Sitaram for SLaTeX, which we use to typeset our programs.

References

[1] W. E. Byrd and D. P. Friedman. From variadic functions to variadic relations. In *Proceedings of the 2006 Scheme and Functional Programming Workshop, University of Chicago Technical Report TR-2006-06, 2006*, pages 105–117, 2006.

[2] R. K. Dybvig. *Chez Scheme Version 7 User's Guide*. Cadence Research Systems, 2005.

[3] D. P. Friedman, W. E. Byrd, and O. Kiselyov. *The Reasoned Schemer*. The MIT Press, 2005.

[4] E. Hilsdale and D. P. Friedman. Writing macros in continuation-passing style. In *Scheme and Functional Programming 2000*, page 53, 2000.

[5] R. Kelsey, W. Clinger, and J. Rees (eds.). Revised[5] report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, Sept. 1998.

[6] O. Kiselyov. Macros that compose: Systematic macro programming. In *GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, pages 202–217, London, UK, 2002. Springer-Verlag.

[7] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten (eds.). Revised[6] report on the algorithmic language Scheme, September 2007.

Higher-Order Aspects in Order

Éric Tanter *
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Santiago, Chile
etanter@dcc.uchile.cl

Abstract

In aspect-oriented programming languages, advice evaluation is usually considered as part of the base program evaluation. This is also the case for certain pointcuts, such as `if` pointcuts in AspectJ, or simply all pointcuts in higher-order aspect languages like AspectScheme. While viewing pointcuts and advice as base level computation clearly distinguishes AOP from reflection, it also comes at a price: because aspects observe base level computation, evaluating pointcuts and advice at the base level can trigger infinite regression. To avoid these pitfalls, aspect languages propose (sometimes insufficient) ad-hoc mechanisms, which make aspect-oriented programming more complex. This paper proposes to clarify the situation by introducing explicit levels of execution in the programming language, thereby allowing aspects to observe and run at specific, possibly different, levels. We adopt a defensive default that avoids infinite regression, and give programmers the means to override this default through explicit level shifting expressions. We implement our proposal as an extension of AspectScheme, and formalize its semantics. This work recognizes that different aspects differ in their intended nature, and shows that structuring execution contexts helps tame the power of aspects and metaprogramming.

1. Introduction

In the pointcut-advice model of aspect-oriented programming [Masuhara et al. 2003, Wand et al. 2004], as embodied in *e.g.* AspectJ [] and AspectScheme [Dutchyn et al. 2006], crosscutting behavior is defined by means of pointcuts and advices. A pointcut is a predicate that matches program execution points, called join points, and an advice is the action to be taken at a join point matched by a pointcut. An aspect is a module that encompasses a number of pointcuts and advices.

A major challenge in aspect language design is to cleanly and concisely express where and when aspects should apply. To this end, expressive pointcut languages have been devised. While originally pointcuts were conceived as purely "meta" predicates that cannot have any interaction with base level code [Wand et al. 2004], the needs of practitioners have led aspect languages to include more expressive pointcut mechanisms. This is the case of the `if` pointcut in AspectJ, which takes an arbitrary Java expression and matches at a given join point only if the expression evaluates to true. Going a step further, higher-order aspect languages like AspectScheme consider a pointcut as a first-class, higher-order function like any other, thus giving the full computational power of the base language to express pointcuts.

While pointcuts were initially conceived of as pure metalevel predicates, advices were seen as a piece of base-level functionality [Wand et al. 2004]. In other words, an advice is just like an ordinary function or method, that happens to be triggered "implicitly" whenever the associated pointcut predicate matches. Considering advice as base-level code clearly distinguishes AOP from runtime metaobject protocols (to many, the ancestors of AOP). Indeed, a metaobject runs, by definition, at the metalevel [Maes 1987]. This makes it possible to consider metaobject activity as fundamentally different from base level computation, and this can be used to get rid of infinite regression [Denker et al. 2008]. In AOP, infinite regression can also happen, and does happen, easily[1]: it is sufficient for a piece of advice to trigger a join point that is potentially matched by itself (either directly or indirectly). This is one of the reasons why a specific kind of join point, which denotes advice execution, has been introduced in AspectJ [Wand et al. 2004].

In recent work, we analyze this issue further and show that AspectJ fails to properly recognize the possibility of infinite regression due to pointcut evaluation [Tanter 2008a]. We proposed a solution that consists in introducing a pointcut execution join point, and a defensive default that avoids aspects matching against their own execution. In a language like AspectScheme, controlling regression in both pointcuts and advice is done using a special primitive (**app/prim**), which makes it possible to apply a function without generating an application join point. This solution however does not scale to join points that are produced in the dynamic extent of the evaluation of pointcuts and advices.

Since all these issues are reminiscent of conflation of levels in reflective architectures [Chiba et al. 1996], we choose to question the basic assumption that pointcut and advice are *intrinsically* either base or meta. For instance, looking at how programmers use advices, it turns out that some advices are clearly base code, while some are not: *e.g.* generic aspects, advices that use `thisJoinPoint` (reification of the current join point to be used in the advice), etc. To get rid of this tension between AOP and MOPs, or between "all is base" and "all is meta", we propose a reconciliating approach in which *execution levels* are managed explicitly (if needed) in a program. By doing so, we allow different (parts of) pointcuts and advice to be run at different levels, and aspects are bound to observe execution of particular levels. This gives programmers complete control over what aspects see and where they run (*i.e.* who sees them). To alleviate the task for non-expert programmers, we also choose a defensive default that avoids regression. In addition, since we decouple pointcut and advice from execution levels, it becomes possible to use execution level shifting

* Partially funded by FONDECYT projects 11060493 & 1090083.

[1] http://www.eclipse.org/aspectj/doc/released/progguide/pitfalls-infiniteLoops.html

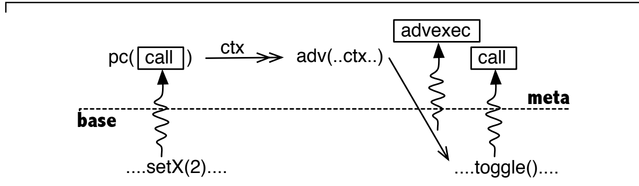


Figure 1. Join points and aspect execution in aspect languages with "meta" pointcuts and base-level advice.

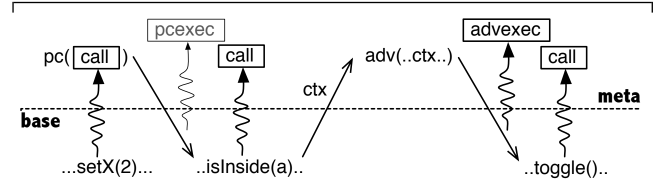


Figure 2. Join points and aspect execution in aspect languages with `if` pointcuts or higher-order pointcuts, and base-level advice.

to explicitly move certain parts of a program execution to a higher level, thereby hiding them from lower-level aspects.

This paper is structured as follows: Section 2 describes the current state of affairs regarding aspect weaving, illustrating the issue of conflation. Section 3 introduces execution levels along with a safe default for aspects, which allows aspects of aspects, but prevents an aspect from stepping on its own tail. Section 3.2 gives control back to the programmer by introducing explicit level shifting expressions, which can be used to move any expression evaluation from a level to another. We formalize the operational semantics of our proposal in Section 4, by modeling a higher-order aspect language with explicit execution levels. Section 5 discusses related work and Section 6 concludes. We include the complete formalization of our proposal in Appendix A.

2. Background and Motivation

First of all, let us consider a simple example that illustrates some of the issues at stake (in AspectJ). We define an `Activity` aspect that highlights whenever a `Point` object is active, that is, when one of its methods is executing. Furthermore, we are interested in highlighting only points that are inside a pre-determined `area`. This aspect could be defined in AspectJ as follows:

```
public aspect Activity {
  Area area = ...;
  Object around(Point p) :
    execution(* Point.*(..)) && this(p)
    && if(p.isInside(area)){
      p.toggle(); // start highlight
      Object r = proceed(p);
      p.toggle(); // stop highlight
      return r;
  }
}
```

This defines exactly the aspect we are interested in. Note that the `this(p)` pointcut is used to expose the currently executing object `p` to both the `if` pointcut and the advice. The advice toggles highlighting, then `proceeds`, *i.e.* executes the original computation on `p`, gets the result, stops highlighting, and then returns the result.

2.1 Issues

The definition of the `Activity` aspect, though natural, is however flawed due to three kinds of reentrancy [Tanter 2008a].

Base-triggered reentrancy. First, if a method of a `Point`, say `move`, internally calls other methods of `Point`, like `setX`, the advice is going to apply several times, yielding repeated toggling of highlighting resulting in incorrect behavior. This is called based-triggered reentrancy. This can be avoided by excluding all join points that are in the control flow of a matched join point. In AspectJ this is done by adding a `!cflowbelow` condition in the pointcut definition, which rules out all join points that occur in the dynamic extent of a matched join point.

Advice-triggered reentrancy. Second, the aspect is subject to advice-triggered reentrancy: highlighting a point object is done by calling its `toggle` method, whose execution is going to be matched by the same aspect, and so on infinitely. To solve advice-triggered reentrancy, similarly to the solution to base-triggered reentrancy, the idea is to exclude join points that occur in the control flow an advice execution. For that, AspectJ includes a specific kind of join point kind called "advice execution". A solution therefore consists in extending the above pointcut as follows:

```
execution(* Point.*(..)) && ...
  && !cflow(adviceexecution());
```

Figure 1 depicts the situation. When a call occurs at the base level, a call join point is created (snaky arrow). The join point (call box) is passed to the pointcut, which returns either false (if there is no match), or a list of bindings (`ctx`) if there is a match. The evaluation of the pointcut occurs entirely at the meta-level [Wand et al. 2004] (without considering `if` pointcuts). The bindings are used to expose context information to the advice. The advice is then called, and runs at the base level. This means another call occurring in the dynamic extent of the advice execution is reified as a call join point, just as visible as the first one.

Pointcut-triggered reentrancy. The definition of the `Activity` aspect also fails because of pointcut-triggered reentrancy: calling `isInside` on the point object within the `if` pointcut results in another method execution that is potentially matched by the aspect. This is because `if` pointcuts are evaluated at the base level in AspectJ. In a higher-order aspect language like AspectScheme, this problem is exacerbated by the fact that there is no such thing as a special `if` pointcut, rather, a pointcut is just a function that runs –as any other– at the base level. The situation with higher-order aspects (and first-order aspects with `if` pointcuts) is depicted on Figure 2: execution of (part of) the pointcut is performed at the base level, therefore the join points produced while executing the pointcut are visible exactly like any other. Consider the following code in AspectScheme.

```
(deploy (let ((area ...))
          (lambda (jp)
            (let ((x (jp-arg 0 jp)))
              (and (Point? x)
                   (is-inside x area)))))
        trace)
```

This (simplified) code defines a pointcut that matches all function applications where the first argument is a value matching the `Point?` predicate, if ever the point resides within a given area. Since the pointcut function runs at the base level, the applications of both `Point?` and `is-inside` generate a call join point that is matched against the same pointcut function, infinitely. In

AspectScheme, one would have to use the special `app/prim` form to apply functions in a way that does not generate join points.

2.2 Controlling Reentrancy

The analysis of the three kinds of reentrancy was formulated in a previous article [Tanter 2008a]. We show that reentrancy can be avoided using well-known patterns (*i.e.* `cflow` checks). However, adding these checks to pointcut definitions makes them much more complex than they should be. Also, current AspectJ compilers (ajc and abc [Avgustinov et al. 2006]) make it impossible to get rid of pointcut-triggered reentrancy without completely refactoring the aspect definition, because they hide join points that occur *lexically* in `if` pointcuts.

We therefore proposed a revised semantics for `if` pointcuts, such that their execution is fully visible to all aspects. In addition, we make it clear that, similarly to the advice case, it is necessary to have a pointcut execution join point in order to discriminate the join points that are produced by pointcut evaluation, and therefore getting rid of pointcut-based reentrancy.

Finally, we adopt a new default semantics according to which an aspect never sees a reentrant join point. This implies that the above definition of `Activity` is correct *as is* with our modified semantics. We introduce means to control reentrancy at a more fine-grained level, as required.

While reentrancy control solves the issues of self-references, it does not solve the more general problem of mutual visibility among aspects. For instance, let us consider a second aspect, `Mirroring`, in charge of maintaining mirrors of certain point objects:

```
aspect Mirroring {
  void around(Point p) :
    execution(* Point.set*(..)) && this(p) {
      proceed(p);
      proceed(lookupMirror(p));
    }
}
```

Whenever a state changing method (denoted with the syntactic method pattern `set*(..)`) executes, the aspect not only proceeds with the original object, but also proceeds with the mirror, in order to keep it in sync with the original point. The fact that AspectJ supports this powerful mechanism is reminiscent of reflective method invocation in Java, and in general, metaprogramming[2]. Indeed, `Mirroring` is an aspect that is similar in many respects to what used to be implemented with a typical metaobject protocol [Zimmermann 1996].

When an object is changed, both `Mirroring` and `Activity` aspects match. Using aspect *precedence* declaration, we can make sure that `Activity` runs first, so as to highlight the point object before doing anything else. Recall that the composition of around advices is a nested chaining, such that when `Activity` calls `proceed`, `Mirroring` advice is executed, and when `Mirroring` in turn calls `proceed`, since it is the last advice in the chain, the original computation is performed.

The precedence declaration does not however make it possible to solve another looping issue that appears: when `Mirroring` advice invokes reflectively the method on the mirror point, `Activity` is going to match. If we have reentrancy control as defined in [Tanter 2008a], execution does not enter an infinite loop, but still, the resulting highlighting behavior is incorrect, because `toggle` is applied twice on the mirror point.

[2] http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg03353.html

2.3 Conflation

As a matter of fact, all this situation is reminiscent of the issue of meta-circularity, which has long been identified in reflective architectures [des Rivières and Smith 1984]. Broadly from the perspective of reflection, the problem is that of meta-circularity: we are trying to use all the power of higher-order functions to redefine, via pointcuts and advice, the meaning of some function applications. Or, in the case of AspectJ, we are using all the power of Java to implement pointcuts and advice. The proven, but ad hoc, solution to this problem is to add base checks that stop regression, such as `!cflow(adviceexec(..))` in AspectJ, or the default reentrancy control summarized above. Another solution is to introduce a more primitive mechanism that is *not* subject to redefinition, like AspectScheme's **app/prim**.

However, as clearly identified by Chiba *et al.*, these approaches eventually fall short, for they fail to address the fundamental problem, which is that of *conflating* levels that ought to be kept separate [Chiba et al. 1996]. As it turns out, an in-depth inspection of the use of control-flow based checks to avoid reentrancy in AspectJ shows that this mechanism is brittle and does not always work. While we postpone the detailed description of these issues to a revised and extended version of this paper, it is easy to see that `cflow` checks do not help if the advice triggers some behavior in a separate thread. Also, confusion arises due to the fact that, with around advice, the execution of the base code (through `proceed`) is also in the control flow of the advice execution. On Figures 1 and 2, conflation is represented by the fact that all join points (boxes) are present at the same "level", *i.e.* they are all similarly visible to all the defined aspects.

This therefore suggests to place pointcut and advice execution at a higher-level of execution ($n + 1$) than "base" code (n). On the one hand, this allows for a stable semantics, where issues of conflation can be avoided [Chiba et al. 1996, Denker et al. 2008]. On the other hand, this boils down to reconsidering AOP as just a form of metaprogramming, a somewhat unpopular view in the AO community. Only Bodden *et al.* have looked at this issue in AOP and proposed a solution based on placing aspects at different levels of execution, recognizing advice execution as a meta activity [Bodden et al. 2006]. However, seeing advice as *inherently* meta defeats the original idea of AOP, where an advice is just another (probably misnamed) piece of code that has the same ontological status as a method [Kiczales 2009].

Recognizing that AOP *can* be (and is) used also for metaprogramming, we propose to resolve this conflict by decoupling the "metaness" concern from the pointcut and advice mechanism. We introduce explicit level shifting in the language, so that programmers can specify their intent with respect to the ontological status of their pointcuts and advices. Also, aspects can be explicitly deployed at higher levels, in order to observe higher-level computation. This said, we opt for a *default* semantics regarding pointcuts and advices that favors stability. That is, by default, we consider both pointcut and advice execution as higher-level computation. This arguable choice is purely motivated by a defensive concern: the unaware programmer should not face infinite regression unless she consciously chooses to.

3. Execution Levels

In this section, we introduce execution levels and discuss how they can be used in conjunction with aspects. Section 3.1 exposes the default way in which pointcuts and advices are evaluated. Section 3.2 gives more control to programmers by exposing level shifting expressions. Section 3.3 briefly discusses an interesting perspective raised by the introduction of execution levels.

Figure 3. Running pointcut and advice at a higher level of execution.

Figure 4. Proceeding to the original computation is done at the lower level.

3.1 Aspects and Levels: Default

Figure 3 depicts the default evaluation of pointcuts and advice with level shifting. As before, we adopt the convention that the evaluation of base code (at level 0) generates join points at level 1 (*e.g.* the call box), where aspects can potentially match and trigger advice. Pointcut and advice execution join points are generated, but at level 2. Similarly the whole evaluation of pointcuts and advices is done at level 1, so the join points produced in the dynamic extent of these evaluations are generated at level 2. This ensures that the call of `isInside` done during pointcut evaluation of `Activity` is not seen at the same level as the call to `setX` (level 0). The same holds for the call to `toggle` in the advice.

Proceed. As briefly explained in Section 2.2, an advice can *proceed* to the computation originally described by the join point. When several aspects match the same join point, like `Activity` and `Mirroring` on `setX`, the corresponding advices are chained such that calling proceed in advice k triggers advice $k + 1$. Only when the last advice proceeds is the original computation performed. The original computation clearly belongs to the same level as the original call. This means that in our default semantics, the last call to proceed in a chain of advices runs the original computation at the lower level. Subsequently, join points generated by the evaluation of the original computation (level 0 in that case) are seen at the same level as before (level 1). This is shown on Figure 4.

Aspects of aspects. The default semantics of computing pointcut and advice at a higher-level ensures that other aspects do not see these computations. In our example with `Activity` and `Mirroring`, this is fine for the case of `Mirroring`: the execution of its advice, which performs a reflective invocation on the mirror object, is done at level 1, so the method execution of the mirror is not visible to `Activity`.

Figure 5. Shifting up. Figure 6. Shifting down.

However, this layering also implies that `Mirroring` cannot see the advice computation of `Activity`; therefore the mirror object is not highlighted when it ought to. In order to allow aspects to observe the activity of other aspects, while keeping the same default semantics, it is necessary to define aspects at higher levels. For instance, in [Bodden et al. 2006], this is done by declaring certain aspects as $meta[n]$ where n is the level at which the aspect stands. The following section introduces a more uniform and flexible solution to this issue.

3.2 Controlling Execution Levels

While installing aspects at higher levels is correct, it stays within the perspective of "aspects are meta". From a software engineering viewpoint, it also implies that at the time `Mirroring` is deployed, it is known that this aspect may be required at higher levels.

As we already mentioned before, AOP is not solely metaprogramming with syntactic sugar: the original idea is that advice is a piece of base-level code. In some cases, advice execution should be visible to aspects that observe base level execution. This is the case for instance of the `Activity` aspect: its advice, which highlights points, should be executed at the base level. This allows `Mirroring` to coherently update the mirror objects. This second alternative is more compatible with the traditional AO view that "advices are base". From an engineering viewpoint, it allows the implementor of the `Activity` aspect to declare that some part of its pointcut and/or advice should be considered as standard base code. `Mirroring` does not need to be aware that some other aspect performs computation of interest to it.

Up and down. In order to reconcile both approaches, we introduce explicit level shifting expressions in the language, such that a programmer can decide at which level an expression is evaluated. Level shifting is orthogonal to the pointcut/advice mechanism, and can be used to move any computation.

Figure 5 shows that shifting up an expression moves the computation of that expression a level above the current level. This implies that join points generated during the evaluation of that expression are visible one level above. Conversely, shifting an expression down moves the computation of that expression a level below the current level, as depicted on Figure 6[3].

Aspects in order. Using **up** and **down**, it is possible to control where aspects are run. One can either use these level shifting expression directly within the definitions of pointcut and advice functions, or use wrappers that embed a whole function in a level shifting expression.

For instance, suppose that the computation of `is-inside` in the pointcut of `Activity` is expensive, and that a memoization aspect is used. It is possible to move the computation of `is-inside` down

[3] For completeness, we consider that execution starts at level 0 and that evaluating a **down** expression at level 0 has no effect.

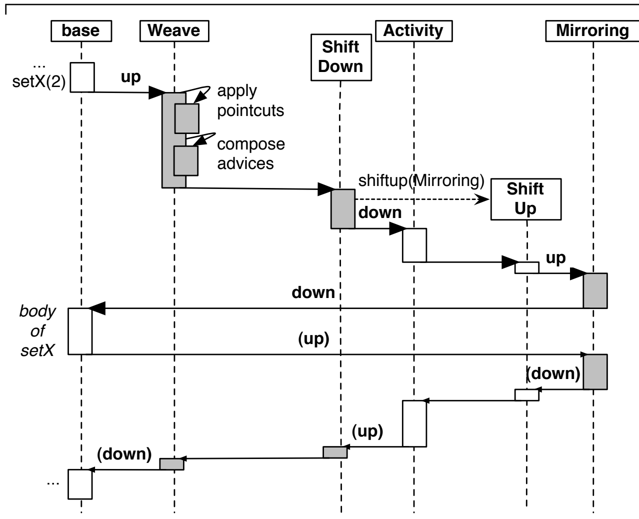


Figure 7. Level shifting with two advices, Activity and Mirroring. Level 0 execution is white, level 1 execution is grey.
(Backward thin arrows are "returns", with their associated level shift.)

to the base level, while still ensuring that the check of the `Point?` predicate is not considered base level computation:

```
(deploy (let ((area ...))
          (lambda (jp)
             (let ((x (jp-arg 0 jp)))
               (and (Point? x)
                    (down (is-inside x area))))))
        trace)
```

We can also define higher-order wrapper functions, like `shift-up`, which takes a function and returns a new function that runs the given function one level above:

```
(define (shift-up f)
  (lambda args (up (apply f args))))
```

It is possible to depart from the chosen default semantics, to express the original AO view according to which pointcuts are metalevel predicates and advice is base code. We can use a `deploy-aj` sugar defined as:

```
(deploy-aj pc adv)
≡ (deploy pc (adv-shift-down adv))
```

Note that here we cannot simply use `shift-down` (similar to `shift-up` defined previously) to transform the advice. Indeed, multiple advices are chained together by means of `proceed`. In a higher-order aspect language , an advice is a function that takes a `proceed` function, context information, and a variable number of arguments at the join point [Dutchyn et al. 2006]. The `proceed` function is used to either call the next advice, or to run the original computation, if it is the last advice in the chain. Therefore, simply shifting the execution level of one advice implies that subsequent advices also run at the modified level. One should rather use the advice shifting function below:

```
(define (adv-shift-down adv)
  (lambda (proceed ctx . args)
    (let ((new-proc (shift-up proceed)))
      (down (apply adv (append (list new-proc ctx)
                               args))))))
```

`adv-shift-down` ensures that the execution levels are maintained appropriately by shifting the `proceed` function in the reverse direction, *i.e.* the advice body is shifted down, while the `proceed` function is shifted up with `shift-up`.

Figure 7 shows the evolution of control flow with our two aspects `Activity` and `Mirroring` that both apply on a call to `setX`. The evaluation of pointcuts is done at level 1, since this is the default. The advices that apply are chained. In this example, the `Activity` advice has been wrapped (using `adv-shift-down`) so as to execute at level 0. When the Shift Down wrapper runs, it creates a Shift Up wrapper (with `shift-up`) for the following advice in the chain. Therefore, when the `Activity` advice proceeds, execution shifts up, and the `Mirroring` advice runs at level 1 (the default). When it proceeds, since it is the last advice, the body of `setX` is run, at level 0.

3.3 Level Shifting and Information Hiding

By moving pointcuts and advices up and down, one actually controls their visibility, with respect to other aspects. As it turns out, level shifting is orthogonal to the pointcut/advice mechanism, to the extent that it applies to any expression, not only pointcuts and advice bodies. This mechanism can therefore be used to run any arbitrary piece of code at another level of execution[4].

For instance, if a function invokes a security manager each time it is applied in order to ensure that its execution is authorized, it can "hide" the invocation and execution of the security manager from aspects observing its execution level by pushing it to a higher level. This means that level shifting can be used to address, to some extent, the issue of information hiding violation that has been raised with respect to standard aspect languages. For instance, in Open Modules [Aldrich 2005], only join points explicitly exposed though pointcuts of the interface of a module are visible to aspects of other modules. This connection suggests interesting extensions of our work towards a flexible notion of "execution domains" (not necessarily sequential levels) that could be used for similar purposes.

Also, the level-shifting operators **up** and **down** are relative only, making it possible to shift execution one level up or down, respectively. It remains to be determined through practical experience whether these operators are sufficient. One could indeed consider a **bottom** operator that moves execution down to level 0, as well as a **top** operator that moves execution to the uppest level, so that execution is invisible to all aspects. The semantics we present in the following section does not consider these operators, though it would be straightforward to accomodate them.

4. Semantics

We now turn to the formal semantics of higher-order aspects with level shifting. We introduce a core language extended with execution levels and aspect weaving. In this section we only present the essential elements, and skip the obvious. The complete consolidated formal description of the language is provided in appendix.

Figure 8 presents the user-visible syntax of the core language, *i.e.* without aspects nor execution levels. The language is a simple Scheme-like language with booleans, numbers and lists, and a number of primitive functions to operate on these. The only expressions considered are multi-arity function application, and **if** expressions. The full language includes also sequencing (**begin**) and binding (**let**) expressions for convenience.

[4] In this work, level shifting is useful only in the presence of aspects, since only aspects are sensitive to levels.

$$
\begin{array}{rlll}
\mathit{Value} & v & ::= & (\lambda(x\cdots)\ e) \mid n \mid \#t \mid \#f \\
& & & \mid (\textbf{list}\ v\cdots) \mid prim \mid \mathit{unspecified} \\
& prim & ::= & \textbf{list} \mid \textbf{cons} \mid \textbf{car} \mid \textbf{cdr} \mid \textbf{empty?} \\
& & & \mid \textbf{eq?} \mid + \mid - \mid \ldots \\
\mathit{Expr} & e & ::= & v \mid x \mid (e\ e\cdots) \mid (\textbf{if}\ e\ e\ e) \\
& v & \in & \mathscr{V}\text{, the set of values} \\
& n & \in & \mathscr{N}\text{, the set of numbers} \\
& list & \in & \mathscr{L}\text{, the set of lists} \\
& x & \in & \mathscr{X}\text{, the set of variable names} \\
& e & \in & \mathscr{E}\text{, the set of expressions} \\
\mathit{EvalCtx} & E & ::= & [\,] \mid (v\cdots\ E\ e\cdots) \mid (\textbf{if}\ E\ e\ e)
\end{array}
$$

Figure 8. Syntax of the core language.

$$
\begin{array}{rlll}
\mathit{Expr} & e & ::= & \ldots \mid (\textbf{up}\ e) \mid (\textbf{down}\ e) \mid \\
& & & (\texttt{in-up}\ e) \mid (\texttt{in-down}\ e) \\
\mathit{EvalCtx} & E & ::= & \ldots \mid (\texttt{in-up}\ E) \mid (\texttt{in-down}\ E)
\end{array}
$$

$$
\begin{array}{rcll}
\langle l, J, E[(\textbf{up}\ e)]\rangle & \hookrightarrow & \langle l+1, J, E[(\texttt{in-up}\ e)]\rangle & \textsc{InUp} \\
\langle l, J, E[(\texttt{in-up}\ v)]\rangle & \hookrightarrow & \langle l-1, J, E[v]\rangle & \textsc{OutUp} \\
\langle l, J, E[(\textbf{down}\ e)]\rangle & \hookrightarrow & \langle l-1, J, E[(\texttt{in-down}\ e)]\rangle & \textsc{InDwn} \\
\langle l, J, E[(\texttt{in-down}\ v)]\rangle & \hookrightarrow & \langle l+1, J, E[v]\rangle & \textsc{OutDwn}
\end{array}
$$

Figure 9. Shifting execution levels.

We describe the operational semantics of our language via a reduction relation $\hookrightarrow$, which describes evaluation steps:

$$\hookrightarrow: \mathscr{L} \times \mathscr{J} \times \mathscr{E} \rightarrow \mathscr{L} \times \mathscr{J} \times \mathscr{E}$$

An evaluation step consists of an execution level $l \in \mathscr{L}$, a join point stack $J \in \mathscr{J}$ and an expression $e \in \mathscr{E}$. The reduction relation takes a level, a stack, and an expression and maps this to a new evaluation step. The reduction rules for the core language are standard [Matthews and Findler 2008] and not presented here. See the appendix for details.

In the following we describe the semantics of execution levels, the join point stack, aspects and their deployment, and the weaving semantics. By convention, when we introduce new user-visible syntax (*e.g.* the aspect deployment expression), we use **bold** font. Extra expression forms added only for the sake of the semantics are written in `typewriter` font.

4.1 Execution Levels

The language supports explicit execution level shifting forms, **up** and **down** (Figure 9). Correspondingly, there are two (not user-visible) marker expressions, `in-up` and `in-down` used to keep track of the level counter. When encountering an **up** expression, the level counter is increased, and an `in-up` marker is placed in the execution context (InUp). When the nested expression has been reduced to a value, the `in-up` mark is disposed, and the level counter is decreased (OutUp). Evaluation of a shift down expression is done similarly (see rules InDown and OutDown).

$$
\begin{array}{rcl}
J & ::= & j + J \mid \bullet \\
j & ::= & \lceil l, k, v, v\cdots \rceil \\
k & ::= & \texttt{call} \mid \texttt{pc} \mid \texttt{adv} \\
l & \in & \mathscr{N} \\
J & \in & \mathscr{J}\text{, the set of join point stacks}
\end{array}
$$

$$
\begin{array}{rlll}
\mathit{Expr} & e & ::= & \ldots \mid \texttt{jp}\ j \mid (\texttt{in-jp}\ e) \\
\mathit{EvalCtx} & E & ::= & \ldots \mid (\texttt{in-jp}\ E)
\end{array}
$$

$$\langle l, j+J, E[\texttt{in-jp}\ v]\rangle \hookrightarrow \langle l, J, E[v]\rangle \qquad \textsc{OutJp}$$

Figure 10. The join point stack.

4.2 Join Point Stack

We follow [Clifton and Leavens 2006] in the modeling of the join point stack (Figure 10). The join point stack J is a list of *join point abstractions* j, which are tuples $\lceil l, k, v, v\cdots \rceil$: the execution level of occurrence l, the join point kind k, the applied function v, and the arguments $v\cdots$. We consider three kinds of join points, `call` for function applications, `pc` for pointcut executions, and `adv` for advice executions[5].

In order to keep track of the join point stack in the semantics we introduce two (not user-visible) expression forms: `jp` j introduces a join point, and (`in-jp` e) keeps track of the fact that execution is proceeding under a given dynamic join point. The definition of the evaluation context is updated accordingly (Figure 10).

A join point abstraction captures all the information required to match it against pointcuts, as well as to trigger its corresponding computation when necessary. For instance, the reduction rule for `call` join points is as follows (we will see other kinds later on):

$$
\begin{array}{ll}
\langle l, J, E[((\lambda(x\cdots)\ e)\ v\cdots)]\rangle & \textsc{App} \\
\quad \hookrightarrow \langle l, J, E[\texttt{jp}\ \lceil l, \texttt{call}, (\lambda(x\cdots)\ e), v\cdots \rceil]\rangle &
\end{array}
$$

This means that when a function is applied to a list of arguments, the expression is reduced to a `jp` expression with the definition of the corresponding join point, which embeds the current execution level l, its kind `call`, the function definition, and the values passed to it. A later rule pushes the thus created join point to the stack J, marking the expression with `in-jp`, and then triggers weaving. Poping a join point from the stack is done by the OutJp rule, when the expression under a dynamic join point has been reduced to a value.

4.3 Aspects and Deployment

As described on Figure 11, an aspect is a tuple $\langle l, pc, adv\rangle$ where l denotes the execution level at which it is defined, pc is the pointcut and adv the advice (both first-class functions). More precisely, a pointcut is a function that takes a join point stack as input and produces either $\#f$ if it does not match, or a (possibly empty) list of context values exposed to the advice. Following [Dutchyn et al. 2006, Dutchyn 2006], higher-order advice is modeled as a function receiving first a function to apply whenever

[5] For simplicity and conciseness, we do not distinguish call and execution join points here. Our implementation (see Section 4.5) does make this difference, and therefore supports four join point kinds.

$$
\begin{array}{llcl}
\mathit{Aspects} & \mathscr{A} & = & \{\langle l_i, pc_i, adv_i\rangle \mid i = 1, \ldots, |\mathscr{A}|\} \\
\mathit{Pointcut} & pc & \in & \mathscr{J} \rightarrow \{\#f\} \cup \mathscr{L} \\
\mathit{Advice} & adv & \in & (\mathscr{V}^* \rightarrow \mathscr{V}) \times \mathscr{L} \times \mathscr{V}^* \rightarrow \mathscr{V}
\end{array}
$$

$$
prim \;::=\; \ldots \mid \textbf{deploy}
$$

$$
\begin{array}{ll}
\langle l, J, E[(\textbf{deploy}\ v_{pc}\ v_{adv})]\rangle & \textsc{Deploy} \\
\quad \hookrightarrow \langle l, J, E[\textit{unspecified}]\rangle \text{ and } \mathscr{A} = \{\langle l, v_{pc}, v_{adv}\rangle\} \cup \mathscr{A} &
\end{array}
$$

Figure 11. Aspects and deployment (global environment $\mathscr{A}$).

$$
\begin{array}{ll}
\langle l, J', E[\texttt{jp}\ \lceil l, k, v_\lambda, v\cdots\rceil]\rangle & \textsc{Weave} \\
\quad \hookrightarrow \langle l, J, E[(\texttt{in-jp}\ (\textbf{up}\ (\texttt{app/prim}\ W[\![|\mathscr{A}|]\!]_J\ v\cdots)))]\rangle & \\
\qquad \text{where } J = j + J' &
\end{array}
$$

and, with $J = \lceil l, k, (\lambda(x\cdots)\ e), v\cdots\rceil + J'$:

$$
\begin{array}{lcl}
W[\![0]\!]_J & = & (\lambda(a\cdots) \\
& & \quad (\textbf{down}\ (\texttt{app/prim}\ (\lambda(x\cdots)\ e)\ a\cdots))) \\
W[\![i]\!]_J & = & (\texttt{app/prim}\ (\lambda(p) \\
& & \quad (\textbf{if}\ (\textbf{eq?}\ l_i\ l) \\
& & \quad\quad (\textbf{let}\ ((c\ (\texttt{app/pc}\ pc_i\ J))) \\
& & \quad\quad\quad (\textbf{if}\ c \\
& & \quad\quad\quad\quad (\lambda(a\cdots)(\texttt{app/adv}\ adv_i\ p\ c\ a\cdots)) \\
& & \quad\quad\quad\quad p)) \\
& & \quad\quad p)) \\
& & \quad\quad\quad W[\![i-1]\!]_J)
\end{array}
$$

Figure 12. Aspect weaving, with level shifting.

the advice wants to proceed, a list of values exposed by the pointcut, and the arguments passed at the original join point.

An aspect environment $\mathscr{A}$ is a set of such aspects. An aspect is deployed with a `deploy` expression (added as a primitive to the language, see Figure 11). To simplify our reduction semantics, in this section we have not included the aspect environment as part of the description of an evaluation step. Rather, we simply "modify" the global aspect environment $\mathscr{A}$ upon aspect deployment[6] (see rule DEPLOY). Also note that we do not model the different scoping strategies of AspectScheme here—we restrain ourselves to deployment in a global aspect environment. For more advanced management of aspect scoping and aspect environments, see [Tanter 2008b]. When an aspect is deployed, it captures its execution level of definition. This means that, when executing at level n, (**deploy** p a) deploys the aspect such that it sees join points representing execution of level n, and (**up** (**deploy** p a)) deploys the aspect a level above, such that it sees join points that denote execution at level $n+1$.

4.4 Weaving

We now turn to the semantics of aspect weaving. The WEAVE rule describes the process. A `jp` expression is converted to an `in-jp` expression, and the join point is pushed onto the stack. The inner expression of `in-jp` is the application, one execution level up, of

$$
\begin{array}{llcl}
\mathit{Expr} & e & ::= & \ldots \mid (f\ e\ e\cdots) \\
\mathit{AppForm} & f & ::= & \texttt{app/pc} \mid \texttt{app/adv} \mid \texttt{app/prim} \\
\mathit{EvalCtx} & E & ::= & \ldots \mid (f\ v\cdots\ E\ e\cdots)
\end{array}
$$

$$
\begin{array}{ll}
\langle l, J, E[((\lambda(x\cdots)\ e)\ v\cdots)]\rangle & \textsc{App} \\
\quad \hookrightarrow \langle l, J, E[\texttt{jp}\ \lceil l, \texttt{call}, (\lambda(x\cdots)\ e), v\cdots\rceil]\rangle & \\
\langle l, J, E[(\texttt{app/pc}\ (\lambda(x\cdots)\ e)\ v\cdots)]\rangle & \textsc{AppPc} \\
\quad \hookrightarrow \langle l, J, E[\texttt{jp}\ \lceil l, \texttt{pc}, (\lambda(x\cdots)\ e), v\cdots\rceil]\rangle & \\
\langle l, J, E[(\texttt{app/adv}\ (\lambda(x\cdots)\ e)\ v\cdots)]\rangle & \textsc{AppAdv} \\
\quad \hookrightarrow \langle l, J, E[\texttt{jp}\ \lceil l, \texttt{adv}, (\lambda(x\cdots)\ e), v\cdots\rceil]\rangle & \\
\langle l, J, E[(\texttt{app/prim}\ (\lambda(x\cdots)\ e)\ v\cdots)]\rangle & \textsc{AppPrim} \\
\quad \hookrightarrow \langle l, J, E[e\{v\cdots/x\cdots\}]\rangle &
\end{array}
$$

Figure 13. Different kinds of application.

the list of advice functions that match the correct join point properly chained together, to the original arguments. Note that the weaving rule applies uniformly over the different kinds of join points.

Our weaving process is closely based on that described by Dutchyn. It only differs in that we deal with execution levels, and introduce both pointcut and advice join points. The W metafunction recurs on the global aspect environment $\mathscr{A}$ and returns a composed procedure whose structure reflects the way advice is going to be dispatched (Figure 12).

For each aspect $\langle l_i, pc_i, adv_i\rangle$ in the environment, W first checks whether the aspect is at the same execution level as the join point, *i.e.* if the aspect can actually "see" the join point. If so, it applies its pointcut pc_i to the current join point stack. If the pointcut matches, it returns a list of context values, c. W then returns a function that, given the actual join point arguments, applies the advice adv_i. All this process is parameterized by the function to proceed with, p. This function is passed to the advice, and if an aspect does not apply, then W simply returns this function. The base case, $W[\![0]\!]_J$ corresponds to the execution of the original function. Note that it is performed by **down**ing the execution level, to reflect the fact that while, by default, pointcuts and advice run at an upper level, the original function runs at its original level of application.

The WEAVE rule uses a number of different application forms, namely `app/pc`, `app/adv` and `app/prim`, described in Figure 13. These forms are introduced to be able to distinguish the different manners of applying a function in the language. Note that these forms are not in user-visible syntax. However, since we use the base language to weave, we need to be able to denote the primitive application of a function, *i.e.* such that it does not trigger a join point. The APPPRIM rule simply performs the classical β_v reduction. `app/prim` is used to hide the activity of the pointcut matcher and the advice dispatcher, as well as to perform an actual function application (*i.e.* when all aspects have proceeded, see $W[\![0]\!]_J$)[7].

W uses two dedicated application forms to apply pointcuts and advice, respectively, `app/pc` and `app/adv`. These forms are introduced so as to generate join points of different kinds, corresponding to different uses of a function. Rule APPPC creates a join point of kind `pc`, and rule APPADV creates a join point of kind `adv`.

[6] The complete semantics given in the appendix properly includes the aspect environment in the evaluation steps.

[7] Note that contrary to AspectScheme, thanks to management of execution levels, it is not necessary for `app/prim` to be in user-visible syntax.

4.5 Availability

We have defined the complete semantics of our language using PLT Redex, a domain-specific language for specifying reduction semantics [Felleisen et al. 2009]. The automatically-generated rendering of the complete language grammar, reduction relation, and weaving metafunction W are given in Appendix A. The full definition, as well as a number of executable test cases can be obtained from: `http://pleiad.cl/research/scope`

We have also implemented our language as an extension of AspectScheme (*i.e.* a language module extending PLT Scheme using macros), available at the same website. The language supports both call and execution join points, in addition to pointcut and advice execution, and includes the different scoping semantics for aspects (statically and dynamically scoped) in addition to global, top-level deployment. It also includes level shifting forms (as macros that handle a dynamically-scoped parameter).

5. Related Work

Reflective towers. Seminal work on reflection focused on the notion of a *reflective tower*. This tower is a stack of interpreters, each one executing the one below. Reification and reflection are level-shifting mechanisms, by which one can navigate in the tower. This idea was first introduced by Brian Smith [Smith 1982] with 2-Lisp and 3-Lisp, and different flavors of it were subsequently explored, with languages like Brown [Wand and Friedman 1988] and Blond [Danvy and Malmkjaer 1988].

2-Lisp focuses on structural reflection, by which values can be moved up and down. An `up` operation reduces its argument to a value and returns (a representation of) the internal structure of that value (*i.e.* its "upper" identity). Conversely, `down` returns the base-level value that corresponds to a given internal structure. 3-Lisp introduces procedural reflection by which *computation* can actually be moved in the tower. This is done by introducing a special kind of abstraction, a *reflective procedure*, which is a procedure of fixed arity that, when applied, runs at the level above[8]. It receives as parameters some internal structures of the interpreter (typically the current expression, environment, and continuation). Control can return back to the level below by applying the evaluation function.

In this framework, one could describe the pointcut-advice mechanism as follows, at least in its original form [Wand et al. 2004]. Pointcuts are reflective procedures, that take as parameter (a representation of) the current join point. In contrast to reflective procedures in reflective languages, they are not explicitly applied; rather, they are "installed" in the interpreter, and their application is triggered by the interpreter at each join point. A pointcut runs at the upper level and, if it matches, returns bindings that are consequently used for the (base-level) execution of the advice.

The level shifting operations we introduce in this work differ from level shifting in the reflective tower in a number of ways. Most importantly, there is no tower of interpreters at all: execution levels are just properties of execution flows. Only aspects (more precisely, pointcuts) are sensitive to this property of execution flows. Pointcuts and advices are all evaluated by the very same interpreter that evaluates the whole program. Level shifting operations just taint the execution flow such that the produced join points are only visible to aspects sitting at the corresponding level. This "illusion of the tower" also explains why there is no explicit wrapping and unwrapping of values between levels (as opposed to *e.g.* 2-Lisp).

[8] Interestingly, Blond makes the distinction between reflective procedures that run at the level above the level at which they are applied, and procedures that run at the level above that at which they were defined.

Infinite regression. The issue of infinite regression in metalevel architectures has been long identified [des Rivières and Smith 1984, Kiczales et al. 1991]. Chiba, Kiczales and Lamping recognized the ad hoc nature of regression checks, identifying the more general issue of metalevel *conflation* [Chiba et al. 1996]. In the proposed meta-helix architecture, extensions to objects (*e.g.* new fields) are layered on top of each other. Levels are reified, at runtime if necessary, and an object has a representative at each level. An "implemented-by" relation based on delegation keeps level clearly separated.

In previous work, we studied similar issues with a particular kind of aspects, which perform structural adaptations (*a.k.a.* inter-type declarations or introductions). We proposed a mechanism of *visibility* of structural changes introduced by aspects [Tanter 2006, Tanter and Fabry 2009]. The visibility system, implemented in the Reflex AOP kernel, allows one to declare which aspects see the changes made by which other aspects, or to declare that changes made by an aspects are globally visible or globally hidden. While more flexible that a strict layered architecture like the meta-helix, this system is harder to reason about and specifications can easily conflict with each other. Also, in this proposal, it is impossible for base level code to hide certain members so they are not visible to (some) aspects.

Stratified aspects. To the best of our knowledge, the first piece of work directly related to the issue of infinite recursion with the pointcut/advice mechanism is due to Bodden and colleagues. With stratified aspects, aspects are associated with levels, and the scope of pointcuts is restricted to join points of lower levels [Bodden et al. 2006]. The work focuses on advice-triggered reentrancy only, and does not mention the issue related to *e.g.* `if` pointcuts. A more fundamental issue with stratified aspects is that levels are *statically* declared and determined. That is, classes live at level 0, aspects at level 1, meta-aspects at level 2, and so forth. This means that stratified aspects fail to recognize that levels are a property of *execution flows*, not of static declared entities. As a consequence, as recognized by the authors, it is impossible to properly handle shift downs, *i.e.* when an aspect calls a method of a level 0 object.

The meta context. Recently, Denker *et al.* introduced the idea of passing an implicit "meta-context" argument to meta-objects such that they can determine at which level they run [Denker et al. 2008]. This generalizes the idea of the meta-helix and recognizes that levels are a property of execution flows. In their system, metaobjects always run at their level, and execution only shift downs when a metaobject calls `proceed` on the reification of an execution event (*i.e.* a join point in AO terminology). While close to ours, the work really remains in the domain of metalevel architectures and therefore cannot reconcile with the original AO view, according to which advice is base level. Here, in addition, we uncouple level shifting from the behavioral reflection/pointcut-advice mechanism. Finally, the level of execution of activation conditions (the equivalent of pointcut residues in that model) is left unspecified.

Controlling reentrancy. In previous work, we analyzed the issue of unwanted applications of aspects in a general setting [Tanter 2008a]. We identify three kinds of aspect reentrancy: base-triggered reentrancy (*e.g.* caused by a recursive program), pointcut-triggered reentrancy (*e.g.* caused by an `if` pointcut), and advice-triggered reentrancy. After showing the somewhat surprising strategies of current AspectJ compilers with respect to `if` pointcuts, we propose a safe default semantics for aspects, according to which the activity of an aspect is invisible to itself, and an aspect is immune to iterative/recursive refactorings. To deal with pointcut-triggered reentrancy, we introduce the pointcut execution

join point. We also allow for well-defined scoped reentrancy to be introduced, for instance to match reentrant join points whenever the executing objects differ. The reentrancy control proposal does not deal with levels at all. It is only concerned with properly dealing with the conflation when it occurs (as indicated by the original AO view). In fact, in an aspect language with execution levels as proposed in this paper, reentrancy control is still needed, for cases where the programmer causes conflation to occur (*e.g.* by running an advice at a lower level).

6. Conclusion

We have proposed a higher-order aspect language design in which pointcuts and advices are regular functions and yet, by default, infinite regression never occurs. This is done by introducing a notion of *levels of execution* that help discriminate the context in which functions are being used. Explicit level shifting expressions make it possible to control the visibility of computation. We believe this work reconciles the (usually unwanted or embarassing) “metaness” of aspects with the (usually unrecognized) “baseness” of runtime metaobject protocols. The key point lies in viewing metaness not as an intrinsic/static property of a piece of program, but as a property of execution flows, ultimately under control of the programmer. We are working on adding execution levels to practical aspect languages like AspectJ in order to empirically validate the usefulness of the proposed design.

Acknowledgments. We thank Gregor Kiczales for discussions on this topic and proposal, as well as the anonymous reviewers for their insightful comments and suggestions.

References

[Aldrich 2005] Aldrich, J. (2005). Open modules: Modular reasoning about advice. In Black, A. P., editor, *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, number 3586 in Lecture Notes in Computer Science, pages 144–168, Glasgow, UK. Springer-Verlag.

[Avgustinov et al. 2006] Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. (2006). abc: an extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 293–334. Springer-Verlag.

[Bodden et al. 2006] Bodden, E., Forster, F., and Steimann, F. (2006). Avoiding infinite recursion with stratified aspects. In *Proceedings of Net.ObjectDays 2006*, Lecture Notes in Informatics, pages 49–54. GI-Edition.

[Chiba et al. 1996] Chiba, S., Kiczales, G., and Lamping, J. (1996). Avoiding confusion in metacircularity: The meta-helix. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer-Verlag.

[Clifton and Leavens 2006] Clifton, C. and Leavens, G. T. (2006). MiniMAO$_1$: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming*, 63:312–374.

[Danvy and Malmkjaer 1988] Danvy, O. and Malmkjaer, K. (1988). Intensions and extensions in a reflective tower. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 327–341, Snowbird, Utah, USA. ACM Press.

[Denker et al. 2008] Denker, M., Suen, M., and Ducasse, S. (2008). The meta in meta-object architectures. In *Proceedings of TOOLS Europe*, Lecture Notes in Business and Information Processing, Zurich, Switzerland. Springer-Verlag. To appear.

[des Rivières and Smith 1984] des Rivières, J. and Smith, B. C. (1984). The implementation of procedurally reflective languages. In *Proceedings of the Annual ACM Symposium on Lisp and Functional Programming*, pages 331–347.

[Dutchyn 2006] Dutchyn, C. (2006). *Dynamic Join Points: Model and Interactions*. PhD thesis, University of British Columbia, Canada.

[Dutchyn et al. 2006] Dutchyn, C., Tucker, D. B., and Krishnamurthi, S. (2006). Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239.

[Felleisen et al. 2009] Felleisen, M., Findler, R. B., and Flatt, M. (2009). *Semantics Engineering with PLT Redex*. The MIT Press. To appear.

[Kiczales 2009] Kiczales, G. (2009). Personal communication.

[Kiczales et al. 1991] Kiczales, G., des Rivières, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press.

[Maes 1987] Maes, P. (1987). Concepts and experiments in computational reflection. In Meyrowitz, N., editor, *Proceedings of the 2nd International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 87)*, pages 147–155, Orlando, Florida, USA. ACM Press. ACM SIGPLAN Notices, 22(12).

[Masuhara et al. 2003] Masuhara, H., Kiczales, G., and Dutchyn, C. (2003). A compilation and optimization model for aspect-oriented programs. In Hedin, G., editor, *Proceedings of Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag.

[Matthews and Findler 2008] Matthews, J. and Findler, R. B. (2008). An operational semantics for Scheme. *Journal of Functional Programming*, 18(1):47–86.

[Smith 1982] Smith, B. C. (1982). Reflection and semantics in a procedural language. Technical Report 272, MIT Laboratory of Computer Science.

[Tanter 2006] Tanter, É. (2006). Aspects of composition in the Reflex AOP kernel. In Löwe, W. and Südholt, M., editors, *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *Lecture Notes in Computer Science*, pages 98–113, Vienna, Austria. Springer-Verlag.

[Tanter 2008a] Tanter, É. (2008a). Controlling aspect reentrancy. *Journal of Universal Computer Science*, 14(21):3498–3516. Best Paper Award of the Brazilian Symposium on Programming Languages (SBLP 2008).

[Tanter 2008b] Tanter, É. (2008b). Expressive scoping of dynamically-deployed aspects. In *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*, pages 168–179, Brussels, Belgium. ACM Press.

[Tanter and Fabry 2009] Tanter, É. and Fabry, J. (2009). Supporting composition of structural aspects in an AOP kernel. *Journal of Universal Computer Science*, 15(3):620–647.

[Wand and Friedman 1988] Wand, M. and Friedman, D. P. (1988). The mystery of the tower revealed: a non-reflective description of the reflective tower. *Lisp and Symbolic Computation*, 1(1):11–37.

[Wand et al. 2004] Wand, M., Kiczales, G., and Dutchyn, C. (2004). A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910.

[Zimmermann 1996] Zimmermann, C. (1996). *Advances in Object-Oriented Metalevel Architectures and Reflection*. CRC Press.

A. Complete definition of the semantics

This appendix includes the complete grammar of the language (Figure 15), the weaving function W (Figure 14), and the complete reduction relation (Figure 16), all automatically generated from the PLT Redex definition.

$$
\begin{array}{rl}
e ::= & (e\ e\ ...) \mid (\textsf{if}\ e\ e\ e) \mid x \mid v \\
 & \mid (\textsf{let}\ ((x\ e)\ ...)\ e) \\
 & \mid (\textsf{begin}\ e\ e\ ...) \\
 & \mid (\textsf{up}\ e) \mid (\textsf{down}\ e) \\
 & \mid (\textsf{in-up}\ e) \mid (\textsf{in-down}\ e) \\
 & \mid (\textsf{jp}\ j) \mid (\textsf{in-jp}\ e) \\
 & \mid (f\ e\ e\ ...) \\
f ::= & \textsf{app/pc} \mid \textsf{app/adv} \mid \textsf{app/prim} \\
v ::= & (\lambda\ (x\ ...)\ e) \mid (\textsf{list}\ v\ ...) \mid \mathit{number} \mid \#\mathrm{t} \mid \#\mathrm{f} \\
 & \mid \textsf{string} \mid \mathit{prim} \mid \textsf{unspecified} \\
\mathit{prim} ::= & + \mid - \mid \textsf{list} \mid \textsf{cons} \mid \textsf{car} \mid \textsf{cdr} \mid \textsf{empty?} \mid \textsf{eq?} \mid \textsf{deploy} \\
P ::= & (l\ J\ A\ E) \\
l ::= & \mathit{number} \\
J ::= & (\textsf{stack}\ j\ ...) \\
j ::= & (l\ k\ v\ v\ ...) \\
k ::= & \text{“call”} \mid \text{“pc”} \mid \text{“adv”} \\
A ::= & (\textsf{asps}\ a\ ...) \\
a ::= & (l\ v\ v) \\
E ::= & (v\ ...\ E\ e\ ...) \\
 & \mid (\textsf{if}\ E\ e\ e) \\
 & \mid (\textsf{begin}\ E\ e\ e\ ...) \\
 & \mid (\textsf{in-up}\ E) \mid (\textsf{in-down}\ E) \mid (\textsf{in-jp}\ E) \\
 & \mid (f\ v\ ...\ E\ e\ ...) \\
 & \mid [\,] \\
x ::= & \mathit{variable\text{-}not\text{-}otherwise\text{-}mentioned}
\end{array}
$$

Figure 15. Complete grammar (generated automatically from PLT Redex).

```
W[[(l1 k (λ (x ...) e) v ...), (asps (l2 v1 v2) a ...)]] = (app/prim (λ (p)
                                                             (if (eq? l1 l2)
                                                               (let ((c (app/pc v1 (list k (λ (x ...) e) v ...))))
                                                                 (if c
                                                                   (λ (x ...) (app/adv v2 p c x ...))
                                                                   p))
                                                               p))
                                                           W[[(l1 k (λ (x ...) e) v ...), (asps a ...)]] )
W[[(l1 k (λ (x ...) e) v ...), (asps)]]                 = (λ (x ...) (down (app/prim (λ (x ...) e) x ...)))
```

Figure 14. Weaving meta-function (generated automatically from PLT Redex).

```
P[(+ number1 number2)] ⟶ P[(+ number1 number2)]                                    [+]
P[(- number1 number2)] ⟶ P[(- number1 number2)]                                    [-]
P[(if #f e1 e2)] ⟶ P[e2]                                                           [if-f]
P[(if v1 e1 e2)] ⟶ P[e1]                                                           [if-t]
    where v1
P[(let ((x e) ...) e0)] ⟶ P[(app/prim (λ (x ...) e0) e ...)]                       [let]
P[(begin v e1 e2 ...)] ⟶ P[(begin e1 e2 ...)]                                      [seq]
P[(begin e)] ⟶ P[e]                                                                [outseq]
P[(eq? v1 v2)] ⟶ P[#t]                                                             [eq-t]
    where (eq? v1 v2)
P[(eq? v1 v2)] ⟶ P[#f]                                                             [eq-f]
    where (not (eq? v1 v2))
P[(cons v0 (list v1 ...))] ⟶ P[(list v0 v1 ...)]                                   [cons]
P[(car (list v1 v2 ...))] ⟶ P[v1]                                                  [car]
P[(cdr (list v1 v2 ...))] ⟶ P[(list v2 ...)]                                       [cdr]
P[(empty? (list))] ⟶ P[#t]                                                         [empty?-t]
P[(empty? v1)] ⟶ P[#f]                                                             [empty?-f]
    where (not (equal? v1 (list)))
(l1 J A E[(up e)]) ⟶ ((add1 l1) J A E[(in-up e)])                                  [InUp]
(l1 J A E[(in-up v)]) ⟶ ((sub1 l1) J A E[v])                                       [OutUp]
(l1 J A E[(down e)]) ⟶ ((sub1 l1) J A E[(in-down e)])                              [InDwn]
(l1 J A E[(in-down v)]) ⟶ ((add1 l1) J A E[v])                                     [OutDwn]
(l1 J (asps a ...) E[(deploy v1 v2)]) ⟶ (l1 J (asps (l1 v1 v2) a ...) E[unspecified])   [Deploy]
P[(app/prim (λ (x ...) e) v ...)] ⟶ P[subst-n[[(x v), ..., e]] ]                   [AppPrim]
(l1 J A E[((λ (x ...) e) v ...)]) ⟶ (l1 J A E[(jp (l1 "call" (λ (x ...) e) v ...))])     [App]
(l1 J A E[(app/pc (λ (x ...) e) v ...)]) ⟶ (l1 J A E[(jp (l1 "pc" (λ (x ...) e) v ...))])   [AppPc]
(l1 J A E[(app/adv (λ (x ...) e) v ...)]) ⟶ (l1 J A E[(jp (l1 "adv" (λ (x ...) e) v ...))]) [AppAdv]
(l1 (stack j ...) (asps a ...) E[(jp (l1 k (λ (x ...) e) v ...))]) ⟶ (l1 (stack (l1 k (λ (x ...) e) v ...) j ...) (asps a ...)   [Weave]
                                        E[(in-jp (up (app/prim W[[(l1 k (λ (x ...) e) v ...),
                                                                  (asps a ...)]]
                                                               v ...)))])
(l1 (stack (l1 k v0 v ...) j ...) A E[(in-jp v1)]) ⟶ (l1 (stack j ...) A E[v1])    [OutJp]
```

Figure 16. Complete set of reduction rules (generated automatically from PLT Redex).

Fixing Letrec (reloaded)

Abdulaziz Ghuloum
Indiana University
aghuloum@cs.indiana.edu

R. Kent Dybvig
Indiana University
dyb@cs.indiana.edu

Abstract

The Revised[6] Report on Scheme introduces three fundamental changes involving Scheme's recursive variable binding constructs. First, it standardizes the sequential recursive binding construct, `letrec*`, which evaluates its initialization expressions in a strict left-to-right order. Second, it specifies that internal and library definitions have `letrec*` semantics. Third, it prohibits programs from invoking the continuation of a `letrec` or `letrec*` init expression more than once. The first two changes increase the incentive for handling `letrec*` efficiently, while the third change gives the compiler more options for transforming `letrec` and `letrec*` expressions.

This paper extends an earlier effort of Waddell, Sarkar, and Dybvig to handle the Revised[5] Report `letrec` and the (then nonstandard) `letrec*` efficiently. It presents more aggressive transformations for `letrec` and `letrec*` that take advantage of the new prohibition on invoking the continuations of initialization expressions multiple times. The implementation employs Tarjan's algorithm for finding strongly connected components in a graph that encodes the dependencies among the bindings.

Keywords Scheme, recursive binding construct, internal definitions, mutual recursion, mutual definition, continuations, optimization, letrec

Proceedings of the 2009 Scheme and Functional Programming Workshop
California Polytechnic State University Technical Report CPSLO-CSC-09-03

1. Introduction

Scheme's `letrec` form, used to create recursive bindings, is easily transformed into a standard combination of `let` and `set!` expressions [4]. Unfortunately, the standard transformation introduces unnecessary assignments that may inhibit subsequent optimization of the form. An alternative transformation is described by Waddell, Sarkar, and Dybvig [9]. This transformation often succeeds in avoiding unnecessary assignments while maintaining the Revised[5] Report semantics for `letrec`. Waddell, et al., also define a variant of `letrec`, called `letrec*` by analogy to `let*`, that evaluates its initialization expressions from left to right, and they present a similar optimizing transformation for `letrec*`.

The Revised[6] Report on Scheme [5] changes the status quo by including `letrec*` as well as `letrec` in the language and, more importantly, by changing the semantics of internal defines so that `define`-bound variables behave as if bound by `letrec*` rather than `letrec`. An immediate consequence of this changes is that R[6]RS programs (on average) will contain more variables bound according to the `letrec*` semantics. Thus, optimizing `letrec*` has become even more important. Over time, we also expect programmers to take advantage of this change by using the values of earlier bindings for the purpose of initializing subsequent ones. Unfortunately, this will result in more of the so-called *complex* bindings for which the Waddell, et al., transformation, like the naive transformation, often produces unnecessary assignments.

The Revised[6] Report makes one other semantic change in its recursive binding constructs, which is to prohibit programs from invoking the continuation of a `letrec` or `letrec*` init expression more than once. This gives the implementation more flexibility to avoid producing assignments whose absence could otherwise

have been detected by invoking the continuation of an initialization expression multiple times.

This paper presents a new transformation algorithm that often produces fewer assignments than the Waddell, et al., transformation, especially for `letrec*`. At worst, it produces the same number of assignments as the Waddell, et al., transformation. It accomplishes this by an aggressive partitioning of bindings into minimal mutually dependent groups, aided by Tarjan's algorithm [6] for finding strongly connected components, and by taking advantage of the new prohibition against invoking the continuations of initialization expressions multiple times.

The rest of this paper is organized as follows. Section 2 gives an overview of the syntax and semantics of Scheme's `letrec` and `letrec*` forms along with the run-time restrictions imposed by the standard. Section 3 gives the naive but straightforward implementation of these forms based on source-level macro transformation. Section 4 summarizes the *fixing letrec* algorithm of Waddell, Sarkar, and Dybvig [9]. Section 5 motivates our work by showing simple examples where the original algorithm yields undesirable results. Section 6 describes the constraints that our transformation must follow, the analysis it requires, and how it encodes the information it needs in a graph form. Section 7 shows how the algorithm uses the strongly connected-components to perform the actual transformation. Finally, Section 8 provides some concluding remarks.

2. Semantics of `letrec` and `letrec*`

The syntax of the `letrec` and `letrec*` forms are identical, except for the opening keyword:

```
(letrec ([var init] ...) body ...)
(letrec* ([var init] ...) body ...)
```

The lexical scoping rules for `letrec` and `letrec*` are the same: all of *var*s are visible in all *init* expressions as well as in the *body* definitions and expressions. For both forms, evaluation proceeds as follows:

1. All *var*s are bound to fresh locations.
2. The *init* expressions are evaluated and the value of each *init* is assigned to the corresponding *var*.
3. The *body* is evaluated and the values of the last expression are returned.

The two forms differ in the specifics of item 2. For `letrec`, all *init* expressions are evaluated first, in some unspecified order, before all bindings are initialized to the computed values. For `letrec*`, the bindings are initialized sequentially: each *init* is evaluated, and the corresponding *var* is set before the next binding is initialized.

2.1 Restrictions

Although all *var*s are visible in all *init* expressions, the actual evaluation of the *init* expressions must obey additional restrictions. For `letrec`, each *init* expression must evaluate without referencing or assigning the value of any of the *var*s. During the evaluation of `letrec*` *init* expressions, however, references and assignments to previously initialized bindings (e.g., ones that appear earlier in the list of bindings) are permitted. The R6RS requires that implementations *must* detect and report violations of this restriction [5], though some implementations currently ignore this requirement.

The R6RS also prohibits programs from invoking the continuation (returning from the evaluation) of an *init* expression. The report specifies that implementation *should* detect and report such violations. Thus, portable programs cannot depend on the implementation's behavior when the program violates this prohibition.

Implementations can enforce both of these restrictions via a source code transformation that inserts additional checks.

A transformation that guards against the first restriction with minimal overhead is described by Waddell, et al. [9]. The transformation works as a pre-pass to the *fixing letrec* algorithm. It first inserts validity checks amounting to binding one *initialized?* variable per `letrec` expression, or one per `letrec*` binding, and inserting validity checks anywhere a possible violation of the restriction might occur. Because these checks are separate from the actual `letrec` or `letrec*` bindings, the transformation does not inhibit optimizations involving the bindings. Because it works independently, as a prepass, of the transformation of `letrec` and `letrec*` into more primitive forms, it is applicable regardless of the transformation used, whether it be the naive transformation, the one described by Waddell, et al., or the more sophisticated transformation described in this paper.

To enforce the second restriction, an implementation can transform an unchecked `letrec` or `letrec*` ex-

pression into a checked one via a simple transformation:

```
(define-syntax checked-letrec
  (syntax-rules ()
    [(_ ([var init] ...) b b* ...)
     (letrec ([var (once init)] ...)
        b b* ...)]))
```

where once is a primitive keyword that behaves according to the following definition:

```
(define-syntax once
  (syntax-rules ()
    [(_ expr)
     (let ([returned? #f])
       (let ([v expr])
         (when returned?
           (assertion-violation ---))
         (set! returned? #t)
         v))]))
```

The once wrapper should be omitted at least for *init* expressions that cannot possibly invoke their continuations multiple times, including constants, lambda expressions, variable references, and applications of most primitives to simple values. An implementation might also attempt to omit the once wrapper for other expressions it can prove do not invoke their continuations multiple times.

The remainder of this paper assumes that all uses of letrec and letrec* are either correct (i.e, do not violate the two restrictions of the report stated above) or that the required checks have already been inserted by a prior transformation.

3. Naive letrec / letrec* transformation

The task of the *fixing letrec* pass of the compiler is to rewrite the general Scheme letrec and letrec* forms into simpler forms that subsequent passes of the compiler can easily handle.

The simplest (and most naive) transformation for letrec* can be achieved at the source level via a simple macro:

```
(define-syntax letrec*
  (syntax-rules ()
    [(_ ([var init] ...) b b* ...)
     (let ([var #f] ...)
       (set! var init) ...
       (let () b b* ...))]))
```

It is easy to see how the output of this transformation performs the required letrec* evaluation semantics. (It enforces none of the restrictions, but we are assuming these are enforced by some earlier transformation.) The same transformation can be used for letrec expressions, but it unnecessarily forces the initialization expressions to be evaluated from left to right, preventing the compiler from subsequently choosing a different order that might result in more efficient code. The following definition of letrec avoids specifying a particular evaluation order:

```
(define-syntax letrec
  (lambda (stx)
    (syntax-case stx ()
      [(_ ([var init] ...) b b* ...)
       (with-syntax ([(tmp ...)
                      (generate-temporaries
                        #'(var ...))])
         #'(let ([var #f] ...)
             (let ([tmp init] ...)
               (set! var tmp) ...
               (values))
             (let () b b* ...)))])))
```

These naive transformations have two unfortunate consequences. First, the compiler cannot always "undo" the transformation since the resulting code overspecifies the intended behavior. That is, the compiler cannot tell whether the resulting code is intended to behave according to the letrec / letrec* semantics, or whether it really means binding some variables to a constant (#f) followed by assigning these variables to some other values. Second, some optimizing compilers of Scheme cannot handle assigned variables as efficiently as unassigned ones. In addition to inhibiting inlining, copy-propagation, constant folding, direct jumps to local procedures, and other optimizations, assigned variables often end up being *boxed* in heap-allocated locations [1], thus introduce additional runtime overhead for heap overflow checks when the bindings are introduced, additional memory traffic when the bindings are used, and possibly additional garbage collection overhead.

4. Waddell's letrec transformation

The Waddell, et al., algorithm for *fixing letrec* works by transforming a letrec expression into a set of let and fix binding forms. The fix binding form is a

restricted form of letrec in which all bound *var*s are unassigned and all *init*s are lambda expressions. The transformation partitions the letrec bindings into four sets:

1. [xu eu] ... *unreferenced*
2. [xs es] ... *simple*
3. [xl el] ... *lambda*
4. [xc ec] ... *complex*

Unreferenced bindings are those evaluated for effect only: their computed values are never used in the program. Bindings that are unreferenced in the program are eliminated as are assignments to these unreferenced bindings. The *simple* bindings are those satisfying the following criteria: (1) the *var* is unassigned, (2) the *init* is not a lambda expression and does not contain a reference to any *var* bound in the same letrec, and (3) evaluating its *init* expression cannot capture and invoke its continuation more than once. (In R⁵RS Scheme, an initialization expression is permitted to invoke its continuation more than once, but expressions that do are not considered *simple*.) The *lambda* bindings set contains the bindings where the *var* is unassigned and the *init* is a lambda expression. The *lambda* bindings are bound using fix in the output of the transformation. All other bindings are considered *complex*.

After partitioning the bindings, a letrec expression is transformed to:

```
(let ([xs es] ...         ; simple bindings
      [xc #f] ...)        ; complex bindings
  (fix ([xl el] ...)      ; lambda bindings
    eu ...                ; unreferenced
    (let ([xt ec] ...) ; complex values
      (set! xc xt) ...)
    body))
```

with the inner let expression omitted if no bindings are *complex*.

For letrec*, the bindings are similarly partitioned into *unreferenced*, *simple*, *lambda*, and *complex* bindings following the same criteria used for letrec expressions. The only difference in the transformation is that the *unreferenced init* expressions and the assignments to *complex* bindings must be interleaved in order to preserve the left-to-right evaluation order required for letrec*. Assuming no unreferenced bindings, a letrec* expression is transformed to:

```
(let ([xs es] ...         ; simple bindings
      [xc #f] ...)        ; complex bindings
  (fix ([xl el] ...)      ; lambda bindings
    (set! xc ec) ...      ; complex inits
    body))
```

Because some expressions considered *simple* for purposes of the letrec transformation might raise exceptions or perform side effects, the rules for simple expressions given above are too liberal to preserve the strict left-to-right evaluation order constraint for letrec* initialization expression. Thus, the Waddell, et al., transformation treats as *complex* any otherwise *simple* binding whose right-hand side is not effect free" [9]. For example, the expression (car x) is not considered *simple* for purposes of the letrec* transformation, since the car procedure may raise an exception if x is not a pair. In general, an expression cannot be considered effect free if it modifies state, exits from the program, raises an exception, or might not terminate. This has the unfortunate consequence of making many bindings *complex*, leading to all the drawbacks of assigned variables mentioned earlier. To be effective, *fixing letrec* should have a better story for handling such *init* expressions than to treat them as *complex*.

5. Why does it matter?

Consider the following simple program fragment:

```
(let ()
  (define q 8)
  (define f (lambda (x) (+ x q)))
  (define r (f q))
  (define s (+ r (f 2)))
  (define g (lambda () (+ r s)))
  (define t (g))
  t)
```

which should be understood in terms of a straightforward transliteration into letrec*, with the bindings appearing in the same order.

According to the Waddell, et al., partitioning algorithm, the binding for q is considered *simple*, the bindings for f and g are *lambda*, and the bindings for r, s, and t are *complex*. After *fixing letrec*, the code is transformed to:

```
(let ([q 8])
  (let ([r #f] [s #f] [t #f])
    (fix ([f (lambda (x) (+ x q))]
```

```
        [g (lambda () (+ r s))])
    (set! r (f q))
    (set! s (+ r (f 2)))
    (set! t (g))
    t)))
```

Because the variables q, f, and g are unassigned, they are straightforward targets for inlining, copy propagation, and other optimizations performed by a source optimizer such as the one described by Waddell and Dybvig [7, 8]. The assigned variables r, s, and t are not, so the resulting code after optimization might look like the following less-than-ideal code:

```
(let ([t #f] [s #f] [r #f])
  (set! r 16)
  (set! s (+ 10 r))
  (set! t (+ r s))
  t)
```

Compare this with the following program, which at the source level appears less efficient as it contains several more lambda expressions and procedure calls:

```
(let ()
  (define q (lambda () 8))
  (define f (lambda (x) (+ x (q))))
  (define r (lambda () (f (q))))
  (define s (lambda () (+ (r) (f 2))))
  (define g (lambda () (+ (r) (s))))
  (define t (lambda () (g)))
  (t))
```

It is discomforting and counterintuitive that the Waddell and Dybvig source optimizer boils this down to just the constant "42" while the simpler program shown at the beginning of this section does not, all because the letrec* transformation produces unnecessary assignments.

The goal of an efficient letrec and letrec* transformation is to reduce the number of emitted variable assignments (set!s) by turning as many of the bindings into simple let or fix bindings as possible. The original *fixing letrec* algorithm fails to do so, in essence, because it assumes that all of the bindings are possibly mutually dependent and thus must be grouped together in the output.

The new algorithm removes this restriction by grouping a set of bindings together only if they are mutually dependent. It can thus handle each nonrecursive complex binding in a group by itself without introducing an assignment. For example, it transforms the program given above into the following assignment-free program:

```
(let ([q 8])
  (fix ([f (lambda (x) (+ x q))])
    (let ([r (f q)])
      (let ([s (+ r (f 2))])
        (fix ([g (lambda () (+ r s))])
          (let ([t (g)])
            t))))))
```

6. Constraints, analysis, and encoding

Ideally, the transformation would place each letrec or letrec* binding in its own let or fix expression, each nested inside the previous ones, and so avoid all assignments. It cannot do so, however, due to semantic constraints that must be obeyed in order to handle the full generality of letrec and letrec*.

6.1 Constraints due to lexical scope

The lexical scope rule for let bindings requires that the right-hand-side expressions cannot reference any of the left-hand-side variables or any variables not bound in an outer scope. So, if x and y are two letrec bindings, we cannot place x in an outer let to y if x's *init* expression refers to y.

The lexical scope rule for fix bindings allows for mutual recursion among lambda expressions: the variables at the left-hand-side can appear at the right-hand-side lambda expressions as well as in the fix body. Consider the following example:

```
(let ()
  (define f (lambda () (even? 5)))
  (define even?
    (lambda (x)
      (or (zero? x) (odd? (- x 1)))))
  (define odd?
    (lambda (x) (not (even? x))))
  (define t (f))
  t)
```

Because even? and odd? are mutually recursive, they must be bound by the same fix. The procedure f references even?, but neither even? nor odd? references f, so f is placed in an inner fix. (Since it is nonrecursive, it could be bound by let instead, but we bind it using fix to facilitate a later, independent transformation that combines nested fix expressions to simplify the block

allocation and wiring together of procedures.) The variable binding `t` is placed inside the binding for `f` since it references `f`, and none of the other bindings reference it. The final product of the transformation shows the effect of scoping constraints on the output of the transformation.

```
(fix ([even? (lambda (x) --- odd? ---)]
      [odd? (lambda (x) --- even? ---)])
  (fix ([f (lambda () (even? 5))])
    (let ([t (f)])
      t)))
```

The transformation cannot transform all `letrec` and `letrec*` forms into arbitrarily nested `let` and `fix` bindings and thus avoid ever producing an assignment. For example, the program

```
(letrec ([x (list (lambda () x))]) x)
```

necessarily requires the introduction of an assignment to establish the cyclic relationship between the procedure and the pair. It is thus transformed into the following equivalent program

```
(let ([x #f])
  (set! x (list (lambda () x)))
  x)
```

Similarly, an assignment may be introduced for more than one recursively defined nonprocedure binding. In the following example, `x` and `y` are mutually recursive nonprocedure bindings, both requiring an assignment. The binding for `f` refers to both `x` and `y` but it can appear in an inner binding because neither `x` nor `y` refers to it.

```
(let ()
  (define x (list (lambda () y)))
  (define f (lambda () (cons x y)))
  (define y (list (lambda () x)))
  (define t (f))
  t)
=>
(let ([x #f] [y #f])
  (fix ([f (lambda () (cons x y))])
    (set! x (list (lambda () y)))
    (set! y (list (lambda () x)))
    (let ([t (f)])
      t)))
```

If the initialization expressions for two complex bindings are not mutually recursive, however, the assignments can be avoided. For example, assuming `x`, `f`, and `g` are defined outside of the following expression and both `y` and `z` are referenced within `body`:

```
(let ()
  (define y (f x))
  (define z (g x))
  body)
```

is transformed into the following.

```
(let ([y (f x)])
  (let ([z (g x)])
    body))
```

This transformation would also be valid for the equivalent `letrec` expression, with the R[6]RS semantics. It would not be valid for internal defines following the R[5]RS `letrec` semantics, however, since a continuation catpured by `f` might be invoked multiple times, each causing a new location for `z` to be created, possibly holding a different value each time.

6.2 Constraints due to evaluation order

For `letrec*` (but not for `letrec`), an additional constraint on the transformation is forced by the requirement that the *init* expressions be evaluated sequentially from left to right. Evaluating an *init* expression may, however, cause side effects. The `letrec*` transformation therefore *must* preserve the order of observable side effects in the *init* expressions as they appear in the input program.

Whether an *init* expression causes an observable side effect is undecidable in general, so the analysis to determine whether it does so must be conservative. Our current implementation considers an *init* expression to have a side effect if it contains a procedure call or a `set!` expression occurring outside of a `lambda` expression. This gives the implementation the freedom to reorder the simple bindings comprising of constants, variable references, primitive references, `lambda` expressions, and the combination of simple expressions to form certain primitive calls, `if`, `begin`, `let`, `letrec`, and `letrec*` expressions.

6.3 Viewing constraints as a dependency graph

The first part of our algorithm works by encoding each `letrec` and `letrec*` instance as a directed graph G

in which the nodes are the set of $vars$ and the arcs represent the dependencies between bindings.

The dependencies are derived from the lexical scope and (for `letrec*`) evaluation order constraints and control the overall structure of the result of the transformation. For example, the lexical scope rule dictates that at the output of the transformation, an *init* in an outer `let` or `fix` binding must not reference a variable placed in an inner binding. Or, stated differently, if x_i appears free in some $init_j$ expression, x_i has to be bound before (or at the same time) $init_j$ is evaluated but not later. Thus, the binding for x_j is said to depend on the binding for x_i as in the following definition:

Dependency graph for `letrec`:
Given a set of bindings $\{\langle x_i, init_i\rangle *\}$, the dependency graph is $G = \langle V, E\rangle$ where
$V = \{x_i*\}$ and
$E = \{(x_j, x_i) : x_i \in FV(init_j)\}$.

Constraints due to the specified evaluation order for `letrec*` are encoded in a similar fashion. If two *init* expressions might perform observable side effects, their order must be preserved in the output of the transformation. Thus, the edges of the dependency graph for `letrec*` must contain an arc $x_j \rightarrow x_i$ if $init_j$ must be evaluated after $init_i$.

Dependency graph for `letrec*`:
Augments the edges of the corresponding `letrec` graph with the set
$\{(x_j, x_i) :$ $init_j$ and $init_i$ are *complex* and $j > i\}$.

Dependencies are transitive, e.g., if $(x_i, x_j) \in E$ and $(x_j, x_k) \in E$, then $(x_i, x_k) \in E$ and thus need not be encoded explicitly. In fact, our implementation adds at most $N - 1$ order-of-evaluation edges for a `letrec*` containing N bindings and not the $O(N^2)$ edges required if all pairs of dependent bindings are connected.

The dependency graphs for the programs presented so far are shown in figure 1. It is instructive to compare each dependency graph with the corresponding code after transformation.

6.4 Strongly connected components

Once the dependencies among bindings are determined, the resulting graph is partitioned into strongly connected components (SCCs) using Tarjan's algorithm [6]. An SCC in G is the largest subgraph of G in which every node is reachable when starting from every other node in the SCC.

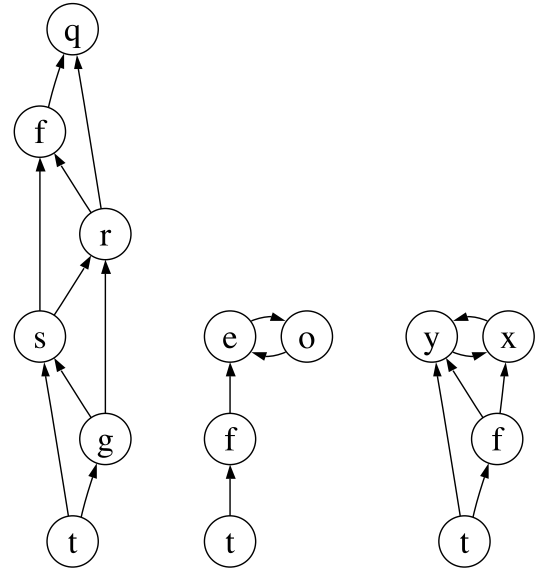


Figure 1. The dependency graphs for the first three source programs shown in Section 6. The left-most graph is for the program at the beginning of Section 5. The two shorter graphs are for 6.1 and 6.2. The edges in these graphs record the constraints due to lexical scope and evaluation order.

The Tarjan algorithm works by visiting all reachable nodes starting from some node x_s, in a depth-first-order traversal, ranking each node with its depth of traversal, and combining cycles as they are encountered.

In its simplest case, an SCC contains just a single node (which may or may not be pointing back to itself). If an SCC contains more than one node, these nodes are mutually dependent, either because they are mutually recursive or because one node references another that must follow it in the left-to-right evaluation order of `letrec*`.

The arcs of G not only determine the set of SCCs but also defines a partial order relation on the SCCs. Our implementation of Tarjan's algorithm returns an ordered list of SCCs such that SCC_i does not depend upon SCC_j if $i < j$, but SCC_j might depend on SCC_i.

7. Transformation based on SCCs

The `letrec` and `letrec*` transformation partitions the set of bindings into strongly connected components.

We only need to consider how to generate code for a single SCC. This is because the list of SCCs obtained from the Tarjan algorithm is ordered according to the required dependencies, so the bindings for each SCC need merely be nested within the bindings for previous SCCs.

Each SCC is handled in a manner similar to the way the Waddell, et al., transformation handles the entire set of `letrec` and `letrec*` bindings, except that we treat specially the case where an SCC contains only one binding.

Single bindings: Code generation for an SCC containing a single binding $\langle var, init \rangle$ is handled according to one of the following cases:

– If *init* is a `lambda`, and *var* is unassigned:
```
(fix ([var init]) rest)
```

– If $var \notin FV(init)$:
```
(let ([var init]) rest)
```

– Otherwise, we resort to assignment:
```
(let ([var #f])
  (set! var init)
  rest)
```

The *rest* code in the transformation denotes the code for subsequent SCCs and the transformed *body* expression.

Multiple bindings Code generation for an SCC containing multiple bindings is done by partitioning the bindings into two parts:

1. $\langle var_\lambda, init_\lambda \rangle$ if *init* is a `lambda` expression and *var* is unassigned.
2. $\langle var_c, init_c \rangle$ otherwise.

Tarjan's algorithm does not guarantee an order of returned elements for each SCC, so, for `letrec*`, the complex bindings need to be sorted according to their occurrence in the original `letrec*` form. The two partitions are used to produce the following code:

```
(let ([var_c #f] ...)
  (fix ([var_λ init_λ] ...)
    (set! var_c init_c) ...
    rest))
```

The graphs shown in Figure 1 tell the whole story. The first graph shows a long chain of dependencies, but all SCCs are of size 1. This is why the transformed code (shown at the end of Section 5) has a deeply nested `let` and `fix` forms, each binding a single variable. The second graph (the `even?/odd?` example) shows two mutually recursive bindings in one SCC (producing a `fix` binding in the output) and two singleton SCCs (producing two `let` bindings). The last graph also shows an SCC with two bindings, but this time, `x` and `y` cannot be bound with `fix`, and thus an assignment is needed.

8. Conclusions

The `letrec` and `letrec*` transformation algorithm described in this paper improves on the handling of `letrec` and `letrec*` by Waddell, et al. [9], by producing fewer assignments, thus reducing direct overhead from assignments as well as the indirect overhead from the inhibition of certain optimizations. Each binding that would be considered *simple* by the Waddell, et al., transformation ends up in its own SCC. Since simple bindings cannot reference any of the left-hand-side variables, they are handled by the second single-binding case above, i.e., they are bound by `let` expressions. Similarly, all *lambda* bindings end up bound by `fix` expressions. The difference between the Waddell, et al., transformation and ours is in the treatment of bindings the former considers *complex*. While the Waddell, et al., transformation always introduces assignments for *complex* bindings, our transformation avoids assignments for those that are nonrecursive and end up in their own SCC, which appears in our initial experiments to be by far the most common case. For `letrec*`, some apparently *simple* bindings must be considered complex, such as primitive calls that might raise exceptions, so this improvement is particularly important for R^{6}RS, which employs `letrec*` semantics for internal definitions.

We have implemented the original and new algorithms and wired them into the Ikarus [3] and Chez Scheme [2] compilers, with a parameter to select which algorithm to run. Using both systems, we compared the effectiveness of the two algorithms in eliminating complex bindings while bootstrapping the compiler and run-time libraries, which for both systems involves thousands of `letrec` or `letrec*` bindings. 10% of Ikarus's bindings and 7.2% of Chez Scheme's bindings are considered complex by the original algorithm versus only .6% and .1% for the new. Thus, in both systems, the complex bindings are a substantial fraction of

all letrec/letrec* bindings using the original algorithm, but an insignificant fraction using the new.

Although the bootstrapping times for Ikarus improve by just under 11% when switching from the original to new algorithms, the bootstrapping times for Chez Scheme do not improve significantly. This is likely due, in part, to the fact that most of the code that affects Chez Scheme's compile-time was written before letrec* was introduced and internal definitions were changed to use letrec* semantics. Furthermore, most of the code was developed with an even less effective algorithm for handling letrec than the Waddell, et al., algorithm, and this has caused the developers to shy away from potentially complex bindings in time-critical portions of the system.

Based partly on our experience, we believe that programmers will take advantage of the letrec* semantics for internal definitions, especially with the knowledge that letrec* can be implemented effectively, and that the improvement afforded by the new transformation will become increasingly more valuable over time. It will be interesting to test this hypothesis once a large corpus of programs written specifically for R^6RS becomes available.

Acknowledgments

We thank the reviewers for their helpful comments and suggestions.

Implementation

The transformation presented in this paper is implemented in Ikarus Scheme. As of revision 1825, the implementation is in the file ikarus.compiler.optimize-letrec.ss. The current source code can be viewed online at http://ikarus-scheme.org/optimize-letrec.html.

References

[1] R. Kent Dybvig. *Three Implementation Models for Scheme*. PhD thesis, University of North Carolina, Chapel Hill, April 1987.

[2] R. Kent Dybvig. *Chez Scheme Version 7 User's Guide*. Cadence Research Systems, 2005.

[3] Abdulaziz Ghuloum. *Ikarus Scheme User's Guide*, 2009.

[4] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised[5] report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[5] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten (editors). Revised[6] report on the algorithmic language Scheme. 2007.

[6] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[7] Oscar Waddell. *Extending the Scope of Syntactic Abstraction*. PhD thesis, Indiana University Computer Science Department, August 1999.

[8] Oscar Waddell and R. Kent Dybvig. Extending the scope of syntactic abstraction. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 203–213, New York, NY, 1999.

[9] Oscar Waddell, Dipanwita Sarkar, and R. Kent Dybvig. Fixing letrec: A faithful yet efficient implementation of Scheme's recursive binding construct. *Higher-Order and Symbolic Computation*, 18(3-4):299–326, 2005.

The Scribble Reader

An Alternative to S-expressions for Textual Content

Eli Barzilay
Northeastern University
eli@barzilay.org

Abstract

For decades, S-expressions have been one of the fundamental advantages of languages in the Lisp family — a major factor in shaping these languages as an ideal platform for symbolic computations, from macros and meta-programming to symbolic data exchange and much more. As convenient as this minimalist syntax has proven to be, it is unfitting for dealing with textual content. In this paper we describe the reader used by Scribble — the PLT Scheme documentation system. The reader implements a syntax that is easy to use, uniform, and it meshes well with the Scheme philosophy. The syntax makes "here-strings" and string interpolation easy, yet it is more powerful than a combination of the two.

1. Introduction

In Scheme, textual content comes in the form of plain old double-quoted strings, with simple backslash escapes. Dealing with rich textual content is possible, but highly inconvenient to the point of making such kind of programing nearly impractical. Writing a complete textual document in plain Scheme syntax is possibly more difficult than implementing a symbolic interpreter in Fortran. In contrast, practically all modern languages come with a variety of tools that promote textual content: multiple kinds of string quotations, "here-strings" (also called "here-documents"), and string interpolation syntax. In this regard, Scheme is severely lagging behind the times.

The convenience that comes with these tools is more than a mere technicality; it is important for enabling textual computing in much the same way that S-expressions, quotes, and quasiquotes in Scheme enable symbolic computing. Neither of these facilities is required, yet without them, textual manipulation and symbolic programming becomes a hair-pulling experience. Being limited to Scheme strings means that all text must be modified by escaping double quotes and backslashes (a major hassle for text that has Scheme code in it), and to mix text and code, we need to split the text into separate strings, then recombine them with `string-append` — making this quite similar in nature to an implementation of meta-programming when all you have is `make-symbol` and `make-pair`.

In PLT Scheme, we have designed a new concrete syntax to address the problem of textual content. This is implemented as a reader macro that is used as part of the Scribble documentation system[5]. The new syntax is similar in spirit to S-expressions, and indeed it is both elegant and useful in a similar way. At the conceptual level, the syntax builds on a similar uniformity to that of S-expressions — the result is even more convenient and general than popular textual facilities, specifically, both here-strings and string interpolation are achieved in a nearly trivial way. The new syntax has proven itself in a massive migration of thousands of documentation pages from a messy LaTeXbased system to Scribble, extending it with much more text, and the result is higher quality renderings, with much improved utility to end users.

An additional similarity with S-expressions is in the syntax's versatility and independent utility: the Scribble syntax is useful in many textual contexts, going well beyond "a documentation language"; parallel to the utility of S-expressions for many tasks that revolve around symbolic computations and more. It is essentially an alternative S-expression "skin" that can ease textual applications at the concrete syntax level while keeping the usual Scheme flexibility. As a matter of fact, this approach can be used to extend *any* language, it is the conceptual approach and the use of S-expressions that makes it particularly fitting for Scheme.

In short, the Scribble syntax, not only brings PLT Scheme up to speed with respect to textual programming: it provides it with the same edge that Scheme always had with respect to symbolic programming.

The Scribble syntax was not made up in a vacuum; various Scheme implementations have come up with a few solutions to varying degrees of completeness. Indeed, PLT Scheme itself has SCSH-style "here-strings"[10], two different preprocessor tools, a PLaneT library for string interpolation, and all of these come in addition to a decade-long sequence of various experiments with syntaxes in trying to find a good solution.

Many Scheme implementations provide a readtable-like facility for extending their concrete syntax, and can therefore implement the Scribble extensions conveniently and achieve the same benefits. Moreover, if adopted by more implementations, these benefits can carry to the Scheme community as a whole.

2. Scribble Syntax by Example

In this section we present the Scribble syntax through a not-so-short sequence of examples. As usual with readtable-based parsers, the syntax is hooked onto a "reader macro" character, which makes it an extension of the existing S-expression reader — and allowing a natural mix of Scheme code in both S-expression and Scribble syntax, as well as making the implementation relatively simple by localizing the required extension rather than requiring a completely new parser.

Proceedings of the 2009 Scheme and Functional Programming Workshop
California Polytechnic State University Technical Report CPSLO-CSC-09-03

2.1 Basic @-Expressions

Scribble forms are often called @-expressions or @-forms because they begin with a @ character. Embedded in Scheme code, a simple @-expression can look as follows:

```
(define (greetings) @list{Hello, world!})
```

Following the @, the reader proceeds to read `list` as a Scheme identifier, and any text that appears within the following curly braces gets parsed as a sequence of Scheme strings, and finally the identifier and the strings are wrapped in a list. We can use the fact that this is an extension of the Scheme reader, and inspect how it parses an @-form using `quote`[1]:

```
> '(define (greetings) @list{Hello, world!})
(define (greetings) (list "Hello, world!"))
```

The @ character was chosen by counting the frequency of the first character used for symbols in the complete PLT code-base, and choosing the least frequent one. Another (unexpected) advantage of @ is that it is illegal to start an identifier with it according to the R6RS[11] standard. The Scheme reader is extended by introducing @ as a "non-breaking" reader macro character, so less code may be broken when switching to it (i.e., a @ inside an identifier has no special meaning). In the rare case where @ *is* needed at the beginning of a symbol, PLT Scheme's escape conventions for symbols (also used by several other implementations) can be used — for example, `\@foo` and `|@foo|` are plain Scheme identifiers. This is a direct benefit of using a readtable extension: these escapes mean that the Scribble reader will never see these symbols.

Reading @-forms as plain S-expressions is crucial to the utility of the Scribble syntax. It allows us to leave the meaning of these expressions to the usual Scheme semantics — using any kind of bindings: existing or imported, procedures or macros.

```
> @display{Hello, world!}
Hello, world!
> (define (greet str) (printf "Hello, ~a!\n" str))
> @greet{world}
Hello, world!
```

The textual content of an @-expression is nearly free-form text. It can have newlines, double and single quotes, backslashes, etc. The S-expression that it parses to will have a number of strings: usually one string per line and one for each newline. The indentation of the whole form is ignored, but indentation *inside* the form is preserved and is parsed as a separate string of spaces.

```
> '@string-append{Backslashes escape quotes:
                  "x\"y"}
(string-append "Backslashes escape quotes:" "\n" "\"x\\\"y\"")
> '@string-append{1. First
                    - sub
                  2. Second}
(string-append "1. First" "\n" "   " "- sub" "\n" "2. Second")
```

This makes @-forms extremely convenient for common tasks where text contains code snippets. In almost all case, the author only needs to deal with forms (function applications, macros, or quoted lists) that have multiple strings as a "textual body".

```
> (define (show . strings)
    (for-each display strings))
> @show{(define (show . strings)
          (for-each display strings))}
(define (show . strings)
  (for-each display strings))
```

The reason for separating the text into separate strings for newlines and indentation is that in some cases it is useful to process the text based on them. We also use syntax properties (a PLT Scheme feature) that contain additional information, making it possible to know the precise original syntax. In some rather rare cases where this is needed, this information can be used by macros[2].

Several subtle yet important decisions are made here. Ignoring the indentation of an @-forms makes it possible to preserve the indentation structure of Scribble-extended Scheme code, a priceless feature for Schemers. In fact, the "overly verbatim" nature of Scheme strings and popular here-string syntaxes greatly reduces their popularity, and indeed, they are mostly used only in toplevel positions, and still, Schemers often prefer `string-append` to preserve the textual layout of their code. The importance of preserving this textual layout leads to another feature of the Scribble reader: a newline that begins the textual body or one that ends it are ignored *if* the body contains text. If the body contains no text, then it is assumed that the newlines are all intentional.

```
> @list{
    blah blah blah
  }
("blah blah blah")
> @list{
  }
("\n")
> @list{

    blah

  }
("\n" "blah" "\n")
```

This decision is a typical case that demonstrates an important principle of the Scribble syntax design: it should be convenient and natural to use for *humans authors* facing tasks that involve writing text in code. This stands in contrast to plain Scheme strings, where uniformity and terseness is strongly favored over convenience. (It is also interesting to compare this with textual quotations in modern languages.) Obviously, there is an important tradeoff here: uniformity and terseness are important for any feature of a programming language, even more so for its concrete syntax. As a result, designing the Scribble syntax has been a tedious experience of finding the golden line between uniformity and convenience[3].

The issue of indentation demonstrate this tension in two additional decisions. First, the indentation of a whole @-form is ignored, but what if after the opening brace there are some spaces and then text? In this case, the assumption is that the spaces are intentional, and they are not ignored. Second, when @-forms are used as markup language, a textual body might be *outdented* relative to the first line, to avoid disturbing the flow of text. A simple solution to both of these issues is to simply ignore any text that follows the opening brace when determining the indentation of the whole @-expression.

```
> @list{  One
          Two}
("  One" "\n" "  " "Two")
> @list{This sentence is split
  across two lines.}
("This sentence is split" "\n" "across two lines.")
```

In a similar way to its treatment of indentation, Scribble ignores spaces at the end of lines — except for spaces right before the closing brace, if there is text on that line.

```
> @list{  One
          Two  }
("  One" "\n" "  " "Two  ")
```

[1] To try these examples in a DrScheme or MzScheme REPL, enable the Scribble reader with `(require scribble/reader)` and `(use-at-readtable)`.

[2] A macro is required since the source and its properties do not exist at runtime.

[3] There are still a number of corner cases where it is not clear whether the Scribble syntax has followed the right choice.

2.2 Escapes and "Here Strings"

Clearly, the textual content is not completely free form: it is terminated by a closing brace. However, balanced braces are still allowed. This greatly reduces the need for escaping — most uses of a textual container that require braces will have balanced braces, so we favor not requiring escapes for this case over the alternative of always requiring them, or the asymmetric alternative of requiring them *only* for closing braces.

```
> @list{int add1(int i) {
          return i+1;
        }}
("int add1(int i) {" "\n" "  " "return i+1;" "\n" "}")
```

Typical cases that require an unbalanced single brace are programs that construct text programmatically, and for such uses we can still use conventional Scheme strings. In other words, we choose yet again the option that has the greater advantage for most texts, over the more uniform but less convenient alternatives.

Still, we wish for the syntax to be complete and to accommodate an unbalanced brace in *some* way when it is needed. A good rule-of-thumb to see how such a syntax scale is to observe how it handle reflective texts, for example, writing texts *about* the system itself[4].

A common sequence of events at this point is (a) decide to use an escape character, (b) go with the familiar backslash, (c) require backslashes to be escaped too, (d) end up with yet another level of backslash-escapes. An alternative that we considered is backslashes that escape only braces, and otherwise are part of the text: a sequence of n backslashes followed by a brace would stand for a text that consists of $n - 1$ backslashes and the brace, for example "\{" quotes "{", and "\\\{" quotes "\\{". This non-uniform rule comes at a cost: it is confusing in its non-uniform behavior, and it is impossible to have a backslash appear before an *unescaped* brace (as the last character of a textual body). Regardless of the choice, an escape character is a poor choice for reflective texts (failing our rule-of-thumb), and was therefore rejected.

Instead of an escape mechanism, we have turned to a more convenient approach that fits cases that call for extra "freedom" in the text part of an `@`-expression — a set of alternative delimiters can be used, making braces lose their special role. Using a different shape of parentheses, as done in some languages that have both single- and double-quoted strings, only shifts the problem elsewhere, forcing the author to be aware of the current delimiters. Instead, we have settled on alternative delimiters that are longer than single braces: we still use braces (so delimiters still look similar, making them easy to read and to remember), but the open/close delimiters we use `|{` and `}|`.

```
> @list|{ }-{ }|
(" }-{ ")
```

The vertical bars are not an arbitrary choice: they have a similar role in delimiting symbols in the PLT Scheme reader, so playing a similar role here makes this choice more memorable.

Going back to our rule-of-thumb, it becomes apparent that just a single alternative is insufficient, for example when this alternative is itself documented. The delimiter syntax is therefore further generalized to arbitrary user-specified delimiters in a way that is analogous to here-strings: the opening delimiter can have any sequence of punctuation characters (excluding curly braces) between the vertical bar and the brace, making the expected closing delimiter to hold the same sequence (in reversed order, and with reversed parentheses)[5]. Using such delimiters makes it possible to have the text arbitrarily free-form.

```
> @show|--<<{Use @foo|{...}| to type free braces}>>--|
Use @foo|{...}| to type free braces
```

For extreme cases, the Scribble reader is generalized on the macro character to use. For example, the textual domain might use `@` excessively (e.g., documenting Scribble), or you might want to intentionally make a syntax that looks like an existing language (e.g., create a language that is close in its look to LaTeX)[6]. Yet even with our extensive experience of (re)writing the PLT Scheme documentation in Scribble, there was no need for this feature. It is therefore only available through procedures in the reader API that can construct such custom readers.

Finally, the Scribble syntax has yet another way to locally escape text using the usual Scheme string syntax. This is done by following a `@` with a Scheme string. This can be handy in some difficult cases, where it is used for very short strings, or when authors prefer it over using the alternative delimiter syntax.

```
> @show{An open brace: #\@"{".}
An open brace: #\{.
```

As we shall see next, this is actually a limited (and slightly modified) example of nested Scribble forms.

2.3 Nested @-Forms

So far, the Scribble syntax that we have covered serves as only a convenient alternative to quoting and to "here-strings". Specifically, we did not address the problem of combining text and code, commonly addressed via "string interpolation". Schemers quickly recognize string interpolation as similar in nature to quasi-quoting. In fact, such a facility is sometimes named "quasi-strings", and implemented as a string-like extension using similar characters.

Our syntax implements a conceptually different solution — a general approach that is powerful enough to support string interpolation as a trivial byproduct. At its core, the basic property of the Scribble syntax is that `@`-forms have the *same meaning* whether they appear in Scheme code or nested in other `@`-forms. The direct implication of this principle is simple: if the textual body of an `@`-expression has a nested expression, then the nested one is read recursively, and the resulting expression contains the nested form among the strings of its textual body, which means that the textual body of an `@`-form is no longer just a sequence of strings.

```
> '@foo{abc @bar{ijk}
        xyz}
(foo "abc " (bar "ijk") "\n" "xyz")
```

This approach is the primary key to the convenience and flexibility of the Scribble syntax. The first thing to note is a convenience point: `@`-expressions preserve their meaning when they move into and out of other `@`-expressions. But this is important at a much more fundamental level: it means that `@`-expressions follow the same rules as S-expressions — an occurrence of `@foo{...}` denotes an application of the procedure bound to `foo` (or a `foo` macro form), regardless of where it appears in the code (barring quoted contexts). This is what makes the Scribble syntax be an *alternative* to S-expressions, one that is well suited for textual content — not just a mere combination of string quotation and unquotation.

```
> (define mytext @list{A @vector{B} C})
> mytext
("A " #("B") " C")
```

If this is compared to a conventional string interpolation (using a fictitious syntax),

[4] For example, this paper and the Scribble documentation pages are both written using the Scribble reader.

[5] Alphanumeric characters are forbidden here to avoid mistakes; curly braces are forbidden to avoid ambiguity; and the reason for reversing parentheses is to make it more convenient to edit the text and making it fit well inside the `|{...}|` delimiters

[6] To experiment with this, the source of this paper uses a Scribble reader customized to use a backslash, making the source mostly compatible with LaTeX.

```
(define mytext (list "A $(vector "B") C"))
```

several differences become apparent:

- In such mechanisms, the interpolated expressions are expected to evaluate to a string, or are coerced to a string value, to keep the value a string. In contrast, nested `@`-forms can be any expression and have any kind value.
- This is an indication of a more important difference: in the string interpolation example, there is no single interpolated *expression* — only an interpolated *value*. To put this in rough Scheme terms, to get an interpolated expression, we would need to somehow "expand the string" into a sequence of expressions that are spliced into the expression that contains the string — and make sure that such strings always appear in expressions that expect one or more string values. (This kind of expansion requires a global transformation, or a kind of a macro expander that can expand string literals and deals with transformers that return a to-be-spliced value.)
- Finally, note that the vector expression needs to be escaped, while the list expression must not be escaped. We therefore need to be aware of the lexical surrounding of an expression to know if it should be escaped or not — which can be difficult with bigger pieces of code, and it therefore encourages restricting interpolated expressions to relatively small bits. This is similar in nature to quasi-quoting, where we have a "textual context" for text, and we can escape out of it back to Scheme code. This is in sharp contrast to Scribble *expressions* — which are an extension of the language rather than just a new kind of context.

Going back to our reader, nested `@`-forms make perfect sense: they can be conveniently used with any kind of code, and any kind of context. We only need to make sure that every `@`-expression that we use has a proper binding, and that these bindings expect a variable number of "textual body" arguments, mostly strings. For example, a simple adjustment to the previous definition of `show` makes the following example work as expected:

```
> (define (show . text)
    (let loop ([x text])
      (if (list? x) (for-each loop x) (display x))))
> (define (greet str) (list "Hello, " str "!"))
> @show{Once again: @greet{world}}
Once again: Hello, world!
```

As the `@` character turns into a special character in a textual body, it is also subjected to the alternative delimiter syntax as the open and end braces. Specifically, when `|{...}|` are used for a textual body, then nested expressions should similarly begin with a `|@`, and the same goes for the variants with extra punctuations in the delimiter.

```
> @list|{123 @vector{456} 789}|
("123 @vector{456} 789")
> @list|{123 |@vector{456} 789}|
("123 " #("456") " 789")
> @show|{@greet{world} --> |@greet{world}}|
@greet{world} --> Hello, world!
> @show|--{@greet|{world}| --> |--@greet|{world}|}--|
@greet|{world}| --> Hello, world!
```

Note that the nested form determines its own variant of delimiters: the `greet` expressions above do not have to use the alternative delimiters.

At this point the Scribble syntax combines convenient facility for quoting free-form text, with an extension that addresses the same kind of problems as string interpolation devices. But it is not a complete replacement for string interpolation — yet.

2.4 Expression Escapes, String Interpolation

With Scheme S-expressions, parenthesized expressions denote procedure applications (we ignore macros for now), with the head of the expression denoting the procedure to apply. Of course, identifiers can appear in places other than the head of a parenthesized expression, making it a simple reference to a value. The Scribble syntax builds on S-expressions by reading an `@`-form as an S-expression, where the identifier after the `@` stands in the "head position" of the expression, and the following curly braces denote the (textual body) arguments. It is therefore intuitive to make the Scribble syntax correspond to S-expressions: if `@`$\langle id \rangle$ is not followed by a curly-braced textual body, then the result of reading the form is just $\langle id \rangle$.

```
> '@foo{x @y z}
(foo "x " y " z")
> (define name "earth")
> @show{Using "@name": @greet{@name}}
Using "earth": Hello, earth!
```

Furthermore, there is no reason to restrict the head part of an `@`-expression to identifiers only — it can just as well have *any* S-expression. With a textual body, this is equivalent to an application expression that has an application expression in the function position.

```
> '@(foo){bar}
((foo) "bar")
> @show{I repeat: @(compose greet string-upcase){@name}}
I repeat: Hello, EARTH!
```

Without a textual body, we get a generic escape for arbitrary Scheme expressions.

```
> @show{I repeat: @(greet (string-upcase name))}
I repeat: Hello, EARTH!
> '@show{1+2 = @(+ 1 2)}
(show "1+2 = " (+ 1 2))
> @show{1+2 = @(+ 1 2)}
1+2 = 3
```

An interesting result of making the syntax uniform via uses of the Scheme reader is that the head part of an `@`-expression can itself be an `@`-expression. With an input code of `@@foo{bar}{baz}`, the Scribble reader gets invoked by the first `@`, it will then read the head part for the `@`-form which is `@foo{bar}` (resulting in `(foo "bar")`), and finally it reads the textual body and constructs the resulting expression, `((foo "bar") "baz")`. In some cases, this can be useful for procedures that expect more than a single textual body: write the appropriate curried definition where each step consumes a "textual body" as a rest argument, and to use it, specify the right number of `@`s at the beginning of the `@`-form.

```
> (define ((features . pros) . cons)
    @list{Pros: @|pros|.
          Cons: @|cons|.})
> @show{@@features{good}{bad, ugly}}
Pros: good.
Cons: bad, ugly.
```

There is a small exception to such Scheme escapes: when the escaped expression is a literal Scheme string, the string is combined with the surrounding text rather than become a separate expression. As discussed above, this allows using the same syntax for escaping arbitrary text using familiar Scheme quotes.

```
> @list{1 @2 @"3" 4}
("1 " 2 " 3 4")
```

This is another case where Scribble favors utility over uniformity. In general, `@`-forms are not used when precise control over the number of arguments is needed — instead, it is used with a sequence of "textual body arguments", with functions (or macros) where `(f "x" "yz")`, `(f "xy" "z")`, `(f "x" "y" "z")`, and `(f "xyz")` are all equivalent. Given this, the escaped string exception is both

harmless and redundant; however, it is useful in some cases where a single string argument is expected, and in cases where it is important that a chunk of text be kept as a single string (e.g., when each string is later post-processed).

Using the Scribble reader's expression escapes, we can now get string interpolation as a relatively boring special case of using `@string-append{...}` and @-expression escapes that evaluate to plain strings. Combining this further with configurable delimiters, we get the functionality of here-strings too.

```
> (define qs string-append) ; qs is for quasi-string
> @qs{... name=@name ...}
"... name=earth ..."
> @qs|{...} @name=|@name {...}|
"...} @name=earth {..."
```

Since the Scheme reader is used to read escaped expressions, we need some way to separate identifiers from adjacent text that will otherwise be read as part of the identifier, or from braces that should not be part of the @-form. One solution that a few Scheme implementations of a textual syntax choose is to avoid the problem: require using an escaped `begin` expression when delimiting identifiers. In Scribble, we have avoided such solutions for a few reasons: (1) it is inconvenient, especially when it breaks the textual flow with an otherwise redundant identifier; (2) a `(begin id)` expression is not always equivalent to `id` (e.g., in a quoted list); and (3) given the uniform Scribble syntax, it makes more sense to have a way to avoid a following brace from becoming part of an @-expression, rather than leave it to the less convenient escapes.

The Scribble solution for this is to use `|...|` delimiters. The vertical bar character was chosen partly because it is also used by the alternative delimiter syntax (so the motivation is to keep a small set of "special" characters), and partly because in PLT Scheme, `|id|` is read as the `id` identifier.

```
> @show{Hello, @name.}
reference to undefined identifier: name.
> @show{Hello, @|name|.}
Hello, earth.
> @show{Hello, @|name|{}.}
Hello, earth{}.
```

Note, however, that the two uses are different: if the vertical bars were left for the Scheme reader, the above examples would not have worked (the second would still read the period as part of the identifier, and the third would still use the braces as the form's textual body). This is not a problem in practice: @-forms are for human-authored texts, so unconventional identifiers are not used; even if there is a case where such an identifier is needed, it is possible to use backslash escapes as usual in the PLT reader.

2.5 Further Extensions

2.5.1 Scheme-Mode Arguments

While we have a convenient way for specifying textual @-forms and Scheme S-expressions in both code and text contexts, there are cases where it is convenient to have a form with both S-expression subforms and a textual body. Such a mixture of the two kinds of subforms is particularly useful in a documentation system, where many typesetting procedures consume text (as a 'rest' argument) as well as keyword arguments for various customization options, but also for functions that expect a few non-text argument before the textual body. Yet, using the Scribble syntax that we have seen so far make this inconvenient and error prone.

```
> (define (shout #:level n . text)
    (list text (make-list n "!")))
> @show{@shout{@|#:level|@|3|Greetings}}
Greetings!!!
```

The way that this is addressed in Scribble is by introducing another part to @-forms where additional "Scheme-mode" arguments can appear. This part is specified using square bracket delimiters.

```
> @show{@shout[#:level 3]{Greetings}}
Greetings!!!
```

The textual body part of an @-form is still optional — making it is possible to specify only the Scheme-mode arguments with no textual body at all. Such @-expressions are read as a simple parenthesized list with the head followed by the arguments, making it an alternative form for plain S-expressions.

```
> @list{@list[1 2 3] @(list 1 2 3)}
((1 2 3) " " (1 2 3))
```

This apparent redundancy is a by-product of making the Scribble syntax uniform. While both of the @-expression and the escaped S-expression read as exactly the same code, we use a convention where the first form is for expressions that produce text, and the second for other expressions. This makes more sense in places where the default reader mode for a whole file is the Scribble reader's "text mode", which is used, for example, in the PLT documentation sources. In these files, a function like `itemize` that expects items as arguments is written using the first form, while `require` and `define` expressions are written using the second.

```
#lang scribble/manual
@(require some-module)
...text...
@itemize[@item{...text...}
         @item{...text...}]
...text...
```

It is interesting to note that initially, we intended to use Scheme-mode part of @-forms only for keyword arguments, and used `@itemize{...}`, which means that the `itemize` function needs to ignore all-whitespace string arguments (and throw an error for other string arguments). Only later we 'discovered' that the square-bracket form makes more sense.

2.5.2 Headless Expressions

The full syntax of an @-form is therefore an @, a Scheme-syntax form head, a sequence of Scheme-syntax arguments in square brackets, and a sequence of text-mode arguments delimited by square braces. Only *one* of these is required — if only the head is included, we get an expression escape; and if the head is not included, then the Scribble reader will read it as a parenthesized expression with only the Scheme-mode and/or textual-mode arguments. This is mostly useful inside a Scheme quote:

```
> '@{Hello,
     world!}
("Hello," "\n" "world!")
> '@{Hello @{world}!}
("Hello " ("world") "!")
> `@{Hello @|,name|!}
("Hello " "earth" "!")
```

2.5.3 Dealing with Scheme Punctuations

There is a minor problem that is related to Scheme quasiquotes. Say that we want to have a quasiquoted list, and use unquoted @-forms in this list. The @ character in `,@foo` is going to be read as the short notation for `unquote-splicing` — a problem that appears in a few places in the PLT documentation sources[7]. An ugly hack around this problem is to add a space after the comma.

```
> `(html ,@string-titlecase{hello world!})
unquote-splicing: expected argument of type ⟨proper list⟩; given #⟨procedure:string-titlecase⟩
> `(html , @string-titlecase{hello world!})
(html "Hello World!")
```

[7] This is similar to one reason that identifiers are not allowed to begin with a @ in standard Scheme, as it makes `,@foo` ambiguous.

A related problem occurs when we try to unquote `@`-forms: if we follow the rules, we need `` `@{...@, @f{}...} `` — but Schemers want their unquotes short and convenient (e.g., reader shorthands).

Scribble solves both problems by "pulling out" any of the shorthand punctuations (`` ' ` , ,@ #' #` #, #,@ ``) outside of an `@`-form's head when they are found there, and wraps them around the whole expression. This solves both problems conveniently:

```
> `(html @,string-titlecase{hello world!})
(html "Hello World!")
> `@html{... @,string-titlecase{hello world!} ...}
(html "... " "Hello World!" " ...")
```

2.5.4 Extending Scheme Escapes

Since we have a specific syntax for specifying Scheme expression escapes (`@|...|`), it is natural to generalize it: instead of a single Scheme expression, it can hold more expressions, or none at all. When more than one expressions are used, they are all spliced into the containing `@`-form.

```
> @list{foo @|(+ 1 2) (* 3 4)| bar}
("foo " 3 12 " bar")
```

Note that this means that it possible to use this for keyword arguments, e.g., `@para{@|#:style 'small|blah blah}` — but this is inconvenient: it is heavy on punctuation (therefore hard to remember and easy to mistakes), and it is unnatural in the sense that the Scheme-mode part is nested inside the textual part. The square-bracket syntax is vastly easier to remember and simpler to use.

A Scheme escape with no expressions serves a special purpose as a "separator token" that can be used to split some text into two strings. This is rarely used since, as mentioned above, Scribble code tend to not rely on a particular separation of the textual-body arguments. A slightly more useful use of zero-expression escapes is forcing the reader to include an initial or a final newline, more spaces at the beginning or at the end of a line.

```
> @list{
    foo @|| bar
    }
("foo " " bar")
> @list{@||
    @|| foo @||
    @||}
("\n" " foo " "\n")
```

2.5.5 Comments

Finally, the Scribble reader has syntax for `#||#`-like balanced comments and `;`-like line comments. Note that we cannot use Scheme comments in escape expressions; it seems that an escape expression can be used with a balanced Scheme comment (making it an empty escape expression): `@|#|...|#|`. Ignoring the hieroglyphic look of this construct, the problem is that the commented portion is a comment in Scheme syntax. For example, if the commented part contains `|#`, then the comment will terminate prematurely. A proper way to achieve balanced comments without an extension to the Scribble syntax is therefore even more hieroglyphic: `@|#;@{...}|`. Line comments are impossible, since we will still need to insert something on the following line.

A balanced comment in Scribble is written as `@;{...}`. The body is still parsed in text-mode, so it can hold balanced braces, and it can use the alternative delimiter syntax.

```
> @list{foo@;{commented
             text}bar}
("foobar")
> @list{foo@;{...{}...}bar}
("foobar")
> @list{foo@;|{...}...}|bar}
("foobar")
```

A line comment starts with a `@;` (and followed by a character other than `;`), and it ends at the first non-whitespace character in the following line. This kind of comment can be useful in the same way as LaTeXcomments.

```
> @list{Some text, @; a comment
        more text.}
("Some text, more text.")
```

There is no syntax for an "expression comment", because defining an "expression" in a textual context is more difficult, and also because we had no need for this so far.

3. Text-Mode Source Code

While the Scribble extension is extremely useful, there are cases where we want to change our "perspective". Instead of thinking about source code as a Scheme file that contains some textual data, we wish to view it as a text file that contains some embedded Scheme code.

There are a number of applications where this shift in view is desirable. The most obvious case in PLT Scheme, and in fact one of the major motivations for the Scribble syntax, is the Scribble documentation system itself. Files are written in one of the Scribble languages (e.g., `#lang scribble/manual`), where all text in the source (except for the `#lang` line itself) is read as strings by default, and rendered as text in the target manual. The exception to this is `@`-forms (including `@` escape expressions) which are read as usual in the Scribble syntax. For example, the source of the Scribble manual starts with:

```
#lang scribble/manual
@(require scribble/bnf "utils.ss")
@title{@bold{Scribble}: PLT Documentation Tool}
@author["Matthew Flatt" "Eli Barzilay"]
Scribble is a collection of tools for creating prose
...
```

Unfortunately, implementing a reader for such languages cannot be done with a readtable as it is no longer an extension of the Scheme reader. Doing so will require overriding *all* characters (including Unicode characters). But there is no need for a separate reader implementation: at the conceptual level, the syntax of these files is as if the source is all contained in the textual body part of an `@`-form that surrounds the whole contents. This description provides a concrete hint for implementing such a reader — the only thing that we need is to invoke the part of the Scribble reader that parses a textual body. Indeed, the Scribble reader's API makes its textual body parser available as an "inside" kind of reader function, and this function can be used as a module reader (via the `#lang` mechanism) to parse the source. When invoked this way, the toplevel textual parser is only different in that it is expecting an end-of-file to end the text, rather than a `}` closing delimiter.

The result of this inside reader has a different type from the normal Scheme reader (`read` and `read-syntax` in PLT Scheme) — instead of returning a single syntax value, it returns a list of such values. Most items in this list are strings and the rest are `@`-forms. The list then becomes the body expressions in the module that is constructed by the `#lang`-specified language. The textual languages then use a special macro that can easily change the semantics of the module's toplevel expressions — `#%module-begin`; this macro is essentially wrapped around the module's body, and therefore it can rewrite these toplevel expressions.

3.1 Specific Use Cases

The "inside reader" feature is a powerful tool for creating textual languages. In the PLT codebase, it is used in a few places in addition to the documentation language. Each of these languages

```
#lang scribble/text
@(require scheme/list scheme/string)
@(define map/nl (compose add-newlines map))
@(define (itemize #:bullet [b "*"] . items)
   (map/nl (lambda (item) @list{@b @item})
           items))
@(define (pseudo-loop statements)
   @list{begin
           @; mix braces and begin/end: the joy of pseudo
           while (true@;{use NIL for dramatic effect}) {
             @(map/nl (lambda (s)
                        (let* ([s (string-append* s)]
                               [s (string-downcase s)]
                               [s (regexp-replace*
                                   #px"\\s+" s "_")])
                          @list{@s();}))
                      statements)
           }
         end})
@(define (both . items)
   @itemize[@list{In text:
                  @(apply itemize #:bullet "-" items)}
            @list{And repeating in a pseudo-code:
                  @pseudo-loop[items]}
            ])
@(define summary
   @list{If that's not enough,
         I don't know what is.})
Todo:
@both[@list{Hack some}
      @list{Hack more}
      @list{Sleep some}
      @list{Hack some
            more}]
@summary
```

```
Todo:
* In text:
  - Hack some
  - Hack more
  - Sleep some
  - Hack some
    more
* And repeating in a pseudo-code:
  begin
    while (true) {
      hack_some();
      hack_more();
      sleep_some();
      hack_some_more();
    }
  end
If that's not enough,
I don't know what is.
```

Figure 1. Preprocessor example

has a specific twist on the concept of text files with embedded Scheme code.

- In the documentation languages, the (mostly textual) expressions are all collected into an implicit global definition that is made available for later rendering,

  ```
  (define doc (list ...textual-contents...))
  (provide doc)
  ```

  except for definition expressions and `require` declarations that are kept at the top-level.

  As usual, the contents of the `doc` definition expression is made of strings and `@`-forms, which evaluate to a hierarchy of structs that the documentation system uses to represent the text, and the Scribble renderers can translate the resulting value to HTML, LaTeX, or text. The documentation languages come with bindings for typesetting markup, facilities to do LaTeX-like processing (grouping parts, splitting to paragraphs, and shorthands like `` and --). Additional libraries provide manual-specific bindings (e.g., forms for documenting procedures and for producing examples with automatic sandbox evaluation to show the results of these examples); bindings for articles, and more.
- Another case where the textual file reader is used is the a preprocessor language, `#lang scribble/text`. In this language toplevel expressions are displayed one at a time on the standard output, using a procedure that is similar to the `show` definition in the above examples. In fact, this article is written using this language, where the source can be programmable in Scheme, an obvious improvement over the underlying LaTeX. (In addition, the reader uses `\` as the character for `@`-forms, making it a hybrid language, where both LaTeXand Scheme can be used to define new commands.)

  The actual procedure that displays the text has the added capability of handling indentation and more[8], making it possible to tackle difficult tasks where whitespace matters. For example, this language is used as a preprocessor for the PLT foreign interface, where it needs to gracefully handle oddities such as the requirement that C preprocessor (CPP) directives start at the beginning of a line.

  (Note that our preprocessor cannot *replace* the C preprocessor. This will require implementing C-specific functionality such us finding include files, and knowing which CPP symbols are defined by the local C compiler. In other words, CPP is not just a preprocessor tool — it also serves as a an extension language of the C compiler, and indeed it is implemented as part of the compiler.)

  The Scribble preprocessor language can also be used from Scheme files (extended with the `@`-form syntax) — we use this approach in the source code of the `plt-scheme.org` web pages.
- In addition to these, the PLT web server implements template-based servlets using this facility: a servlet "includes" a template file using `include/reader`, which injects the textual content into a lexical scope in the servlet — for example, one that binds identifiers that are used in the template. The resulting language is similar in nature to existing template systems like the Cheetah template engine for Python[12] — where template files do not even have a `#lang` line.

The Scribble system is still quite young, and we expect to have additional uses for textual languages in the future, in addition to using it in extended-Scheme-syntax files. For example, most uses of the slideshow[3] language still use strings in the source code; switching to `@`-expressions can make writing slides considerably easier, and a textual language might further improve it. Also, we consider implementing some form of a wiki, where `@`-forms (and Scheme) are used as for convenient markup.

To get a rough feeling of how working with Scribble in a textual language looks like, see Figure 1 (real code tend to have much more textual content, of course).

4. Syntax Design, DSLs, and Why Macros are So Great

Scribble is essentially the last version in a decade of various experiments with different approaches to combining textual content and Scheme code. Two older systems are still part of the PLT distribution: `mzpp` is a template based preprocessor, which allows interleaving of text and Scheme code in a conventional way; `mztext` is a preprocessor that is similar in nature to TeX— when `@` is found in the input stream, a function name is read and applied on the stream, allowing it to parse arbitrary amount of text in arbitrary ways (of-

[8] This requires a PLT-specific feature: syntax values contain the position of the expression in the source.

ten a {...} piece of text), and return a modified stream that can contain new tokens.

Our experience of the development and implementation of the Scribble syntax demonstrates that extending a language at the concrete syntax level is *hard*. The end result seems sensible now, but the road that leads up to it is paved with subtle decisions. A few examples:

- How do we deal with quoting delimiters? As discussed above, there are several options with different trade-offs.
- Should we stick to the Syntax of scheme identifiers even though it sometimes require extra quoting? Maybe change it to break on periods, colons, etc?
- What is the best way to solve the possible problem with `,@`?
- Even the seemingly minor issue of how whitespaces are handled can be important. For example, if a space is either allowed or forbidden before the braced textual body (either parsing `@foo {x}` as a single @-expression or as `foo` followed by `" x"`), we may run into confusing mistakes; which choice is more consistent and/or natural? (The Scribble chooses the latter.) Perhaps such occurrences should just lead to a read error?
- Another question is how should the concrete syntax translate to S-expressions. An earlier implementation used a `dispatch` form, with subforms for the head, the Scheme sub-expressions, and the textual body. The idea was that it would be convenient to have a central point of control for assigning meaning to Scribble forms, by importing or defining `dispatch` — but this did not work out well. Having a central point was not useful (the only `dispatch` definition that was used expanded to an application form), and is better left for the underlying language (in this case, to Scheme macros and/or PLT's `#%app` form); meanwhile, the `dispatch` symbol would show up in quoted forms, making a common Scheme idiom (quoting and quasiquoting) less useful.

  This particular issue has lead to the "principle of least surprise", and a decision that the Scribble reader should not inject any new identifiers into the input that were not originally there. In a sense, this is the same confusion Scheme newbies run into with the `quote` character shorthand, e.g., `(define 'x 4)`.

There is an important lesson about (Scheme) macros and language design here. If the only tool you have to extend your language is a concrete syntax extension, then the difficulty involved in that considerably raises the bar for implementing such extensions, and at the same time extension code is more fragile. At the S-expression level, Scheme gets a clear win by separating the concrete level parser from syntactic extensions, which means that there is only one place to worry about the concrete syntax. Similarly, adding hygiene continues this trend of going higher than the concrete text of the code, as lexical scope is also separated into an independent lower level.

Considering concrete syntax in this light, the utility of @-expressions seems even greater. We get to keep the advantage of a separate layer to do the concrete parsing, while making more "text-oriented" information available at the higher level (i.e., in user code). With this, the Scribble reader helps in providing an additional bridge to the textual level of the source code. But this should not come as a surprise, as strings have always been this kind of flexible data containers. (Perhaps too flexible, as seen in languages like Perl and TCL where strings are used for arbitrary semi-structured data, similarly to S-expression abuse in Scheme and Lisp.)

This additional utility of @-expressions can be demonstrated by a simple use of here-strings, to specify code in a DSL. For example, a Scheme interface to generate equation images specifies shell commands in one string, and the LaTeX equation in another string.

```
(define commands
 @string-append{
   pdflatex x.tex
   convert -density 96x96 x.pdf -trim +repage x.png})
(define (latex . body) ...)
...
@latex{\sum_{i=0}^{\infty}\lambda_i}
```

An obvious extension to this use is to add interpolation with escape expressions. Taking it further, we can gradually choose textual constructs to abstract over in the foreign syntax using Scheme functions to generate thos syntax, while keeping the text-friendly property of our source code. For example, the preprocessed C code of our foreign interface has a `cdefine` function

```
@cdefine[ffi-lib 1 2]{
  ...C code...
}
```

that generates the boilerplate C function header, adds a CPP `#define` before the function for error messages and an `#undef` after it, and finally registers the function's name and arity to be used later in the C initialization code to create the proper binding.

5. PLT Specifics

Generally speaking, the Scribble reader does not require any features specific to PLT Scheme. In fact, we *hope* that other implementors will consider doing so, which will provide their implementations with the same benefits, as well as benefit the whole Scheme community.

There are, however, a number of specific PLT features that are worth mentioning in this context.

- The syntax objects that are used in PLT Scheme contain source location information. Making the Scribble reader use and record locations is an important feature: parsing errors are properly reported and easy to find, and even in the case of syntax errors and runtime errors, we know where the error is (e.g., with highlights in DrScheme) without resorting to dumping the S-expression that were read for manual inspection. In other words, it makes the Scribble syntax be a real part of the language, rather than a loose add-on.
- The Scribble reader records information about the original form in the parsed syntax values using syntax properties. For example, we can distinguish syntax that was written as an S-expression from one that was written as an @-expression, and we can tell which of its subforms were specified as part of a textual body. This feature can be useful in some rare cases where we want the choice of concrete form to affect the meaning of the code.
- The implementation of keyword arguments in PLT Scheme[4] allows keywords to come before other arguments, which is convenient for use in textual forms, where the customization options are better kept next to the function name. This is not specific to the Scribble syntax — it fits well with any similar markup language (e.g., XML attributes).
- In PLT, a `#lang` line at the top of a file is used to specify the language for the file. This specification works by choosing a reader that will wrap the code in a `module` form, which is how both the syntax and the semantics of the language are determined. As mentioned above, there are several languages that use the "inside reader" — for example, Scribble documentation files start with `#lang scribble/manual` and preprocessor files start with `#lang scribble/text`. In addition, there is a 'special' `at-exp` language that is used as a prefix for other languages, for example: `#lang at-exp scheme`. The `at-exp` language will delegate to the `scheme` reader, but will mix-in

the Scribble @-expression reader as an extension. This makes it easy to enrich your language with @-forms. In both case, the main benefit of #lang is localizing the reader to a single file, making it possible to use different concrete syntaxes for different source files, without a damaging global effect.

- Some of the special syntax identifiers that are used in PLT Scheme – like #%app and #%module-begin – can be used to create customized languages like the preprocessor language, where the result of evaluating a toplevel expression is printed using the preprocessor printer, or the documentation language where they are collected into a definition.

Again, missing these features does not prevent implementing the Scribble reader and getting its benefits. These are features that are generally useful, and enhance the utility of the reader in various ways — not having them means that the respective benefit is lost, but nothing else.

6. Alternative Approaches and Related Work

There are numerous approaches to representing textual content in code — and more than a few have been used by Schemers.

- Many Schemers still use plain Scheme syntax. Some attempts at making things a little better include quasiquoting, and the simple idea of using no spaces around double quotes, which can be seen as a very limited form of interpolation.

```
(string-append "Today is "(date)".")
```

  There are also uses of multi-line Scheme strings, yet it seems that these are disliked enough to be rather rare.

- SCSH[10] was the first implementation to popularize here strings, with a syntax that was adopted by several implementations, including PLT Scheme

```
#<<END
...
END
```

- The Skribe[6] documentation system has greatly influenced the design of Scribble, though not at the concrete syntax level. Skribe has a simple string interpolation facility, where square brackets delimit a string, and a ,(starts a Scheme expression escape, requiring all such escapes to be parenthesized expressions. Other implementations have a similar functionality, for example, Gauche[8] and JScheme[1] have a built-in facility, and other implementations (PLT included) have add-on libraries. Implementations are mostly simple readtable-based extensions, which usually expand to a string-append expression. However, there is no consensus for either the syntax or the details (e.g., whether any value can be used in an expression escape or just strings).
- BRL ("Beautiful Report Language"[9]) and later Kawa[2], use the textual template approach: a source file is parsed as text by default, with Scheme code in square brackets. An interesting aspect of the BRL syntax is that in Scheme code, reversed square brackets delimit strings, which makes it possible to view brackets as marks around Scheme code which is possible not continuous. For example:

```
Are we there yet?
[(if (< (time) eta) ]no[ ]yes![)]
```

  This approach is based on several server-side page generation systems like PHP, JSP, and ASP.

- Finally, we have mentioned a few precursors to the current Scribble syntax, two of which (mzpp and mztext) are still include in the PLT preprocessor collection. Several other experiments were never made public, including an early template implementation where escapes could be nested in a way that is similar to Kawa, except that expression escapes can nest, forming a tower of interpreters, and in addition each level can run a *different* language. This was an example of a powerful system that was not useful at all for practical purposes.

7. Conclusion

In PLT Scheme, the Scribble reader has proven itself as a valuable tool. It is more powerful than similar facilities in modern languages, and at the same time it is a relatively simple and uniform syntax, which is critical to its acceptance. The reader plays a major role in the success of our documentation system: in our content migration from LaTeXto Scheme, and in adding significant amounts of text — we now have thousands of pages, and the quality of the documentation is much higher than it ever was before. It was also successful in creating a better preprocessor tool, and a template-based generation in our web server. The design process has been long and difficult; getting the advantages of string syntaxes in modern languages, doing so in a framework that fits well with the Scheme midset, and improving on it, are all factors that make this a real challenge.

Any Scheme implementation can gain the same benefits, and in fact, the Ikarus implementation[7] has been recently extended with the Scribble syntax. We hope that additional implementations will follow, leading to an even greater value of the syntax. Scribble is easy to try out in PLT, and it is even possible to use PLT to "manually expand" code with Scribble syntax to plain S-expressions (by simply using read on source files), so users of other implementations can try out the syntax using MzScheme as a preprocessor before implementing it.

References

[1] Ken Anderson, Tim Hickey, and Peter Norvig. JScheme. http://www.norvig.com/jscheme.html.

[2] Per Bothner. The Kawa language framework. http://www.gnu.org/software/kawa/.

[3] Robert Bruce Findler and Matthew Flatt. Slideshow: Functional presentations. In *ACM SIGPLAN International Conference on Functional Programming*, pages 224–235, 2004.

[4] Matthew Flatt and Eli Barzilay. Keyword and optional arguments in PLT Scheme. In *Proceedings of the Tenth Workshop on Scheme and Functional Programming*, 2009.

[5] Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the book on ad hoc documentation tools. In *ACM SIGPLAN International Conference on Functional Programming*, 2009.

[6] Erick Gallesio and Manuel Serrano. Skribe: a functional authoring language. *Journal of Functional Programming*, 15(5):751–770, 2005.

[7] Abdulaziz Ghuloum. Ikarus Scheme. http://ikarus-scheme.org/.

[8] Shiro Kawai. Gauche. http://practical-scheme.net/gauche/.

[9] Bruce R. Lewis. BRL: the beautiful report language. http://brl.sourceforge.net/.

[10] Olin Shivers. A Scheme shell. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1994.

[11] Michael Sperber (Ed.). The revised^6 report on the algorithmic language Scheme, 2007.

[12] Cheetah, the python powered template engine. http://www.cheetahtemplate.org/.

Interprocedural Dependence Analysis of Higher-Order Programs via Stack Reachability

Matthew Might Tarun Prabhu

University of Utah

{might,tarunp}cs.utah.edu

Abstract

We present a small-step abstract interpretation for the A-Normal Form λ-calculus (ANF). This abstraction has been instrumented to find data-dependence conflicts for expressions and procedures.

Our goal is parallelization: when two expressions have no dependence conflicts, it is safe to evaluate them in parallel. The underlying principle for discovering dependences is Harrison's principle: whenever a resources is accessed or modified, procedures that have frames live on the stack have a dependence upon that resource. The abstract interpretation models the stack of a modified CESK machine by mimicking heap-allocation of continuations. Abstractions of continuation marks are employed so that the abstract semantics retain proper tail-call optimization without sacrificing dependence information.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors–Optimization

General Terms Languages

Keywords A-Normal Form (ANF), abstract interpretation, control-flow analysis dependence analysis, continuation marks

1. Introduction

Compiler- and tool-driven parallelization of sequential code is an attractive option for exploiting the proliferation of multicore hardware and parallel systems. Legacy code is largely sequential, and parallelization of such code by hand is both cost-prohibitive and error-prone. In addition, decades of computer science education have created ranks of programmers trained to write sequential code. Consequently, sequential programming has inertia—an inertia which means that automatic parallelization may be the only feasible option for improving the performance of many software systems in the near term. Motivated by this need for automatic parallelization, this work explores a static analysis for detecting parallelizable expressions in sequential, side-effecting higher-order programs.

When parallelizing a sequential program, two questions determine where parallelization is appropriate:

1. Where is parallelization safe?
2. Where is parallelization beneficial?

The safety question is clearly necessary because arbitrarily parallelizing parts of a program can change the intended behavior and meaning of the program. The benefit question is necessary because cache effects, communication penalties, thread overheads and context-switches attach a cost to invoking parallelism on real machines. Our focus is answering the safety question, and we answer it with a static analysis tuned to pick up resource-conflict dependences between procedures. We leave the question of benefit to be answered by the programmer, heuristics, profiling or further static analysis.

When determining the safety of parallelization, the core principle is *dependence*: given two computations, if one computation *depends* on the other, then they may not be executed in parallel. On the other hand, if the two computations are *independent*, then executing the computations in parallel will not change the meaning of either one.

Example Consider the following code:

```
(let ((a (f x))
      (b (g y)))
 (h a b))
```

If possible, we would like to transform this code into:

```
(let|| ((a (f x))
        (b (g y)))
 (h a b))
```

where the form `(let|| ...)` behaves like an ordinary `let`, except that it may execute its expressions in parallel. In order to do so, however, the possibility of a dependence between the call to `f` and the call to `g` must be ruled out. □

Dependences may be categorized into *control* dependences and *data* dependences. If the execution of one computation determines whether or not another computation will happen, then there is a control dependence between these computations. Fortunately, functional programming languages make finding intraprocedural control dependences easy: lexical scoping exposes control dependences without the need for an intraprocedural data-flow analysis.

Example In the following code:

```
(if (f x)
    (g y)
    (h z))
```

there is a control dependence from the expression `(g y)` upon `(f x)` and from `(h z)` upon `(f x)`. □

Proceedings of the 2009 Scheme and Functional Programming Workshop
California Polytechnic State University Technical Report CPSLO-CSC-09-03

If, on the other hand, one computation modifies a resource that another computation accesses or modifies, then there is a data dependence between these computations.

Example In the following code:

```
(let* ((z 0)
       (f (λ (r) (set! z r)))
       (g (λ (s) z)))
 (let ((a (f x))
       (b (g y)))
  (h a b)))
```

it is unsafe to transform the interior `let` into a `let||` form, because the expression `(f x)` writes to the address of the variable `z`, and the expression `(g y)` reads from that address. □

1.1 Goal

Our goal in this work is a static analysis that conservatively bounds the resources read and written by the evaluation of an expression in a higher-order program.

The trivial case of such an analysis is an expression involving only primitive operations, *i.e.*, no procedures are invoked, and there are no indirect accesses to memory. For example, it is clear that the expression `(+ x y)` reads the addresses of the variables `x` and `y`, but writes nothing.

A harder case is when an expression uses a value through an alias. In this case, we can use a standard value-flow analysis such as k-CFA [24, 25] to unravel this aliasing.

The hardest case, and therefore the focus of this work, is when the evaluation of an expression invokes a procedure. For example, the resources read and written during the evaluation of the expression `(f x)` depend on the values of the variables `f` and `x`. Syntactically separate occurrences of the same expression may yield different reads and writes, and in fact, even temporally separated invocations of the *same* expression can yield different reads and writes. To maximize precision, our analysis actually provides resource-dependence information for each calling context of every procedure. Combined with control-flow information, this procedure-dependence data makes it possible to determine the data dependences for any given expression.

1.2 Approach

Harrison's dependence principle [12] inspired our approach:

Principle 1.1 (Harrison's Dependence Principle)**.** *Assuming the absence of proper tail-call optimization, when a resource is read or written, all of the procedures which have frames live on the stack have a dependence on that resource.*

Phrased in terms of procedures instead of resources, the intuition behind Harrison's principle is that a procedure depends on

1. all of the resources which it reads/writes directly, and
2. transitively, all of the resources which its callees read/write.

Harrison's principle implies that if an analysis could examine the stack in every machine state which accesses or modifies a resource, then the analysis could invert this information to determine all of the resources which a procedure may read or write during the course of execution. Obviously, a compiler can't expect to examine the real execution trace of a program: it may be non-terminating, or it may depend upon user input. A compiler can, however, perform an abstract interpretation of the program that models the program stack. From this abstract interpretation, the compiler can conservatively bound the resources read and written by each procedure.

Example In the following program,

```
(define r #f)

(define (f) (g))
(define (g) (h))
(define (h) (set! r 42))

(f)
```

at the assignment to the variable `r`, frames on behalf of the procedures `f`, `g` and `h` are on the stack, meaning each has a write-dependence on the variable `r`. □

A modification of Harrison's principle generalizes to the presence of a semantics with proper tail-call optimization by recording caller and context information inside continuation marks [4]. A **continuation mark** is an annotation attached to a frame (a continuation) on the stack. This work exploits continuation marks to reconstruct the procedures and calling contexts live on the stack at any one moment. The run-time stack is built out of a chain of continuations, and each time an existing continuation is adopted as a return point, the adopter is placed in the mark of the continuation; this allows multiple dependent procedures to share a single stack frame. It is worth going through the effort of optimizing tail calls in the concrete semantics, because abstract interpretations of tail-call-optimized semantics have higher precision [19].

Our approach also extends Harrison's principle by allowing dependences to be tracked separately for every context in which a procedure is invoked. For example, when λ_{42} is invoked from call site 13, it may write to resources a and b, but when invoked from call site 17, it may write to resources c and d. By discriminating among contexts, parallelizations which appeared to be invalid may be shown safe.

Clarification It is worth pointing out that our approach does *not* work with shared-memory multi-threaded programs. The analysis works only over sequential input programs, and then finds places where parallelism may be safely introduced. By restricting our focus to sequential programs, we avoid the well-known state-space explosion problem in static analysis of parallel programs. Finding mechanisms for introducing additional parallelism to parallel programs is a difficult problem reserved for future work.

1.3 Abstract-resource dependence graphs

The output of our static analysis is an *abstract-resource dependence graph*. In such a graph, there is a node for each abstract resource, and a node for each abstract procedure invocation. Each abstract resource node represents a set of mutable concrete resources, *e.g.*, heap addresses, I/O channels. An abstract procedure invocation is a procedure plus an abstract calling context. In the simplest case, all calling contexts are merged together and there is one node for each procedure, as in 0CFA [24, 25]. We distinguish invocations of procedures because each invocation may use different resources.

An edge from a procedure's invocation node to an abstract resource node indicates that during the extent of a procedure's execution within that context, a *write* to a resource represented by that node may occur. An edge from an abstract resource node to a procedure's node indicates that, during the extent of a procedure's execution within that context, a *read* from a resource represented by that node may occur. If there is a path from one invocation to another, then there is a write/read dependence between these invocations, and if two invocations can reach the same resource, then there is a write/write dependence.

Example The write or the read may not be lexically apparent from the body of the procedure itself, as it may happen inside

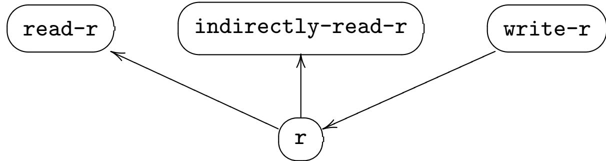
another procedure invoked indirectly. For example, consider the code:

```
(define r #f)

(define (read-r) r)
(define (indirectly-read-r) (read-r))
(define (write-r) (set! r #t))

(write-r)
(indirectly-read-r)
```

This would produce a dependence graph of the form:



In this example, we did not have to concern ourselves with discriminating on context: there is a single context for each procedure. Since there is only one binding of the variable `r`, it has its own abstract resource node. □

1.4 Road map

A-normal form [ANF] (Section 2) is the language that we use for our dependence analysis. Our analysis consists of an abstract interpretation of a specially constructed CESK-like machine for administrative normal form. To highlight the correspondence between the concrete and the abstract, we'll present the concrete and abstract semantics simultaneously (Section 3). Following that, we'll discuss instantiating parameters to obtain context-insensitive (Section 4) and context-sensitive (Section 5) dependence graphs. We'll conclude with a discussion of related work (Section 8) and future efforts (Section 9).

1.5 Contributions

Our work makes the following contributions:

1. A direct abstract interpretation of ANF
2. enabled by abstractions of "heap-allocated" continuations.
3. A garbage-collecting abstract interpretation of ANF.
4. A dependence analysis for higher-order programs
5. enabled by abstractions of continuation marks.
6. A *context-sensitive*, *interprocedural* dependence analysis.

2. A-Normal Form (ANF)

The forthcoming semantics and analysis deal with the administrative normal form λ-calculus (ANF) augmented with mutable variables (Figure 1). In ANF, all arguments in a procedure call must be immediately evaluable; that is, arguments can be λ terms and variables, but not procedure applications, let expressions or variable mutations. As a result, procedure calls must be either `let`-bound or in tail-position. A single imperative form (`set!`) allows the mutation of a variable's value.

The ANF language in Figure 1 contains only serial constructs. After the analysis is performed, it is not difficult to add a parallel `let||` form [13] to the language which performs the computation of its arms in parallel.

Why not continuation-passing style? It is possible to translate this analysis to continuation-passing style (CPS), but this analysis is a rare case in which ANF simplifies presentation over CPS.

$$
\begin{aligned}
u \in \mathsf{Var} &= \text{a set of identifiers} \\
lam \in \mathsf{Lam} &::= (\lambda\ (u_1 \cdots u_n)\ e_{\mathrm{body}}) \\
f, x \in \mathsf{Arg} &= \mathsf{Lam} + \mathsf{Var} \\
e \in \mathsf{Exp} &::= x \\
&\mid (f\ x_1 \cdots x_n) \\
&\mid (\mathtt{let}\ ((u\ e_{\mathrm{val}}))\ e_{\mathrm{body}}) \\
&\mid (\mathtt{set!}\ u\ x_{\mathrm{val}}\ e_{\mathrm{body}})
\end{aligned}
$$

Figure 1. A-normal form (ANF) augmented with mutable variables.

Because the analysis is stack-sensitive, the continuation-passing style language would have to be partitioned as in ΔCFA [18]. This partition introduces a notational overhead that distracts from presentation, instead of providing the simplification normally afforded by CPS.

In addition to the syntactic partitioning, the semantics would also need to be partitioned, so that true closures are kept separate from continuation closures. Without such a semantic partitioning, there would be no way to install the necessary continuation marks solely on continuations.

The use of continuation-passing style would also require a constraint that continuation variables not escape—that `call/cc`-like functions not be used in the direct-style source. This constraint comes from the fact that Harrison's principle expects stack-usage to mimick dependence. It is not readily apparent whether Harrison's principle can be adapted to allow the stack-usage patterns of unrestricted continuations. ANF without `call/cc` obeys the standard stack behavior expected by Harrison's principle.

3. Concrete and abstract semantics

Our goal is to determine all of the possible stack configurations that may arise at run-time when a procedure is read or written. Toward that end, we will construct a static analysis which conservatively bounds all of the machine states which could arise during the execution of the program. By examining this approximation, we can construct conservative models of stack behavior at resource-use points.

This section presents a small-step, operational, concrete semantics for ANF concurrently with an abstract interpretation [6, 7] thereof. The concrete semantics is a CESK-like machine [9] except that instead of having a sequence of continuations for a stack (*e.g.*, $Kont^*$ or $Frame^*$), each continuation is allocated in the store, and each continuation contains a pointer to the continuation beneath it. The standard CESK components are visible in the "Eval" states. The semantics employ the approach of Clements and Felleisen [4, 5] in adding marks to continuations; these allow our dependence analysis to work in the presence of tail-call optimization. (Later, these marks will contain the procedure invocations on whose behalf the continuation is acting as a return point.)

3.1 High-level structure

At the heart of both the concrete and abstract semantics are their respective state-spaces: the infinite set $State$ and the finite set $\widehat{State}$. Within these state-spaces, we will define semantic transition relations, $(\Rightarrow) \subseteq State \times State$ for the concrete semantics and $(\leadsto) \subseteq \widehat{State} \times \widehat{State}$ for the abstract semantics, in case-by-case fashion.

To find the meaning of a program e, we inject it into the concrete state-space with the expression-to-state injector function $\mathcal{I} : \mathsf{Exp} \to State$, and then we trace out the set of visitable states:

$$\mathcal{V}[\![e]\!] = \{\varsigma \mid \mathcal{I}[\![e]\!] \Rightarrow^* \varsigma\}.$$

Similarly, to compute the abstract interpretation, we also inject the program e into the initial abstract state, $\hat{\mathcal{I}} : \mathsf{Exp} \to \widehat{State}$. After this, a crude (but simple) way to imagine executing the abstract interpretation is to trace out the set of visitable states:

$$\hat{\mathcal{V}}[\![e]\!] = \{\hat{\varsigma} \mid \hat{\mathcal{I}}[\![e]\!] \leadsto^* \hat{\varsigma}\}.$$

(Of course, in practice an implementor may opt to use a combination of widening and monotonic termination testing to more efficiently compute or approximate this set [16].)

Relating the concrete and the abstract The concrete and abstract semantics are formally tied together through an abstraction relation. To construct this abstraction relation, we define a partial ordering on abstract states: $(\widehat{State}, \sqsubseteq)$. Then, we define an abstraction function on states: $\alpha : State \to \widehat{State}$. The abstraction relation is then the composition of these two: $(\sqsubseteq) \circ \alpha$.

Finding dependence Even without knowing the specifics of the semantics, we can still describe the high-level approach we will take for computing dependence information. In effect, we will examine each abstract state $\hat{\varsigma}$ in the set $\hat{\mathcal{V}}(e)$, and ask three questions:

1. From which abstract resources may $\hat{\varsigma}$ read?
2. To which abstract resources may $\hat{\varsigma}$ write?
3. Which procedures may have frames live on the stack in $\hat{\varsigma}$?

For each live procedure and for each resource read or written, the analysis adds an edge to the dependence graph.

3.2 Correctness

We can express the correctness of the analysis in terms of its high-level structure. To prove soundness, we need to show that the abstract semantics simulate the concrete semantics under the abstraction relation. The key inductive lemma of this soundness proof is a theorem demonstrating that the abstraction relation is preserved under a single transition:

Theorem 3.1 (Soundness)**.** *If:*

$$\varsigma \Rightarrow \varsigma' \text{ and } \alpha(\varsigma) \sqsubseteq \hat{\varsigma},$$

then there exists an abstract state ς' such that:

$$\hat{\varsigma} \leadsto \hat{\varsigma}' \text{ and } \alpha(\varsigma') \sqsubseteq \hat{\varsigma}'.$$

Or, diagrammatically:[1]

$$\begin{array}{ccc} \varsigma & \xrightarrow{(\Rightarrow)} & \varsigma' \\ \sqsubseteq\circ\alpha \downarrow & & \downarrow \sqsubseteq\circ\alpha \\ \hat{\varsigma} & \underset{(\leadsto)}{\cdots\cdots>} & \hat{\varsigma} \end{array}$$

Proof. Because the transition relations will be defined in a case-wise fashion, a proof of this form is easiest when factored into the same cases. There is nothing particularly interesting about the cases of this proof, so they are omitted. □

3.3 State-spaces

Figure 2 describes the state-space of the concrete semantics, and Figure 3 describes the abstract state-space. In both semantics, there are five kinds of states: head evaluation states, tail evaluation states, closure-application states, continuation-application states, and store-assignment states. Evaluation states evaluate top-level syntactic arguments in the current expression into semantic values, and then transfer execution based on the type of the current expression: calls move to closure-application states; simple expressions return by invoking the current continuation; `let` expressions move to another evaluation state for the arm; and `set!` terms move directly to a store-assignment state.

Every state contains a time-stamp. These are meant to increase monotonically during the course of execution, so as to act as a source of freshness where needed. In the abstract semantics, time-stamps encode a bounded amount of evaluation history, *i.e.*, context. (They are exactly Shivers's contours in k-CFA [25].)

The semantics make use of a binding-factored environment [17, 19, 25] where a variable maps to a binding through a local environment (β), and a binding then maps to a value through the store (σ). That is, a binding acts like an address in the heap. A binding-factored environment is in contrast to an unfactored environment, which takes a variable directly to a value. We use binding-factored environments because they simplify the semantics of mutation and make abstract interpretation more direct.

A return point (rp) is an address in the store that holds a continuation. A continuation, in turn, contains an variable awaiting the assignment of a value, an expression to evaluate next, a local environment in which to do so, a pointer to the continuation beneath it, and a mark to hold annotations. The set of marks is unspecified for the moment, but for the sake of finding dependences, the mark should at least encode all of the procedures for whom this continuation is acting as a return point.[2]

In order to allow polyvariance to be set externally [25] as in k-CFA, the state-space does not implicitly fix a choice for the set of times (contours) or the set of return points.

The most important property of an abstract state is that its stack is exposed: the analysis can trace out all of the continuations reachable from a state's current return point. This stack-walking is what ultimately drives the dependence analysis.

Abstraction map The explicit state-space definitions also allow us to formally define the abstraction map $\alpha : State \to \widehat{State}$ in terms of an overloaded family of interior abstraction functions, $|\cdot| : X \to \hat{X}$:

$$\begin{aligned} \alpha(e, \beta, \sigma, rp, t) &= (e, |\beta|, |\sigma|, |rp|, |t|) \\ \alpha(\chi, \vec{v}, \sigma, rp, t) &= (|\chi|, |\vec{v}|, |\sigma|, |rp|, |t|) \\ \alpha(\kappa, v, \sigma, t) &= (|\kappa|, |v|, |\sigma|, |t|) \\ \alpha(\vec{a}, \vec{v}, Eval) &= (|\vec{a}|, |\vec{v}|, \alpha(Eval)) \end{aligned}$$

$$\begin{aligned} |\beta| &= \lambda v.|\beta(v)| \\ |\sigma| &= \lambda \hat{a}. \bigsqcup_{|a|=\hat{a}} |\sigma(a)| \end{aligned}$$

$$\begin{aligned} |\langle v_1, \ldots, v_n \rangle| &= \langle |v_1|, \ldots, |v_n| \rangle \\ |(lam, \beta)| &= \{(lam, |\beta|)\} \\ |(u, e, \beta, rp, m)| &= \{(u, e, |\beta|, |rp|, |m|)\} \end{aligned}$$

$|a|$ is fixed by the polyvariance

$|m|$ is fixed by the context-sensitivity.

Injectors With respect to the explicit state-space definitions, we can now define the concrete state injector:

$$\mathcal{I}[\![e]\!] = ([\![e]\!], [], [], rp_0, t_0),$$

[1] The dotted line means "there exists a transition."

[2] Tail-called procedures share return points with their calling procedure.

$$
\begin{array}{rcll}
\varsigma & \in & \mathit{State} & = \mathit{Eval} + \mathit{ApplyFun} + \mathit{ApplyKont} + \mathit{SetAddrs} \\
& & \mathit{Eval} & = \mathit{EvalHead} + \mathit{EvalTail} \\
& & \mathit{EvalHead} & = \mathsf{Exp} \times \mathit{BEnv} \times \mathit{Store} \times \mathit{Kont} \times \mathit{Time} \\
& & \mathit{EvalTail} & = \mathsf{Exp} \times \mathit{BEnv} \times \mathit{Store} \times \mathit{RetPoint} \times \mathit{Time} \\
& & \mathit{ApplyFun} & = \mathit{Clo} \times \mathit{Val}^* \times \mathit{Store} \times \mathit{RetPoint} \times \mathit{Time} \\
& & \mathit{ApplyKont} & = \mathit{Kont} \times \mathit{Val} \times \mathit{Store} \times \mathit{Time} \\
& & \mathit{SetAddrs} & = \mathit{Addr}^* \times \mathit{Val}^* \times \mathit{EvalTail} \\
\\
\beta & \in & \mathit{BEnv} & = \mathsf{Var} \rightharpoonup \mathit{Addr} \\
\sigma & \in & \mathit{Store} & = \mathit{Addr} \rightharpoonup \mathit{Val} \\
\\
a & \in & \mathit{Addr} & = \mathit{Bind} + \mathit{RetPoint} \\
b & \in & \mathit{Bind} & = \mathsf{Var} \times \mathit{Time} \\
\\
v & \in & \mathit{Val} & = \mathit{Clo} + \mathit{Kont} \\
\chi & \in & \mathit{Clo} & = \mathsf{Lam} \times \mathit{BEnv} \\
\kappa & \in & \mathit{Kont} & = \mathsf{Var} \times \mathsf{Exp} \times \mathit{BEnv} \times \mathit{RetPoint} \times \mathit{Mark} \\
\\
\mathit{rp} & \in & \mathit{RetPoint} & = \text{a set of addresses for continuations} \\
m & \in & \mathit{Mark} & = \text{a set of stack-frame annotations} \\
t & \in & \mathit{Time} & = \text{an infinite set of times}
\end{array}
$$

Figure 2. State-space for the concrete semantics.

$$
\begin{array}{rcll}
\hat{\varsigma} & \in & \widehat{\mathit{State}} & = \widehat{\mathit{Eval}} + \widehat{\mathit{ApplyFun}} + \widehat{\mathit{ApplyKont}} + \widehat{\mathit{SetAddrs}} \\
& & \widehat{\mathit{Eval}} & = \widehat{\mathit{EvalHead}} + \widehat{\mathit{EvalTail}} \\
& & \widehat{\mathit{EvalHead}} & = \mathsf{Exp} \times \widehat{\mathit{BEnv}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{Kont}} \times \widehat{\mathit{Time}} \\
& & \widehat{\mathit{EvalTail}} & = \mathsf{Exp} \times \widehat{\mathit{BEnv}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{RetPoint}} \times \widehat{\mathit{Time}} \\
& & \widehat{\mathit{ApplyFun}} & = \widehat{\mathit{Clo}} \times \widehat{\mathit{Val}}^* \times \widehat{\mathit{Store}} \times \widehat{\mathit{RetPoint}} \times \widehat{\mathit{Time}} \\
& & \widehat{\mathit{ApplyKont}} & = \widehat{\mathit{Kont}} \times \widehat{\mathit{Val}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{Time}} \\
& & \widehat{\mathit{SetAddrs}} & = \widehat{\mathit{Addr}}^* \times \widehat{\mathit{Val}}^* \times \widehat{\mathit{EvalTail}} \\
\\
\hat{\beta} & \in & \widehat{\mathit{BEnv}} & = \mathsf{Var} \rightharpoonup \widehat{\mathit{Addr}} \\
\hat{\sigma} & \in & \widehat{\mathit{Store}} & = \widehat{\mathit{Addr}} \rightharpoonup \widehat{\mathit{Val}} \\
\\
\hat{a} & \in & \widehat{\mathit{Addr}} & = \widehat{\mathit{Bind}} + \widehat{\mathit{RetPoint}} \\
\hat{b} & \in & \widehat{\mathit{Bind}} & = \mathsf{Var} \times \widehat{\mathit{Time}} \\
\\
\hat{v} & \in & \widehat{\mathit{Val}} & = \mathcal{P}(\widehat{\mathit{Clo}} + \widehat{\mathit{Kont}}) \\
\hat{\chi} & \in & \widehat{\mathit{Clo}} & = \mathsf{Lam} \times \widehat{\mathit{BEnv}} \\
\hat{\kappa} & \in & \widehat{\mathit{Kont}} & = \mathsf{Var} \times \mathsf{Exp} \times \widehat{\mathit{BEnv}} \times \widehat{\mathit{RetPoint}} \times \widehat{\mathit{Mark}} \\
\\
\widehat{\mathit{rp}} & \in & \widehat{\mathit{RetPoint}} & = \text{a set of addresses for continuations} \\
\hat{m} & \in & \widehat{\mathit{Mark}} & = \text{a set of stack-frame annotations} \\
\hat{t} & \in & \widehat{\mathit{Time}} & = \text{a finite set of times}
\end{array}
$$

Figure 3. State-space for the abstract semantics.

and the abstract state injector:

$$\hat{\mathcal{I}}[\![e]\!] = ([\![e]\!], [], [], \widehat{rp}_0, \hat{t}_0).$$

Partial order We can also define the partial ordering on the abstract state-space explicitly:

$$(e, \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t}) \sqsubseteq (e, \hat{\beta}, \hat{\sigma}', \widehat{rp}, \hat{t}) \text{ iff } \hat{\sigma} \sqsubseteq \hat{\sigma}'$$
$$(\hat{\chi}, \vec{\hat{v}}, \hat{\sigma}, \widehat{rp}, \hat{t}) \sqsubseteq (\hat{\chi}, \vec{\hat{v}}', \hat{\sigma}', \widehat{rp}, \hat{t}) \text{ iff } \vec{\hat{v}} \sqsubseteq \vec{\hat{v}} \text{ and } \hat{\sigma} \sqsubseteq \hat{\sigma}'$$
$$(\hat{\kappa}, \hat{v}, \hat{\sigma}, \hat{t}) \sqsubseteq (\hat{\kappa}, \hat{v}', \hat{\sigma}', \hat{t}) \text{ iff } \hat{v} \sqsubseteq \hat{v}' \text{ and } \hat{\sigma} \sqsubseteq \hat{\sigma}'$$
$$(\vec{\hat{a}}, \vec{\hat{v}}, \hat{\varsigma}) \sqsubseteq (\vec{\hat{a}}, \vec{\hat{v}}, \hat{\varsigma}') \text{ iff } \hat{\varsigma} \sqsubseteq \hat{\varsigma}'$$

$$\hat{\sigma} \sqsubseteq \hat{\sigma}' \text{ iff } \hat{\sigma}(\hat{a}) \sqsubseteq \hat{\sigma}'(\hat{a}) \text{ for all } \hat{a} \in dom(\hat{\sigma})$$

$$\langle \hat{v}_1, \ldots, \hat{v}_n \rangle \sqsubseteq \langle \hat{v}'_1, \ldots, \hat{v}'_n \rangle \text{ iff } \hat{v}_i \sqsubseteq \hat{v}'_i \text{ for } 1 \leq i \leq n$$
$$\hat{v} \sqsubseteq \hat{v}' \text{ iff } \hat{v} \subseteq \hat{v}'.$$

3.4 Auxiliary functions

The semantics require one auxiliary function to ensure that the forthcoming transition relation is well-defined. The semantics make use of the concrete argument evaluator: $\mathcal{E} : \mathsf{Arg} \times BEnv \times Store \rightharpoonup Val$:

$$\mathcal{E}([\![lam]\!], \beta, \sigma) = ([\![lam]\!], \beta)$$
$$\mathcal{E}([\![u]\!], \beta, \sigma) = \sigma(\beta[\![u]\!]),$$

and its counterpart, the abstract argument evaluator: $\hat{\mathcal{E}} : \mathsf{Arg} \times \widehat{BEnv} \times \widehat{Store} \rightharpoonup \widehat{Val}$:

$$\hat{\mathcal{E}}([\![lam]\!], \hat{\beta}, \hat{\sigma}) = \{([\![lam]\!], \hat{\beta})\}$$
$$\hat{\mathcal{E}}([\![u]\!], \hat{\beta}, \hat{\sigma}) = \hat{\sigma}(\hat{\beta}[\![u]\!]).$$

Given an argument, an environment and a store, these functions yield a value.

3.5 Parameters

There are three external parameters for this analysis, expressed in the form of three concrete/abstract function pairs. The only constraint on each of these pairs is that the abstract component must simulate the concrete component.

The continuation-marking functions annotate the top of the stack with dependence information:

$$mark^{\flat} : Clo \times State \to Kont \to Kont$$
$$mark^{\sharp} : \widehat{Clo} \times \widehat{State} \to \widehat{Kont} \to \widehat{Kont}.$$

Without getting into details yet, a reasonable candidate for the set of abstract marks is the power set of λ-terms: $\widehat{Mark} = \mathcal{P}(\mathsf{Lam})$.

The next-contour functions are parameters that dictate the polyvariance of the heap, where the heap is the portion of the store that holds bindings:

$$succ^{\flat} : State \to Time$$
$$succ^{\sharp} : \widehat{State} \to \widehat{Time}.$$

For example, in 0CFA, set of times is a singleton: $\widehat{Time} = \{\hat{t}_0\}$.

The next-return-point-address functions will dictate the polyvariance of the stack, where the stack is the portion of the store that holds continuations. In fact, there are two pairs of these functions, one to be used for ordinary `let`-form transitions:

$$alloca^{\flat} : State \to RetPoint$$
$$alloca^{\sharp} : \widehat{State} \to \widehat{RetPoint},$$

and another pair to be used for non-tail application evaluation:

$$alloca^{\flat} : Clo \times State \to RetPoint$$
$$alloca^{\sharp} : \widehat{Clo} \times \widehat{State} \to \widehat{RetPoint}.$$

For example, in 0CFA, the set of return points is the set of expressions: $RetPoint = \mathsf{Exp}$, and first allocation function yields the current expression, while the second allocation function yields the λ-term inside the closure.

We will explore marks and marking functions in more detail later. In brief, the polyvariance functions establishes the trade-off between speed and precision for the analysis. For more detailed discussion of choices for polyvariance, see [16, 25].

3.6 Return

In a return state, the machine has reached the body of a λ term, a `let` form or a `set!` form, and it is evaluating an argument term to return: x. The transition evaluates the syntactic expression x into a semantic value v in the context of the current binding environment β and the store σ. Then the transition finds the continuation awaiting the value of this expression: $\kappa = \sigma(rp)$. In the subsequent application state, the continuation κ receives the value v. In every transition, the time-stamp is incremented from time t to $succ^{\flat}(\varsigma)$.

$$\overbrace{([\![x]\!], \beta, \sigma, rp, t)}^{\varsigma \in EvalTail} \Rightarrow \overbrace{(\kappa, v, \sigma, t')}^{\varsigma' \in ApplyKont},$$
$$\text{where } \kappa = \sigma(rp)$$
$$v = \mathcal{E}([\![x]\!], \beta, \sigma)$$
$$t' = succ^{\flat}(\varsigma).$$

As will be the case for the rest of the transitions, the abstract transition mirrors the concrete transition in structure, with subtle differences. In this case, it is worth noting that the abstract transition nondeterministically branches to all possible abstract continuations:

$$\overbrace{([\![x]\!], \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t})}^{\hat{\varsigma} \in \widehat{EvalTail}} \leadsto \overbrace{(\hat{\kappa}, \hat{v}, \hat{\sigma}, \hat{t}')}^{\hat{\varsigma}' \in \widehat{ApplyKont}},$$
$$\text{where } \hat{\kappa} \in \hat{\sigma}(\widehat{rp})$$
$$\hat{v} = \hat{\mathcal{E}}([\![x]\!], \hat{\beta}, \hat{\sigma})$$
$$\hat{t}' = succ^{\sharp}(\hat{\varsigma}).$$

3.7 Application evaluation: Head call

From a "head-call" (*i.e.*, non-tail) evaluation state, the transition first evaluates the syntactic arguments $f, x_1, \ldots, x_n$ into semantic values. Then, the supplied continuation is marked with information about the procedure being invoked and then inserted into the store

at a newly allocated location: rp'.

$$\overbrace{(\llbracket(f\ \ x_1 \cdots x_n)\rrbracket, \beta, \sigma, \kappa, t)}^{\varsigma \in EvalHead} \Rightarrow \overbrace{(\chi, \langle v_1, \ldots, v_n\rangle, \sigma', rp', t')}^{\varsigma' \in ApplyFun},$$
$$\begin{aligned} \text{where } v_i &= \mathcal{E}(\llbracket x_i \rrbracket, \beta, \sigma) \\ t' &= succ^\flat(\varsigma) \\ \chi &= \mathcal{E}(\llbracket f \rrbracket, \beta, \sigma) \\ rp' &= alloca^\flat(\chi, \varsigma) \\ \sigma' &= \sigma[rp \mapsto mark^\flat(\chi, \varsigma)(\kappa)]. \end{aligned}$$

In the abstract transition, execution nondeterministically branches to all abstract procedures:

$$\overbrace{(\llbracket(f\ \ x_1 \cdots x_n)\rrbracket, \hat\beta, \hat\sigma, \hat\kappa, \hat t)}^{\hat\varsigma \in \widehat{EvalHead}} \rightsquigarrow \overbrace{(\hat\chi, \langle \hat v_1, \ldots, \hat v_n\rangle, \hat\sigma', \widehat{rp}', \hat t')}^{\hat\varsigma' \in \widehat{ApplyFun}},$$
$$\begin{aligned} \text{where } \hat v_i &= \hat{\mathcal{E}}(\llbracket x_i \rrbracket, \hat\beta, \hat\sigma) \\ \hat t' &= succ^\sharp(\hat\varsigma) \\ \hat\chi &\in \hat{\mathcal{E}}(\llbracket f \rrbracket, \hat\beta, \hat\sigma) \\ \widehat{rp}' &= alloca^\sharp(\hat\chi, \hat\varsigma) \\ \hat\sigma' &= \hat\sigma \sqcup [\widehat{rp} \mapsto mark^\sharp(\hat\chi, \hat\varsigma)(\hat\kappa)]. \end{aligned}$$

3.8 Application evaluation: Tail call

From a tail-call evaluation state, the transition evaluates the syntactic arguments $f, x_1, \ldots, x_n$ into semantic values. At the same time, the current continuation is marked with information from the procedure being invoked:

$$\overbrace{(\llbracket(f\ \ x_1 \cdots x_n)\rrbracket, \beta, \sigma, rp, t)}^{\varsigma \in EvalHead} \Rightarrow \overbrace{(\chi, \langle v_1, \ldots, v_n\rangle, \sigma', rp, t')}^{\varsigma' \in ApplyFun},$$
$$\begin{aligned} \text{where } v_i &= \mathcal{E}(\llbracket x_i \rrbracket, \beta, \sigma) \\ t' &= succ^\flat(\varsigma) \\ \chi &= \mathcal{E}(\llbracket f \rrbracket, \beta, \sigma) \\ \sigma' &= \sigma[rp \mapsto mark^\flat(\chi, \varsigma)(\sigma(rp))]. \end{aligned}$$

In the abstract transition, execution nondeterministically branches to all abstract procedures, and *all* of the current abstract continuations are marked:

$$\overbrace{(\llbracket(f\ \ x_1 \cdots x_n)\rrbracket, \hat\beta, \hat\sigma, \widehat{rp}, \hat t)}^{\hat\varsigma \in \widehat{EvalHead}} \rightsquigarrow \overbrace{(\hat\chi, \langle \hat v_1, \ldots, \hat v_n\rangle, \hat\sigma', \widehat{rp}, \hat t')}^{\hat\varsigma' \in \widehat{ApplyFun}},$$
$$\begin{aligned} \text{where } \hat v_i &= \hat{\mathcal{E}}(\llbracket x_i \rrbracket, \hat\beta, \hat\sigma) \\ \hat t' &= succ^\sharp(\hat\varsigma) \\ \hat\chi &\in \hat{\mathcal{E}}(\llbracket f \rrbracket, \hat\beta, \hat\sigma) \\ \hat\sigma' &= \hat\sigma[\widehat{rp} \mapsto mark^\sharp(\hat\chi, \hat\varsigma)(\hat\sigma(\widehat{rp}))]. \end{aligned}$$

3.9 Let-binding applications

If a `let`-form is evaluating an application term, then the machine state creates a new continuation κ set to return to the body of the `let`-expression, e'. (The mark in this continuation is set to some default, empty annotation, m_0.) Then, the transition moves on to a head-call evaluation state.

$$\overbrace{(\llbracket\texttt{(let ((}u\ \ e\texttt{))}\ \ e'\texttt{)}\rrbracket, \beta, \sigma, rp, t)}^{\varsigma \in EvalTail} \Rightarrow \overbrace{(\llbracket e \rrbracket, \beta, \sigma, \kappa, t')}^{\varsigma' \in EvalHead},$$
$$\begin{aligned} \text{where } t' &= succ^\flat(\varsigma) \\ \kappa &= (u, \llbracket e \rrbracket, \beta, rp, m_0). \end{aligned}$$

The abstract transition mirrors the concrete transition:

$$\overbrace{(\llbracket\texttt{(let ((}u\ \ e\texttt{))}\ \ e'\texttt{)}\rrbracket, \hat\beta, \hat\sigma, \widehat{rp}, \hat t)}^{\hat\varsigma \in \widehat{EvalTail}} \rightsquigarrow \overbrace{(\llbracket e \rrbracket, \hat\beta, \hat\sigma, \hat\kappa, \hat t')}^{\hat\varsigma' \in \widehat{EvalHead}},$$
$$\begin{aligned} \text{where } \hat t' &= succ^\sharp(\hat\varsigma) \\ \hat\kappa &= (u, \llbracket e \rrbracket, \hat\beta, \widehat{rp}, \hat m_0). \end{aligned}$$

3.10 Let-binding non-applications

From a `let`-binding evaluation state where the expression is not an application, the transition creates a new continuation κ set to return to the body of the `let` expression, e'. After allocating a return point address rp' for the continuation, the transition inserts the continuation into the new store, σ'.

$$\overbrace{(\llbracket\texttt{(let ((}u\ \ e\texttt{))}\ \ e'\texttt{)}\rrbracket, \beta, \sigma, rp, t)}^{\varsigma \in EvalTail} \Rightarrow \overbrace{(\llbracket e \rrbracket, \beta, \sigma', rp', t')}^{\varsigma' \in EvalTail},$$
$$\begin{aligned} \text{where } t' &= succ^\flat(\varsigma) \\ \kappa &= (u, \llbracket e \rrbracket, \beta, rp, m_0) \\ rp' &= alloca^\flat(\varsigma) \\ \sigma' &= \sigma[rp' \mapsto \kappa]. \end{aligned}$$

The abstract transition mirrors the concrete transition, except that the update to the store happens via joining ($\sqcup$) instead of shadowing:

$$\overbrace{(\llbracket\texttt{(let ((}u\ \ e\texttt{))}\ \ e'\texttt{)}\rrbracket, \hat\beta, \hat\sigma, \widehat{rp}, \hat t)}^{\hat\varsigma \in \widehat{EvalTail}} \rightsquigarrow \overbrace{(\llbracket e \rrbracket, \hat\beta, \hat\sigma', \widehat{rp}', \hat t')}^{\hat\varsigma' \in \widehat{EvalTail}},$$
$$\begin{aligned} \text{where } \hat t' &= succ^\sharp(\hat\varsigma) \\ \hat\kappa &= (u, \llbracket e \rrbracket, \hat\beta, \widehat{rp}, \hat m_0) \\ \widehat{rp}' &= alloca^\sharp(\hat\varsigma) \\ \hat\sigma' &= \hat\sigma \sqcup [\widehat{rp}' \mapsto \{\hat\kappa\}]. \end{aligned}$$

3.11 Binding mutation

From a `set!`-mutation evaluation state, the transition looks up the new value v, finds the address $a = \beta\llbracket u \rrbracket$ of the variable and then transitions to an address-assignment state.

$$\overbrace{(\llbracket\texttt{(set!}\ \ u\ \ x\ \ e\texttt{)}\rrbracket, \beta, \sigma, rp, t)}^{\varsigma \in EvalTail} \Rightarrow \overbrace{(\langle a \rangle, \langle v \rangle, (\llbracket e \rrbracket, \beta, \sigma, rp, t'))}^{\varsigma' \in SetAddrs},$$
$$\begin{aligned} \text{where } t' &= succ^\flat(\varsigma) \\ v &= \mathcal{E}(\llbracket x \rrbracket, \beta, \sigma) \\ a &= \beta\llbracket u \rrbracket. \end{aligned}$$

Once again, the abstract transition directly mirrors the concrete transition:

$$\overbrace{(\llbracket \texttt{(set!}\ u\ x\ e\texttt{)} \rrbracket, \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t})}^{\hat{\varsigma} \in \widehat{EvalTail}} \rightsquigarrow \overbrace{(\langle \hat{a} \rangle, \langle \hat{v} \rangle, (\llbracket e \rrbracket, \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t}'))}^{\hat{\varsigma}' \in \widehat{SetAddrs}},$$
$$\text{where } \hat{t}' = succ^\sharp(\hat{\varsigma})$$
$$\hat{v} = \hat{\mathcal{E}}(\llbracket x \rrbracket, \hat{\beta}, \hat{\sigma})$$
$$\hat{a} = \hat{\beta}\llbracket u \rrbracket.$$

3.12 Continuation application

The continuation-application transitions move directly to address-assignment states:

$$\overbrace{(\kappa, v, \sigma, t)}^{\varsigma \in AppKont} \Rightarrow \overbrace{(\langle a \rangle, \langle v \rangle, (\llbracket e \rrbracket, \beta, \sigma, rp, t'))}^{\varsigma \in SetAddrs},$$
$$\text{where } t' = succ^\flat(\varsigma)$$
$$\kappa = (u, \llbracket e \rrbracket, \beta, rp, m)$$
$$a = (u, t').$$

The abstract exactly mirrors the concrete:

$$\overbrace{(\hat{\kappa}, \hat{v}, \hat{\sigma}, \hat{t})}^{\hat{\varsigma} \in \widehat{AppKont}} \rightsquigarrow \overbrace{(\langle \hat{a} \rangle, \langle \hat{v} \rangle, (\llbracket e \rrbracket, \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t}'))}^{\hat{\varsigma} \in \widehat{SetAddrs}},$$
$$\text{where } \hat{t}' = succ^\sharp(\hat{\varsigma})$$
$$\hat{\kappa} = (u, \llbracket e \rrbracket, \hat{\beta}, \widehat{rp}, \hat{m})$$
$$\hat{a} = (u, \hat{t}').$$

3.13 Procedure application

Procedure-application states also move directly to assignment states, but the transition creates an address for each of the formal parameters involved:

$$\overbrace{(\chi, \vec{v}, \sigma, rp, t)}^{\varsigma \in ApplyFun} \Rightarrow \overbrace{(\vec{a}, \vec{v}, (\llbracket e \rrbracket, \beta', \sigma, rp, t'))}^{\varsigma' \in SetAddrs},$$
$$\text{where } \chi = (\llbracket \texttt{(}\lambda\ \texttt{(}u_1 \cdots u_n\texttt{)}\ e\texttt{)} \rrbracket, \beta)$$
$$t' = succ^\flat(\varsigma)$$
$$a_i = (\llbracket u_i \rrbracket, t')$$
$$\beta' = \beta[\llbracket u_i \rrbracket \mapsto a_i].$$

Once again, the abstract directly mirrors the concrete:

$$\overbrace{(\hat{\chi}, \vec{\hat{v}}, \hat{\sigma}, \widehat{rp}, \hat{t})}^{\hat{\varsigma} \in \widehat{ApplyFun}} \rightsquigarrow \overbrace{(\vec{\hat{a}}, \vec{\hat{v}}, (\llbracket e \rrbracket, \hat{\beta}', \hat{\sigma}, \widehat{rp}, \hat{t}'))}^{\hat{\varsigma}' \in \widehat{SetAddrs}},$$
$$\text{where } \hat{\chi} = (\llbracket \texttt{(}\lambda\ \texttt{(}u_1 \cdots u_n\texttt{)}\ e\texttt{)} \rrbracket, \hat{\beta})$$
$$\hat{t}' = succ^\sharp(\hat{\varsigma})$$
$$\hat{a}_i = (\llbracket u_i \rrbracket, \hat{t}')$$
$$\hat{\beta}' = \hat{\beta}[\llbracket u_i \rrbracket \mapsto \hat{a}_i].$$

3.14 Store assignment

The store-assignment transition assigns each address a_i its corresponding value v_i in the store:

$$\overbrace{(\vec{a}, \vec{v}, (\llbracket e \rrbracket, \beta, \sigma, rp, t))}^{\varsigma \in SetAddrs} \Rightarrow \overbrace{(\llbracket e \rrbracket, \beta, \sigma', rp, t')}^{\varsigma' \in EvalTail},$$
$$\text{where } \sigma' = \sigma[a_i \mapsto v_i]$$
$$t' = succ^\flat(\varsigma).$$

In the abstract transition, the store is modified with a join ($\sqcup$) instead of over-writing entries in the old store. Soundness requires the join because the abstract address could be representing more than one concrete address—multiple values may legitimately reside there.

$$\overbrace{(\vec{\hat{a}}, \vec{v}, (\llbracket e \rrbracket, \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t}))}^{\hat{\varsigma} \in \widehat{SetAddrs}} \rightsquigarrow \overbrace{(\llbracket e \rrbracket, \hat{\beta}, \hat{\sigma}', \widehat{rp}, \hat{t}')}^{\hat{\varsigma}' \in \widehat{EvalTail}},$$
$$\text{where } \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{v}_i]$$
$$\hat{t}' = succ^\sharp(\hat{\varsigma}).$$

4. Computing data dependence from the stack

Against the backdrop of the abstract interpretation, we can define how to extract dependence information from an individual state. Harrison's principle calls for marking each stack frame with the procedure being invoked, and then, looking at the stack of each state to determine the dependents of any resource being accessed in that state.

The simplest possible marking function uses a set of λ terms for the mark:

$$Mark = \widehat{Mark} = \mathcal{P}(\mathsf{Lam}).$$

In this case, we end up with an analysis function that tags continuations with the λ term from the currently applied closure. The default mark is the empty set: $m_0 = \hat{m}_0 = \emptyset$. The concrete marking function is then:

$$mark^\flat((\llbracket lam \rrbracket, \beta), \varsigma)(\kappa) = (u_\kappa, e_\kappa, \beta_\kappa, rp_\kappa, m_\kappa \cup \{\llbracket lam \rrbracket\}),$$

which means that the abstract marking function is:

$$mark^\sharp((\llbracket lam \rrbracket, \hat{\beta}), \hat{\varsigma})(\hat{\kappa}) = (u_{\hat{\kappa}}, e_{\hat{\kappa}}, \hat{\beta}_{\hat{\kappa}}, \widehat{rp}_{\hat{\kappa}}, \hat{m}_{\hat{\kappa}} \cup \{\llbracket lam \rrbracket\}).$$

To compute the dependence graph, we need a function which accumulates all of the marks for a given state, and then we'll need functions to compute the resources read or written by that state. To accumulate the marks for a given state, we need to walk the stack. Toward this end, we can build an adjacency relation on continuations, $(\rightarrow_{\hat{\varsigma}}) \subseteq \widehat{Kont} \times \widehat{Kont}$:

$$(u, \llbracket e \rrbracket, \hat{\beta}, \widehat{rp}, \hat{m}) \rightarrow_{\hat{\varsigma}} \hat{\kappa} \text{ iff } \hat{\kappa} \in \hat{\sigma}_{\hat{\varsigma}}(\widehat{rp}).$$

We can then use the function $\hat{\mathcal{S}} : \widehat{State} \to \mathcal{P}(\widehat{Cont})$ to find the set of continuations reachable in the stack of a state $\hat{\varsigma}$:

$$\hat{\mathcal{S}}(\hat{\varsigma}) = \{\hat{\kappa} \mid \hat{\kappa}_{\hat{\varsigma}} \rightarrow^*_{\hat{\varsigma}} \hat{\kappa}\}.$$

Using this reachability function, the function $\hat{\mathcal{M}} : \widehat{State} \to \widehat{Mark}$ computes the aggregate mark on the stack:

$$\hat{\mathcal{M}}(\hat{\varsigma}) = \bigcup_{\hat{\kappa} \in \hat{\mathcal{S}}(\hat{\varsigma})} \hat{m}_{\hat{\kappa}}.$$

Using the aggregate mark function, we can construct the dependence graph. For each abstract state $\hat{\varsigma}$ visited by the interpretation, every item in the set $\hat{\mathcal{M}}(\hat{\varsigma})$ has a read dependence on every abstract

address read (via the evaluator $\hat{\mathcal{E}}$), and a write dependence for any address which is the destination of a `set!` construct. The function $\hat{\mathcal{R}} : \widehat{State} \to \mathcal{P}(\widehat{Addr})$ computes the set of abstract addresses read by each state:

$$\hat{\mathcal{R}}([\![x]\!], \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t}) = \hat{\mathcal{A}}(\hat{\beta})\langle x\rangle$$
$$\hat{\mathcal{R}}([\![\texttt{(}f\ x_1 \cdots x_n\texttt{)}]\!], \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t}) = \hat{\mathcal{A}}(\hat{\beta})\langle f, x_1, \ldots, x_n\rangle$$
$$\hat{\mathcal{R}}([\![\texttt{(}f\ x_1 \cdots x_n\texttt{)}]\!], \hat{\beta}, \hat{\sigma}, \hat{\kappa}, \hat{t}) = \hat{\mathcal{A}}(\hat{\beta})\langle f, x_1, \ldots, x_n\rangle$$
$$\hat{\mathcal{R}}([\![\texttt{(let ((}u\ e\texttt{))}\ e'\texttt{)}]\!], \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t}) = \hat{\mathcal{A}}(\hat{\beta})\langle e\rangle$$
$$\hat{\mathcal{R}}([\![\texttt{(set!}\ u\ x\ e\texttt{)}]\!], \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t}) = \hat{\mathcal{A}}(\hat{\beta})\langle x\rangle,$$

where the function $\hat{\mathcal{A}} : \widehat{BEnv} \to \mathsf{Exp}^* \to \mathcal{P}(\widehat{Addr})$ computes the addresses immediately read by expressions:

$$\hat{\mathcal{A}}(\hat{\beta})\langle\rangle = \emptyset$$
$$\hat{\mathcal{A}}(\hat{\beta})\langle e\rangle = \begin{cases} \{\hat{\beta}(e)\} & e \in \mathsf{Var} \\ \emptyset & \text{otherwise} \end{cases}$$
$$\hat{\mathcal{A}}(\hat{\beta})\langle e_1, \ldots, e_n\rangle = \hat{\mathcal{A}}(\hat{\beta})\langle e_1\rangle \cup \ldots \cup \hat{\mathcal{A}}(\hat{\beta})\langle e_n\rangle,$$

and, for all inputs where the function $\hat{\mathcal{R}}$ is undefined, it yields the empty set.

The function $\hat{\mathcal{W}} : \widehat{State} \to \mathcal{P}(\widehat{Addr})$ computes the set of abstract addresses written by a state:

$$\hat{\mathcal{W}}([\![\texttt{(set!}\ u\ x\ e\texttt{)}]\!], \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t}) = \{\hat{\beta}(u)\},$$

and for undefined inputs, the function $\hat{\mathcal{W}}$ yields the empty set.

5. Context-sensitive dependence analysis

It may be the case that a procedure accesses different addresses based on where and/or how it is called. The analysis can discriminate among context-sensitive dependences by enriching the information contained within marks to include context.

For example, the mark could also contain the site from which the procedure was called:

$$\widehat{Mark} = \mathcal{P}(\mathsf{Lam} \times \mathsf{Exp}).$$

Then, if a procedure is called from different call sites, the dependences at each call site will be tracked separately.

Example In the following code:

```
(define a #f)
(define b #f)

(define (write-a) (set! a #t 0))
(define (write-b) (set! b #t 1))

(define (unthunk f) (f))

(unthunk write-a) ; write-dependent on a
(unthunk write-b) ; write-dependent on b
```

there are two calls to the function `unthunk`. Without including context information in the marks, both calls to `unthunk` will be seen as having a write-dependence on both the addresses of `a` and `b`. By including context information, it sees that `unthunk` writes to the address of `a` in the first call, and to the address of `b` in the second call, which means that both calls to the function `unthunk` could actually be made in parallel. □

As the prior example demonstrates, it is possible to have a context-sensitive dependence analysis while still having a context-insenstive abstract interpretation.

Alternatively, the context-sensitivity of the dependence analysis could be synchronized with the context-sensitivity of the stack:

$$\widehat{Mark} = \mathcal{P}(\mathsf{Lam} \times \widehat{RetPoint}),$$

or of the heap:

$$\widehat{Mark} = \mathcal{P}(\mathsf{Lam} \times \widehat{Time}).$$

6. Abstract garbage collection

The non-recursive, small-step nature of the semantics given here ensures its compatibility with abstract garbage collection [19]. Abstract garbage collection removes false dependences that arise from the monotonic nature of abstract interpretation. Without abstract garbage collection, two independent procedures which happen to invoke a common library procedure may have their internal continuations, and hence their dependencies, merged. Moreover, the arguments to that library procedure will appear to merge as well. Abstract garbage collection collects continuations and arguments between invocations of the same procedure, cutting off this channel for spurious cross-talk.

To implement abstract garbage collection for this analysis, we define a garbage collection function on evaluation states:

$$\hat{\Gamma}(\hat{\varsigma}) = \begin{cases} (e, \hat{\beta}, \hat{\sigma}|\widehat{Reaches}(\hat{\varsigma}), \widehat{rp}, \hat{t}) & \hat{\varsigma} = (e, \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t}) \\ (e, \hat{\beta}, \hat{\sigma}|\widehat{Reaches}(\hat{\varsigma}), \hat{\kappa}, \hat{t}) & \hat{\varsigma} = (e, \hat{\beta}, \hat{\sigma}, \hat{\kappa}, \hat{t}), \end{cases}$$

where the function $\widehat{Reaches} : \widehat{State} \to \mathcal{P}(\widehat{Addr})$ finds all of the addresses reachable from a particular state:

$$\widehat{Reaches}(\hat{\varsigma}) = \{\hat{a} : \hat{a}_0 \hookrightarrow^*_{\hat{\sigma}_{\hat{\varsigma}}} \hat{a} \text{ and } \hat{a}_0 \in \widehat{Roots}(\hat{\varsigma})\},$$

and the relation $(\hookrightarrow) \subseteq \widehat{Addr} \times \widehat{Store} \times \widehat{Addr}$ determines which addresses are adjacent in the supplied store:

$$\hat{a} \hookrightarrow_{\hat{\sigma}} \hat{a}' \text{ iff } \hat{a}' \in \widehat{Touches}(\hat{\sigma}(\hat{a})),$$

and the overloaded function $\widehat{Touches}$ determines which addresses are touched by a particular abstract value:

$$\widehat{Touches}(\hat{v}) = \{\hat{a} \mid \hat{y} \in \hat{v} \text{ and } \hat{a} \in \widehat{Touches}(\hat{y})\}$$
$$\widehat{Touches}(lam, \hat{\beta}) = range(\hat{\beta})$$
$$\widehat{Touches}(u, e, \hat{\beta}, \widehat{rp}, \hat{m}) = range(\hat{\beta}) \cup \{\widehat{rp}\}.$$

7. Implementation

The latest implementation of this analysis for a macroless subset of Scheme is available as part of the Higher-Order Flow Analysis (HOFA) toolkit. HOFA is generic Scheme-based static analysis middle-end currently under construction. The latest version of HOFA is available online:

`http://ucombinator.googlecode.com/`

Figure 4 contains an example of a dependence diagram for the Solovay-Strassen cryptographic benchmark.

8. Related work

The semantics for dependence analysis are related to the semantics for ΓCFA for continuation-passing style (CPS) [16]. In fact, care was taken during this transfer to ensure that both abstract garbage collection and abstract counting are just as valid for these semantics. The notion of store-allocated continuations is reminiscent of SML/NJ's stack-handling [2], though because we do not impose an ordering on addresses, we could be modeling either stack-allocated continuations or store-allocated continuations. As these semantics demonstrate, changing from CPS to direct-style adds complexity

Figure 4. Solovay-Strassen benchmark dependence graph

in the form of additional transition rules. This dependence analysis exploits the fact that direct-style programs lead to computations that use the stack in a constrained fashion: stacks are never captured and restored via escaping continuations. It is not clear whether Harrison's principle extends to programs which use full, first-class continuations to restore popped stacked frames.

Abstract interpretation [6, 7] has long played a role in program analysis and automatic parallelization. Bueno *et al*. [3] used abstract interpretation of logic programs for automatic parallelization. Ricci [21] investigated the use of abstract interpretation for automatic parallelization of iterative constructs. Harrison [12] employed abstract interpretation is his approach to automatic parallelization of low-level Scheme code.

The notion of continuation marks, a mechanism for annotating continuations, is due to Clements and Felleisen [4, 5]. Clements used them previously to show that stack-based security contracts could be enforced at run-time even with proper tail-call optimization [5]. Using continuation marks within an abstract interpretation is novel. Our work exploits continuation marks for the same purpose: to retain information otherwise lost by tail-call optimization. In this case, the information we retain are the callers and calling contexts of all procedures that would be on a non-tail-call-optimized stack.

The idea of computing abstractions of stack behavior in order to perform dependence analysis appears in Harrison [12]. Harrison's work involved using abstract procedure strings to compute possible stack configurations. However, abstract procedure strings cannot handle tail calls properly, and they proved a brittle construct in practice, making the analysis both imprecise and expensive. Might and Shivers improved upon these drawbacks in their generalization to frame strings in ΔCFA [18, 20], but in handling tail calls properly, they removed the ability to soundly detect dependencies. The analysis presented here simplifies matters because it avoids constructing a stack model out of strings, opting to use the actual stack threaded through the store itself.

At present, this framework does not fully exploit Feeley's `future` construct [8], yet it could if combined with Flanagan and Felleisen's work [10] on removing superfluous `touch`es. The motivating `let||` construct may be expressed in terms of futures; that is, the following:

```
(let|| ((v e) ...)
 body)
```

could be rewritten as:

```
(let ((v (future e)) ...)
 (begin (touch v) ...
  body))
```

but, the present analysis does not determine if it is safe to remove the calls to `touch`, since it does not know if there will be resource usage conflicts with the continuation. Generalizing this analysis to CPS should also make it possible to automatically insert `future` constructs without the need for calls to `touch`, since it would be possible to tell if the evaluation of an expression has a dependence conflict with the current continuation.

Other approaches to automatic parallelization of functional programs include Schreiner's work [23] on detecting and exploiting patterns of parallelism in list processing functions. Hogen et al [1] presented a parallelizing compiler which used strictness analysis and generated an intermediate functional program with a special syntactic "letpar" construct which indicated that a legal parallel execution of subexpressions was possible. Parallelizing compilers have been implemented for functional programming languages such as EVE [15] and SML [22]. More theoretical work in this space includes [11] and more recently [14].

9. Future work

It tends to be harder to transfer an analysis from CPS to ANF: CPS is a fundamentally simpler language, requiring no handling of return-flow in abstract interpretation, and hence, no stack. This analysis marks a rare exception to that rule, in part because it is directly focused on working with the stack. Continuation-passing style can invalidate Harrison's principle when continuations escape. The two most promising routes for taming these unrestricted continuations are modifications of ΔCFA [20, 16] and an abstraction of higher-order languages to push-down automata.

References

[1] Andrea, G. H., Kindler, A., and Loogen, R. Automatic parallelization of lazy functional programs. In *Proc. of 4th European Symposium on Programming, ESOP'92, LNCS 582:254-268* (1992), pp. 254–268.

[2] Appel, A. W. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] Bueno, F., de la Banda, M. G., and Hermenegildo, M. Effectiveness of abstract interpretation in automatic parallelization: a case study in logic programming. *ACM Transactions on Programming Languages and Systems 21*, 2 (1999), 189–239.

[4] Clements, J. *Portable and high-level access to the stack with Continuation Marks*. PhD thesis, Northeastern University, 2005.

[5] Clements, J., and Felleisen, M. A tail-recursive machine with stack inspection. *Transactions on Programming Languages and Systems* (2004).

[6] Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Los Angeles, California, 1977), ACM Press, New York, NY, pp. 238–252.

[7] Cousot, P., and Cousot, R. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, 1979), ACM Press, New York, NY, pp. 269–282.

[8] Feeley, M. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors*. PhD thesis, Brandeis University, April 1993.

[9] Felleisen, M., and Friedman, D. A calculus for assignments in higher-order languages. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages* (1987), pp. 314–325.

[10] Flanagan, C., and Felleisen, M. The semantics of future and its use in program optimization. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1995), ACM, pp. 209–220.

[11] Geser, A., and Gorlatch, S. Parallelizing functional programs by generalization. In *Journal of Functional Programming* (1997), vol. 9, pp. 46–60.

[12] HARRISON, W. L. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation 2*, 3/4 (Oct. 1989), 179–396.

[13] HOGEN, G., KINDLER, A., AND LOOGEN, R. Automatic Parallelization of Lazy Functional Programs. In *ESOP '92, 4th European Symposium on Programming* (Rennes, France, February 26–28, 1992), B. Krieg-Brückner, Ed., vol. 582, Springer, Berlin, pp. 254–268.

[14] HURLIN, C. Automatic parallelization and optimization of programs by proof rewriting. In *SAS '09: Proceedings of the 16th international symposium on Static Analysis (to appear)*.

[15] LOIDL, H. W. A parallelizing compiler for the functional programming language eve, 1992.

[16] MIGHT, M. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, 2007.

[17] MIGHT, M. Logic-flow analysis of higher-order programs. In *Proceedings of the 34th Annual ACM Symposium on the Principles of Programming Languages (POPL 2007)* (Nice, France, January 2007), pp. 185–198.

[18] MIGHT, M., AND SHIVERS, O. Environment analysis via ΔCFA. In *Proceedings of the 33rd Annual ACM Symposium on the Principles of Programming Languages (POPL 2006)* (Charleston, South Carolina, January 2006), pp. 127–140.

[19] MIGHT, M., AND SHIVERS, O. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the 11th ACM International Conference on Functional Programming (ICFP 2006)* (Portland, Oregon, September 2006), pp. 13–25.

[20] MIGHT, M., AND SHIVERS, O. Analyzing the environment structure of higher-order languages using frame strings. *Theoretical Computer Science 375*, 1–3 (May 2007), 137–168.

[21] RICCI, L. Automatic loop parallelization: An abstract interpretation approach. In *International Conference on Parallel Computing in Electrical Engineering* (Los Alamitos, CA, USA, 2002), vol. 00, IEEE Computer Society, p. 112.

[22] SCAIFE, N., HORIGUCHI, S., MICHAELSON, G., AND BRISTOW, P. A parallel sml compiler based on algorithmic skeletons. *J. Funct. Program. 15*, 4 (2005), 615–650.

[23] SCHREINER, W. On the automatic parallelization of list-based functional programs. In *Proceedings of the Third International Workshop on Compilers for Parallel Computers* (1992). Invited paper.

[24] SHIVERS, O. Control-flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)* (Atlanta, Georgia, June 1988), pp. 164–174.

[25] SHIVERS, O. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

A. Appendix: Conventions

We make use of the natural meanings for the lattice operation $\sqcup$, the order relation $\sqsubseteq$ and the elements $\bot$ and $\top$, *i.e.*, point-wise, component-wise, member-wise liftings.

The notation $f[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$ means "the function f, except at point x_i, yield the value y_i."

Given a function $f : X \to Y$, we implicitly lift it over a set $S \subseteq X$:

$$f(S) = \{f(x) \mid x \in S\}.$$

The function $f|X$ denotes the function identical f, but defined only over inputs in the set X.

Descot: Distributed Code Repository Framework

Aaron W. Hsu
Indiana University
awhsu@indiana.edu

Abstract

Programming language communities often have repositories of code to which the community submits libraries and from which libraries are downloaded and installed. In communities where many implementations of the language exist, or where the community uses a number of language varieties, many such repositories can exist, each with their own toolset to access them. These diverse communities often have trouble collaborating accross implementation boundaries, because existing tools have not addressed inter-repository communication. Descot enables this collaboration, making it possible to collaborate without forcing large social change within the community. Descot is a metalanguage for describing libraries and a set of protocols for repositories to communicate and share information. This paper discusses the benefits of a public interface for library repositories and details the library metalanguage, the server protocol, and a server API for convenient implementation of Descot-compatible servers.

1. Introduction

All programming language communities must share code to be effective. Issues of portability and ease of distribution arise often within most language communities. In order to share code effectively, programmers must be able to run code portably over different implementations of a language, and they must have some means of distributing their code to users who need an easy way to install and manage their collection of libraries. In communities with a single dominant implementation, the first requirement is usually moot, and a central repository of portable libraries usually satisfies the second requirement (e.g. — CPAN [15]). However, in diverse communities, where many language standards and implementations may actively coexist in close proximity to one another, portability and easy distribution and installation can elude the community as a whole.

For example, the Scheme community actively uses at least four standards [19, 8, 21, 29] and even more actively developed and maintained Scheme implementations. Other communities share these same features. The Scheme community has made progress through efforts like the R6RS [29] `library` form at improving the overall portability of Scheme code. The Scheme community also has a number of repositories and tools for managing libraries. Many of these are implementation specific, but some, like Snow [12] are portable across implementations and try to store portable code packages.

In communities like Scheme, standardizing a single set of tools and a repository for managing, locating, and installing libraries of code is a difficult proposition at best. Rather than trying to create a standard toolchain and a central repository within a community that promotes diverse solutions and approaches, it would be better if many tools could be developed and many repositories created in such a way that they could all interoperate and communicate with one another. This would allow the tools to grow as needed around the segments of the community to which they were best suited, and would prevent any other segment of the community from losing out on advances made by the rest of the community. If a public interface existed for library repositories and tools to communicate among themselves in an effective, extensible manner, the benefits of a central repository could be retained, as well as the ability to develop different approaches to the issue.

Descot [18] realizes just such an interface by utilizing an RDF-based Schema [5, 24] to define a language for expressing library metadata and defining a set of protocols for interfacing with servers. It defines protocols for querying the server about code, retrieving specific library metadata, submitting code to the repository, and for mirroring one repository from another. Note that Descot does not try to replace existing management tools, which already take care of installing code, and it does not attempt to establish any specific repository. Instead, Descot enables the communication between repositories and among repositories and tools. Descot does not attempt to deal with the portability of the code itself, and leaves such efforts to other standards such as R6RS [29]; it deals only with the metadata of a library and the means to access this metadata.

With diverse communities like Scheme, collaboration between libraries will often break down into as much of a social as a technical problem. Descot cannot hope to solve social opposition, but it does enable the community to collaborate while maintaining the normal benefits of decentralized, separate development. Descot was specifically designed to minimize the impact on the social structure of the community that adopts it. This paper details the first steps towards enabling collaboration, by providing the techincal foundation. The author intends to undertake further efforts, such as the development of easy tools and libraries for deploying and integrating Descot, which will further reduce the barriers that usually exist when trying to improve the collaboration of largely separate efforts.

Proceedings of the 2009 Scheme and Functional Programming Workshop
California Polytechnic State University Technical Report CPSLO-CSC-09-03

Among the largest of obstacles, implementors and designers of repositories often have their own ideas about the design and use of the repository. Descot enables an unbounded number of clients, each with their own unique features, to operate on a wide variety of repositories. The basic design of Descot also enables easy extension to the metalanguage, which means that additional features for specific repositories may be added easily, without making it impossible for existing tools to work with the basic metadata. With Descot almost every feature and detail is left to the designer of the system, except for the parts necessary for useful communication, and even these are made flexible enough to allow a tremendous range of freedom.

While Descot is not tied to one specific language community, the remainder of this paper discusses Descot within the context of the Scheme community. Section 2 discusses the existing tools surrounding library distribution that are necessary for Descot to be useful. Section 3 details the Descot system itself. Section 4 lists some of the work others have done which relates to Descot. Section 5 contains concluding remarks.

2. Background

In order to effectively share code in a community such as Scheme, there must be a way of running another author's code, and there must be a way of searching, installing, and submitting code to the public. Scheme implementations often have a central repository for that implementation to which authors usually submit their code [31, 26, 30]. In order to manage large programs effectively, most Scheme implementations provide a module system that helps to control the visibility of procedures and macros defined in a block of code. Additionally, these repositories have convenient tools that allow libraries to be automatically downloaded and installed if desired. Often, merely specifying a requirement for one or more libraries is enough to guarantee that an user of a program can automatically install the libraries, assuming that they are visible in the central repository.

Other repositories attempt to host portable libraries that work across implementations. Snow [12] is a good example of this family of repositories (see Section 4 for more examples of these systems), and has a number of useful tools, including command line management tools and a packaging system. Because Snow tries to be portable across implementations, the tools themselves are able to run on a variety of Scheme implementations, and the libraries available in the Snow repository often run on more than one implementation.

Traditionally, authors of Scheme libraries would simply host their files in tarballs or flat files, and would maintain a set of dependencies that their code used. (See, for example, see Oleg Kiselyov's collection of Scheme code [23].) User's wanting to use their code would then either use the semi-portable libraries provided, with a little work, or would attempt to find them in their implementation's repository. This effort has been made somewhat easier by the recent standardization of the R6RS library form [29], which defines a standard library syntax, enabling code to be more easily shared among implementations and users.

Still, there are a wide variety of tools for library management, and many different module systems in active use. Clearly, the cooperation of these various tools, repositories, and implementations would benefit the community as a whole.

Library	Binding	SCM	CVS	Retrieval-method
Archive	Single-file	License	Person	Implementation

Table 1. Descot Classes

3. Descot

The Descot system itself divides roughly into the schema [17], which is the actual language for libraries, the server protocol, which specifies how servers ought to behave, a query protocol, for handling server queries, and an API that assists in the development of Descot servers. Descot itself consists of the first three elements, and the API exists as a convenience for developers.

3.1 Schema

Descot defines an RDF Schema [17, 24] for describing libraries of code. It augments the existing default RDF Schema [5] and is itself written using RDF. RDF is a specification for describing metainformation as directed graphs and has a number of syntactic representations. Current Descot tools support arbitrary representation formats, but by default, use SRDF (see the Appendix). The author chose RDF as the basic metalanaguage because it already has existing tools written around it and is relatively mature. RDF was designed specifically with this sort of problem in mind, and allows extensions as a matter of course. This makes it ideal as the basic language from the perspective of market share and technical features. The XML representation of RDF, however, is tedious and unpleasant to write by hand. SRDF is an S-expression based RDF format designed to mirror Turtle [3]. SRDF makes it easy to write RDF graphs by hand, while remaining easy to manipulate and parse using basic Scheme functions. The author actually began by writing his own S-expression based metalanguage, but soon realized that it was essentially a reimplementation of RDF. By using RDF, many features and semantics may be left to the RDF designers, greatly simplifying the specification of Descot's metalanguage. Descot also supports Turtle out of the box provided that the necessary libraries exist. Since the Schema itself is based on RDF, it is also format neutral; any other RDF format could be used, including, for example, SXML [23]. The Schema itself is a set of URIs to which we ascribe semantic meaning, and is used in the description of RDF Triples. All the URIs start with the prefix:

`http://descot.sacrideo.us/10-rdf-schema#`

All terms mentioned in this section are the tails of URIs prefixed by the above string.

The terms are divided roughly into Classes (see Table 1) and Properties. Most of the properties apply directly to Libraries (Table 3), but there are some general, person, and CVS properties as well (Table 2).

Every class is a type for a specialized node in a Descot Graph. Every node in a descot graph is expected to have a type property associated with it to identify its class.

`Library` nodes represent libraries, and most of the properties stem from Library nodes. Library nodes are also the main root node for most retrievals.

`Binding` nodes represent information about a procedure or macro that is exported or imported from a library. These nodes can be used to store information such as alternate names for procedures. They may also point to documentation about a specific procedure, but the only required property is the name.

name	alts	desc	homepage	e-mail
cvs-root	cvs-module			

Table 2. General/Miscellaneous Descot Properties

deps	names	license
creation	modified	contact
authors	categories	copyright-year
exports	location	implementation
copyright-owner	version	

Table 3. Descot Library Properties

`Archive` nodes contain file archive download information. Generally, they may point directly to the location of an Archive, such as a tarball. As such, these will usually be end nodes in a Descot graph, because they will not contain further information.

`Single-file` nodes are similar to `Archive` nodes, but they point to single Scheme files instead of archives. Generally, single files do not need to be processed by Descot clients further before being fed into a compatible implementation.

`License` nodes contain information about a License type, such as ISC, BSD, GPL, or a proprietary license of some sort. They may point somewhere else as the main reference, and have only a short description of the actual license in the graph, or they may contain the entire text of the license as the description. A short name should be provided that servers can use when they want to display licensing information without presenting the entire description, usually given on one line.

`Person` and `Implementation` nodes follow a similar pattern, describing people and implementations, respectively. People have names and e-mail addresses associated with them, and may have additional information. Implementations generally have a web site and a name associated with them.

`SCM` is a general class for "Source Control" based libraries. That is, `SCM` is a sub-class of `Retrieval-method` like `Archive` and `Single-file` are, but it describes a retrieval via some source control module, like CVS. `CVS` is the sub-class of the `SCM` class that describes CVS server modules particularly. Generally, one would use the `CVS` module or some other equivalent (such as for SVN or Darcs) rather than using `SCM`, but `SCM` properties may be defined to give generic information about a source module to a server that may not recognize the particular type of source control used.

Every node may be associated with a particular `name` which can be anything, and is not specific to the type. Library names are generally strings, but they could be extended to include other information or other types if a server desired. Generally, however, it is recommended to stick with the same types for existing classes, and change the range of the `name` property only for new classes introduced specifically for some specific server or purpose, so that other Descot-compatible systems do not have to work much harder on classes that are already defined.

For any given node, it may also happen that there are alternate nodes that would work in place of the given node. `alts` is expected to point to an rdf `Alt` node that will list the alternates. For example, a library may be implemented by a number of authors, and each library could be listed as an alternate to the others.

`desc` is a property pointing to a string node that contains a description of the node. This could be the license text in the case of a `License` node, or may be a human-readable description of a library for `Library` nodes.

`homepage` can be used where applicable to associate a given homepage to a node. The homepage referenced should be a Resource, and not, for example, a blank node.

The `CVS` node class also has two properties associated with it: `cvs-root` and `cvs-module`. These point to strings which contain the root of the CVS server and the module name for the library, respectively. This is enough information, generally, to obtain the library via CVS, but servers may wish to list additional information, such as the supported protocols for the CVS server.

`email` associates a string representation of an e-mail address with a given `Person` class node. The author did not use e-mail as a unique ID for people because e-mail addresses do not map directly in a one-to-one fashion to people. However, implementations may want to resolve conflicts of people who have the same name by differentiating them by their e-mail addresses.

The following properties all expect to have `Library` nodes as their domains/subjects.

`deps` points to a `List` of Libraries upon which the subject library is dependent.

`names` is a `List` of strings of short library names. These are expected to be alternative short names frequently used to identify the library, as opposed to the long `name` property string, which identifies the normal title of the library.

`exports` is a `List` of `Binding` nodes which represent the procedures and macros that the given library exports.

`license` points to a `License` node that is the license of the given `Library` node.

`authors` is a `List` of `Person` nodes that represents the authors of the library, but not necessarily the maintainer of the Descot metainformation.

`creation` points to a date time string that is the date of creation for the library metainformation, *not* necessarily the creation date of the library itself.

`modified` points to a date time string that represents the date and time of the last modification made to the library metadata, and not necessarily the date and time of the last update to the library itself.

`contact` points to a single person who has claimed responsibility for maintaining the metadata of a given library. This field must exist, and the `authors` property is not a substitute.

`implementation` points to an `Implementation` node, which identifies the implementation or language for which the code was designed to run. This could be a literal implementation, or may be an R6RS `Implementation` node to represent all R6RS compliant Scheme implementations, for example.

`version` is a string that identifies the version of the library. This could be a version number such as "3.5" or it could be something like "-Current". The later is useful for storing the metadata of the latest snapshot of development for a library, such as what one might find from a CVS server.

`location` points to a `Retrieval-method` node or a node of a type that is a sub-class of `Retrieval-method`. This node should tell a Descot client how to obtain the library itself. Notice that this is a very extensible property, and sophisticated servers may provide new `Retrieval-method` sub-classes to describe the details of library retrieval. PLT's

PLaneT, for example, may have a class for libraries that are distributed through the PLaneT packaging system.

`categories` points to a `List` of strings that are categories or tags for the given library. These tags are assumed to be case-insensitive for all intents and purposes.

`copyright-year` and `copyright-owner` are two parts of the Copyright information. `copyright-year` points to a year string, while `copyright-owner` may point to a `Person` or a `List` of `Person` nodes.

3.2 Server Protocol

Descot-compatible servers follow a simple set of rules that allow them to interact with one another. Servers handle three types of requests: mirroring, library/node requests, and queries. Queries are handled in Section 3.3. This section details only mirroring and node requests.

Every server must have a mirroring URI. When a request for this URI comes into the server, the server must respond with the RDF graph consisting of every library node in the server's store with one and only one branch. That branch must be the `modified` property pointing to the last modification time of the referenced library node. In this way, a server which is mirroring the content of another server may identify which libraries need to be updated, and pull only the given information into its own store.

The format of transmission should be arranged in an appropriate manner by the servers or server and client. No specific format is required, and no format need be recognized.

Servers and clients may also make node requests to a server. These are requests for the relevant information about a given node. For example, a client may wish to obtain the metadata for a library for some URI. It does so by accessing the URI and parsing the response from the server. The method of access depends on the protocol specified by the URI. HTTP will likely be a common protocol, but others, such as FTP, Gopher, or HTTPS could also be used. The response should be an RDF graph in either the format requested by the client [server] or the attempt by the server if it does not support the requested format. (Again, the way to request a particular format is protocol dependent, and not specified here.)

The graph returned by a server handling a node request contains a subset of the entire store on the server. Its root or starting node has the URI of the request. The server should then walk the paths going out from the requested URI in the store and return the graph that it walks. The server should stop pursuing a particular path when it encounters a node which has its own unique, accessible URI that can be requested individually. That is, the returned graph contains the descendants or the paths starting from the node with the URI requested, stopping at nodes which themselves have valid URIs. Blank nodes, then, are the only means by which the depth of the graph may grow beyond one. When encountering a blank node while walking the graph, a server will descend into it and continue its walk, but otherwise, the server will not descend into a node, which will have a valid URI if it is not a blank node.

These two request methods provide enough structure for servers and clients to communicate clearly and efficiently. No other behavior is required of a Descot server, though handling query requests is permitted and defined for any Descot server. Most servers will not handle queries, and instead, specific Descot servers will develop to mirror smaller servers and index them to provide a place to search many repositories at once.

3.3 Query Server

Since Descot uses RDF to describe its metadata, it may also utilize the tools available to RDF graphs. SPARQL is a query language and protocol for querying RDF graphs. If a Descot server wishes to provide Querying, then it should follow the protocols and language laid down in the SPARQL specification [28, 7, 2]. Implementing query request handling for a Descot server is not required.

Query-enabled servers enable lightweight clients to interact in useful and interesting ways with servers. Many systems which allow multiple repositories to be used often require that clients cache data about the repositories that it searches. This is fine when there are only a few repositories, but in systems where every developer may potentially have a repository, it may not make sense to cache all the data on every client. While nothing stops a client from caching server data from a Descot server, lightweight clients may use query-enabled Descot servers that mirror other repositories to search and find libraries and code which may have been obscured if the user of the client had to find and install repository information manually.

Query-enabled servers may thus become hubs among the web of Descot servers, providing users the benefit of a central repository, without many of the disadvantages.

3.4 Server API

A Server API has been developed to assist designers in writing Descot servers easily and quickly. They can also be utilized by scripts to assist in dealing with Descot stores. While the current Descot source code contains a number of additional modules, the utilities, printing, and server modules will generally help the most.

This code is currently available via revision control, and a packaged release will be made once some of the features have been completed. This API is the one used by the Descot server that runs (currently only as a proof of concept) at the Descot homepage [18]. The API is provided to assist developers of servers and clients, and implementors may opt to implement the Descot protocol and specification in other ways.

The `rdf-printing` module provides three procedures for printing RDF graphs in Turtle format.

`write-rdf-triple->turtle` takes an RDF triple and an optional port argument, and writes out that triple in Turtle form. `write-rdf-triples->turtle` and `write-rdf-graph->turtle` work the same way but take a list of triples and an RDF graph as their first argument respectively.

The `descot-rdf-utilities` module defines and exports common RDF and Descot URIs for use in other applications. It also defines the following procedures and macros.

`store-categories` : ⟨*graph*⟩ → ⟨*category list*⟩

Produces from an Descot RDF graph a list of all the categories found in the store.

`libraries-in-category` : ⟨*cat*⟩ → ⟨*library list*⟩

Produces a list of libraries that have a category ⟨*cat*⟩.

`in-rdf-list` : ⟨*store*⟩ ⟨*node*⟩ → #⟨void⟩

`in-rdf-list` is a foof loop [6] iterator over RDF `List` nodes. It allows one to iterate over RDF lists in the same way one might iterate over a normal Scheme list.

The iterator is used in `for` clauses of foof loops, as in, `(for elem rest (in-rdf-list store list-head-node))`.

`parse-turtle-file` : ⟨*file*⟩ [⟨*graph*⟩] → ⟨*graph*⟩

Parses a given ⟨*file*⟩ into a given ⟨*graph*⟩ or an empty graph if none is given.

`library-ids` : ⟨*store*⟩ → ⟨*id list*⟩
`library-title` : ⟨*rdf-map*⟩ → ⟨*library name*⟩
`library-names` : ⟨*store*⟩ ⟨*rdf-map*⟩ → ⟨*name list*⟩
`library-description` : ⟨*rdf-map*⟩ → ⟨*desc string*⟩
`library-copyright` : ⟨*store*⟩ ⟨*rdf-map*⟩ → ⟨*copy pair*⟩
`library-homepage` : ⟨*rdf-map*⟩ → ⟨*uri*⟩
`library-license-name` : ⟨*store*⟩ ⟨*rdf-map*⟩ → ⟨*name*⟩
`library-authors` : ⟨*store*⟩ ⟨*rdf-map*⟩ → ⟨*author list*⟩
`library-contact` : ⟨*store*⟩ ⟨*rdf-map*⟩ → ⟨*person pair*⟩
`library-created` : ⟨*rdf-map*⟩ → ⟨*date string*⟩
`library-modified` : ⟨*rdf-map*⟩ → ⟨*date string*⟩
`library-version` : ⟨*rdf-map*⟩ → ⟨*version string*⟩
`library-implementation` : ⟨*store*⟩ ⟨*rdf-map*⟩ → ⟨*impl pair*⟩
`library-location` : ⟨*store*⟩ ⟨*rdf-map*⟩ → ⟨*location*⟩

The above procedures are standard accesser procedures to different elements of a Descot library node. They can be used to quickly get pieces of the graph instead of walking the graph explicitly. ⟨*store*⟩ refers to the Descot store, and ⟨*rdf-map*⟩ refers to a specific RDF map containing the child nodes of a given library node.

The actual `descot-server` module available in the Descot source provides a generalized, format-neutral API for handling server requests. Currently, it handles node requests, mirroring requests, and provides conveniences for handling submissions of new libraries into the existing store.

The Descot Server API uses a file system hierarchy to store the RDF graph in a manner that makes it convenient to retrieve server request information. The entire graph is stored under a single ⟨*root*⟩ directory, and for any subject node with a valid URI, there exists a single file which holds the information necessary to serve a node request for that URI. The path to this file is formed by the following scheme:

```
<root>/<scheme>/<domain>/<path>[#<fragment>]
```

where ⟨*domain*⟩ is the domain of the URI with the terms reversed and separated by forward slashes rather than dots. The API provides a procedure for generating this path from a given URI:

`descot-uri->store-path` : ⟨*uri string*⟩ → ⟨*path string*⟩

and also defines a parameter `descot-store` to hold the root location.

The API also defines reader and writer parameters for the store. The reader parameter `descot-api-reader` contains a procedure

`reader` : ⟨*fname*⟩ [⟨*graph*⟩] → ⟨*rdf graph*⟩

that will read the files in the store. This allows the format of the store to be any format for which an user can provide a proper reader. This parameter defaults to `parse-srdf-file` from the `srdf` module (see the Appendix).

The `descot-api-triples-writer` parameter holds a procedure that will be used whenever a graph must be written to a file. It defaults to `write-rdf-triples->srdf` and any procedure that replaces the default should have the same signature (see the Appendix). This writer is also used when no preferred format is detected for an incoming node request. Since detection of format preference is not yet built into the API, this parameter effectively controls all RDF output from the API, and not just the format from the store.

The above parameters are used to separate the api from the format of the repository. They are not expected to change after initializing a server using this API.

`write-descot-request` : ⟨*subject uri*⟩ ⟨*port*⟩ → #⟨void⟩

`write-descot-request` handles node requests for the server and writes out the proper response to the given ⟨*port*⟩.

`write-descot-updates` : ⟨*port*⟩ → #⟨void⟩

`write-descot-updates` writes out the mirroring graph to the given port.

`write-descot-store` : ⟨*graph*⟩ → #⟨void⟩

When new libraries are submitted to a server, normally they will go through a vetting process, after which, they must be stored in the main repository database. `write-descot-store` allows a store to be written safely to the store and is the main procedure to use when adding new data to the store.

Since the API does not yet provide enough detailed access to make direct graph walking along the graph easy, a convenience procedure is exported from the server API to allow applications to read in the entire store for work.

`read-descot-store` : ⟨*root*⟩ → ⟨*RDF graph*⟩

It works with any subdirectory of the root location and the root location itself as the ⟨*root*⟩ value, so one can selectively graph pieces of a graph if necessary.

3.5 Example

The following is a complete example of a relatively self contained graph with all the information necessary to serve all the node requests. It is written in the SRDF format defined in the Appendix.

```
(= authors
   "http://descot.sacrideo.us/rdf/authors/")
(= impls
   "http://descot.sacrideo.us/rdf/impls/")
(= licenses
   "http://descot.sacrideo.us/rdf/licenses/")
(= bindings
   "http://descot.sacrideo.us/rdf/bindings/")
(= dscts
   "http://descot.sacrideo.us/10-rdf-schema#")
(= rdf
 "http://www.w3.org/1999/02/22-rdf-syntax-ns#")
(= xsd
   "http://www.w3.org/2001/XMLSchema#")
(= dsct
   "http://descot.sacrideo.us/rdf/libs/system/")

((: dsct "malloc#chez")
 ((: rdf "type") (: dscts "Library"))
 ((: dscts "name")
  (& "Garbage Collected Malloc" en))
 ((: dscts "names")
  (($ "malloc") ($ "gc-malloc")))
 ((: dscts "desc")
  ($ "Create malloced regions of memory that
      are handled by the garbage collector."))
 ((: dscts "exports") ((: bindings "gc-malloc")))
 ((: dscts "license")
  (: licenses "public-domain"))
```

```
 ((: dscts "authors") ((: authors "dybvig")))
 ((: dscts "creation")
  (^ "2009/03/08 23:33:10" (: xsd "dateTime")))
 ((: dscts "modified")
  (^ "2009/05/12 00:41:44" (: xsd "dateTime")))
 ((: dscts "copyright-year")
  (^ "2008" (: xsd "gYear")))
 ((: dscts "copyright-owner")
  (: authors "dybvig"))
 ((: dscts "contact") (: authors "arcfide"))
 ((: dscts "version") ($ "1.0"))
 ((: dscts "location")
  (* ((: rdf "type") (: dscts "CVS"))
     ((: dscts "cvs-root")
      ($ "anoncvs@anoncvs.sacrideo.us:/cvs"))
     ((: dscts "cvs-module") ($ "lib/malloc.ss"))))
 ((: dscts "implementation") (: impls "chez"))
 ((: dscts "categories") (($ "system"))))

((: licenses "public-domain")
 ((: rdf "type") (: dscts "Licenses"))
 ((: dscts "name") ($ "Public Domain")))

((: bindings "gc-malloc")
 ((: rdf "type") (: dscts "Binding"))
 ((: dscts "name") ($ "malloc"))
 ((: dscts "desc")
  ($ "Garbage Collected Malloc")))

((: authors "dybvig")
 ((: rdf "type") (: dscts "Person"))
 ((: dscts "name") ($ "R. Kent Dybvig"))
 ((: dscts "email") ($ "dyb@scheme.com"))
 ((: dscts "homepage") "http://www.scheme.com"))

((: authors "arcfide")
 ((: rdf "type") (: dscts "Person"))
 ((: dscts "name") ($ "Aaron W. Hsu"))
 ((: dscts "email") ($ "arcfide@sacrideo.us"))
 ((: dscts "homepage")
  "http://www.sacrideo.us"))

((: impls "chez")
 ((: rdf "type") (: dscts "Implementation"))
 ((: dscts "name") ($ "Chez Scheme"))
 ((: dscts "homepage") "http://www.scheme.com"))
```

If a node request came it, it would come for one of the top-level s-expressions defined above. The data transmitted back to the requesting client would be equivalent to the data contained in that top-level s-expression. That is, if a request for

```
(: dsct "malloc#chez")
```

came in to a server, it would return only the data found in the s-expression above that has

```
(: dsct "malloc#chez")
```

as the first element. The server would ignore the other top-level s-expressions.

4. Related Work

Since Descot only describes a library and does not attempt to make it portable across implementations or languages, efforts to make portable code, such as those from Snow [12] and especially module systems like R6RS libraries [29] contribute invaluable features to a complete repository system.

Snow is only one of many repositories that exist in Scheme, each with its own unique features and focus. These include library suites such as SLIB [20] and implementation-specific repositories such as those found for PLT [26], Chicken [31], and Bigloo [30].

Other attempts at portable library repositories include CSAN [9] and CxAN [27]. The latter is unique because it is not a Scheme specific project.

Implementations often support libraries internally without making them into separate libraries, or they may package libraries with their distributions, which makes it interesting to deal with that information. Descot is general enough to represent these internal libraries, which almost all Scheme implementations have, even though they are not generally considered repositories [22, 11, 25].

Other projects have created distributed networks of repositories quite successfully, though not specifically focused on library code distribution. The Debian packaging system [1], often known as apt, caches server information on the clients to enable multiple repositories to be used by one client. The client can then download the desired packages and install them as appropriate. System such as the RPM-based [10] yum [14] also behave in a similiar manner. An user specifices a series of repositories to use, and the client caches information about the software packages available from the repositories listed. Sites such as RPMfind [4] also make packages available via web browser. These clients will often scan many repositories over all different distributions to obtain their indexes.

While the above are similiar to Descot by their distributed nature, the packages they references are actual software packages and contain all the binaries or source code inside them. The BSD family of operating systems (and Gentoo, which follows a similar pattern [13]) uses a series of files that contain metadata about how to build and install a given software package. Descot's metadata representation more closely resembles these so called ports systems than the packaging used by systems like apt or yum. When, say, an OpenBSD user wishes to build a package, rather than install it via binary package, the user would navigate to a prefilled filesystem containing port metadata. The user would then run a command that would fetch, build, and install the package [16]. Similar tools could be made for Descot repositories.

5. Conclusion

The Descot system described above provides the means by which fragmented or diverse communities can cooperate and leverage development efforts that previously existed in isolation of one another. Since most communities do not lack for tools or repositories of code, but rather, a means of common access, Descot focuses entirely on fostering the communication among existing systems, rather than trying to rewrite existing tools and change previous workflows. Since Descot is extensible and dynamic, it can fit into a wide range of domains, and can adapt to handle the needs of a community, rather than trying to fit different communities or sub-cultures into a single methodology. Descot is built on common, well documented technologies and so should easily travel where less standards-based systems may not. Descot provides an open, clearly specified infrastructure so that communities can collaborate together and avoid redundant work. It provides the convenience of central code distribution

without forcing large, top-down changes on a community that may not respond well to such pressure.

6. Acknowledgments

The author would like to thank Kent Dybvig for his comments, which led to improvements in the presentation this this paper.

References

[1] Osamu Aoki. *Debian Reference*, June 2009. *http://www.debian.org /doc/manuals/debian-reference/*.

[2] Dave Beckett and Jeen Broekstra. Sparql query results xml format. W3c recommendation, W3C, January 2008. *http://www.w3.org/TR/rdf-sparql-XMLres/*.

[3] David Beckett and Tim Berners-Lee. Turtle - terse rdf triple language. W3c team submission, W3C, January 2008. *http://www.w3.org/TeamSubmission/turtle/*.

[4] Fabrice Bellet. Rpmfind, June 2009. *http://www.rpmfind.net*.

[5] Dan Brickley and R. V. Guha. Rdf vocabulary description language 1.0: Rdf schema. W3c recommendation, W3C, February 2004. *http://www.w3.org/RDF/*.

[6] Taylor Campbell. foof loop, June 2009. *http://mumble.net/ ˜ campbell/darcs/foof-loop/loop.scm*.

[7] Kendall Grant Clark, Lee Feigenbaum, and Elias Torres. Sparql protocol for rdf. W3c recommendation, W3C, January 2008. *http://www.w3.org/TR/rdf-sparql-protocol/*.

[8] William Clinger and Jonathan Rees. *Revised^4 Report on the Algorithmic Language Scheme*, September 1991. ftp://ftp.cs.indiana.edu/pub/scheme-repository/doc/standards/r4rs.ps.gz.

[9] CSAN. Comprehensive scheme archive network, June 2009. *http://www.cliki.net/Community*.

[10] Alexandre de Abreu. *All you have to know about RPM*, March 2004. *http://fedoranews.org/alex/tutorial/rpm/*.

[11] R. Kent Dybvig. *Chez Scheme Version 7 User's Guide*. Cadence Research Systems, July 2007. *http://www.scheme.com/csug7/*.

[12] Marc Feeley. *Scheme Now! Documentation*, June 2009. *http://snow.iro.umontreal.ca/?tab=Documentation*.

[13] Gentoo Foundation. Gentoo linux, June 2009. *http://www.gentoo.org*.

[14] Michael Hideo. *Red Hat Enterprise Linux 5 Deployment Guide*. Red Hat Inc., Raleigh, NC, 5 edition, November 2008. *http://www.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/5/html/Deployment _Guide/index.html*.

[15] Jarkko Hietaniemi. Comprehensive perl archive network. http://www.cpan.org.

[16] Nick Holland. *The OpenBSD packages and ports system*. OpenBSD, May 2009. *http://www.openbsd.org*.

[17] Aaron W. Hsu. Descot rdf schema. Rdf schema, May 2009. *http://descot.sacrideo.us/10-rdf-schema*.

[18] Aaron W. Hsu. Descot technical documentation. Programmer's documentation, May 2009. *http://descot.sacrideo.us*.

[19] IEEE. *1178-1990 IEEE Standard for the Scheme Programming Lanuage*, 1990.

[20] Aubrey Jaffer. *SLIB: The Portable Scheme Library*, February 2008. *http://people.csail.mit.edu/jaffer/slib_toc.html*.

[21] Richard Kelsey, William Clinger, and Jonathan Rees. *Revised^5 Report on the Algorithmic Language Scheme*, February 1998. http://www.schemers.org/Documents/Standards /R5RS/r5rs.ps.

[22] Richard Kelsey, Jonathan Rees, and Mike Sperber. *The Incomplete Scheme 48 Reference Manual for release 1.8*, January 2008. *http://www.s48.org/1.8/manual/manual.html*.

[23] Oleg Kiselyov. Scheme hash, June 2009. *http://okmij.org/ftp/Scheme*.

[24] Frank Manola and Eric Miller. Rdf primer. W3c recommendation, W3C, February 2004. *http://www.w3.org/TR/rdf-primer/*.

[25] Massachusetts Institute of Technology. *MIT/GNU Scheme 7.7.90+ Reference Manual*, 2008. *http://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/index.html*.

[26] Jacob Matthews. PLaneT: Automatic package distribution. Reference Manual PLT-TR2009-planet-v4.2, PLT Scheme Inc., June 2009. *http://plt-scheme.org/techreports/*.

[27] Hans Oesterholt. Cxan, July 2004. *http://cxan.sourceforge.net/*.

[28] Eric Prud'hommeaux and Andy Seaborne. Sparql query language for rdf. W3c recommendation, W3C, January 2008. *http://www.w3.org/TR/rdf-sparql-query/*.

[29] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten. *Revised^6 Report on the Algorithmic Language Scheme*, September 2007. http://www.r6rs.org/final/r6rs.pdf.

[30] Vladimir Tsichevski. *Bigloo libraries*, December 2003. *http://bigloo-lib.sourceforge.net/*.

[31] Felix Winkelmann. *The CHICKEN User's Manual*, April 2009. *http://chicken.wiki.br/man/4/The User's Manual*.

Appendix: SRDF Format

SRDF is an s-expression based format for describing RDF graphs. It is meant to be mostly equivalent in its form to Turtle. Since the language is S-expression based, it is easier for Scheme and Lisp parsers to parse it. Parsers for other languages can also be written very easily. This makes it particularly nice for use in automated systems or in areas where S-expressions are the natural representation format. SRDF is designed to work for most Scheme's `read` procedures.

SRDF documents are composed of a series of RDF triples and, possibly, prefix definitions. Prefixes take the form `(= name "uri")`, and associate a given Scheme symbol with a URI string. Otherwise, the form is an RDF triple or a set of triples.

Normal triples are just a list of three elements, each a URI. Multiple triples with the same subject can be declared in one expression by replacing the list that would hold the single predicate and object with a list of such predicates and objects. Likewise, one can specify more objects to be associated with a given subject and predicate by doing the same thing with the object list, and replacing the list tail that would normally hold the object with a list of such objects.

If the second element of a predicate pair contains a list of objects, this represents a collection of objects, and is created in the same way that a turtle collection syntax is created: by associating a series of blank nodes with the right predicates with each of the objects listed.

An object that differs from a list of objects that are each associated with the subject and predicate. The following is an instance of the former:

```
("subject-uri" "predicate-uri"
  ("object1" "object2" ...))
```

Whereas the following is an instance of the latter:

```
("subject-uri" "predicate-uri"
  "object1"
  "object2"
  "object3")
```

Normal RDF triples take the form:

```
("subject-uri" "predicate-uri" "object-uri")
```

A blank node may be inlined into the graph by using a '*' as the beginning symbol in an object context like so:

```
("subject" "pred" (* "pred" "object"))
```

Of course, blank nodes may have anything that is a valid predicate `cdr` as its `cdr` so the following is also valid:

```
("subject" "pred"
 (* ("pred1" "object1") ("pred2" "object2")))
```

URIs may be described by their full path names as strings, as prefix combined paths, or as blank node paths. The following are all valid URIs:

```
"http://some.domain/path/to#blah"
"blah"
(: prefix "blah")
(_ "uniqueid")
```

We use ':' for prefixes and '_' for blank nodes. In addition to URIs, we permit literals as valid `cars` for objects. Literals can be strings, numbers, booleans, or may be strings with either languages or types associated with them. The following are examples of languages and types, respectively:

```
($ "Language unspecified.")
(& "English Sentence lies here." en)
(^ "2008/01/03 14:00" (: xsd "date"))
```

The following is a fairly formal BNF grammar with the exception of tokens such as strings, numbers, and booleans being undefined and presumed to be defined lexical values. Additionally, we define S-expression in terms of atoms and pairs, so the BNF grammar is also defined in the "longhand" notation for pairs and lists. This means that while the BNF Grammar states something like `("subj" . ("pred" . ("obj" . ())))` as the valid simplistic RDF triple, it is also legal in practice to use the shorthand version of this: `("subj" "pred" "obj")`

⟨*rdf sexp*⟩→ ⟨*rdf triple*⟩ | ⟨*rdf triple*⟩ ⟨*rdf sexp*⟩

⟨*rdf triple*⟩→ **'('** ⟨*uri*⟩ **'.'** ⟨*rdf subject tail*⟩ **')'**
  | **'('** **'='** name string **')'**

⟨*rdf subject tail*→ ⟨*rdf predicate*⟩
  | ⟨*rdf predicate list*⟩

⟨*rdf pred list*⟩→ **'()'**
  | **'('** ⟨*rdf predicate*⟩ **'.'** ⟨*rdf pred list*⟩ **')'**

⟨*rdf predicate*⟩→ **'('** ⟨*uri*⟩ **'.'** ⟨*rdf object list*⟩ **')'**

⟨*rdf object list*⟩→ **'()'**
  | **'('** ⟨*rdf object*⟩ **'.'** ⟨*rdf object list*⟩ **')'**

⟨*rdf object*⟩→ ⟨*uri*⟩ | ⟨*literal*⟩
  | ⟨*rdf object list*⟩ | ⟨*blank node list*⟩

⟨*blank node list*⟩→ **'('** **'*'** **'.'** ⟨*rdf subject tail*⟩ **')'**

⟨*uri*⟩→ *uri*
  | **'('** **':'** **'.'** ⟨*uri list*⟩ **')'**
  | **'('** **'_'** *name* **')'**

⟨*uri list*⟩→ **'()'**
  | **'('** ⟨*uri name*⟩ **'.'** ⟨*uri list*⟩ **')'**

⟨*uri name*⟩→ *uri* | *name*

⟨*literal*⟩→ *number* | *boolean*
  | **'('** **'$'** *string* **')'**
  | **'('** **'&'** *string name* **')'**
  | **'('** **'^'** *string* ⟨*uri*⟩ **')'**

Keyword and Optional Arguments in PLT Scheme

Matthew Flatt
University of Utah and PLT
mflatt@cs.utah.edu

Eli Barzilay
Northeastern University and PLT
eli@ccs.neu.edu

Abstract

The **lambda** and procedure-application forms in PLT Scheme support arguments that are tagged with keywords, instead of identified by position, as well as optional arguments with default values. Unlike previous keyword-argument systems for Scheme, a keyword is not self-quoting as an expression, and keyword arguments use a different calling convention than non-keyword arguments. Consequently, a keyword serves more reliably (e.g., in terms of error reporting) as a lightweight syntactic delimiter on procedure arguments. Our design requires no changes to the PLT Scheme core compiler, because **lambda** and application forms that support keywords are implemented by macros over conventional core forms that lack keyword support.

1. Using Keyword and Optional Arguments

A rich programming language offers many ways to abstract and parameterize code. In Scheme, first-class procedures are the primary means of abstraction, and procedures are unquestionably the right vehicle for parameterizing code with respect to a few run-time values. For parameterization over larger sets of values, however, Scheme procedures quickly become inconvenient.

Keyword and optional arguments support tasks that need more arguments than fit comfortably into procedures, but where radically different forms—such as **unit** or **class** in PLT Scheme—are too heavyweight conceptually and notationally. At the same time, keyword and optional arguments offer a smooth extension path for existing procedure-based APIs. Keyword arguments can be added to a procedure to extend its functionality without binding a new identifier (which always carries the danger of colliding with other bindings) and in a way that composes with other such extensions.

Keyword arguments in PLT Scheme are supported through a straightforward extension of the **lambda**, **define**, and application forms. Lexically, a keyword starts with `#:` and continues in the same way as an identifier; for example, `#:color` is a keyword.[1]

A keyword is associated with a formal or actual argument by placing the keyword before the argument name or expression. For example, a `rectangle` procedure that accepts two by-position arguments and one argument with the `#:color` keyword can be written as

```
(define rectangle
  (lambda (width height #:color color)
    ....))
```

or

```
(define (rectangle width height #:color color)
  ....)
```

This `rectangle` procedure could be called as

```
(rectangle 10 20 #:color "blue")
```

A keyword argument can be in any position relative to other arguments, so the following two calls are equivalent to the preceding one:

```
(rectangle #:color "blue" 10 20)
(rectangle 10 #:color "blue" 20)
```

The `#:color` formal argument could have been in any position among the arguments in the definition of `rectangle`, as well. In general, keyword arguments are designed to look the same in both the declaration and application of a procedure.

In a procedure declaration, a formal argument can be paired with a default-value expression using a set of parentheses—or, by convention, square brackets. The notation for a default-value expression is the same whether the argument is by-position or by-keyword. For example, a `rectangle`'s height might default to its width and its color default to pink:

```
(define (rectangle width
                   [height width]
                   #:color [color "pink"])
  ....)
```

This revised `rectangle` procedure could be called in any of the following ways:

```
(rectangle 10)
(rectangle 10 20)
(rectangle 10 20 #:color "blue")
(rectangle 10 #:color "blue")
(rectangle #:color "blue" 10 20)
(rectangle #:color "blue" 10)
(rectangle 10 #:color "blue" 20)
```

Our goals in a design for keyword and optional arguments include providing especially clear error messages and enforcing a consistent syntax for keyword arguments. Toward these goals, two aspects of our design set it apart from previous approaches in Lisp and Scheme:

- Keywords are distinct from symbols, and they are not self-quoting as expressions.

  For example, the form

  ```
  #:color
  ```

  in an expression position is a syntax error, while

  ```
  (rectangle #:color "blue")
  ```

[1] See Section 7.7 for a discussion on this choice of keyword syntax.

Proceedings of the 2009 Scheme and Functional Programming Workshop
California Polytechnic State University Technical Report CPSLO-CSC-09-03

is a call to `rectangle` with the `#:color` argument `"blue"`. In the latter case, the procedure-application form treats the `#:color` keyword as an argument tag, and not as an expression. Every keyword in an application must be followed by a value expression, so the form

```
(rectangle #:color #:filled? #f)
```

is rejected as a syntax error, because `#:color` lacks an argument expression; if keywords could be expressions, the call would be ambiguous, because `#:filled?` might be intended as the `#:color` argument to `rectangle`.

- Keywords are not passed as normal arguments to arbitrary procedures, where they might be confused with regular procedure arguments. Instead, a different calling convention is used for keyword arguments.

  For example,

  ```
  (cons #:color "blue")
  ```

  does not create a pair whose first element is a keyword and second element is a string. Evaluating this expression instead reports a run-time error that `cons` does not expect keyword arguments.

Although a keyword is not self-quoting as an expression, a keyword is a first-class value in PLT Scheme. A keyword can be quoted to produce a valid expression, as in `'#:color` and `(cons '#:color "blue")`, where the latter creates a pair whose first element is a keyword. Keyword values and quoted-keyword expressions are useful for creating a procedure that accepts arbitrary keyword arguments and processes them explicitly. Keyword values are also useful in reflective operations that inspect the keyword requirements of a procedure. By convention, PLT Scheme programmers do not use keywords for run-time enumerations and flags, leaving those roles to symbols and reserving keywords for syntactic roles.

The rest of the paper proceeds as follows. Section 2 describes the syntax and semantics of keyword and optional arguments in PLT Scheme. Section 4 describes our implementation of keyword arguments. Section 5 provides some information on the performance of keyword and optional arguments. Section 6 reports on our experience using keywords in PLT Scheme. Section 7 describes previous designs for keywords in Lisp and Scheme and relates them to our design.

2. Syntax and Semantics

Figure 1 shows the full syntax of PLT Scheme's **lambda**. In a ⟨kw-formals⟩, all ⟨id⟩s must be distinct, including ⟨rest-id⟩, and all ⟨keyword⟩s must be distinct. A required non-keyword argument (i.e., the first case of ⟨formal-arg⟩) must not follow an optional non-keyword argument (i.e., the second case of ⟨formal-arg⟩).

A **lambda** form that is constructed using only the ⟨id⟩ form of ⟨fomal-arg⟩ has the same meaning as in standard Scheme (Sperber 2007). A **lambda** form that uses only the ⟨id⟩ and `[`⟨id⟩ ⟨default-expr⟩`]` forms of ⟨formal-arg⟩ can be converted to an equivalent **case-lambda** form; the appendix shows the conversion precisely in terms of **syntax-rules**. For each optional argument that is not supplied in an application of the procedure, the corresponding ⟨default-expr⟩ is evaluated just before the procedure body is evaluated. The environment of each ⟨default-expr⟩ includes the preceding arguments, and if multiple ⟨default-expr⟩s are evaluated, then they are evaluated in the order that they are declared. When a "rest argument" is declared after optional arguments, arguments in an application are first consumed by the optional-argument positions, so the rest argument is non-empty only when more arguments are provided that the total number of required and optional arguments.

```
(lambda ⟨kw-formals⟩ ⟨body⟩ ...+)

⟨kw-formals⟩ = (⟨formal-arg⟩ ...)
             | (⟨formal-arg⟩ ...+ . ⟨rest-id⟩)
             | ⟨rest-id⟩

⟨formal-arg⟩ = ⟨id⟩
             | [⟨id⟩ ⟨default-expr⟩]
             | ⟨keyword⟩ ⟨id⟩
             | ⟨keyword⟩ [⟨id⟩ ⟨default-expr⟩]
```

... means "zero or more," ...+ means "one or more," ⟨id⟩ or ⟨rest-id⟩ matches an identifier, ⟨expr⟩ or ⟨default-expr⟩ matches an expression, ⟨keyword⟩ matches a keyword, and ⟨body⟩ matches a definition or expression in an internal-definition context

Figure 1: Extended grammar for **lambda**

```
(⟨proc-expr⟩ ⟨actual-arg⟩ ...+)
⟨actual-arg⟩ = ⟨expr⟩
             | ⟨keyword⟩ ⟨expr⟩
```

Figure 2: Extended grammar for procedure application

```
> (define polygon
    (lambda (n [side-len (/ 12 n)] . options)
      (list n side-len options)))
> (polygon)
procedure polygon: no clause matching 0 arguments
> (polygon 3)
(3 4 ())
> (polygon 3 7)
(3 7 ())
> (polygon 3 7 'solid 'smooth)
(3 7 (solid smooth))
```

When the ⟨keyword⟩ ⟨id⟩ or ⟨keyword⟩ `[`⟨id⟩ ⟨default-expr⟩`]` forms of ⟨formal-arg⟩ are used to construct a **lambda** expression, the resulting procedure accepts keyword-tagged arguments in addition to the arguments that would be accepted without the keyword-tagged arguments. Arguments using the ⟨keyword⟩ ⟨id⟩ form are required, while arguments using the ⟨keyword⟩ `[`⟨id⟩ ⟨default-expr⟩`]` form are optional. As with the keywordless `[`⟨id⟩ ⟨default-expr⟩`]` form, each keyword-tagged ⟨default-expr⟩ is evaluated for a given application of the procedure if no actual argument is tagged with the corresponding ⟨keyword⟩, and the preceding argument ⟨id⟩s are in the environment of each ⟨default-expr⟩. When ⟨default-expr⟩s are evaluated for multiple arguments, they are evaluated in the order declared in the **lambda** expression, independent of whether the arguments have a keyword tag or the order of keyword tags on actual arguments. Actual arguments that are tagged with a keyword can be supplied in any order with respect to each other and with respect to by-position arguments.

```
> (define polygon
    (lambda (n [side-len (/ 12 n)]
             #:color [color "blue"]
             #:rotate theta
             . options)
      (list n side-len color theta options)))
> (polygon 4)
polygon: requires an argument with keyword #:rotate, not
supplied; arguments were: 4
> (polygon 4 #:rotate 0)
(4 3 "blue" 0 ())
> (polygon 4 7 #:rotate 0 #:color "red" 'solid)
(4 7 "red" 0 (solid))
```

The above examples use the extended syntax of procedure applications shown in Figure 2, which allows arguments tagged with keywords. Each ⟨keyword⟩ in an application must be distinct. Crucially, the grammar of ⟨expr⟩ in PLT Scheme (not shown here) does not include an unquoted ⟨keyword⟩, so the grammar for procedure application is unambiguous.

Naturally, the result of ⟨proc-expr⟩ in an application must be a procedure. For each keywordless argument ⟨expr⟩, the result is delivered to the procedure as a by-position argument, while each other ⟨expr⟩ is provided with the associated ⟨keyword⟩. PLT Scheme always evaluates the sub-expressions of a procedure application left-to-right, independent of whether the argument is tagged with a keyword. If the applied procedure evaluates ⟨default-expr⟩s for unsupplied arguments, it does so only after all of the ⟨expr⟩s in the procedure application are evaluated. Similarly, the expected and supplied arguments (in terms of arity and keywords) are checked after all of the argument ⟨expr⟩s are evaluated but before the any ⟨default-expr⟩s would be evaluated (so no ⟨default-expr⟩s are evaluated if the number of supplied by-position arguments is wrong, if a required keyword argument is missing, or if an unexpected keyword is supplied).

The **define** shorthand for procedure is extended in the obvious way to support keyword and optional arguments. PLT Scheme also supports the MIT curried-function shorthand, which composes seamlessly with keyword and optional arguments.

```
> (define ((rect w [h w] #:color [c "pink"])
           canvas x y)
    (set-pen-color! canvas c)
    (draw-rectangle! canvas x y w h))
> ((rect 10 #:color "blue") screen 0 0)
```

In addition to the **lambda**, **define**, and application syntactic forms, our design extends and adds a few procedures. An extended **apply** procedure accepts arbitrary keyword arguments, and it propagates them to the given procedure.

```
> (apply polygon 4 7 #:rotate 0 '(solid smooth))
(4 7 "blue" 0 (solid smooth))
```

Keyword arguments to **apply** are analogous to arguments between the procedure and list argument in the standard **apply**; that is, they are propagated directly as provided. The **keyword-apply** procedure generalizes **apply** to accept a list of keywords and a parallel list of values, which are analogous to the last argument of **apply**.

```
> (keyword-apply polygon
                 '(#:color #:rotate)
                 '("blue" 0)
                 4 7
                 '(solid smooth))
(4 7 "blue" 0 (solid smooth))
```

The list of keywords supplied to **keyword-apply** must be sorted alphabetically, for reasons explained in Section 4.

The **make-keyword-procedure** procedure constructs a procedure like **apply** that accepts arbitrary keyword arguments. The argument to **make-keyword-procedure** is a procedure that accepts a list of keywords for supplied arguments, a parallel list of values for the supplied keywords, and then any number of by-position arguments.

```
> (define trace-call
    (make-keyword-procedure
     (lambda (kws kw-vals proc . args)
       (printf ">>~s ~s ~s<<\n" kws kw-vals args)
       (keyword-apply proc kws kw-vals args))))
> (trace-call polygon 6 #:rotate 0)
>>(#:rotate) (0) (6)<<
(6 2 "blue" 0 ())
```

Finally, the reflection operations **procedure-arity** and **procedure-reduce-arity** in PLT Scheme inspect or restrict the arity of a procedure. The additional procedures **procedure-keywords** and **procedure-reduce-keyword-arity** extend the set of reflection operators to support keywords. The **procedure-keywords** procedure reports the keywords that are required and allowed by a given procedure. The **procedure-reduce-keyword-arity** procedure converts a given procedure with optional keyword arguments to one that allows fewer of the optional arguments and/or makes some of them required. A typical use of **procedure-reduce-keyword-arity** adjusts the result of **make-keyword-procedure** (for which all keywords are optional) to give it a more specific interface.

PLT Scheme does not extend **case-lambda** to support keyword or optional arguments; the extension would be straightforward, but there has been no demand. Similarly, continuations in PLT Scheme do not support keyword arguments. Extended variants of **call-with-values**, **values**, and **call/cc** procedures could support keyword results and continuations that accept keyword arguments. We have not tried that generalization, but an implementation could use continuation marks (Clements and Felleisen 2004) that are installed by **call-with-values** and used by **values** and **call/cc** to connect a keyword-accepting continuation with its application or capture.

3. Keywords in Other Syntactic Forms

The PLT Scheme macro system treats keywords in the same way as a number or a boolean. For example, a pattern for a macro can match a literal keyword:

```
(define-syntax show
  (syntax-rules ()
    [(_ #:canvas c expr ...)
     (call-with-canvas c (lambda () expr ...))]
    [(_ expr ...)
     (show #:canvas default-canvas expr ...)]))
```

This macro recognizes an optional **#:canvas** specification before a sequence of drawing expressions to select the target of the drawing operations. For example, the first pattern in the **syntax-rules** form matches

```
(show #:canvas my-canvas (draw-point! 0 0))
```

while the second clause matches

```
(show (draw-point! 0 0))
```

The second clause also matches

```
(show #:dest my-canvas (draw-point! 0 0))
```

in which case **#:dest** is used as an expression, and a syntax error after expansion reports the misuse of **#:dest**. That is, the pattern matcher for macros does not constrain arbitrary pattern variables against matching literal keywords. The error message "**#:dest** is not an expression" is less clear than "the **show** form expects **#:canvas** and does not recognize **#:dest**," and a **syntax-case** implementation of **#:draw** could more thoroughly check its sub-forms. Similarly, the first clause in the **show** macro does not match

```
(show (draw-point! 0 0) #:canvas my-canvas)
```

since it recognizes **#:canvas** only at the beginning of the form. Again, a **syntax-case** implementation of **show** could allow **#:canvas** in later positions, if desired.[2]

[2] A better solution would be a variant of **syntax-rules** that handles keyword constraints and ordering automatically—along with related constraints, such as requiring an identifier.

```
(define-struct ⟨id⟩ (⟨field⟩ ...) ⟨struct-option⟩ ...)

        ⟨field⟩ = ⟨field-id⟩
                | [⟨field-id⟩ ⟨field-option⟩ ...]

⟨struct-option⟩ = #:super ⟨super-expr⟩
                | #:auto-value ⟨auto-expr⟩
                | #:property ⟨prop-expr⟩ ⟨val-exr⟩
                | #:transparent

 ⟨field-option⟩ = #:mutable
                | #:auto
```

Figure 3: Partial grammar for PLT Scheme's **define-struct**

Syntactic forms in PLT Scheme that use keywords include the **define-struct** form and the **->*** contract constructor. Both are typical in that they allow keywords only in specific places (instead of anywhere between the form's parentheses). For example, the syntax of **define-struct** is shown in Figure 3, where keyword-tagged options appear only within ⟨field⟩s and after the ⟨field⟩ sequence. In the allowed positions, however, keywords are used in a more flexible way than in an application form; the **#:transparent**, **#:mutable**, and **#:auto** keywords need no corresponding argument expression, while the **#:property** keyword is followed by two expressions. This combination of constraints (i.e., requiring keywords in certain positions) and generalizations (i.e., allowing different numbers of expressions associated with a keyword) compared to procedure application is the prerogative of a syntactic form.

At the same time, **define-struct** relies on the prohibition of unquoted keywords as expressions to provide good error messages when parsing a set of ⟨struct-option⟩s, such as when the **#:super** keyword lacks a corresponding expression before the next keyword. The consistent role of keywords as non-expression delimiters has encouraged the use of keywords within syntactic forms for PLT Scheme.

4. Implementation

Although **lambda** and the procedure-application form in PLT Scheme support keyword and optional arguments, the core compiler does not directly support them. Instead, support for keyword and optional arguments is implemented as a macro in a library, in the same way that **unit** and **class** are implemented as macros over the core **lambda** form. The only core support for keywords is a keyword datatype and reader syntax.

The library is implemented so that the keyword-supporting application form is equivalent to the core application form when no keywords are supplied, and a **lambda** form with no keyword or optional arguments is equivalent to the core **lambda** form. Furthermore, a procedure with only optional keyword arguments can be called through the core application form. These constraints on the design preserve the performance of keywordless procedure applications and provide good interoperability between libraries that use and do not use keyword-supporting syntactic forms.

A PLT Scheme library can implement an extended application form, because an application form implicitly uses the **#%app** binding in its lexical environment. For example, in

```
(require (rename-in scheme [#%app orig-#%app]))
(define-syntax-rule (#%app expr ...)
  (begin
    (orig-#%app printf "at ~s\n" '(expr ...))
    (orig-#%app expr ...)))
(+ 1 (+ 2 3))
```

each application of **+** prints debugging information before evaluating the application:

```
at (+ 1 (+ 2 3))
at (+ 2 3)
6
```

The library that implements keyword and optional arguments supplies an **#%app** macro in addition to **lambda** and **define** to replace the core bindings. The replacement macros expand a keyword-supporting **lambda**, **#%app**, or **define** into a combination of primitive forms and run-time functions (such as **make-keyword-procedure**) that implement keyword arguments.

To allow procedures with optional keywords to be applied through the core application form, the implementation relies on a second PLT Scheme facility that predates support for keywords: applicable structure types. When the core application form encounters a value to apply that is not a procedure, it checks whether the value is an instance of a structure type that has an associated application operation (which is itself represented as a procedure). If so, it uses the associated operation to apply the structure to the given arguments. For example, another way to create noisy procedure applications is to wrap the base procedure in a **traced** structure:

```
(define-struct traced (f)
  #:property prop:procedure ; => applicable
             (lambda (t . args)
               (let ([f (traced-f t)])
                 (printf "~s\n" (cons f args))
                 (apply f args))))
(define traced-cons (make-traced cons))
(traced-cons 1 2)
```

Internally, the keyword-handling part of a procedure is represented by a core procedure that accepts a list of keywords, a list of corresponding values, and then the by-position arguments—just like a procedure given to **make-keyword-procedure**. This internal representation is wrapped in an applicable structure, where the application operation (which is used by a non-keyword application form) calls the internal procedure with empty keyword and keyword-value lists. The application form with keywords, meanwhile, extracts the internal procedure and applies it to non-empty keyword and value lists. The list of keywords is always sorted alphabetically, so that the supplied keywords can be checked against an expected set without sorting or searching when the internal procedure is called. The internal procedure is not directly accessible, since it is wrapped in an opaque structure.

The keyword-supporting application form sorts a set of supplied keywords at compile time. Compile-time sorting is possible because keywords in an application are statically apparent; keywords that act as argument tags are syntactic literals, while expressions that produce keyword values are never treated as argument tags. The list of keywords also can be allocated once per call site (as a quoted list of keywords), while the list of corresponding values must be allocated for each call. This detail explains why the internal representation of a keyword-accepting procedure accepts a list of keywords separate from the list of arguments.

Finally, an applicable structure that represents a keyword procedure has an associated property to generates a string description of the procedure's arity and expected keywords. This property is used when a procedure that accepts only optional keyword arguments is applied to the wrong number of by-position arguments. In that case, the arity-mismatch error not only describes the expected number of by-position arguments, but also the optional keyword arguments. This arity-description property is built into the run-time system, since it must be used when reporting an arity mismatch from the core application form.

5. Performance

In PLT Scheme, application of a keyword-accepting procedure is somewhat slower than a keywordless procedure, but the design presented here significantly outperforms our earlier, more conventional implementation. The performance cost relative to plain procedures has several causes: applications without optional keywords must extract a procedure from an applicable structure; keyword arguments are always collected into a list; keyword arguments must be checked against the expected set of keywords; and the compiler currently cannot inline keyword-accepting procedures. Procedures with optional (but no keyword) arguments expand to **case-lambda**, in which case the relative cost is lower (no applicable structure, no keyword checking, and not collecting arguments into a list), but the compiler currently does not inline multi-clause **case-lambda** procedures.

The following loops serve as rough micro-benchmarks:

```
; A plain procedure
(define (sub1 n) (- n 1))
(let loop ([n 10000000])
  (unless (zero? n) (loop (sub1 n))))

; With an optional argument
(define (sub1/opt [n 0]) (- n 1))
(let loop ([n 10000000])
  (unless (zero? n) (loop (sub1/opt n))))

; With unsupplied keyword argument
(define (sub1/kw/unused n #:m [m 1]) (- n m))
(let loop ([n 10000000])
  (unless (zero? n) (loop (sub1/kw/unused n))))

; Pass the argument in a list
(define (sub1/list nl) (- (car nl) 1))
(let loop ([n 10000000])
  (unless (zero? n) (loop (sub1/list (list n)))))

; With a required keyword argument
(define (sub1/kw #:n n) (- n 1))
(let loop ([n 10000000])
  (unless (zero? n) (loop (sub1/kw #:n n))))

; Required and unsupplied optional
(define (sub1/kw2 #:n n #:m [m 1]) (- n m))
(let loop ([n 10000000])
  (unless (zero? n) (loop (sub1/kw2 #:n n))))

; Many optional keywords
(define (sub1/kws #:a [a 0] #:n [n 5]
                  #:q [q 0] #:z [z 1])
  (- n z))
(let loop ([n 10000000])
  (unless (zero? n)
    (loop (sub1/kw4 #:n n #:z 1))))
```

Since the variations of **sub1** merely perform a fixnum subtraction that will be inlined by the compiler, the micro-benchmarks compare just the overhead of different forms of procedure application. The run times for these versions are shown in Figure 4. For those runs, the benchmarks are executed outside of a module, where the compiler cannot inline definitions (but it can still inline the subtraction operation).

The "optional" case demonstrates the cost of **case-lambda** versus **lambda**, while the "unused keyword" case demonstrates the overhead of an applicable structure. The "list" case demonstrates the overhead of putting a single argument into a list and extracting it in the called function, as happens to a keyword argument in our implementation. The "keyword" case demonstrates the additional overhead of checking provided keyword arguments against the expected set. The "keywords and unused" case demonstrates how the checking overhead grows with both required and optional keywords, and the "many optional" case demonstrates how the overhead grows as additional optional keywords are added.

program	*CPU time (msec)*	*relative*
plain	327	1.0
optional	348	1.0
unused keyword	784	2.3
list	503	1.5
keyword	1115	3.4
required plus optional	1470	4.4
many optional	1999	6.1

Figure 4: Micro-benchmark results for PLT Scheme 4.2.1, on an 2GHz Core Duo MackBook running Mac OS X 10.5.7; results are median run times over three runs as measured using the **time** form

implementation	*plain*	*...hide*	*keywords*	*...hide*	*many kws*
PLT	48	327	1115	1103	1999
Old PLT	48	327	2869	2866	7891
Gambit-C, default	221	222	989	992	1437
Gambit-C, fast	43	118	897	930	1287
Chicken, default	1066	1079	2478	2502	7881
Chicken, fast	8	353	1203	1430	4529
R6RS, Ikarus	100	100	130*	1237	2054
R6RS, Larceny	66	143	66*	2237	4752
R6RS, PLT	50	361	50*	4644	9001
SBCL	126	217	232	332	473
Allegro CL	150	230	450	520	780

Figure 5: Micro-benchmark results on an 2GHz Core Duo MackBook running Mac OS X 10.5.7; PLT Scheme version 4.2.1; Gambit-C version 4.4.0, with **(declare (standard-bindings) (block) (fixnum) (not safe))** for the "fast" variant; Chicken version 4.0.0 with the **-Ob** compiler flag for the "fast" variant; Ikarus version 0.0.4-rc1+ revision 1827; Larceny version 0.97b1; SBCL version 1.0.23; Allego CL express edition version 8.1; both SBCL and Alllegro CL use **(declaim (optimize (speed 3) (safety 1) (space 0) (debug 0)))**; results are median run times over three runs as measured using a **time** form

As a further check on the performance of our keyword implementation, we provide a comparison to several other implementations:

- An older and more conventional implementation of keyword arguments in PLT Scheme, where keywords are self-quoting and keywords are passed as normal procedure arguments.
- Keyword-argument support as provided by Gambit-C (Feeley 2009) and Chicken (Winkelmann et al. 2009), first with the default compiler settings, and then with settings for faster performance.
- Eddington's R6RS library for keyword procedures[3] as run in Ikarus (Ghuloum 2009), Larceny (Clinger et al. 2009), and PLT Scheme.
- SBCL (SBCL 2009) and Allego CL (Franz, Inc. 2009) using Common Lisp standard keyword functions (Steele 1990).

For each implementation, Figure 5 reports run times for the plain, single-keyword, and many-keyword micro-benchmarks. The plain and single-keyword benchmarks are each run in two ways: one

[3] `http://bazaar.launchpad.net/~derick-eddington/scheme-libraries/xitomatl/files`, revision 180

with a direct use of a **define**d function within a compilation unit (e.g., within a module), and another where the function name is **define**d as **#f** and then **set!**ed to the function (or, in the case of Common Lisp, **setf**ed and then called via **funcall**). The latter corresponds to the *...hide* column, and the intent is to defeat inlining and other static analyses. We measure this difference because our approach to implementing keywords, if ported to other systems, might discourage static analysis. For similar reasons, we check the effect of different compiler settings in Gambit-C and Chicken. In the R6RS cases, the non-*...hide* case for keywords is special, because it works in the opposite direction: it uses a **define/kw** form that binds a macro to statically convert keyword arguments to by-position arguments at the call site.

Not surprisingly, Common Lisp implementations perform keyword applications with the lowest overhead relative to plain applications. Keywords in Common Lisp are standard and widely used, so implementors are motivated to tune their compilers for keyword arguments. Along similar lines, the result for the new keyword system in PLT Scheme reflects a 30% speed boost from a JIT-specialized primitive that fits the structure-unpacking needs of a keyword application (although the JIT is oblivious to the use of this primitive for keyword applications). Overall, the results illustrate that keywords in PLT Scheme have a typical overhead, even while providing a better separation of keyword arguments from by-position arguments and providing more flexibility in the placement of keyword arguments relative to by-position arguments.

As the first row in Figure 5 shows, the PLT Scheme compiler can greatly improve performance through procedure inlining, but inlining is not currently available for procedures that accept keywords. The performance of inlining could be recovered with a form analogous to **define/kw** in Eddington's R6RS library, which binds the name of a keyword-accepting procedure to an identifier macro. The macro expands direct applications of the keyword-accepting procedure to call a plain procedure—statically converting keyword arguments into by-position arguments, and thus enabling the usual inlining optimizations for plain procedures. Note that keyword arguments always can be detected statically by such a macro with our design, since keywords are a syntactic part of a keyword application, instead of dynamically detected as expression results. The performance of keyword applications without macros has been good enough, however, that we have not yet explored this approach.

6. Experience

Support for optional arguments was one of the first macros that we included in PLT Scheme. We managed, however, to avoid supporting keyword arguments for over a decade. The protocol for keyword arguments seemed inherently complex, so we tried to live without them.

Eventually, however, we ended up with too many functions consuming too many optional arguments, where supplying the nth optional argument required supplying also the n-1 preceding optional arguments. We also created many functions that were small, non-composable variations of each other. For example, Slideshow (Findler and Flatt 2004) provided a **slide** function for generating a slide, a **slide/title** function for generating a slide with a title, a **slide/center** for generating a slide with centered content, and a **slide/title/center** function for generating a slide with a title and centered content. Further variations of **slide** included three slashes.

Our initial design for keywords in PLT Scheme was based on Common Lisp, but with even more extensions and with some attempts to clean up the mingling of keywords and rest arguments. For example, while an argument tagged with **#:rest** includes any supplied keyword arguments (analogous to Common Lisp), an argument tagged with **#:body** includes only extra arguments that follow keyword arguments—and those extra arguments need not have keywords. We used **#:body** frequently; for example, a keyword-based **slide** procedure must accept keywords for configuration but arbitrary rest arguments for the content of the slide.

Many other beautiful generalizations in our initial keyword system, such as the ability to nest optional and keyword syntax in place of a **#:body** identifier, went completely unused. Worse, concerns with error messages and with accidental consumption of keywords as arguments lead to a relatively restrained use of keywords in our libraries. To some degree, the complexity of the syntax for defining keyword-accepting procedures (and notably its lack of connection to the application syntax) also limited adoption. Finally, having to import an extra library to obtain the keyword-supporting **lambda** form was a significant obstacle.

The design presented here arose from an effort to make keyword arguments more widely acceptable in PLT Scheme: to simplify their semantics, to streamline their syntax, and to integrate them into our main dialect of Scheme. Subjectively, the design feels right, and we now use keyword arguments in many more functions and in parts of the language that are closer to the core. For example, **call-with-output-file** used to accept optional arguments to select text versus binary mode and to indicate handling for a file that exists already. Since the arguments were optional, they were placed at the end of the argument list, which is after the callback procedure that is often a **lambda** expression:

```
(call-with-output-file
 dest
 (lambda (out)
   ....) ; many lines
 'truncate
 'text)
```

The distance between the file name and the mode flags made the code difficult to read and write, and the specification of the extra arguments was awkward to document (i.e., up to two extra arguments that are distinct symbols from certain sets). Using keyword arguments, we write the above expression as

```
(call-with-output-file
 dest #:exists 'truncate #:mode 'text
 (lambda (out)
   ....))
```

In this form, the callback procedure regains its place at the end, where it belongs. The file name is still the first argument, where it belongs. The extra optional arguments are more clearly tagged via keywords, and they can be placed in the middle of the by-position arguments, which is where they work best. The specification of the optional arguments (i.e., keyword-tagged with simple defaults) is straightforward and easy to document.

Before deploying our current design for keyword argument, we anticipated problems with the pattern **(lambda args ...)** to accept arbitrary arguments or **(lambda args (apply ... args))** to propagate all arguments. Those patterns work only for by-position arguments; generalizing any use of those patterns requires a switch to **make-keyword-procedure** and **keyword-apply**, which is more verbose and more difficult to remember. For example, the **traced** example of an applicable structure in Section 4 does not support tracing of keyword arguments, and it should be generalized as follows:

```
(define-struct traced (f)
  #:property prop:procedure
  (make-keyword-procedure
   (lambda (kws vals t . args)
     (let ([f (traced-f t)])
       (printf "~s\n" (list* f kws vals args))
       (keyword-apply f kws vals args)))))
```

For similar reasons, some PLT Scheme library procedures have not automatically worked with keywords on a first iteration, such as the `const` function to produce another function that accepts any arguments and returns a constant. Such problems are easy to fix, and occasional missing support for keywords has not been a significant problem so far, but we expect to provide syntactic support for the `make-keyword-procedure` and `keyword-apply` pattern.

The initial implementation of our design for keywords did not include the extra property for arity reporting that is described at the end of Section 4. As a result, if the keyword-based `call-with-output-file` was applied to four by-position arguments, the error message simply reported that the procedure expects one to two arguments without mentioning that the procedure also accepts optional `#:exists` and `#:mode` arguments. Indeed, such an error message often appeared as a result of a call to `call-with-output-file` using old-style optional symbols instead of the new keyword arguments. PLT Scheme users immediately requested improvement in the error message, which reflects the demand for clear error reporting that our design was created to satisfy.

7. Related Work

We know of three major designs for keywords in Lisp and Scheme: keywords in Common Lisp (Steele 1990), keywords in DSSSL (ISO 1996), and SRFI-89 (Feeley 2007). At least one other design has been implemented through portable Scheme macros. Ada, Python, and OCaml, support keyword arguments, while keyword arguments in Smalltalk are fundamentally different. We take each of these in turn in the following sections, and we end with a brief discussion of the syntax of keywords in Scheme.

7.1 Common Lisp

In Common Lisp, keywords are the same datatype as symbols, but they are written with a `:` prefix and they are self-quoting as an expression. (This is actually a trick related to packages; see Section 7.7.)

A Lisp procedure definition can include the special identifiers `&optional` or `&key` before a set of arguments to declare them as optional or by-keyword. In the latter case, the local name of the argument effectively doubles as the keyword. Keyword arguments are always optional, and the default value for optional and keyword arguments is `nil` if none is declared. An `&allow-other-keys` declaration suppresses rejection of keywords for actual arguments that have no corresponding `&key` formal argument. A "rest" argument can be specified with the `&rest` declaration, which must appear before any `&key` declarations. (The full syntax is somewhat more complex, but those are the main points.)

For example, a `rectangle` procedure that accepts a width, an optional height that default to the width, and and an optional keyword-tagged color argument that defaults to `"pink"` is written and called as

```
(defproc (rectangle width
                    &optional (height width)
                    &key (color "pink"))
  ....)

(rectangle 10 20 :color "blue")
```

The semantics of `&optional` and `&key` declarations is essentially to extend the number of arguments accepted by the procedure, and then post-process the list of extra by-position arguments to match them with optional and keyword arguments. When the function consumes keyword arguments, the total number of arguments after the by-position arguments must be even, and the keywords that tag arguments are interleaved with the argument values–i.e., the argument list is used as a *plist*. When both `&key` and `&rest` are used, arguments that are candidates for keyword arguments (including the keywords themselves) are collected into a `&rest` argument, and the number of arguments must be even.

With keywords as part of the standard, many standard procedures in Common Lisp can exploit keyword arguments. For example, the `member` function accepts a comparison procedure as a `test:` argument, in contrast to Scheme's proliferation of separate `member`, `memv`, `memq`, and `memp` procedures.

An advantage of implementing keyword-argument passing as normal arguments, as in Common Lisp, is that procedures like `apply` work with keywords automatically, and the `&rest`-argument convention accommodates arbitrary keywords (at least when `&allow-other-keys` is declared). Separate `keyword-apply` and `make-keyword-procedure` procedures are unnecessary.

Compared to our design, however, the Common Lisp design suffers several drawbacks:

- Since optional- and keyword-argument values are drawn from the same set of actual arguments, and since the keywords that are meant as tags are passed the same as ordinary arguments, keywords can be accidentally consumed as optional arguments. As noted by Seibel (2005), "Combining `&optional` and `&key` parameters yields surprising enough results that you should probably avoid it altogether."
- Although folding keyword arguments into a `&rest` arguments makes sense in combination with `&allow-other-keys`, it means that a procedure cannot generally accept both by-position rest arguments and keyword arguments. Instead, using keywords forces the rest argument to be a plist.
- Keyword arguments must be placed last in a procedure application. That is, keywords can be in any order relative to each other, but they must appear after all required and optional by-position arguments.

The first two drawbacks, in particular, inhibit the use of keywords to extend existing procedures that already use optional or rest arguments. Our design accommodates such extensions, while producing more consistent error messages and being simpler to explain overall.

Dylan (Shalit 1996) supports keyword arguments in much the same way as Common Lisp, except that only keyword arguments can be optional. Furthermore, Dylan distinguishes keyword tags in applications from argument expressions, so that a keyword intended as a tag is never accepted as an argument value. Dylan thus achieves many of the goals in our design of providing a better separation between keyword and by-position arguments, but it does so by restricting the Common Lisp model. A remaining drawback is that keyword arguments cannot be mixed with by-position arguments.

7.2 DSSSL

DSSSL includes an expression language that is based on Scheme, but it includes keyword arguments similar to those of Common Lisp. Keywords in DSSSL are a separate datatype from symbols; they are written like symbols, but with a trailing `:`. Instead of identifiers like `&key` that are treated specially in argument lists, DSSSL uses the special constants `#!key`, `#!optional`, and `#!rest` (and it omits the other declarations of Common Lisp). The semantics of procedure calls and argument processing are as in Common Lisp.

The `rectangle` example in DSSSL syntax looks like the Common Lisp version, but with `&` changed to `#!` and a colon in the application moved to the end of the keyword:

```
(define (rectangle width
                   #!optional (height width)
                   #!key (color "pink"))
  ....)

(rectangle 10 20 color: "blue")
```

DSSSL-style keyword and optional arguments is implemented by several Scheme implementations, including Bigloo (Serrano 2009), Chicken (Winkelmann et al. 2009), and Gambit (Feeley 2009), though details vary slightly. For example, **#!key** is a symbol in Chicken. Compared to our design, keyword and optional arguments in DSSSL have the same advantages and drawbacks as in Common Lisp.

7.3 SRFI-89

Like DSSSL, SRFI-89 distinguishes keyword values from symbols, uses a trailing **:** for the syntax of keywords, and keywords are self-quoting. Unlike DSSSL, SRFI-89 regularizes the syntax of procedures with keyword and optional arguments by making the procedure syntax more closely match the application syntax.

A keyword is associated with an argument in a procedure expression by placing the keyword before the formal argument; a small difference to our syntax is that the keyword and argument identifier are grouped by parentheses. An optional argument is declared by placing a default-value expression after the formal argument, and then grouping the two with parentheses. A keyword argument can be required, or it can be made optional by adding a default-value expression after identifier, within the parentheses that group it with the keyword.

The **rectangle** example could be written with SRFI-89 as follows:

```
(define (rectangle width
                   (height width)
                   (color: color "pink"))
  ....)

(rectangle 10 20 color: "blue")
```

Our design mostly imitates the SRFI-89 syntax, because we value the syntactic similarly of declarations and applications. We depart from SRFI-89 syntax in not grouping a keyword with a formal argument in a procedure declaration, because that change further strengthens the similarity to applications (where a keyword and its argument expression are not grouped with parentheses).

SRFI-89 separates a rest argument from keyword arguments; an argument is consumed either as a keyword argument or collected into the rest argument, but never both. SRFI-89 also generalizes keyword support by allowing keyword arguments to appear before by-position arguments. Unlike our design, however, keywords are either grouped together before by-position arguments or together after by-position arguments, and the order for a given procedure is determined by the procedure declaration. A drawback of this approach is that callers of a procedure must remember which order is used for a given procedure. Our design more completely separates by-position and by-keyword arguments, so that keyword arguments can always appear in any order relative to by-position arguments.

As in Common Lisp and DSSSL, optional- and keyword- argument handling is defined in terms of post-processing a sequence of by-position arguments, where keyword tags are mingled with argument values. As a result, it suffers from the many of the same problems in terms of accidental treatment of a keyword tag as a direct argument.

7.4 Implementation via Macros

Scheme macros support a portable implementation of optional and keyword arguments, although no such implementation has become widely used. One recent effort is Eddington's implementation for R6RS, which we used for performance measurements in Section 5. A lack of documentation for the library makes a detailed comparison difficult, but as we noted in Section 5, the library supplies a **define/kw** form for binding names that resolve keyword arguments statically. Having no syntactic distinction between keywords as expressions and keywords as argument tags, however, makes the library's static resolution inconsistent with its dynamic resolution.[4]

A variant of our design appears to be possible as a portable implementation using macros. Keywords could be identified through a **keyword** form that signals a syntax error when used as an expression, while an explicit **with-keyword** form would serve the role of a keyword-allowing application form that detects **keyword** tags. The combination of **keyword** and **with-keyword** enables the distinction between keyword tags and argument expressions, though it is syntactically more verbose than a built-in syntax of keywords or allowing **#%app** to be refined. To allow a procedure with optional keywords to be called through a normal application form, keyword-accepting procedures would be represented as plain procedures (since Scheme standards do not include applicable structure types); the protocol for supplying keyword arguments could use a special value as a regular argument to indicate that certain other arguments provide lists of keywords and associated values.

7.5 Ada, Python, and OCaml

Every function in Ada or Python supports keyword arguments, where the name of each formal argument doubles as the keyword for the argument. In a function call, by-position arguments are provided first and matched to formal arguments in order, and then keyword arguments can appear (in any order) to supply values for the remaining arguments. As in our design, keyword arguments are syntactically distinguished from by-position arguments in a function call. Unlike our design, by-position arguments must be supplied first.

Ada's double role for every formal argument as both a by-position and by-keyword argument is different from Lisp and Scheme systems, where formal argument names are purely local. Exposing all argument names as keywords in Scheme conflicts with other important aspects of the language, such as alpha renaming. A workable syntax might have the programmer annotate identifiers that should double as by-position and by-keyword arguments.

OCaml supports *labels* on function arguments that are similar to Ada's keyword arguments. A programmer explicitly designates labeled formal arguments using $\sim$ on the argument (and, optionally, a label that is separate from the argument's local identifier). The label of an argument becomes part of the function's type, which means that the compiler can always statically adjust the order of labeled arguments in a function call—even changing the order of curried applications to match the declaration order. Labelled arguments also can be optional (which, again, is exposed in the type of the function).

Finally, the PLT Scheme class system behaves much like Ada, in that class initialization arguments (i.e., constructor arguments) are usually supplied by name, but they can also be supplied by position. If arguments are supplied by position, the order of the names in the class declaration is used to match them with arguments values. This design pre-dates general keyword support in PLT Scheme, and it mainly provided backward compatibility with a previous iteration of the class system that supported only by-position initialization arguments.

Allowing keyword arguments to be supplied by position, as in Ada, conflicts somewhat with allowing keyword arguments in

[4] http://groups.google.com/group/ikarus-users/browse_thread/thread/fb3a813c198311ff

any order relative to by-position arguments; perhaps sensible rules could be specified to govern a mixed order of arguments with and without keywords. Ada-style argument handling also conflicts with combining keyword arguments and a by-position rest argument. More generally, we have not found much need for passing keyword arguments by position.

7.6 Smalltalk

In Smalltalk, most methods arguments are tagged with names, but the tags are not keywords in the sense of this paper. The tag on a Smalltalk method argument is simply part of the method name that is interleaved with the arguments; the tags and arguments cannot be reordered, and individual arguments are not optional.

7.7 Keyword Lexical Syntax

A keyword in Common Lisp is prefixed with `:`. This choice of syntax is related to Common Lisp's notion of package-specific symbols, where the empty package name corresponds to the "keyword" package. Conceptually, keywords are self-quoting because all symbols in the keyword package are bound to themselves.

In DSSSL and many Scheme systems, a keyword is *suffixed* with `:`, instead of prefixed with `:`. To many programmers, the suffix better connects the keyword with its argument, while others argue that a prefix is more appropriate for a prefix-oriented language like Scheme.

PLT Scheme uses a `#:` prefix. Chicken also supports a `#:` prefix in addition to a `:` suffix, though the keyword in both cases is equivalent to a symbol. The `#:` choice is natural for Scheme, since a `#` is normally used to extend the reader syntax, and `:` is normally allowed in symbols (i.e., some symbols and identifiers in existing code might break if a `:` prefix or suffix becomes the syntax of keywords). Many argue, however, that `#:` looks too heavy, while the whole point of keywords is arguably to add a lightweight grouping syntax to the language (i.e., lighter weight than parentheses). Also, Common Lisp uses the prefix `#:` for uninterned symbols.

We can offer no rationale that will resolve the debate. We chose `#:` because it broke no existing code and because at least one author likes how it stands out. An informal poll among PLT Scheme users suggested roughly equal support for all three choices (prefix `#:`, prefix `:`, and suffix `:`) with a slightly higher preference for `#:`—possibly reflecting the syntax that is already in place. In any case, PLT Scheme's **`#lang`** notation would allow future modules to be written using a different syntax without affecting old modules.

8. Conclusion

Scheme's "rest" arguments and **`case-lambda`** allow flexible handling of procedure arguments, and they easily accommodate keyword-like patterns using symbols and lists. When a pattern is used widely enough, however, converting the pattern to a language construct offers many advantages: better readability, clearer documentation, better error messages, easier composition of libraries, and a central point of control for implementation details of the pattern. For all of these reasons, we believe that specific constructs for keyword and optional arguments are appropriate for dialects of Scheme.

The essential elements of our design are (1) keywords that are distinct from symbols, as in many Scheme systems, (2) a form for creating keyword-based procedures that matches the application syntax, similar to SRFI-89, (3) disallowing unquoted keywords as literal expressions, which is novel in our design, and (4) passing keyword arguments to a procedure in a way that reliably separates them from by-position arguments, which is also novel.

Bibliography

John Clements and Matthias Felleisen. A Tail-Recursive Machine with Stack Inspection. *ACM Trans. Pogramming Languages and Systems* 26(6), pp. 1029–1052, 2004.

William D. Clinger et al. Larceny. 2009. `http://www.ccs.neu.edu/home/will/Larceny/`

Marc Feeley. SRFI-89: Optional Positional and Named Parameters. 2007.

Marc Feeley. Gambit v4.4.3. 2009. `http://www.iro.umontreal.ca/~gambit/`

Robert Bruce Findler and Matthew Flatt. Slideshow: Functional Presentations. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 224–235, 2004.

Franz, Inc. Allegro CL. 2009. `http://www.franz.com/`

Abdulaziz Ghuloum. Ikarus Scheme v3.0+. 2009. `http://ikarus-scheme.org/`

ISO. Document Style Semantics and Specification Language (DSSSL). ISO/IEC 10179:1996, 1996.

SBCL. 2009. `http://sbcl.sourceforge.net/`

Peter Seibel. *Practical Common Lisp*. Apress, 2005.

Manuel Serrano. Bigloo v3.2b-2. 2009. `http://www-sop.inria.fr/mimosa/fp/Bigloo/`

Andrew Shalit. *The Dylan Reference Manual*. Addison-Wesley, 1996.

Michael Sperber (Ed.). The Revised ^{6} Report on the Algorithmic Language Scheme. 2007.

Guy L. Steele Jr. *Common Lisp: The Language*. Second edition. Digital Press, 1990.

Felix Winkelmann, Kon Lovett, and Leonard Frank (elf). Chicken v4.0.0. 2009. `http://www.call-with-current-continuation.org/`

Appendix

Implementation of optional arguments in terms of **`case-lambda`**:

```
(define-syntax lambda
  (syntax-rules ()
    [(lambda (arg ... . rest) . body)
     (letrec ([f (case-lambda* f (arg ...) () ()
                                rest body)])
       f)]))

(define-syntax case-lambda*
  (syntax-rules ()
    [(case-lambda* f () (id ...) (clause ...)
                   rest body)
     (case-lambda clause ...
                  [(id ... . rest) . body])]
    [(case-lambda* f ([opt-id default-expr]
                      . rest-args)
                   (id ...) clauses rest body)
     (case-lambda* f rest-args (id ... opt-id)
                   ([(id ...)
                     (f id ... default-expr)]
                    . clauses)
                   rest body)]
    [(case-lambda* f (req-id . rest-args)
                   (id ...) clauses rest body)
     (case-lambda* f rest-args (id ... req-id)
                   clauses rest body)]))
```

Screen-Replay: A Session Recording and Analysis Tool for DrScheme

M. Fatih Köksal, R. Emre Başar
Department of Computer Science
İstanbul Bilgi University
{fkoksal,reb}@cs.bilgi.edu.tr

Suzan Üsküdarlı
Department of Computer Engineering
Boğaziçi University
suzan.uskudarli@boun.edu.tr

Abstract

Approaches to teaching "Introduction to Programming" vary considerably. However, two broad categories may be considered: product oriented vs process oriented. Whereas, in the former the final product is most significant, in the latter the process for achieving the final product is also considered very important. Process oriented programming courses strive to equip students with good programming habits. In such courses, assessment is challenging, since it requires the observation of how students develop their programs. Conventional methods and tools that assess final products are not adequate for such observation.

This paper introduces a tool for non-intrusive observation of program development process. This tool is designed to support the process oriented approach of "How to Design Programs" (HtDP) and is implemented for the DrScheme environment. The design, implementation and utility of this tool is described with examples.

Keywords Introductory Programming, Development Process, Design Recipe, DrScheme

1. Introduction

The education of a computer science student usually starts with an introductory programming course. The aim of such courses is to equip students with general programming knowledge and prepare them for subsequent courses in the curriculum. Such courses typically teach the fundamental concepts of programming with the use of given programming language, integrated development environment (IDE), and other tools [2]. With these tools and course instruction students are expected to learn how to write, debug and document programs.

While the objectives of introductory programming courses are similar, the content, approach and assessment methods differ. Teaching with examples is frequently used [6], where examples are provided for every concept introduced. These examples are expected to guide students in their assignments. Students often use these examples as a starting point and modify them until they reach the desired solution. Conventional assessment methods evaluate exams and assignments by comparing students code against expected result. The students code in this case is the final product. There is no further information on how the student arrived at the final product.

The TeachScheme! project [7] does not appreciate the programming-by-tinkering methodology. It developed an alternative approach to teaching, described in the text book, "How to Design Programs" (HtDP) [5]. This approach focuses on a design process that starts from problem statement to a well-organized solution. After the publication of HtDP, several universities around the world revised their curriculum in favor of this approach. Most universities use the methodology as described in the book, where others [2] have derived versions [10] according to their needs.

The HtDP and approaches derived from it emphasize the importance of process in comparison to the product. Accordingly, instead of conventional assessment methods, they prefer lab (or live) exams, which they consider to be a more accurate reflection of students progress [4]. Approaches to conduct live exams also vary. Some let students develop programs independently and evaluate results in a conventional manner. In others [2, 1] the development process is observed personally. The observation process is an intrusive approach that may impact student performance.

In order to understand how students develop their programs it is necessary to track their development process. By tracking their process, we aim to answer following questions: Do students follow the suggested design guidelines while they develop programs on their own? Are students, who follow the suggested guidelines, more successful than the others? If not, is there any specific design pattern that is commonly used by successful students? Using an intrusive tracking method may impact students' performance in the programming session. Indeed, it has been reported that some students were disturbed by personal observation of their work [1].

An alternative approach for tracking program development is to embed the tracking ability into the development tool. Such a tool would need to record as well as replay the development process. This work describes a program development tracking tool for DrScheme [8] that enables a student to record his/her programming session. This recorded session can, then, be replayed and analyzed by an observer.

The rest of this paper is organized as follows: Section 2 further discusses our motivation to analyze students' programming sessions in order to answer questions we stated above. Section 3 investigates related work regarding product and process oriented approaches and their assessment techniques. Underlying concepts and implementation details are given in Section 4, followed by a discussion in Section 5. Finally, in Sections 6 and 7, we discuss future work to be done and conclude our work.

Proceedings of the 2009 Scheme and Functional Programming Workshop
California Polytechnic State University Technical Report CPSLO-CSC-09-03

2. Motivation

The first year curriculum for Computer Science Department at İstanbul Bilgi University was revised effective of 2007-2008 academic year. Courses were divided into sections of at most 20 students, in order to have better control over the course and increase student-instructor interaction. With this change, we have been able to intensively follow our students to see if they meet our educational approaches.

The introductory programming course (Comp149/150-HtDP) at İstanbul Bilgi University, is a part of the meta-course Comp149/150, which also includes the courses: Academic Skills (Comp149/150-AS), Meta Skills (Comp149/150-MS) and Discrete Mathematics (Comp149/150-DM). This meta-course is mandatory to Computer Science, Financial Mathematics and Business Informatics majors. Comp149/150-HtDP uses "How to Design Programs" (HtDP) [5] as the text book, Scheme as the programming language and DrScheme [8] as the development environment.

The first semester of the course (Comp149-HtDP) covers first four parts of the book, which basically includes primitive, compound and recursive data types, conditionals, and abstraction. Generative recursion, graphs, vectors and iterative programming are taught in the second semester (Comp150-HtDP).

Each semester consists of 13 weeks. Every week there are two hours of lectures and two hours of labs. In lecture hours, instructors present the material and write programs in front of the students by following the design recipe as suggested by HtDP. Additionally, each week students are assigned a project, which they must complete within one week. In the final weeks of the second semester assignments become more complicated and students are given at least two weeks to complete. During lab sessions students present their project solutions to their classmates.

During this course students are given four live exams. Each exam consists of one or two questions that have to be solved in approximately 1.5 hours. Exams are completed on computers, where students only have access to the text book and DrScheme. All networking is disabled during the exams. Grades of weekly projects and live exams determine the course grade of students. Final grade of a student from this course is combined with grades from other parts of the meta-course using a formula that rewards even performance. This grading policy was established based on the belief that students must have sufficient knowledge of mathematics, critical reading/thinking skills and the ability to express their thoughts properly in order to develop well structured programs. Starting from the 2008-2009 academic year, students are examined by a jury at the end of the year by their instructors of this meta-course.

The main objective of the entire course is to teach "How to solve it?" [11] and the process is central to this idea. The following section describes the design recipe methodology of HtDP that, in theory, meets the aim of our introductory programming course.

2.1 HtDP and the Design Recipe

HtDP is defined by its authors as "... the first book on programming as the core subject of a liberal arts education". It focuses on the design process that leads from problem statements to well-organized solutions rather than studying the details of a specific programming language, algorithmic minutiae, and specific application domains [5]. It includes design guidelines, which are formulated as a number of *program design recipes* leading students from a problem statement to a computational solution in step-by-step fashion with well-defined intermediate *products*.

A design recipe is a checklist that helps students to organize their thoughts through the problem solving process. Basic steps of the design recipe are as follows;

0. *Data definition*: describe the class of problem data
1. *Contract*: name your function and give input-output relation in terms of data type used
2. *Purpose*: informally specify the behavior of your program
3. *Examples*: illustrate the behavior with examples
4. *Template*: develop your programs template/layout
5. *Code*: transform your template into a complete definition
6. *Tests*: turn your examples into formal test cases.

The version of the design recipe presented here includes 7 steps where the original one has 6. In our version, purpose statement and the contract are split into different steps. It starts from 0, since a data definition can be used by a number of different functions, while other steps are function specific.

Students are expected to use this checklist on a question-and-answer basis to progress towards a solution [5]. Figure 1 shows the application of a design recipe for summing the elements of a list.

2.2 The Strategic War Between Instructors and Students

There are numerous reports of success using HtDP curriculum [12, 2, 13, 3]. Since the adoption of HtDP, we have also observed similar improvements. Specifically, we have observed improvements in student performance with respect to:

- programming abilities,
- overall grades and
- subsequent courses.

These improvements are particularly noticeable in female students.

On the other hand, increased interaction with students revealed some deficiencies in their adoption of the process we use. Students were not applying the design recipe throughout their development process. They were diving into the code without going through the design steps. To tackle this problem, a change in our grading scheme was required. The grading scheme was changed to grade every step of the design recipe separately.

Students responded by faking the process. They were writing the code first and adding the design steps later. This response led us to inspect each student submission more carefully. The forged design steps can be distinguished by checking the inconsistencies between the steps. Considering that, our response was to do a consistency check between the design steps and stopping evaluation of the assignment when an inconsistency was found.

At that point, it was understood that applying more force on following the recipe only created better "design recipe evasion" tactics. With this realization we abandoned the attempt to evaluate the order of construction and only verified presence of correct parts. Currently, the recipe is followed while teaching, and students are encouraged to use for every program they develop. But, the application of the recipe is not enforced or evaluated in any way.

However, we are still interested in tracking our students' development processes to see both how they develop their programs and whether the suggested approach helps them to build well-structured solutions. Therefore we developed a tool for just that purpose.

3. Related Work

To the best of our knowledge, there is no software that deals with the analysis of code/editing sequences in the way Screen-Replay does. This section rather reports approaches that aim to increase both product and process quality of students in programming classes.

In [14], authors report on a controlled experiment to evaluate whether students using continuous testing are more success-

```scheme
;; Data definition:
;; a list of numbers (lon) is either;
;; 1. empty, or
;; 2. a pair of
;;     a) a number and
;;     b) a list of numbers (lon)

;; Contract:
;; sum-lon: lon -> number

;; Purpose:
;; this function consumes a list of numbers
;; and produces the sum of the elements of
;; the given list

;; Examples:
;; empty            -> 0
;; (list 5)         -> 5
;; (list 3 1)       -> 4
;; (list 4 7 -2) -> 9
```

```scheme
;; Template:
;; (define (sum-lon alon)
;;   (cond
;;     ((empty? alon) ...)
;;     (else
;;       ... (first alon)
;;       ... (sum-lon (rest alon)) ...)))

;; Code:
(define (sum-lon alon)
  (cond
    ((empty? alon) 0)
    (else
     (+ (first alon) (sum-lon (rest alon))))))

;; Tests:
(check-expect (sum-lon empty) 0)
(check-expect (sum-lon (list 5)) 5)
(check-expect (sum-lon (list 3 1)) 4)
(check-expect (sum-lon (list 4 7 -2)) 9)
```

Figure 1. Application of the design recipe for summing the elements of a list

ful in completing programming assignments. As the source code is edited, continuous testing uses excess cycles on a developer's workstation to continuously run regression tests in the background against the current version of the code providing feedback about test failures. Their tool aim to give extra feedback during the programming session and improve the productivity of developers. The experimental results indicate that students using continuous testing more likely to complete the assignment by the deadline. It appears that their efforts are on final product quality rather than the programming process.

In their case study [2], instructors from Tübingen and Freiburg Universities report the development of their introductory programming course. For their first-year programming course they adopted the tools developed by the TeachScheme! project, in addition, they supervise their students closely with assisted programming sessions on weekly basis. During assisted programming sessions students solve a set of exercises under the supervision of a doctoral student assisted by one or two teaching assistants to ensure that the students follow the design recipes. Authors report that their students not only performed well on exams, they were also able to transfer their knowledge to other programming languages and IDEs. In our experiences, on the other hand, we observed that some students perform poorly (some even could not do anything) when they are watched "over their shoulders" during programming sessions. Such students perform well when they study in environments where they feel comfortable. As authors state, nearly 15% of the students did not even try to solve the programming assignments during assisted programming sessions. We can not say that this is caused by the same reason, but, further analysis can be done, and the sessions of such students can be observed later using tool support.

This study also points out that, many students avoided asking TAs for help during the session, as they either expected that TAs were not allowed to provide concrete help or they even believed that asking for help was a form of cheating. As reported, the perception of assisted programming changed during the semester as TAs not only provided help upon request but also helped proactively as they noticed students having problems. This approach is helpful for students, who hesitate asking questions. The point is, how do we find out that a student is experiencing a problem applying the design recipe without constantly watching his/her session? As we have already experienced, students' main concern is to have the final running code before the time finishes. Thus, they escape from applying the design recipe and focus back to the code using the programming-by-tinkering method, as soon as they stay uncontrolled. Furthermore, assisting students during programming sessions does not mean that they apply design recipe in exams. One may not attribute the success of students to the success of design recipe, without tracking their process during exams.

Another study [9] points out the importance of exposing the process of development of the solution rather than just presenting the final state of the program. They propose "live coding" as an active learning process. Since instructors do not commit same errors students generally do, they suggest the student-led live coding (where the student writes the code in front of his/her classmates) rather than the instructor-led live coding (in which the instructor writes the code in front of students). Our experiences show that, especially in the first few weeks, students should program by themselves and learn from their mistakes. Interfering as they make mistakes means taking their chance of solving the problem by themselves, therefore learning the importance of design recipe.

Exposing a student's errors in front of his/her classmates might also damage the motivation of other students and lead them to hold back and not participate. Instead, project submissions of students can be replayed without showing the identity of the submission owner to illustrate good and bad programming habits.

For using in online courses or when the class time is limited, authors of this paper also implemented a screen casting software which allows to record narrated video screen captures and then made available to students to review. Keeping track of students' programming sessions and analyzing them can hardly be done using remote desktop or screen-cast applications. Content based information can not be extracted from sessions recorded by such applications. Moreover, these applications are not adequate for resource limited environments.

Finally in [1], instructors teach the programming process using a five steps, test driven, incremental process (STREAM). Every week there is a mandatory assignment. For lab examinations, they propose a method where students are instructed to call upon a TA when they reach a checkpoint to show and demonstrate their solutions. Students approach to the development process as well as their solutions count in the final grade. To evaluate whether students really apply the suggested approach when no guidance is provided,

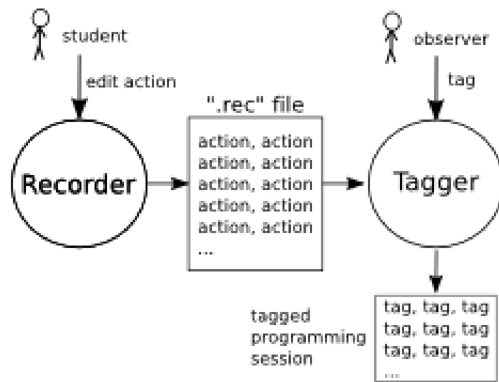


Figure 2. An overview of the Screen-Replay tool

they conduct an experiment. In this experiment students solve the assignments while TAs observe and make note of any violations to the method taught. Authors report that all students followed the process they have been taught. It is unclear whether students were aware of the aim of this experiment. If they were, it is quite possible that it would inpact their programming behaviour.

In summary, none of these methods provide a way for tracking students process while they work on their own. Thus, we see strong viability in favor of our tool in this context.

4. Tracking the Development Process

In order to track how students construct programs we developed a system called Screen-Replay. This system records how students develop their programs and allow evaluators to observe as well as identify the sequence of activities taken during the program construction.

4.1 Requirements

The fundamental requirements of the system are:

1. Record every state the program takes during its construction lifetime. The lifetime begins with the creation of the program session until its completion.
2. Replay the construction of the program.
3. Describe the high level programming activities taken during the program constrution. These activities are the ones described by the HtDP methodology.

The requirement 1 must be satisfied within the development environment in a transparent manner. In another words, construction activities must be recorded in the background while the student is constructing their solution. Requirements 2&3 are meant for evaluators who will inspect and annotate the students program construction.

4.2 Implementation

Screen-Replay mainly consists of two parts: Recorder and Tagger. It implements the requirements within the DrScheme environment. Scheme programming language is used for the implementation. The Recorder and Tagger are decribed in the following sections.

4.2.1 Recorder

The Recorder records all user interactions within the DrScheme's *Definitions* window, which is where programs are defined. The Recorder saves information about any insertion or deletion. The following Scheme structure, *action*, describes the information stored for every user interaction.

```
;; action
;; timestamp (number): current time in seconds
;; operation (symbol): type of the operation.
;;                     Can be 'insert or 'on-delete
;; start (number)     :position of the cursor in
;;                     the definitions window at
;;                     the time of operation
;; len (number)       :length of the action-content
;; content (string)   :the content of the action

(define-struct action (timestamp
                       operation
                       start
                       len
                       content) #:prefab)
```

The following example is an action that indicates that user typed *f*, an insertion of length 1, at position 0 of the definitions window. Position 0 is the starting position.

```
;; For Example
(make-action 1240394142 'insert 0 1 "f")
```

For every text insertion and deletion the Recorder creates a corresponding action. Actions remain in the buffer until the file is saved. The Recorder catches keystrokes by extending `definitions-text` with a mixin. This mixin augments the `insert` and `on-delete` methods with use of a boolean flag to indicate the recording state of the current window. This approach makes it possible to record actions in each window separately.

When a file is saved the buffer content is written to an *actions-file* with a ".rec" extension. An actions-file consists of a series of action structures serialized with the `write` function. The name of the actions-file is formed using the base file name of the program file. Subsequent actions are appended to the actions-file when the file is re-saved. In the case of a save-as operation previous actions are copied from the current actions-file to the new actions-file. Recorded files are replayed using the Tagger.

4.2.2 Tagger

The Tagger has two main functions: (1) To replay the program construction and (2) describe the high-level construction process in terms of the HtDP methodology.

The Tagger allows the observer to see exactly how the program was constructed. While observing the construction process, the observer can describe the programming activity using tags defined for this purpose.

Replaying: The Tagger replays the exact steps taken while the program was written. The observer can see each text insertion or deletion at the same speed of the construction process. Various controls enable more convenient navigation of the construction process:

- *Play*: Start playing actions
- *Pause*: Pause playing
- *Backwards*: Play backwards
- *Speed-Up/Down*: Change the play speed
- *Go-To-Next-Action*: Jump to next action without waiting

A time slider is supplied to enable the observer to directly navigate to a desired action.

A student may jump from one position to another during the programming session. For example, he/she can move to the data definition from the program code. Such jumps can make it difficult for the observer to follow the session. Additional features exist to assist the observer in such cases. For example, the Tagger automatically scrolls to the position within the program that is associated

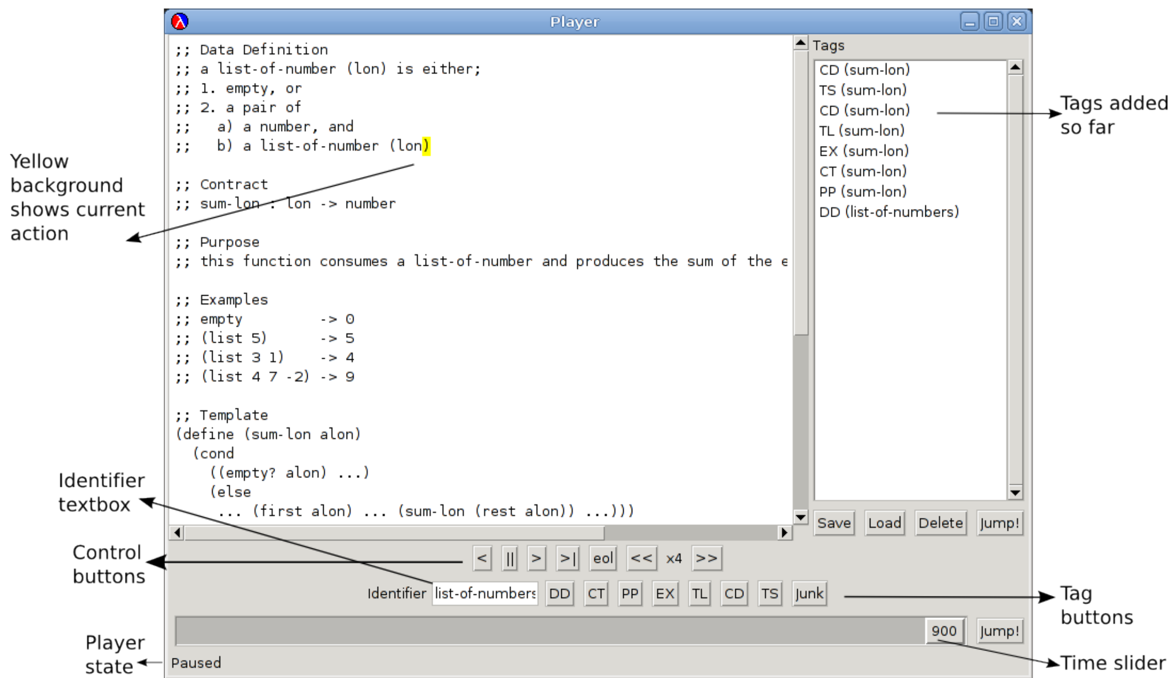


Figure 3. An overview of the Tagger tool

with the current action. This makes the location visible making it easier to follow the flow of construction. The currect action record is shown with a yellow highlight.

When a file is selected to be played, all actions in the actions-file are loaded into a *Tape* structure defined as follows:

```
;; tape
;; actions (vector): contains the actions saved
;;                   by the Recorder
;; pointer (number): index of the current action

(define-struct tape (actions pointer) #:mutable)
```

When the *Play* button is clicked, a thread starts to play the actions in the Tape structure. To play an action is to insert/delete the content to/from the editor according to the timestamp and position information in it. For example, to play

```
(make-action 1240394142 'insert 0 1 "f")
```

will insert the single character "f" in the editors first position. When playing backwards, the action operation is reversed: an *'insert* symbol is interpreted as *on-delete* and an *'on-delete* symbol is interpreted as *insert*.

The Tagger replays the construction at the same speed of the original construction. Scheme semaphores are used in order to make the Tagger wait while playing. The running thread is suspended until the semaphore becomes free. This semaphore is managed by a timer object, which is set to the difference between consecutive actions.

Consider the recorded example shown in Figure 1. The final product shows that all design recipe steps are present. The Tagger enables one to view the process that led to this product. The first column of Figure 4 describes the students process. Replaying this session reveals that the student actually did not follow the design recipe sequence. It appears that the student attempted to fool the instructor. The student first implemented the code and then added the remaining required steps. The tracking process reveals the order of the application of design recipe. It also shows how much time is spent on each step. The ability to observe such a process enables the instructor to discover deficiencies and provide more accurate help.

Tagging: Tagging allows the evaluator to describe that kinds of activities performed during the constructing of a program. Recall that the activities of interest in HtDP are:

0. Data definition
1. Contract
2. Purpose
3. Examples
4. Template
5. Code
6. Tests

To identify each of these activities the corresponding tags DD, CT, PP, EX, TL, CD and TS are defined. During the tagging process the observer specifies a sequence of tags as he/she observes the construction states. The interface includes buttons for each tag. The observer clicks, for example, to the DD button, when the student finishes editing the data definition and moves to another state.

It is possible that the student performs some activity that does not fit within the HtDP methodology. For this, a Junk tag was defined. Junk may not be the best label as the student may do something useful that is not directly meaningful to HtDP. For example, students may write a question or make a check list to assist themselves. On the other hand they may write something totally irrelevant, such as a note to the examiner (i.e. "Dear Professor, for God's sake, I don't want to fail."). In any case, the Junk tag should simply be interpreted as anything that is besides the enumerated tags defined earlier.

The ideal development sequence would be: [DD, CT, PP, EX, TL, CD, TS]. Naturally, one does not expect a perfect program construction. But, rather, hope to observe that the overall order of steps was followed.

Tags are stored in *Tag* structure:

```
;; tag
;; name (symbol)        : name of the tag
;; identifier (symbol): an identifier that
;;                        the tag is tied to
;; end (number)         : value of the tape-pointer
;;                        at the time of tagging

(define-struct tag (name identifier end) #:prefab)

;; For Example
(make-tag 'DD 'list-of-numbers 65)
```

The development process is denoted with a sequence of tags, which are inserted by clicking on the appropriate tag-button. Furthermore, an identifier can be associated with each tag to further describe which function the activity is associated to. For practical reasons, only the position of the tape-pointer at the end of the tag is stored. This makes reorganization of tags easier. When a new tag is generated, the Tagger saves this tag to its *tags-list* and displays it in the panel on the right side of the window.

Recall the programming assignment in Figure 1, which we assumed to be recorded using the Recorder. Second column of the Figure 4 shows responses of the observer to the process of the student. The observer carefully tracks actions of the student and tags the session accordingly. At the end of the tagging process a tag-list, possibly as in the example below, is generated. Tag-end positions may not be easily traceable from the given figure, but they need to be shown in this example.

```
(list (make-tag 'CD 'sum-lon 158)
      (make-tag 'TS 'sum-lon 297)
      (make-tag 'CD 'sum-lon 303)
      (make-tag 'TL 'sum-lon 382)
      (make-tag 'EX 'sum-lon 484)
      (make-tag 'CT 'sum-lon 564)
      (make-tag 'PP 'sum-lon 574)
      (make-tag 'DD 'list-of-numbers 900)
      (make-tag 'JK 'none))
```

Above tag-list, generated from the tagging session, tells us that actions from indices 0 up to 158 are somehow related and have the same context (they form the code for sum-lon function for this particular example). Similarly, actions between 159-297, 298-303 (and so on) have their own context according to the observer. Therefore he/she generated new tags.

The application of design recipe was already revealed with replaying the session, but having the tag-list in hand means much more than just replaying. First of all, once the tag-list is generated there is no need to replay the session to see the process. It is sharable data, which can be sent to someone else for further observation. Tag-lists from different sessions of a student, or from different students can be used together to be analyzed. Even if the tagging process is not finished, tags generated so far can be saved and later loaded (for the same session) for further tagging. The Tagger also allows the observer to jump to a previously tagged position using the tag-list.

Tag-list, by itself, includes some information about the session and can be used for investigation of the construction process. However, using both recorded actions and the tag-list together, a lot more information about the session can be extracted. The following subsection introduces the idea of "processed-tag" which enables more detailed investigation of a session.

4.3 Processing the Tags

While actions and tags are useful by themselves, merging these two sources of data provides a better insight to the students process. A tag, by itself, is actually a collection of actions. Therefore, it should represent characteristics of actions it contains. Using the already available time and position data in actions, tags can be extended with more information to generate a self contained analysis data. Processed-tag is defined as follows to meet this requirement;

```
;; processed-tag
;; step (symbol)          : the name of the tag
;; identifier (symbol)    : an identifier text
;; action-count (number) : total number of actions
;;                          contained in this tag
;; size (number)          : total length of actions
;;                          contained in this tag
;; start-time (number)    : time of the starting
;;                          action of this tag
;; end-time (number)      : time of the last
;;                          action of this tag
;; start-position(number): starting position of this
;;                          tag in the editor
;; end-position (number) : end position of this tag
;;                          in the editor

(define-struct processed-tag (step
                              identifier
                              record-count
                              size
                              start-time
                              end-time
                              start-position
                              end-position) #:prefab)
```

As described above, processed-tag includes much more information about the actions associated a tag. Inspecting a processed-tag provides a summary of its associated actions, i.e duration, begin and end time, the segment in the code, etc.

It is possible to identify when the student switches between design steps. Inspecting these switches might reveal a common pattern in the application of the design recipe.

Another type of information that is possible to extract from processed-tags are the timings. Using processed-tags, it is possible to examine the time distribution among design steps.

It is possible to observe how the overall program progress as well as individual segments. This information might provide insight into students' problem solving techniques.

Sessions can be divided into active or passive parts. Active parts are parts where the user interacts with the editor. Passive parts are the parts where the user does not interact with the editor and there is no information about what he/she is doing. The analysis of relations between these parts together with the segment switching information can provide more accurate information about the students' behavior.

5. Discussion

We compared recorded sessions with source codes to verify that recorded sessions would build the exact source code. All sessions were successfully regenerated from the recorded files, with the exception of regions that were commented out with boxes[1], since this feature has not yet been implemented.

An interesting side effect of the Screen-Replay tool is related to plagiarism, which can be used during analysis. Detecting plagiarism was not a design decision for Screen-Replay, but an analysis of the actions-file helps to detect plagiarism. For example, a student may copy-paste someone else's code. Since the Recorder generates a new action for each keystroke, copy-paste operations end up with actions that has a length greater than 1.

The fuzzy nature of the design recipe makes it hard to automatically detect the design segments. Students apply it in different

[1] DrScheme allows users to comment out regions with a box snip.

Student	Observer
Starts implementing the code for the sum-lon function.	Realizes that the student is implementing the code for the sum-lon function. Types an identifier (may be "sum-lon" for this case) or keeps it blank. Waits until the student switches to some other design step.
Finishes implementing the code. Starts implementing the tests.	Pushes CD button at the time student finishes the code implementation. Waits the student to finish the tests.
Finishes implementing the tests. Goes back to the code (he might get some errors. Tagger doesn't show it) and modifies some parts.	Pushes TS button at the time student finishes tests. Waits the student to finish code modification.
Finishes modifying the code. Starts writing template according to the code, then writes the examples according to tests.	Pushes CD, then TL as the student finishes writing code and template, respectively. Waits the student to finish examples.
Finishes writing examples. Writes contract and purpose for the function. Starts writing data definition for list-of-numbers.	Pushes EX, CT and PP buttons as the student finishes writing examples, contract and purpose, respectively. Realizes that the student is writing data definition for the list-of-number. Updates the identifier. Waits the student to finish the data definition.
Finishes writing data definition.	Pushes DD button as the student finishes writing the data definition.
Writes his/her name and id number.	Pushes JK button (as this is an irrelevant information for the analysis) as the student finishes writing identification information.

Figure 4. Students actions and observers responses in return

orders and in different forms. Steps other than code and tests do not have formal definitions. Some heuristics may be developed, but they can hardly ensure a precise tagging. Therefore, instead of automation we preferred to support the observer with helper functionalities in order to reduce the time and effort required for tagging.

To make tagging easier *go-to-next-record* and *go-to-end-of-the-current-line* buttons are added to the Tagger. The former enables the reviewer to jump to the next action without waiting the action to be occurred. And the latter enables the reviewer to jump to the action that takes place at the end of the current line. Since students change the current line when starting to write a new design step, using this button makes tagging easier.

Another feature that assists the observer is the jump detection function. This function pauses the playing process and warns the observer when the user is about to jump 3 lines above or below from the current line. The observer, then, may put a new tag or continue playing. According to our observations, one or two line jumps mostly appear within the same tag. Therefore, we preferred to warn the observer every time a 3-lines jump occur.

6. Future Work

Implementation of the Screen-Replay as a process tracking tool enabled us to investigate the efficiency of the teaching methods we use. Last three live exams (approximately 100 students each) of our introductory programming course are already recorded. After the end of tagging process we will investigate answers to the following questions;

- Do students follow the suggested design guidelines while they develop programs on their own?
- Are students, who follow the suggested guidelines, more successful than the others?
- If not, is there any specific design pattern that is commonly used by successful students?

Currently, our tool only records and replays text-based actions. To be able to make more accurate analysis, the Screen-Replay tool will be enhanced with support for images or other types of snips.

Finally, we are planning to add support for recording the interaction window of DrScheme. This will allow the investigation for;

- When and how many times students run their programs?
- What are the common errors they get?
- Do students act according to the error messages?

which can not be answered just recording the definitions window.

7. Conclusion

Evaluating how students construct programs is difficult with conventional examinations as they evaluate the result and not the process. Evaluating student process requires observing how they construct their programs in a transparent manner. Else, we run the risk of altering their behavior.

We have developed a tool for transparently observing how students develop their programs. This tool was specifically designed to identify the sequence of activities in terms of the "How to Design Programs" (HtDP) methodology. The tool was implemented and integrated into the DrScheme environment.

Screen-replay was used to record over 100 program consructions during live examinations. Replaying these constructions enabled observers to see the exact manner in which the programs were constructed. With the Tagger tool observers were able to associate student activity with a segment of the construction. Finally, a program construction is associated with a sequence of tags that describes the entire constrution process.

Screen-replay worked well, as it perfectly revealed the entire development process of students. We find the observation process to be very interesting and insightful. It is too early to make any conclusions as of yet. Screen-replay is being used to analyze results of student examinations. The results of the analysis will be reported. Improvements to the tool are also in progress, especially in terms of improving the tagging process.

Acknowledgments

We would like to thank Vehbi Sinan Tunalıoğlu and Bülent Özel for their support to improve this tool and paper. The work in this paper is partially funded by the Boğaziçi University Research Fund BAP 07A107 and 08A103.

References

[1] J. Bennedsen and M.E. Caspersen. Assessing process and product - a practical lab exam for an introductory programming course. pages 16–21, Oct. 2006.

[2] Annette Bieniusa, Markus Degen, Phillip Heidegger, Peter Thiemann, Stefan Wehr, Martin Gasbichler, Michael Sperber, Marcus Crestani, Herbert Klaeren, and Eric Knauel. Htdp and dmda in the battlefield: a case study in first-year programming instruction. In *FDPE '08: Proceedings of the 2008 international workshop on Functional and declarative programming in education*, pages 1–12, New York, NY, USA, 2008. ACM.

[3] Stephen A. Bloch. Scheme and java in the first year. *J. Comput. Small Coll.*, 15(5):157–165, 2000.

[4] Charlie Daly and John Waldron. Assessing the assessment of programming ability. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 210–213, New York, NY, USA, 2004. ACM.

[5] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How To Design Programs*. MIT Press, Cambridge, MA, USA, 2001.

[6] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The structure and interpretation of the computer science curriculum. *J. Funct. Program.*, 14(4):365–378, 2004.

[7] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The teachscheme! project: Computing and programming for every student. *Computer Science Education*, 14(1), 2004.

[8] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. Drscheme: a programming environment for scheme. *J. Funct. Program.*, 12(2):159–182, 2002.

[9] Alessio Gaspar and Sarah Langevin. Restoring "coding with intention" in introductory programming courses. In *SIGITE '07: Proceedings of the 8th ACM SIGITE conference on Information technology education*, pages 91–98, New York, NY, USA, 2007. ACM.

[10] Herbert Klaeren and Michael Sperber. *Die Macht der Abstraktion*. Teubner Verlang, 1st edition, 2007.

[11] George Polya. *How to Solve It (Penguin Science)*. Penguin Books Ltd, April 1990.

[12] Viera K. Proulx and Tanya Cashorali. Calculator problem and the design recipe. *SIGPLAN Not.*, 40(3):4–11, 2005.

[13] Viera K. Proulx and Kathryn E. Gray. Design of class hierarchies: an introduction to oo program design. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 288–292, New York, NY, USA, 2006. ACM.

[14] David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development. *SIGSOFT Softw. Eng. Notes*, 29(4):76–85, 2004.

Get stuffed: Tightly packed abstract protocols in Scheme

John P. T. Moore
Thames Valley University, UK
moorejo@tvu.ac.uk

Abstract

This paper describes a layered approach to encoding and decoding tightly packed binary protocols. The protocols developed are based on an abstract syntax described via an s-expression. This approach utilises simple built-in features of the Scheme programming language to provide a dynamic environment that facilitates the development of extensible protocols. A tool called Packedobjects has been developed which demonstrates this functionality. An example application is presented to illustrate the flexibility of both the tool and the Scheme programming language in this domain. In particular we will show how it is possible to embed this technology into another application programming language such as C to power its network communication. Using the example application we will also highlight the choices available to the developer when deciding whether or not to embed such technology.

1. Introduction

The International Standards Organisation (OSI) 7 Layer reference model provided an academic framework for the design of network protocols and standards [7]. In comparison to the OSI model the TCP/IP model adopted a more simplified approach where amongst other changes the Presentation Layer was consumed by the Application Layer. As such, a network applications programmer needs to consider how to structure their data when transferring it across an internet. A number of competing technologies have been developed and continue to develop in this area. When it comes to structuring data in a human-readable way, XML has dominated. However, approaches to structuring binary data range from serialising native data structures to transforming an abstract syntax into a more concise binary form. Binary protocols continue to play an important role in supporting network applications. Common uses include network games and mobile communication. In addition, Google released their work on Protocol Buffers which was created to address issues they faced in the area of high performance computing [2]. In this paper we discuss Packedobjects, a tool which was originally developed for the Chicken Scheme language and is now being maintained as a Guile module [6]. Before describing Packedobjects we will first provide an overview of some relevant techniques for producing binary protocols.

Proceedings of the 2009 Scheme and Functional Programming Workshop
California Polytechnic State University Technical Report CPSLO-CSC-09-03

2. Zeros and ones

Serialising data structures for transmission across a network is a common technique. The programmer might have to handle differences in byte ordering if communication takes place across different hardware platforms. In addition, the protocol designer is restricted to describing the network protocol in terms of the native data structures available in the language used. An alternative approach might involve using an abstract syntax to describe the network protocol. This introduces some complexity. Ultimately this abstract syntax will need to be represented by the programming language. The traditional way of handling this is not dynamic. A compiler is used to transform the abstract syntax into the native language code. Typically the code generated will be combined with application specific code and linked with vendor supplied code. This is the approach which is taken by numerous Abstract Syntax Notation 1 (ASN.1) tools [1]. ASN.1 originates from the world of telecommunications. The philosophy of ASN.1 is to provide a rich abstract syntax to describe network protocols and this syntax should be transferred into binary before transmission. Different techniques, or encoding rules, can be applied to make this transition from abstract syntax to binary. The abstract syntax allows the protocol designer to think at a higher level and provides a common ground between application developers working in different programming languages. By using the Scheme programming language we can provide a more dynamic approach where s-expressions are used to describe the high level syntax. In keeping with a minimalistic tradition adopted by Scheme, we can represent a subset of the ASN.1 standard when describing our protocols. By simplifying the abstract syntax we can provide a dynamic runtime representation within an s-expression which encourages exploration in the read-eval-print loop (REPL).

3. A layered approach

Figure 1 shows how Packedobjects compares against the OSI model. At the Application Layer, Packedobjects allows the creation of buffers. A buffer contains encoded data either ready to be sent across a network or encoded data ready to be decoded. Packedobjects has been designed to allow buffers to be created within both C and Scheme. For example, the application developer can decide to use C for all network communication and therefore create the buffers in C. In either case Scheme is used to process the contents of those buffers and this takes place at the Presentation Layer. Transportation of the encoded data is shown happening at the Transport Layer. In this case we have indicated UDP is used. Packedobjects can also work over TCP, however some additional work is required to delimit application messages over this byte stream oriented transport protocol.

Application Layer	Buffer manipulation
Presentation Layer	Encoding/Decoding
Session Layer	
Transport Layer	UDP datagrams
Network Layer	Internet
Data Link Layer	
Physical Layer	

Figure 1. The OSI and Packedobjects

4. Data is code

Using an s-expression we can make use of quasiquote, unquote and unquote-splicing to help specify and manipulate our network protocol description and its values. To illustrate some of this flexibility we will use a fictitious protocol which describes shopping for food and drink. In the process we will introduce some abstract data types used by Packedobjects.

```
(define booze
  '(sequence-of
    (beer null)
    (nibbles null)))
```

We start by defining a *sequence-of null* types. The *sequence-of* type is a compound data type which consists of a repeating sequence of other data types, in this case a sequence of two *null* types. The *null* type is one of several atomic data types available in Packedobjects. It is an unusual data type in that it requires no value and is typically used as an acknowledgement in protocol specifications. More familiar atomic data types include integer, boolean, enumerated and various string types.

```
(define grub
  '(sequence
    (pizza null)
    (salad null)))
```

In addition to our drinks we should have some food. In this case we use the *sequence* data type which specifies both pizza and salad.

```
(define trolley
  `(trolley set
    (drink ,@booze)
    (food ,@grub)))

(define basket
  `(basket choice
    (food ,@grub)
    (drink ,@booze)))
```

To carry our food and drink we could use a trolley or use a basket. The trolley is large enough to carry both but we must choose between the food or drink if we use a basket. We have introduced two new compound data types. The *set* data type is a flexible type which allows an unordered sequence of other types. Any item of a set is also optional. Therefore, using the trolley we could decide to only pack some food. Using the basket we must choose between either the food or drink. The *choice* data type is a compound data type which enforces this restriction. In both cases, we have used unquote-splicing to reuse our definitions of food and drink and therefore are able to produce concise protocol descriptions. Having defined a protocol we must specify values according to their description.

```
(define thirsty
  '(basket
    (drink
      ((beer) (nibbles))
      ((beer) (nibbles))
      ((beer) (nibbles))
      ((beer) (nibbles)))))

(define hungry
  '(basket
    (food
      (pizza)
      (salad))))

(define thirsty+hungry
  `(trolley
    ,(cadr hungry)
    ,(cadr thirsty)))
```

Depending on our mood, we might be thirsty, hungry or both. In the case of being both thirsty and hungry we will use a trolley. This example illustrates the use of unquote to reuse our definitions of being hungry and thirsty. Note how we apply *cadr* to represent removing the basket from the value list ready to be placed in the trolley instead. Having defined our protocol and values we are ready to encode the data ready for transmission over a network.

```
(let* ( (bufsize 10)
        (buffer (make-buffer bufsize))
        (encoder (make-encoder buffer trolley))
        (size (encoder 'pack thirsty+hungry))
        (pdu (pdu-from-buffer buffer size)) )
  pdu)
```

The output of the encoder is a tightly packed bit stream. In this case just two bytes are required to represent:

```
(trolley
  (food (pizza) (salad))
  (drink ((beer) (nibbles))
         ((beer) (nibbles))
         ((beer) (nibbles))
         ((beer) (nibbles))))
```

Conversely, the decoder will take a tightly packed bit stream and reproduce a list of values.

```
(let* ( (bufsize 10)
        (buffer
          (make-buffer-from-string pdu bufsize))
        (decoder (make-decoder buffer trolley)))
  (decoder 'unpack))
```

The resulting output will ordinarily be equal to the original value list but in this case the *set* data type was used and this represents a special case. The ordering of decoding will match that of the protocol description. Therefore in this example we get:

```
(trolley
  (drink ((beer) (nibbles))
         ((beer) (nibbles))
         ((beer) (nibbles))
         ((beer) (nibbles)))
  (food (pizza) (salad)))
```

The process of encoding and decoding is completely dynamic. Figure 2 summarises the encoding process. The encoder obtains data by dynamically combining the values supplied with data from

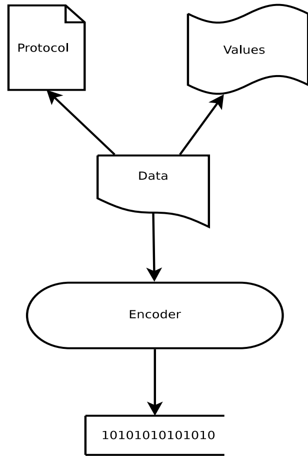


Figure 2. Dynamic encoding

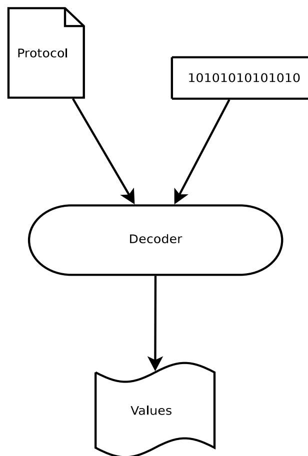


Figure 3. Dynamic decoding

the protocol specification. The encoder simply traverses the result calling the appropriate foreign functions corresponding to the underlying C based encoder. The end product is a Protocol Data Unit (PDU) which is ready to be transported across a transmission medium. The encoding produced is an unaligned bit stream based on Packed Encoding Rules (PER) [3].

The reverse process of decoding the PDU is more straight forward as summarised in figure 3. Here the protocol specification is used to drive foreign function calls to the C decoder which returns back values to Scheme to be combined into a list. Both the encoding and decoding processes illustrate how a clear divide exists between the low level C routines and the high level s-expression used to represent data. In the following section we will further describe the lower layer routines.

5. Bit fiddling

Bit manipulation is sometimes viewed as the practice of hackers [8]. In this section we will attempt to describe the techniques used by Packedobjects to pack and unpack bits in an accessible way to the reader. Both the encoder and decoder operate on words. Thus, working with strings involves multiple calls to encode or decode individual characters. As a result, protocols that are dominated with

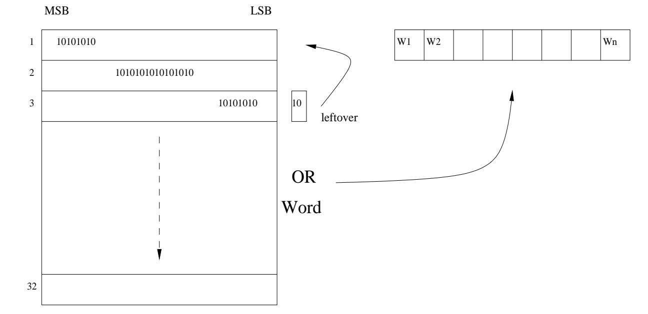


Figure 4. Encode buffers

string data are not well suited to Packedobjects. In a worst case scenario you may try to encode or decode a large amount of 8 bit strings. Packedobjects has no notion of byte boundaries, therefore the strings could start at any bit position within a contiguous block of memory. Reading the contents of such a string would require bit manipulation. The approach taken by Packedobjects is that "every bit counts". This is reinforced by the fact the default string type encodes in 7 bits. Although it may appear an extreme approach to take it does allow for a simplified view of how all types are encoded and decoded into words. The following subsections will illustrate this.

5.1 The encoder

The encoder works using two buffers [4]. One is fixed in size corresponding to the number of bits within a word, the other can be dynamically allocated. The word size is determined by the hardware platform and equals 32 bits in the example given. The fixed buffer can be visualised as an array of 32 words as depicted in figure 4. The dynamic buffer is typically created to accommodate the largest PDU so effectively operates as a statically allocated piece of memory. The fixed buffer is used to construct the bit sequences before they are copied across to the dynamic buffer. A bit sequence is copied to the appropriate word of the fixed buffer and then shifted into position. The bits are aligned so that after an OR operation on the array of words a single word is produced which can then be copied across to the dynamic buffer. This sequence is illustrated in figure 4. To begin with the bit pattern "10101010" is copied to the first word in the fixed buffer. The eight bit pattern must be shifted so that it follows the network byte order and therefore has its most significant bit (MSB) at bit position 32. The next bit pattern "1010101010101010" is added to the second word and shifted so that its MSB starts at bit 24. This leaves room for only eight more bits to be added within the third word. If, for example, the ten bit pattern "1010101010" is to be added, then the eight most significant bits would be copied to the remaining room in the fixed buffer. The entire fixed buffer then has its contents OR'ed and the resulting word is copied to the dynamic buffer. The two bits left over are put back into the fixed buffer starting from the MSB of the first word. The pseudo code for the encode algorithm is provided in figure 5. The algorithm makes just two tests to see whether a word boundary is crossed in the fixed buffer and whether a full word exists already. The algorithm is recursive. It calls itself whenever there is a value left over to encode after a full word has been copied to the dynamic buffer.

5.2 The decoder

The decoder algorithm (figure 6) is slightly more straight forward than the encoder algorithm. A PDU is decoded by masking off the desired bits to form a value. The size of a word determines the size of the window which is placed over the data to decode. The window

```
BEGIN /* encode */

   accept a number and a bit length

   IF the bit pattern is unable to fit into the space
   available in the current word of the fixed buffer THEN
      fit as many bits in as possible
      OR the fixed buffer
      copy the result to the dynamic buffer
      reset the fixed buffer
      GOTO BEGIN to encode the leftover bit pattern
      RETURN
   ENDIF

   copy the number to the correct position in the
   next word of the fixed buffer

   IF we have a full word already THEN
      OR the fixed buffer
      copy the result to the dynamic buffer
      reset the fixed buffer
   ENDIF

END */ encode */
```

Figure 5. Encode algorithm

```
BEGIN /* decode */

   accept a bit length

   IF current bit position has reached a word boundary THEN
      fetch a word from the buffer
      store a copy of the word
   ENDIF

   mask out (AND) the bit pattern from current word

   IF bit pattern crosses word boundary THEN
      fetch the next word from the buffer
      store a copy of the word
      obtain the missing part of the bit pattern
      merge (OR) the two bit patterns together
   ENDIF

   return the result

END */decode */
```

Figure 6. Decode algorithm

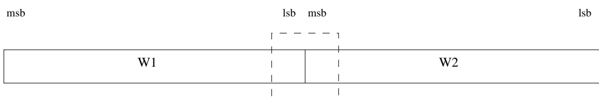


Figure 7. Decode window

moves in word sized increments over the PDU. Provisions must be made to handle bit patterns that cross over word boundaries. Values obtained from different words must be merged together to form a single bit pattern. Figure 7 shows that the area between two words can contain the desired bit pattern. As with the encode algorithm, just two test conditions exist: one to examine whether a new word should be fetched from the PDU buffer and one to examine if the value to extract lies between two word boundaries. By storing a copy of the last word obtained, it may be possible

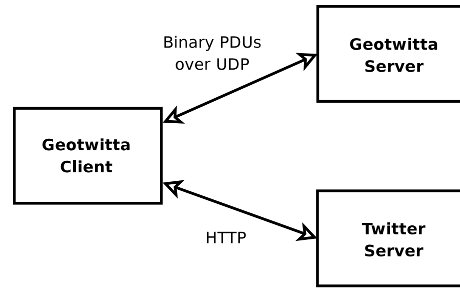


Figure 8. Application architecture

to avoid fetching a word each time the routine is called. Having described the encoder and decoder we will have completed our discussion on the layered approach of the tool and can now focus on a practical example of its use.

6. Example application

The previous sections provided an insight into the flexibility of using an s-expression to represent an abstract syntax and described its transformation into bits. In this section we will provide a more practical example that also highlights the flexibility of the Scheme language itself. We will describe an application that interfaces to the social networking and micro-blogging service Twitter. The application, known as geotwitta, is able to calculate the distance of other users and then post the result to the user's account [5]. Figure 8 summarises the architecture of the application. In order to calculate the distance of other users a server is required to manage the location of each user. Each client simply ”pings” in its coordinates to the server and in response retrieves a list of the distances of other users. Any new responses returned are then posted to Twitter using the HTTP protocol. An example post might appear as follows: #geotwitta @jptmoore appears to be about 18791.955 kilometers away from me.

6.1 The design

Although a simple protocol and simple application, it provides enough scope to show how the Scheme language and in particular Guile can be embedded inside a C application to help power the network protocol. However, we should first state some influencing design criteria for our example application other than the fact it must post to Twitter. Firstly, we want the application to be light-weight in terms of network usage. We also want to be able to easily build a packaged version which could get distributed and installed on well known Linux distributions such as Ubuntu. The first design condition is not really relevant to the client but rather to the server. We would like our low-cost server to be-able to handle multiple client requests without issues of bandwidth or load. Therefore, we shall use UDP to transport the data and Packedobjects to tightly pack the application messages (PDUs). In terms of the second design decision we would like users to be able to easily install the application without compiling from source. Scheme implementations such as Guile provide excellent support for using open source tools such as autoconf. This in turn allows us to easily apply automated routines to transfer the builds into Debian packages. Having provided some background to the design we can now discuss implementation specifics.

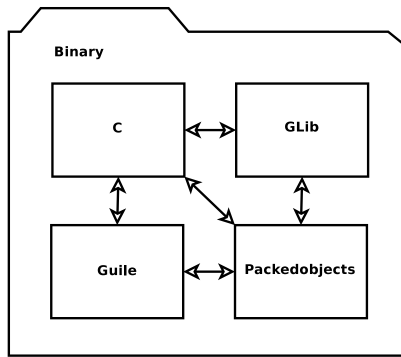


Figure 9. Client technology

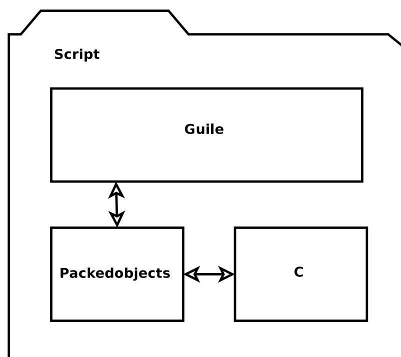


Figure 10. Server technology

6.2 Implementation choices

The client consists of a C application which uses features of GLib [1] to simplify tasks such as calling the Twitter web API. The other main feature of the client is it embeds Guile (including Packedobjects) to facilitate the encoding and decoding of network packets. Figure 9 summarises the technologies built into the client. The end product is an application binary compiled from C source code. Figure 9 shows a relationship between Packedobjects and C. Although Packedobjects is a Guile module it also heavily relies on calls to C for its low level functionality. This highlights the true flexibility with working with an embeddable language where callbacks to the host language may also occur. The server, however, takes a different approach and is completely written in Guile. In this case the end product is a script. Figure 10 summarises the technologies used. The server is simply a Guile script which uses the Packedobjects module. This illustrates one design choice available to the developer when using embeddable Scheme implementations. Do you write the application in Scheme and perhaps interface to C or do you write the application in C and embed Scheme? If the developer decides to embed Scheme into their C application, another choice exists. How much should be done in C and how much should be done in Scheme? In some cases there may be an obvious technical divide. However, often less technical factors influence the decision, such as the ability to re-use code. For example, client software that talks to well known Web 2.0 services is not difficult to find amongst various open source C based projects. Therefore, although it would not be difficult to write this functionality completely in Scheme it was more straightforward to simply use some existing C code. The end product is a binary that is not only easily distributable but also dynamically configurable.

[1] GLib is a utility library developed as part of the GNOME project.

7. Future work

Challenges exist from taking such a dynamic approach to network protocol design. Improvements to the Packedobjects tool can be made in areas such as performance and safety.

In section 4 we saw how expressive a protocol could be but how does this compare to tools like Protocol Buffers? Although subjective it provides a useful additional metric of comparison.

8. Conclusion

The designer of a network protocol must make a number of choices. The choices taken will have an impact on the size and structure of the data communicated. In some cases it is necessary to try and encode the data as efficiently as possible, in which case a binary format may be used. Similar to the way we might migrate from a low-level language and think about a problem in a high-level language, the protocol designer should not think in terms of a low-level binary format. Instead the designer should use a more expressive alternative, one that will still produce equivalent concise binary output. In this paper we presented Packedobjects, a tool which provides such an alternative. By utilising s-expressions from the Scheme programming language, Packedobjects is able to describe network protocols using an abstract syntax. This abstract syntax is dynamically transformed into a tightly packed bit stream for communication across a network. The Scheme programming language provides a number of advantages for the design of such a tool. Firstly the concept of "data is code" eliminates the need for using a compiler to transfer the abstract syntax into a concrete syntax which is usable in the native programming language. Instead we gain the benefits of using a Scheme interpreter to design and test our protocols. In addition, we obtain expressive features such as quasi-quote to help create concise and re-usable protocol definitions. The other main benefit of using Scheme for a tool like Packedobjects is that it provides some implementation specific choices. We have the choice of building solutions completely in Scheme itself but also have the ability to embed the language into a host language such as C. In this paper we have illustrated the benefits of this approach such as code reuse and the ability to easily package and distribute the application. Even though we use C as the host language we are still able to dynamically control the network protocol using the embedded Scheme.

References

[1] DUBUISSON, O. *ASN. 1 Communication between Heterogeneous Systems*. Morgan Kaufmann, 2001.

[2] GOOGLE. Protocol Buffers. http://code.google.com/p/protobuf/, July 2007.

[3] INTERNATIONAL TELECOMMUNICATION UNION. Information Technology — ASN.1 Encoding Rules — Specification of Packed Encoding Rules (PER). ITU-T Recommendation X.691, July 2002.

[4] MOORE, J. *On the Performance of Unaligned Packed Encoding Rules when Applied to a Non-optimised Protocol Specification.* PhD thesis, University of Sheffield, 2001.

[5] MOORE, J. Geotwitta. http://zedstar.org/blog/2009/05/02/geotwitta/, May 2009.

[6] MOORE, J. Packedobjects. http://packedobjects.sourceforge.net/, 2009.

[7] TANENBAUM, A. *Computer Networks*. Prentice hall PTR, 2002.

[8] WARREN, H. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.

Distributed Software Transactional Memory

Anthony Cowley C.J. Taylor
University of Pennsylvania
{acowley, cjtaylor}@seas.upenn.edu

Abstract

This report describes an implementation of a distributed software transactional memory (DSTM) system in PLT Scheme. The system is built using PLT Scheme's Unit construct to encapsulate the various concerns of the system, and allow for multiple communication layer backends. The front-end API exposes true parallel processing to PLT Scheme programmers, as well as cluster-based computing using a shared namespace for transactional variables. The ramifications of the availability of such a system are considered in the novel context of highly dynamic robot swarm programming scenarios. In robotics programming scenarios, difficulty with expressing complex distributed computing patterns often supersedes raw performance in importance. In fact, for many applications the data to be shared among networked peers is relatively small in size, but the manner in which data sharing is expressed leads to tremendous inefficiencies both at development time and runtime. In an effort to maintain focus on behavior specification, we reduce the emphasis on messaging protocols typically found in distributed robotics software, while providing even greater flexibility in terms of how data is mixed and matched as it moves over the network.

1. Introduction

Several well-studied methods for effectively distributing the execution of a program over multiple processors have emerged in response to the difficulties faced by programmers tasked with harnessing such execution platforms. A minimally invasive way to exploit a heterogeneous computing environment is to provide support for remote procedure calls (RPC). This approach has the benefit of potentially requiring only minimal changes to the surface of a program. RPC systems are valued for their ability to keep underlying inter-processor communications abstract from the point of view of the high-level program, thus providing the smooth integration of computing capabilities that are unavailable to the local processor. But when computing resources are not completely orthogonal, that is, the local, calling processor could be doing something useful while a remote processor generates a value, the RPC abstraction can be unsatisfying due to missed opportunities for concurrent execution. Put simply, RPC enables easily distributed *serial* execution.

Concurrent execution, on the other hand, brings with it sweeping implications for the semantics of distributed programs along with the desired more efficient use of available computing resources. Specifically, the original program must be modified in both control flow specification and data access restrictions. In order to avoid leaving a calling processor idle, certain actions analogous to function calls must be asynchronous on some level. That is, the callee need not finish its work before the caller is allowed to proceed. However asynchronous invocation suggest a dual mechanism designed to handle asynchronous returns. This now requires the implementation of handler functions whose ultimate place in the global execution order is not deducible from lexical inspection. To further muddy the waters, the fact that multiple parts of a program are executing simultaneously suggests that no assumption of the data dependency propositions implied by a program's text are safe. For example, Algorithm 1 may no longer be trivially reduced to $y \Leftarrow 1$ if x refers to a shared memory location.

Algorithm 1 A Seemingly Innocent Sequence

1: $x \Leftarrow 1$
2: $y \Leftarrow x$

1.1 Message Passing

One approach to eliminating data dependency ambiguities is adherence to a message passing style design. Such a design, perhaps best exemplified by Erlang [1] and its ideological offspring Termite Scheme [7], makes communication between pieces of serially executed code explicit by differentiating potentially remote communication from the common function call. Instead of being an almost transparent retrofit of standard procedural code as with RPC, the actions of sending and receiving messages are given distinct syntax and sole governorship over the interactions between bits of program code that may otherwise execute fully asynchronously. This separation of messages from function calls may, as in Erlang, be used to isolate serial execution from unintended interference from concurrently executing program code while providing a scaffolding centered around messaging protocols for distributed applications to be built upon.

While structuring programs whose identity is intrinsically distributed around the protocols that define their distribution is a productive endeavour, it can be an ill fit when the distributed nature of the program is secondary to serial algorithm complexity. In such cases, forcing a communication protocol front and center in the program code can actually hide more natural structuring techniques based around

Proceedings of the 2009 Scheme and Functional Programming Workshop
California Polytechnic State University Technical Report CPSLO-CSC-09-03

algorithmic manipulation of abstract values. Another type of situation in which explicit message passing design techniques may fall short is when connectivity between concurrent processes is highly dynamic. In such cases, it may be desirable to abstract complexity at the message passing level from core application-level code. While this is certainly possible to express in a message passing framework, it becomes less clear that message passing should be explicit at all when it is best thought of as an implementation detail.

1.2 Software Transactional Memory

Software Transactional Memory (STM) [17] is a technique for rationalizing shared memory usage in concurrent systems. Beginning with an assumption of atomicity of stores and loads of individual memory locations, composition of memory accessing operations has typically been effected by function abstraction. In this approach, compound memory mutations – in which multiple addresses are read or written – are implemented as sequential operations and often hidden behind the simpler interface of a single function call. However the era of multiprocessor machines has rendered this abstraction technique virtually useless in cases where multiple threads may be accessing overlapping memory segments. STM systems directly address this problem by providing a new abstraction specifically for compositional memory access patterns.

An STM runtime is responsible for providing transactional semantics to programmer-annotated regions of program code. This means that all operations within a particular transaction are seen by all concurrent processes as either all happening at once, or not happening at all. While this desired atomicity may be achieved by manual usage of locks to ensure mutual exclusion, an STM provides the programmer with a much simpler interface that allows for greater composability and modularity [8]. Consider a manual locking scheme governing access to two shared memory addresses identified by variables $a1$ and $a2$. These variables may each be equipped with a lock, say $lock1$ and $lock2$, respectively, that is to be acquired before a variable may be accessed. In order to write a program built on such a foundation, each function must ensure that all necessary locks are acquired before any side effects become visible to other processes, should not acquire more locks than necessary in order to retain all potential concurrency, must ensure that locks are freed in error conditions, and must abide by some agreed-upon lock acquisition ordering policy in order to prevent deadlock with processes with overlapping locking requirements [12].

In some ways, the visibility of a manual locking scheme in concurrent programs is similar to the visibility of a message passing scheme in a distributed program: both expose an underlying implementation detail at many levels of abstraction. In the case of manual locks, composition of two properly synchronized operations is burdened by the need for the composite operation to wrap itself in a union of the locking requirements of the component operations. The locking requirements are never properly abstracted.

2. DSTM in PLT Scheme

Expanding upon the example of a function that manipulates two shared locations, consider a function that transfers money between two bank accounts whose balances are stored in boxes, *a1* and *a2*, shared across multiple processes. This function randomly selects one account to have money withdrawn from it and deposited in the other account after some amount of time has passed (to simulate other work).

```
(define (transfer-unsafe)
  (let-values
      (((src sink) (if (= (random 2) 0)
                       (values a1 a2)
                       (values a2 a1))))
    (let ((amt (random (unbox src))))
      (set-box! src (- (unbox src) amt))
      (sleep (/ (random 1000) 1000.0))
      (set-box! sink (+ (unbox sink) amt)))))
```

Such a function has a social contract that it must obey that is not captured by low-level memory access semantics. First, no more can be transferred from the *src* account to the *sink* account than *src*'s initial balance (i.e. negative balances are not allowed). Second, no concurrent process should see a state where money has apparently disappeared from the system due to it being in-flight from *src* to *sink*. Note that the first constraint may be violated if *src*'s balance is reduced by a concurrent process after *amt* is chosen, while the second is violated by any process operating in the time between the two *set-box!* calls. Both of these concerns are addressed by the Distributed Software Transactional Memory (DSTM) system, here implemented in PLT Scheme [5] due to its robust macro facilities and elegant threading model.

The DSTM system provides several features accessible through a few simple operations,

- *make-tvar* makes a new transactional variable
- *set-tvar!* sets the value of a transactional variable
- *get-tvar* gets the value of a transactional variable
- *atomically* wraps a block in a composable transaction

If the variables *a1* and *a2* now refer to transactional variables, then the function may be rewritten as,

```
(define (transfer-safe)
  (atomically
   (let-values
       (((src sink) (if (= (random 2) 0)
                        (values a1 a2)
                        (values a2 a1))))
     (let ((amt (random (get-tvar src))))
       (set-tvar! src (- (get-tvar src) amt))
       (sleep (/ (random 1000) 1000.0))
       (set-tvar! sink
                  (+ (get-tvar sink) amt))))))
```

In addition to the core STM features, transactional variables are defined in a distributed shared memory space across participating peers. The above program thus demonstrates transactional manipulation of variables efficiently replicated over an abstract communication layer. The key features of this program are:

(a) Mutual exclusion is composable and flexible, requiring no resource identification by the initiator of the transactional behavior.

(b) The protocols of inter-process communication are completely abstract from algorithm specification yet optimized to package composite updates together and integrate both push and pull dissemination strategies to most effectively utilize communication resources.

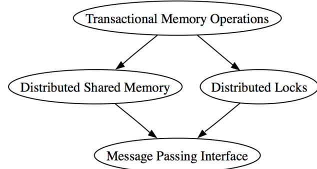


Figure 1. DSTM Architecture

2.1 Implementation Overview

The presented programming model involves the programmer annotating regions of program code that refer to shared variables whose access semantics are to be atomic. That is, any sequence of loads and stores may be treated as occurring free of the effects of any concurrent process. There is but a single annotation, *atomically*, and annotated regions may nest both lexically and dynamically. As a further assurance of proper usage, variables to be shared must be created using *make-tvar*, and may only be accessed by *set-tvar!* and *get-tvar*, which only function when called within the dynamic scope of an *atomically* block. The implementation of this system rests upon several layers of underlying functionality, shown with dependencies indicated in Figure 1. Each layer provides abstract features which enable the layer above.

The system, as described, is implemented as a composition of Units [15]. The Units mechanism provides a way to create modules parameterized by their dependencies. This is an improvement over the traditional syntactic *require* mechanism (`import` in some other languages) because the parameterization becomes part of the runtime object itself, rather than a dependency that is resolved by the compiler before any code is run. The crucial benefits of the Units mechanism to the DSTM implementation are the fact that dependencies are not coded into modules, thus elevating configuration to a first-class operation, and that they provide a clean way to share state in a controlled manner. As an example, message passing functionality is defined with a few primitive operations,

```
#lang scheme/signature
start
wait-for-peer-discovery add-new-peer-handler
fork ! ?
```

This *message-passing^* Signature specifies that a message passing Unit should support basic peer discovery hooks, the ability to *fork* new peers, and mechanisms for asynchronously sending or synchronously receiving a message, *!* and *?*, respectively. The benefit to keeping the message passing layer this abstract is that different messaging implementations may be swapped in without changing the modules that depend on that functionality. The current implementation includes a message passing layer that uses UDP multicast for peer discovery and TCP for packet transfer between peers, as well as another implementation defined entirely on top of Unix-style port operations for situations where network sockets are unavailable.

In order to allow for an expandable number of network consuming protocols, a management layer is wrapped around the low-level message passing interface. This layer is parameterized by the protocol implementations that make use of messaging capabilities, which are themselves parameterized by the underlying message passing functionality.

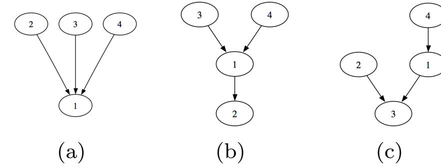


Figure 2. Lock ownership over time. (a) Initially, node 1 creates the lock and retains ownership, a fact known by every node that is aware of the lock. (b) When node 2 requests the lock, node 1 sets its `parent` pointer to node 2. (c) When node 3 requests the lock, it contacts node 1 who forwards the request to node 2. Node 1 uses this incident to update its `parent` pointer. Node 2 updates its `parent` pointer when ownership of the lock is transferred to node 3.

The management layer requires a list of Units exporting the *protocol-handler^* signature,

```
#lang scheme/signature
handle-msg initial-state discovery-handler
```

each of which is parameterized by a minimal messaging interface,

```
#lang scheme/signature
! ? my-node-id
```

which the management layer provides. The high-level DSTM system is built atop a composition of a Distributed Shared Memory (DSM) system, and associated messaging protocol, and a Distributed Locks system and protocol.

2.2 Distributed Shared Memory

The DSM implementation is not very complex because of the way functionality is expressed in a highly modular fashion. The facilities it exposes to the DSTM system are limited to a basic memory interface defined in the *dsm-interface^* signature,

```
#lang scheme/signature
dsm-store dsm-load dsm-snapshot
dsm-invalidate dsm-push dsm-pull
```

that specifies interfaces to, in order, replicate a write operation across all connected peers (remember that the DSM system takes as a parameter an active message passing mechanism), load a value, obtain a snapshot of memory contents, invalidate a particular memory location on a list of selected peers, push a write operation to a list of recipients, and pull a new value from an identified peer. The concrete representation of the DSM state is a functional hash table mapping memory locations to tuples of values and validity bits. The hash table state is passed to the DSM protocol implementation's *handle-msg* function each time a DSM-related message arrives, with the state object returned by *handle-msg* retained by the protocol manager for subsequent invocations.

2.3 Distributed Locks

Distributed mutual exclusion is implemented using a tree-based token passing algorithm due to Raymond [16]. In this algorithm, each node retains a reference to its parent in a spanning tree associated with each lock. When a node wishes to acquire a lock, it sends the request to its parent. When a node receives a request, it can grant the request if

it was holding the lock without retaining exclusive access for itself (that is to say, each lock is always held by *some* node whether or not any node is in the critical section associated with that lock), it can forward the request to its own parent if it is not the root of the tree, it can create a deferred link to the requester if it was already waiting for the specified lock itself, or it can forward the request over an existing deferred link. In this way, each acquisition request is delegated to a node's parent, and requests for a lock queue up as they percolate around the tree. The tree structure itself is dynamically updated by having each node update its parent pointer when forwarding a request up the tree. A key feature of this algorithm is that it allows for sub-groups within the network to form around a locally-contended lock.

The changing shape of the lock spanning tree is illustrated in Figure 2. In this example, a lock is initialized by node N_1. When nodes $N_2 - N_4$ learn of this lock, either during a peer discovery synchronization or on-demand resource discovery, each maintains a record of this ownership information. If N_2 wishes to acquire the lock, it sends this request along its `parent` pointer to N_1 who may grant access to the lock. At this point, N_1 updates its own `parent` pointer, which previously was a self-loop, to point to N_2. In the example, N_3 is the next to request the lock, and it sends this request to its `parent`, N_1, who forwards the request to its `parent`, N_2, and updates its own `parent` pointer with this new information.

A great benefit of this lock acquisition mechanism is the locality of agreement needed for ensuring mutual exclusion. In token ring schemes, by way of comparison, a lock token is passed among every peer in a network. If a node is not waiting to enter a critical section guarded by the lock, it simply passes the token along to the next in line. While this round robin schedule of mutual exclusion can be efficient when nodes are equally likely to be waiting for the lock, it is very inefficient when there is more structure in the patterns of lock acquisition. The tree lock mechanism, on the other hand, is more adaptable to asymmetric access patterns: lock tokens are passed among those trees closest to the root of the lock's spanning tree, while nodes that seldom acquire the lock are pushed to the leaves, and rarely, if ever, consulted.

The specific algorithm used to ensure mutual exclusion is abstract to the DSTM system itself, which simply imports the `lock-interface^` signature,

```
#lang scheme/signature
create-lock acquire release try-acquire?
```

thus leaving the door open to application configurations that rely on alternate distributed lock implementations, such as the aforementioned token ring scheme.

2.4 DSTM

The STM and its interface are heavily inspired by the Haskell implementation of STM present in GHC [8]. It is implemented here as a composition of the distributed computing components, the distributed lock mechanism, and the distributed shared memory system. A nice characteristic of this breakdown is that the locking system does not address memory stores or loads, the DSM system cares not of locks, and neither is dependent on any particular interprocess communication mechanism. When a transaction begins, a DSM snapshot is obtained and a transaction log is started. When a transaction wishes to commit, the necessary distributed locks over all nested transactional scopes are acquired in a specific order or the transaction is aborted. Once the locks are acquired, the transaction log is compared to the current state of the DSM and committed if viable. If a conflict is detected, the log is thrown away, and the transaction is re-started. Finally, the snapshot mechanism allows for Multiversion Concurrency Control (MVCC), so called due to the fact that multiple versions of the data store may be live concurrently. This model has as benefits that read operations do not block because they are guaranteed a consistent world view, and that concurrent execution proceeds optimistically, unhindered by the possibility of long running operations holding locks for their duration.

3. DSTM Applied to Concurrent Robotics

Modern robot software design often mimics a robot's modular hardware construction by building applications from asynchronously executing software modules [2, 6, 19], inspired by early work on process calculi such as CSP [11] and the π-calculus [13], as well as the Actor model of concurrent systems [9, 10, 18]. These methods of isolating concurrent processes from each other obviate concerns about shared mutable state, make potential processor boundaries more explicit via message passing operations, and, arguably, make concurrency design first class by promoting the notion of concurrent execution to a level where it is more clearly represented in the syntax of the program.

Such approaches to software design have pushed the field of multi-robot collaboration forward, yet have seen less uptake in the field of robot swarms. Robot swarm design involves harnessing the capabilities of groups of hundreds or thousands of agents to accomplish some task. In such systems, it is impossible to manually customize behaviors for each agent, so more automated approaches to behavioral differentiation are needed. Some approaches involve behaviors that naturally mutate as they spread across a population in such a way that a desired collective effect is achieved [14], while others involve reactive formulations that allow environmental inputs to guide structured behavior [4]. The latter approach, where structure emerges in response to the environment may be augmented by locally imperative behaviors at varying scales [3]. This ability may be intuitively understood as small coalitions of agents joining together to execute a coordinated action within the larger context of swarming behaviors. The most critical requirement for the expression of this capability is that spontaneous small to medium scale coordination be possible without being crushed under the scaling burden implied by enormous swarm populations.

3.1 Connectivity by Need

The ability to safely update shared estimates of various quantities, such as position, velocity, and appearance, reduces programmer burden for tasks like cooperative target tracking. When a robot observes some features of an identified target, it transactionally updates the estimates of those feature values shared by all connected robots. The ability to atomically reference and update every possible combination of shared data without explicit consideration for locks or message types is powerful, but the underlying information dissemination mechanism can not entirely sacrifice efficiency for convenience. In practice, capturing the connectivity of the network of behavioral modules becomes the meta-programming of a multi-robot system.

An alternative to separate specification of processing and connectivity is to make connectivity an implicit side effect of behavior. This approach has the advantage that it lessens

the tension between procedure design and connectivity specification that can exist in Actor-centric designs. The strategy presented here allows for *both* push and pull data sharing mechanisms to coexist, with situationally appropriate handoff between the two modes of operation. When, for example, two nodes are each repeatedly updating a shared value, the system should push updates generated by each to the other. However, when an agent has neither read nor written a shared location in some time, it is wasteful to push updates to it. Instead, such an agent should pull in fresh data when it next tries to read from the shared memory location.

As a lock is transferred between nodes, one can maintain an updated list of "interested parties" for a given datum. The current DSTM implementation manages this information as an ordered list, referred to as a *push-list*, of the most recent owners of the lock associated with a shared memory location. When a node reads a locally invalidated memory location or acquires a lock, it refreshes its local cache and adds itself to the head of the push-list. At this time, the node also cuts off the tail of the list at the position where it last inserted itself, and sends DSM invalidation messages to all affected nodes, who must then initiate a *pull* the next time they read from that location. The function for managing the push-list associated with a DSTM datum is shown below, with the minor addition that a node already at the head of a push-list will not drop the entire push-list, but rather leave it as is.

```
(define (update push-list my-id)
  (if (or (null? push-list)
          (= my-id (car push-list)))
      (values push-list '())
      (let-values
          (((a b) (break (λ(x) (= my-id x))
                         push-list)))
        (values (cons my-id a)
                (if (null? b) b (cdr b))))))
```

4. Discussion and Future Work

The DSTM system presented here allows for very specific compound data structure definitions and transfer protocols that require no specific programmer effort to establish. Instead, synchronization and communication protocols are a direct consequence of behavior specification: if a behavior depends on multiple values, then those values are safely bundled together for that behavior. The DSTM system is currently being used for simulations of scalable behaviors for mobile robots, but is also intended to serve as an operational model for a forthcoming security system in which the targets move, but the sensors do not. In such a system, one again finds different groups of sensors associated with a given shared datum as time advances. This can be dealt with by explicit target track ownership handoffs between sensor nodes, or implicitly and automatically by a DSTM system.

Acknowledgments

Rajeev Alur's CIS 640 class at UPenn.

References

[1] Joe Armstrong. The development of erlang. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 196–203, New York, NY, USA, 1997. ACM.

[2] Anthony Cowley, Luiz Chaimowicz, and Camillo J. Taylor. Design Minimalism in Robotics Programming. *International Journal of Advanced Robotic Systems*, 3(1):31–37, March 2006.

[3] Anthony Cowley and Camillo J. Taylor. Orchestrating Concurrency in Robot Swarms. In *Proceedings of the IEEE/RJS International Conference on Intelligent Robots and Systems IROS '07*, October 2007.

[4] Michael De Rosa, Seth Copen Goldstein, Peter Lee, Jason D. Campbell, and Padmanabhan Pillai. Programming modular robots with locally distributed predicates. In *Proceedings of the IEEE International Conference on Robotics and Automation ICRA '08*, 2008.

[5] Matthew Flatt et al. Reference: PLT scheme. Reference Manual PLT-TR2009-reference-v4.2, PLT Scheme Inc., June 2009.

[6] B. Gerkey, R. Vaughan, K. Stoy, A. Howard, G. Sukhatme, and M. Mataric. Most Valuable Player: A Robot Device Server for Distributed Control. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1226–1231, 2001.

[7] Guillaume Germain, Marc Feeley, and Stefan Monnier. Concurrency oriented programming in termite scheme. In *Proceedings of the Workshop on Scheme and Functional Programming*, 2006.

[8] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[9] Carl Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intelligence*, June 1977.

[10] Carl Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings 3rd International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.

[11] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1985.

[12] Simon Peyton Jones. *Beautiful Code*, chapter Beautiful Concurrency. O'Reilly, 2007.

[13] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part i. *I and II. Information and Computation*, 100, 1989.

[14] Radhika Nagpal. Programmable self-assembly using biologically-inspired multiagent control. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, July 2002.

[15] Scott Owens and Matthew Flatt. From structures and functors to modules and units. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 87–98, New York, NY, USA, 2006. ACM.

[16] Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 7(1):61–77, 1989.

[17] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[18] Gerald Jay Sussman and Guy Lewis Steele, Jr. The First Report on Scheme Revisited. *Higher-Order and Symbolic Computation*, 11(4):399–404, December 1998.

[19] R. Vaughan, B. Gerkey, and A. Howard. On Device Abstractions For Portable, Reusable Robot Code. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robot Systems*, pages 2121–2427, 2003.

World With Web

A compiler from world applications to JavaScript

R. Emre Başar, Caner Derici, Çağdaş Şenol

İstanbul Bilgi University, Department of Computer Science

{reb,cderici,csenol}@cs.bilgi.edu.tr

Abstract

Our methods for interacting with computers have changed drastically over the last 10 years. As web based technologies improve, online applications are starting to replace their offline counterparts. In the world of online interaction, our educational tools also need to be adapted for this environment. This paper presents WorldWithWeb, a compiler and run-time libraries for mapping programs written in Beginning Student Language of PLT Scheme with World teachpack to JavaScript. This tool is intended to exploit the sharing-enabled nature of the web to support the learning process of students. Although it is designed as an extension to DrScheme, it is also possible to use it in various settings to enable different methods of user interaction and collaboration.

Keywords JavaScript, web, compiler, Scheme

1. Introduction

The role of computers and World Wide Web (WWW) in our life has gone through a series of changes through the last decade. Computers used to be just a static, desktop only tool. The design of WWW also reflected that nature by being a static source of information. Although the WWW is designed to be a source for sharing information [2], it constituted a producer-consumer relationship between the author and the visitor of a web site. In that scenario users had a passive role of accessing the web, using the web browser on their computers.

Mobile networks, wireless access and availability of powerful mobile devices changed the way we interact with computers. This change in the environment also triggered a change in our approach to WWW. By the rapid growth of Web 2.0 and technologies related to it, users dropped their role as passive consumers of information, and became collaborators and creators of the content [11]. This change of role is due to the fact that Web 2.0 enabled users to create and share their content easily, without going through all the hassle of creating and maintaining a personal web site.

While the world has changed, our teaching and development tools mostly stayed the same. Although our tools for teaching programming is much better than the ones we had 10 years ago, they have not adapted to the change. One of the main drawbacks of our current tools is that they are "offline". We believe that in the age of blogs, social networks and other types of sharing media, writing a program that guesses the number you had in mind is useless unless you share it with other people.

To address this need, we developed a compiler, WorldWithWeb[1], that enables users to share their creations easily. Using WorldWithWeb it is possible to create an interactive animation using the "Beginning Student" language of DrScheme with the world.ss teachpack and publish it as a standalone application that runs in the web browser. Being able to produce a self contained, browser-based application makes it possible for the user to share it in all types of online media.

2. WorldWithWeb

The aim of WorldWithWeb is to create a bridge between DrScheme programming environment and the web. This way, users will be able to share their applications easily, enabling them to be a part of the connected world, instead of just a consumer. Using this approach it is much more easier for users to share their creations and get feedback from different people around the world (not just their friends, teachers or families) and more importantly gain online reputation.

To accomplish this goal, we need to enable the user to speak the "lingua franca" of the web: JavaScript [1]. Although JavaScript is a language with a fine set of features and tons of libraries for creating online applications, it is not a perfect fit for pedagogic purposes. A pedagogic language, as defined by Felleisen et al [5] needs to be simple and light as far as possible. DrScheme's "Beginning Student" language (BSL) and other languages at the HtDP category are designed with this purpose in mind, and they are a perfect fit for pedagogic methodology behind HtDP.

For this purpose we created a compiler that compiles programs written in the BSL to JavaScript. This way, a student can create a program in the DrScheme environment, using a language that is designed to help his/her learning process. Then he/she can automatically create a web page containing the application, sharing it over the web with the rest of the world.

2.1 A note on Beginning Student Language and World.ss

DrScheme development environment is able to restrict, or extend a user's access to the underlying language. This feature is called "Languages". One of these languages, is the "Beginning Student" language, which is used by HtDP [4]. This language is trimmed down to restrict the user to a nearly-pure functional subset of Scheme without higher-order functions. Although the BSL provides only some basic utilities for writing programs, it is possible to extend the language with libraries called teachpacks [15].

[1] For source code, screenshots, demos and other information please visit: http://vc.cs.bilgi.edu.tr/trac/worldwithweb

Proceedings of the 2009 Scheme and Functional Programming Workshop
California Polytechnic State University Technical Report CPSLO-CSC-09-03

World.ss [3] is a teachpack designed for creating interactive animations in a purely functional programming style. The imperative parts of creating an animation is handled by internals of world.ss package. This way, the student is left with a purely functional programming interface where he/she can design and code the animation focusing on the design methodology as imposed by HtDP.

3. Related Work

The Moby Scheme Compiler [8] is an effort to make programs written in BSL and world.ss available on mobile devices. It also contains some extensions to world.ss package. Using those extensions, it is possible for the user to create applications that exploits the extra functionality (like GPS or tilting detection) provided by those devices, while staying within the BSL. While Moby opens up new possibilities in front of students for creating applications that runs on devices other than classical PC's, it does not solve the problem of sharing those applications with each other.

scheme2js [9] is a compiler from Scheme to JavaScript, intended to provide complete interoperability between JavaScript and Scheme. While it has similar goals with WorldWithWeb, it should be considered as a foreign function interface between Scheme and JavaScript. Although it is possible to use scheme2js as a tool for implementing low level interfaces of WorldWithWeb, the complex relations between PLT Scheme GUI libraries and underlying OS makes it impossible to use that kind of low-level implementation directly.

Processing.js[2] is an implementation of the Processing [14] for JavaScript. Although it presents the same API to the users, it does not provide any tools for converting from Java to JavaScript automatically. To be able to run a Processing application with Processing.js the user needs to re-write the entire application using JavaScript.

O'browser, which is developed within the Ocsigen project[3] is a virtual machine for OCaml, written in JavaScript. It supports loading OCaml bytecode directly into the VM without any need for recompiling. Although it provides the advantages of a statically typed functional programming language, users need to have previous knowledge about HTML and DOM to create applications for the web. This disadvantage creates a barrier for the beginning student to create applications using that framework.

4. Implementation

The design of WorldWithWeb is based on two distinct parts. One part is the core compiler, which translates the Scheme code to JavaScript. The second part is the runtime libraries, implemented in pure JavaScript. This strict separation allows us to freely experiment on the implementation of runtime, while keeping the compiler as small as possible.

4.1 Compiler

Since the web provides the user with many different methods (i.e. blogs, forums, social networking sites) for publishing the content, our implementation also needs to be adaptable to various media. Therefore, WorldWithWeb is designed as a library that can be used by various frontends to create final output. This way, the user can create new frontends that can read code from any kind of resource and to create output that is appropriate for the destination medium. Currently the only output format is HTML, linked with required JavaScript libraries and user code.

The output of the compiler is a `pinfo` structure which is defined as:

[2] http://www.processingjs.org

[3] http://www.ocsigen.org

```
(define-struct pinfo (code
                      function-mappings
                      tests
                      images))
```

Code field of the structure is the generated JavaScript code for the program. The source code is contained as a string, so it can be used directly with `display` or similar functions to create the main JavaScript file.

Function mappings are provided as a bridge between BSL functions and the JavaScript libraries that implements them. They simply rename the functions available from the libraries to their Scheme counterparts. Since the inclusion of this map might be accomplished using different methods in different environments, it's left to the backend to decide to include them or not.

The tests in DrScheme testing framework need a different evaluation order. They need to be evaluated after all of the other top-level expressions. Otherwise it becomes hard to test functions, especially for mutually recursive cases. To satisfy this need, tests are stored separate from the user's code and presented as extra information to the frontend application. This also allows the frontend to remove the tests if the destination platform is not appropriate for running tests.

DrScheme allows a user to embed images directly into source code. Since there is no direct method for accomplishing this in JavaScript, through the compilation process, embedded images are extracted from source code, and saved as image files to the disk. Images in source code are then replaced with a function call that loads those images on demand. The images field of the `pinfo` structure contains these names, in the order of appearance in the source code.

4.2 Data type correspondence

One of the most important challenges when translating one language to another is to define the data structures of the source language in the terms of destination language. Scheme, and more generally Lisp family of languages, are especially famous in that area because of their unorthodox nature of implementing various language features, such as numbers, Object Oriented Programming techniques etc... While JavaScript has some properties of functional programming languages, like higher order functions, it also lacks some features, like exact numbers or symbols of Scheme. Therefore, while it is possible to create a one to one correspondence in some data types like functions or booleans, many other data types requires special handling.

4.2.1 Numbers

Numbers in WorldWithWeb follows the Scheme number model closely, the numeric tower is implemented fully, including support for big numbers. The support for exact numbers is implemented using Matthew Crumley's BigInteger library[4].

The numeric tower is implemented using a class hierarchy that matches the hierarchy of numbers in the numeric tower. Inexact numbers of Scheme are double precision floating point numbers as defined by IEEE754 [7]. Since this model exactly fits to the JavaScript number model, JavaScript numbers are directly used for implementing inexact numbers.

4.2.2 Symbols

Symbols are one of the most interesting data structures that differentiate Lisp family of languages from others. Although a symbol is similar to a string in other languages, unlike a string it is guaranteed to be unique. While in many other scenarios symbols might

[4] http://silentmatt.com/biginteger/

be simply represented as strings, keeping the uniqueness invariant is important in this case, since the identity equality depends on that uniqueness feature. To accomplish this, symbol constructors are wrapped within a function that maintains a hash table of symbols produced so far. That way, if a requested name is already in the symbol table, it is returned. If the symbol is not in that table, a new symbol object is created and added to the table.

4.2.3 Characters

In many programming languages strings are just modeled as an array of characters. JavaScript, however, follows a different approach. In JavaScript there is no concept of a character. Instead, characters are modeled as strings with length 1. While a pragmatic approach might recommend to implement the same method, it is impossible to follow this method for implementing Scheme characters.

In Scheme, strings and characters are two distinct data types. There are also many functions that operate between these domains. For this reason, we decided to follow the Scheme approach and put a clear distinction between characters and strings. A character in WorldWithWeb is represented as a simple structure, holding an exact number, the Unicode code point of that character.

4.2.4 Structures

Structures in Scheme are simple data types to hold compound data. In BSL, a structure definition consists of a name and a list of fields. That definition introduces not only the structure itself but also a constructor, field accessors and equality tester. Since BSL is a purely functional subset of Scheme, structure definitions in BSL do not introduce field mutators.

Structures in WorldWithWeb are modeled as objects in the JavaScript's prototype based object system. A structure definition introduces an object which is built by the constructor function. Fields of the structure correspond to the the object's fields. Also field accessors and equality tester are defined as ordinary functions that work on object's fields.

4.2.5 Images

Being able to embed images directly in source code of a program is one of the most interesting features of DrScheme. This way, images can be used and modified like any other value in the language. Unfortunately JavaScript has no support for directly embedding images in the source code[5]. To handle this problem in a compatible way, images in the source code are saved as resources and they are replaced with a call to a function which loads the image on demand.

4.3 JavaScript Libraries

The second part of WorldWithWeb is the supporting libraries, written in pure JavaScript. These libraries imitate the primitive functions found in BSL and the world.ss teachpack. The libraries are designed to be exact imitations of their Scheme counterparts. For this reason, they consume the same number of parameters, produce same types of values as their Scheme counterparts and raise the same kinds of error messages.

4.3.1 Beginning Student Language

BSL contains the most basic functions for users to create applications. While its contents are limited to a subset of Scheme, it is big enough to let the users create useful programs. It mainly consists of number and string operators.

The BSL functions are implemented as ordinary JavaScript functions, using the datatypes mentioned above. The naming of the functions follows the naming scheme used by Moby Scheme Compiler.

4.3.2 Testing

There are two motivations behind the testing implementation of WorldWithWeb. First motivation is that testing is a crucial part of programming education. Getting used to writing proper test cases is important and the user should get feedback for his/her tests. While the user can test his/her program in DrScheme environment, it is also an extra safety measure to see that his/her tests pass on the web interface too.

Secondly, all projects need testing. WorldWithWeb is no exception. All test cases for WorldWithWeb libraries are written in Scheme and then compiled to JavaScript. This way we can be sure that as long as the tests pass in both environments our libraries are fully compatible with their Scheme counterparts.

The Scheme testing library provides three testing primitives. Two of them, `check-expect` and `check-within` are no different than any other function call. They are directly implemented as functions. On the other hand `check-error`, which checks if a given expression produces a certain error message cannot be implemented directly.

The error mechanism in JavaScript works using exceptions and in the evaluation order of JavaScript it is not possible to catch exceptions that happen while the arguments of a function are being evaluated. To handle this case, WorldWithWeb compiler wraps the expression that is expected to raise the error in an anonymous function, effectively delaying the evaluation. Later, when the test is evaluated the function is called and the expected exception is catched by ordinary exception handlers, and tested against the expected value.

4.3.3 Images

The world.ss teachpack consists of two parts. One part is the image.ss library which concentrates on creation and manipulation of images and shapes in various ways. The other part is the world.ss which manages the events from the outside world and controls the flow of world state between handlers for those events.

The images library is an imitation of image.ss teachpack found in DrScheme and used by the world.ss package. The library provides all of the functions that enables user to create static images and compose them in various ways. The JavaScript implementation also provides same primitive shapes and covers most of the image manipulation functions.

Images in WorldWithWeb are modeled as objects. All image objects provide a set of methods that makes it possible to use them in a generic manner. These methods include width & height calculations, pinhole alignment and a method called `__draw`.

The Scheme implementation of images relies on the underlying drawing primitives, provided by the GUI [6] framework. The images in that implementation are created as bitmap instances. This method allows the images library to use the methods of bitmap objects for all kinds of width/height calculations. Unfortunately, this approach is not possible in WorldWithWeb since it will require the implementation of a GUI style drawing system in JavaScript. Width and height calculations in WorldWithWeb is done directly by images themselves. For most of the primitive shapes, that approach works directly by applying appropriate formula for the shape. It is also possible to calculate this data for composite images. Although most shapes are implemented in a straightforward way, some of them (like triangles) do not produce the same result with their Scheme counterparts. This approach also fails on calculating the width and height of text objects.

The main challenge about images is the drawing of images on an HTML canvas element. The canvas element is a simple container,

[5] Actually, images can be embedded in data urls with base64 encoding, but this feature is not available in all browsers.

for a drawing context object. The drawing context provides simple drawing primitives like lines, bezier curves etc... The drawing of a world scene is accomplished by passing this drawing context through all the image objects in the current scene.

All image objects are designed to have a method called `__draw` which has three parameters. The first parameter is a drawing context, provided by a canvas element. The second and third parameters are the coordinates that the image object should draw itself. While ordinary objects just draw themselves in the drawing context, overlays, scenes and other composite objects manage the drawing of their sub-objects on the drawing context, passing the context from one element to another in the correct order.

4.3.4 World

As we mentioned in the previous topic, the world.ss library provides the abstraction mechanisms for the events coming from outside world. Although this is the main role of world.ss, it also provides some extra drawing primitives for creating "scenes".

In world.ss terminology, a scene is an image with a pinhole at 0,0 coordinates. For this reason, we implemented scenes as an extension of ordinary drawing primitives, provided by images library.

The main role of the world.ss is handling events. This is implemented in terms of timers and DOM events [13]. Mainly all handlers are defined as wrappers around the user-defined handler functions. For each event, the world is updated by the results of the handler function, and redrawn.

There are mainly three kinds of events in world.ss model. The first kind of event is the tick event, which is independent of the user interaction. It is the simplest event, which calls the handler function in each time tick. This handler is implemented as a JavaScript timer. JavaScript provides a `setTimeout` function which calls the provided handler in given periods.

The second kind of event is the user input. In plain world.ss, the only input user can provide is by using keyboard and mouse. The input from those devices are handled by key and mouse event handlers. In JavaScript, it is possible to access these events using event listeners. Each DOM object provides an interface, `addEventListener`, which allows the user to hook into the events occurring on that element. Using this interface, handler functions can access the details of the event (character code, mouse coordinates, modifier keys etc...). The handler wrappers for mouse and keyboard events get the raw JavaScript events and provide that information into user's handler functions in a format that imitates the Scheme interface.

The last event type is the redraw events. Redraw of the scene is actually not a real event but it is triggered by all kinds of handler events for the scene to represent the current state of the world in the window. Redraw events are currently invoked by handler functions, in a manual fashion. Each handler function invokes the redraw handler after changing the world.

Another handler, which is not tied to an event is `stop-when`. The `stop-when` handler decides when the animation should stop. When the condition checked by the handler is satisfied, all other event handling stops, effectively stopping the animation.

5. Possible Applications

While WorldWithWeb is mainly designed to be used within DrScheme programming environment, it is possible to use it as a library to provide different kinds of functionality from all types of applications. This way user might be provided with a richer environment which is adapted to his/her development environment. While it is possible to use the library to integrate user's application to various web services as discussed before, it might also be used in many different setups providing different services.

5.1 Interactive Environment

As Papert [12] noted in 1980's, programming plays an important role in a child's learning process. Unfortunately, many of the students around the world have no access to a personal computer of their own and need to use public computers instead. Because of the locked down nature of those public access computers, most of the time, the student will not have access to DrScheme environment. For many types of software, this problem is solved by rich Internet applications. This way, users have the opportunity to access their spreadsheets, instant messaging systems and all kinds of documents from anywhere in the world.

Following this idea of online applications, it is possible to create an online programming editor that will provide a simple editing environment for Scheme code. The user can write his/her code in that environment, and then send the code to the server by clicking the "Run" button on the web page. The server will compile the application to JavaScript and send it as a response to the user's web browser, to be evaluated. That way, the user can see the result of his/her code directly in the web browser without the need for any other tool.

5.2 Gadget-like applications

As we mentioned earlier, sharing lies in the heart of web. While there are many different methods for sharing different kinds of content, social networking sites became hubs where the sharing gets centralized. One of the most important feature of social networking sites are the "gadgets" they present to their users. A gadget is a small application, created using JavaScript and HTML. With initiatives like OpenSocial [10], it is possible to create a gadget and share it with other people across different social networking sites.

Since gadgets are just composed of HTML and JavaScript embedded inside a meta data container, it is possible to use WorldWithWeb to create these gadgets automatically. All that's required is to create a frontend that generates appropriate XML structure from the generated code.

Enabling this kind of sharing might be a real boost for the user motivation, since the user's application directly becomes a part of a network that is built especially for sharing purposes.

6. Conclusion & Future Work

World.ss is a library that provides abstractions that enables users to create complicated animations and simulations without going into the imperative roots of creating an animation. WorldWithWeb takes this one step forward, providing the user to share his/her creation on the web without worrying about the underlying protocols or languages. Although it does provide the user with everything he/she needs to create an interactive animation, there is still room for improvement.

6.1 Universe.ss

Universe.ss is a teachpack extending world.ss by enabling the user to create multiuser client/server applications like multiplayer games while staying in a purely functional programming environment. The core concept in universe.ss is the "message". A message is a simple packet of information contained in an S-Expression [15]. The client environment communicates with the server, and other clients using messages.

The application model proposed by universe.ss fits perfectly into the development model of Web 2.0 applications. A Web 2.0 application communicates with the server and other clients using simple messages encoded using various methods. One of these encoding methods is the JavaScript Object Notation (JSON) which makes it possible to encode any kind of JavaScript data in a simple text only format. Most Web 2.0 applications work by passing Java-

Script objects encoded in JSON format between client and server. The server application distributes this data to other clients using polling techniques.

This correspondence in methodology makes it possible to extend WorldWithWeb to cover universe.ss. The implementation of single user applications in universe.ss is trivial, since that part of the universe.ss is nearly the same with world.ss. The main difference of universe.ss shows itself in multi-user applications. To implement multi-user applications, the messages need to be encapsulated and sent/received using XMLHttpRequest's [16].

The implementation of the server side is a much more interesting problem. That application can be modeled as a proxy in front of the universe.ss server. The proxy can capture the JSON encoded objects, create the corresponding Scheme object, and pass it to the actual universe.ss server. Then the inverse of this method can be used to encode objects from universe.ss server going to the clients. The most important advantage of this method for implementing the server side is that the server application can be used as-is without any modifications.

6.2 Better Scheme Semantics

As noted by Loitsch and Serrano, compiling Scheme to JavaScript while keeping the Scheme semantics intact is not a simple process. The lack of first class continuations and similar techniques makes it impossible to make a direct translation between two languages.

Since WorldWithWeb just focuses on a subset of Scheme language, some of these problems (like first class continuations) disappear by themselves. However, some other things like exact numbers and proper tail calls remains. As we mentioned earlier, WorldWithWeb already implements exact numbers using a fractional numbers library.

Proper tail calls are currently not implemented. While this is not a serious problem for small applications, as the applications get more complicated, it is possible to hit a memory barrier. We are currently working on various methods for implementing tail calls in JavaScript.

6.3 Other Language Levels

While compiling BSL programs might motivate the student to a certain level, the tools should be available to him/her through the learning process. This requires adding support for other language levels to the compiler.

Although it is rather simple process to add support for language levels like "BSL with List Abbreviations" or "Intermediate Student" by just extending the library functions, the addition of "Intermediate Student with Lambda" language will require the addition of higher order functions and anonymous functions. This addition, probably will not be so hard, considering that JavaScript already has support for higher order functions and anonymous functions.

Adding the "Advanced Student" language will probably be the hardest part, since that language level introduces mutation. Introduction of mutation into the language creates two important requirements for the compiler: Sequencing and function call semantics.

In purely functional languages the evaluation order of the expressions do not effect the result of the computation. Breaking purity by introducing mutation requires that expressions are evaluated in correct order. When values can be mutated, the programmer should be able to know beforehand when the mutation will happen to be able to reason about the program. For this reason compiling a language with mutation should not effect the evaluation order. Since JavaScript is not designed to be a purely functional language, it already contains proper sequencing constructs. The only thing to be done is to make sure that JavaScript sequencing semantics are in correspondence with the Scheme sequencing semantics.

The second problem is preserving the function call semantics. In the presence of mutation, the results from functions might depend on the function call strategy used. To make sure that the JavaScript version of the program produces the same results with Scheme version, function calls might require special treatment.

Acknowledgments

We would like to acknowledge Chris Stephenson, M. Fatih Köksal and E. Pınar Hacıbeyoğlu for their support through the development process of WorldWithWeb and the preparation of this paper.

References

[1] *ECMAScript Language Specification*. 1999.

[2] Tim Berners-Lee. Information management: A proposal. *CERN, March*, 1989.

[3] Matthias Felleisen, Robert B. Findler, Kathi Fisler, Matthew Flatt, and Shriram Krishnamurthi. How to design worlds, 2008.

[4] Matthias Felleisen, Robert B. Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs*. MIT Press Cambridge, Mass, 2001.

[5] Matthias Felleisen, Robert B. Findler, Matthew Flatt, and Shriram Krishnamurthi. Structure and interpretation of the computer science curriculum. *Journal of Functional Programming*, 2004.

[6] Matthew Flatt, Robert B. Findler, and John Clements. GUI: PLT graphics toolkit. Reference Manual PLT-TR2009-gui-v4.1.5, PLT Scheme Inc., March 2009.

[7] Matthew Flatt and PLT Scheme. Reference: PLT scheme. Reference Manual PLT-TR2009-reference-v4.1.5, PLT Scheme Inc., March 2009.

[8] Shriram Krishnamurthi. The moby scheme compiler for smartphones. In *Proceedings of the International Lisp Conference*, 2009.

[9] Florian Loitsch and Manuel Serrano. Compiling Scheme to JavaScript.

[10] J. Mitchell-Wong, R. Kowalczyk, A. Roshelova, B. Joy, and H. Tsai. Opensocial: From social networks to social ecosystem. pages 361–366, Feb. 2007.

[11] Tim OReilly. What is web 2.0: Design patterns and business models for the next generation of software.

[12] Seymour Papert. Redefining childhood: The computer presence as an experiment in developmental psychology. In *Proceedings of the 8th World Computer Congress: IFIP Congress*, 1980.

[13] Tom Pixley. Document object model (DOM) level 2 events specification. *W3C Recommendation, November*, 2000.

[14] Casey Reas, Ben Fry, and John Maeda. *Processing: A Programming Handbook for Visual Designers and Artists*. MIT Press Cambridge, Mass, 2007.

[15] PLT Scheme. Teachpacks. Reference Manual PLT-TR2009-teachpack-v4.1.5, PLT Scheme Inc., March 2009.

[16] Anne van Kesteren and Dean Jackson. The XMLHttpRequest object. *World Wide Web Consortium, Working Draft WD-XMLHttpRequest-20070618*, 2007.

Peter J. Landin (1930–2009)

Olivier Danvy
Department of Computer Science,
Aarhus University
Aabogade 34, DK-8200 Aarhus N, Denmark
danvy@cs.au.dk

Abstract

This note is a prelude to a forthcoming special issue of HOSC dedicated to Peter Landin's memory.

—

One of the founding fathers of everything lambda in programming languages passed away in June 2009: Peter J. Landin.

Peter Landin spent the last years of his life as Professor Emeritus at Queen Mary, where his colleagues included Edmund Robinson and Peter O'Hearn. Over the last decade, he served in the advisory board of HOSC and thus received a complimentary copy of each issue. HOSC published two of his articles: a tribute to Christopher Strachey [21] in 2000 and a reprint of his 1965 technical note "A generalization of jumps and labels" [20] in 1998, for which Hayo Thielecke wrote an introduction [30]. In the editorial of our 1998 issue [11], Carolyn Talcott and I presented this reprint as follows:

> This paper describes a real conceptual discovery, namely the idea to make control facilities first-class entities in a programming language, through the "J operator." Its exposition is typical of the simplicity, directness, clarity and honesty of Landin's writing that makes his articles such a pleasure to read.

This spring, for the last time, I sent him a copy of a scientific compliment: a joint re-visitation with Ken Shan and Ian Zerny of his direct-style embedding of Algol 60 into applicative expressions with the J operator [15]. There, we retarget his embedding to the Rhino implementation of JavaScript with continuation objects. Indeed, whereas call/cc captures the current continuation, the J operator captures the continuation of the caller of the current method [7]. This feature fitted Landin's embedding then and it fits a JavaScript implementation with a local stack for each method now [3]. Playfully, the title of our re-visitation is thus "J is for JavaScript" [10].

—

In 2004, I paid Peter Landin a visit at the occasion of Josh Berdine's PhD defense [2] and found him in his office, patiently helping an undergraduate student. In turn, I patiently waited for him to be done with the student before presenting him my rational deconstruction of his SECD machine [5]. I then showed him how the SECD machine could be put into defunctionalized form [9, 25] and could then be refunctionalized [8] into a continuation-passing evaluator à la Lockwood Morris [22]. I thus enthusiastically concluded how much the SECD machine made sense, and that even though he might not have discovered continuation-passing style (see Appendix), defunctionalization and refunctionalization provided a concrete argument why his name should be added to the list of the discoverers of continuations [26]. Throughout, he was as patient with me as with the undergraduate student, and in the end he smiled, his eyes sparkled amusedly, and then he made some incredibly modest comments to the effect that he had been lucky.

Peter Landin was indeed so modest that in 1998, he did not attend the MFPS XIV session held in his honor at Queen Mary,[1] eliciting Dana Scott's quip as to whether Peter Landin was the Bourbaki of Computer Science. He did, however, get to read hardcopies of the slides displayed at his session.

—

I initially got in touch with Peter Landin in 1996 by e-mail and by phone and we met for the first time in January 1997 in Paris, at the occasion of the Second ACM SIGPLAN Workshop on Continuations [4], which I was chairing and where he gave a keynote speech. For the proceedings, he wrote the masterfully idiosyncratic "Histories of

[1] <http://www.dcs.qmul.ac.uk/~edmundr/mfps/>

Proceedings of the 2009 Scheme and Functional Programming Workshop
California Polytechnic State University Technical Report CPSLO-CSC-09-03

Discoveries of Continuations: Belles-Lettres with Equivocal Tenses" [19].[2]

After his keynote speech at CW'97, he handed out copies of some of his old research reports [16, 17]. There naturally was a stampede, and to Olin Shivers who asked his copies to be autographed he said "I am not the Beatles," and then signed them.

When introducing him before his keynote speech, I pointed out that independently of all his accomplishments, he was a rare breed of computer scientist with a control operator as his middle name (which is "John" and is abbreviated "J" in his publications). He flashed a look at me that to this day makes me wonder whether it was such a good idea to mention this coincidence at all.

My favorite moment with Peter Landin occurred when we met: he was arriving from London to attend CW'97, I picked him up at the train station, and together with John Reynolds and Andrzej Filinski, we sat at the terrace of a French café. I took the opportunity of a pause in the conversation to venture the question as to whether in their mind, the evaluation order of the meta-language of denotational semantics was call by value or call by name. Peter and John immediately, and simultaneously, answered "call by value of course" (for Peter) and "call by name of course" (for John). For a second of eternity, they looked at each other. Then it was like they were mentally telling each other "let's not have this discussion again" and the universe resumed its course. The rest of the evening was warm and pleasant, the following day was as wonderful as each continuation workshop somehow manages to be, and eventually I took him back to the train station.

—

What happened before is history: his impression that computer science was turning "too theoretical" for him, his quiet move away from the programming-language limelight, and his ascension to programming-language legend. Peter Landin was indeed gifted with an uncanny, almost prophetic, computational sense. To (boldly) quote from the introduction of my rational deconstruction of his SECD machine [5]:

> Forty years ago, Peter Landin wrote a profoundly influential article, "The Mechanical Evaluation of Expressions" [14], where, in retrospect, he outlined a substantial part of the functional-programming research programme for the following decades. This visionary article stands out for advocating the use of the λ-calculus as a meta-language and for introducing the first abstract machine for the λ-calculus (i.e., in Landin's terms, applicative expressions), the SECD machine. However, and in addition, it also introduces the notions of 'syntactic sugar' over a core programming language; of 'closure' to represent functional values; of circularity to implement recursion; of thunks to delay computations; of delayed evaluation; of partial evaluation; of disentangling nested applications into where-expressions at preprocessing time; of what has since been called de Bruijn indices; of sharing; of what has since been called graph reduction; of call by need; of what has since been called strictness analysis; and of domain-specific languages—all concepts that are ubiquitous in programming languages today.

And did I mention that together with his embedding of Algol 60 into applicative expressions, his 700 article [18] is generally recognized as the origin of domain-specific languages today?

On so many fundamental and tasteful ways Peter Landin was unerringly right. He has now passed away, but his writings stay and his discoveries, his inventions, and his middle name live on.

—

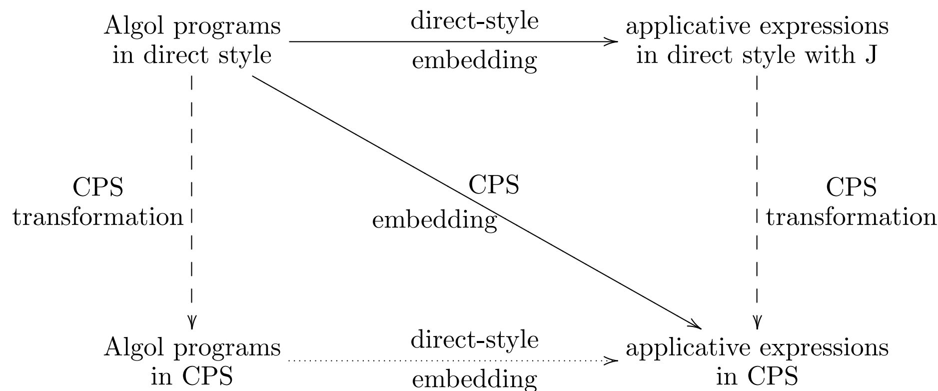
Appendix: As John Reynolds pointed out in our columns [26], Peter Landin did not discover continuation-passing style—instead, he invented control operators and first-class continuations (see Figure).

The left vertical arrow is a tour de force due to Adriaan van Wijngaarden [31] and James Morris [23]. The top horizontal arrow is due to Peter Landin [15]. ("Applicative expressions" is Peter Landin's words for "λ-terms.") The diagonal arrow is variously due to Christopher Strachey and Christopher P. Wadsworth [29] and to Kamal Abdali [1], and its unstaged version is due to Lockwood Morris in the form of a definitional interpreter in continuation-passing style [22, 25]. In the pure case (i.e., without the J operator), the right vertical arrow is due to Michael Fischer [12] and has been formalized by Gordon Plotkin [24] and put to compiler use by Guy Steele [28], who extended it to the impure case and introduced the acronym "CPS" and the term "CPS transformation." The bottom horizontal arrow is obvious. In Landin's direct-style embedding, label declarations are mapped to an occurrence of the J operator that gives rise to a 'program closure' (known today as a "first-class continuation" [13]), and jumps to a label are mapped to an application of the program closure lexically associated to this label. Peter Landin used to joke that he had smuggled the J operator into a galley proof [15].

"In those days [the 1960's],
many successful projects started out
as graffitis on a beer mat
in a very, very smoky pub."
Peter J. Landin, 2004

Acknowledgments: This note benefited from Irène Danvy, Julia Lawall, Karoline Malmkjær, and Ian Zerny's sensible proof-reading.

[2] Including "So these continuations have continuations." which beautifully anticipates the CPS hierarchy [6].



References

[1] S. Kamal Abdali. A lambda-calculus model of programming languages, part II: Jumps and procedures. *Computer Languages*, 1(4):303–320, 1976.

[2] Josh Berdine. *Linear and Affine Typing of Continuation-Passing Style*. PhD thesis, Queen Mary, University of London, 2004.

[3] John Clements, Ayswarya Sundaram, and David Herman. Implementing continuation marks in JavaScript. In Will Clinger, editor, *Proceedings of the 2008 ACM SIGPLAN Workshop on Scheme and Functional Programming*, pages 1–9, Victoria, British Columbia, September 2008.

[4] Olivier Danvy, editor. *Proceedings of the Second ACM SIGPLAN Workshop on Continuations (CW'97)*, Technical report BRICS NS-96-13, Aarhus University, Paris, France, January 1997.

[5] Olivier Danvy. A rational deconstruction of Landin's SECD machine. In Clemens Grelck, Frank Huch, Greg J. Michaelson, and Phil Trinder, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, number 3474 in Lecture Notes in Computer Science, pages 52–71, Lübeck, Germany, September 2004. Springer-Verlag. Recipient of the 2004 Peter Landin prize. Extended version available as the research report BRICS RS-03-33.

[6] Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

[7] Olivier Danvy and Kevin Millikin. A rational deconstruction of Landin's SECD machine with the J operator. *Logical Methods in Computer Science*, 4(4:12):1–67, November 2008.

[8] Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Science of Computer Programming*, 74(8):534–549, 2009. Extended version available as the research report BRICS RS-08-04.

[9] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the research report BRICS RS-01-23.

[10] Olivier Danvy, Chung-chieh Shan, and Ian Zerny. J is for Javascript: A direct-style correspondence between Algol-like languages and Javascript using first-class continuations. In Walid Taha, editor, *Domain-Specific Languages, IFIP TC 2 Working Conference, DSL 2009*, number 5658 in Lecture Notes in Computer Science, pages 1–19, Oxford, UK, July 2009. IFIP, Springer.

[11] Olivier Danvy and Carolyn L. Talcott, editors. *Special Issue on the Second ACM Workshop on Continuations (CW 1997), Part I*, volume 11, number 2 of *Higher-Order and Symbolic Computation*, 1998.

[12] Michael J. Fischer. Lambda-calculus schemata. *LISP and Symbolic Computation*, 6(3/4):259–288, 1993. Available at <http://www.brics.dk/~hosc/vol06/03-fischer.html>. A preliminary version was presented at the ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1, January 1972.

[13] Daniel P. Friedman and Christopher T. Haynes. Constraining control. In Mary S. Van Deusen and Zvi Galil, editors, *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 245–254, New Orleans, Louisiana, January 1985. ACM Press.

[14] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[15] Peter J. Landin. A correspondence between Algol 60 and Church's lambda notation, Parts 1 and 2. *Communications of the ACM*, 8:89–101 and 158–165, 1965.

[16] Peter J. Landin. A generalization of jumps and labels. Research report, UNIVAC Systems Programming Research, 1965. Reprinted in Higher-Order and Symbolic Computation 11(2):125–143, 1998, with a foreword [30].

[17] Peter J. Landin. Getting rid of labels. Research report, UNIVAC Systems Programming, July 1965.

[18] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[19] Peter J. Landin. Histories of discoveries of continuations: Belles-lettres with equivocal tenses. In Danvy [4], pages 1:1–9.

[20] Peter J. Landin. A generalization of jumps and labels. *Higher-Order and Symbolic Computation*, 11(2):125–143, 1998. Reprinted from a technical report, UNIVAC Systems Programming Research (1965), with a foreword [30].

[21] Peter J. Landin. My years with Strachey. *Higher-Order and Symbolic Computation*, 13(1/2):75–76, 2000.

[22] F. Lockwood Morris. The next 700 formal language descriptions. *Lisp and Symbolic Computation*, 6(3/4):249–258, 1993. Reprinted from a manuscript dated 1970.

[23] James H. Morris Jr. A bonus from van Wijngaarden's device. *Communications of the ACM*, 15(8):773, August 1972.

[24] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[25] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363-397, 1998, with a foreword [27].

[26] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, 1993.

[27] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

[28] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474.

[29] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, 1974. Reprinted in Higher-Order and Symbolic Computation 13(1/2):135–152, 2000, with a foreword [32].

[30] Hayo Thielecke. An introduction to Landin's "A generalization of jumps and labels". *Higher-Order and Symbolic Computation*, 11(2):117–124, 1998.

[31] Adriaan van Wijngaarden. Recursive definition of syntax and semantics. In T. B. Steel, Jr., editor, *Formal Language Description Languages for Computer Programming*, pages 13–24. North-Holland, 1966.

[32] Christopher P. Wadsworth. Continuations revisited. *Higher-Order and Symbolic Computation*, 13(1/2):131–133, 2000.