Run Your Research

On the Effectiveness of Lightweight Mechanization

Casey Klein John Clements Christos Dimoulas Carl Eastlund Matthias Felleisen
Matthew Flatt Jay A. McCarthy Jon Rafkind Sam Tobin-Hochstadt Robert Bruce Findler

Abstract

Formal models serve in many roles in the programming language community. In its primary role, a model communicates the idea of a language design; the architecture of a language tool; or the essence of a program analysis. No matter which role it plays, however, a faulty model doesn't serve its purpose.

One way to eliminate flaws from a model is to write it down in a mechanized formal language. It is then possible to state theorems about the model, to prove them, and to check the proofs. Over the past nine years, PLT has developed and explored a lightweight version of this approach, dubbed Redex. In a nutshell, Redex is a domain-specific language for semantic models that is embedded in the Racket programming language. The effort of creating a model in Redex is often no more burdensome than typesetting it with LaTeX; the difference is that Redex comes with tools for the semantics engineering life cycle.

In this paper we report on a validation of this form of lightweight mechanization. The largest part of this validation concerns the formalization and exploration of nine ICFP 2009 papers in Redex, an effort that uncovered mistakes in all nine papers. The results suggest that Redex-based lightweight modeling is effective and easy to integrate into the work flow of a semantics engineer. This experience also suggests lessons for the developers of other mechanization tools.

1. The Role of Language Models

Programming language researchers use formal models to communicate ideas in a concise manner. Many of their models explain a small piece of language design, perhaps a new linguistic construct or a new type system. Other models express the essence of a compiler transformation, the software architecture of an IDE tool, or the workings of a program analysis. For decades researchers have used paper and pencil to develop these models. Paper-and-pencil models come with flaws, however. Since flawed models can lead to miscommunications, researchers state and prove theorems about models, which forces them to "debug" the model.

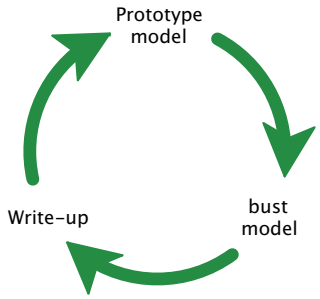
Some flaws nevertheless survive this paper-only validation step, and others are introduced during typesetting. These mistakes become obstacles to communication. For example, Martin Henz from National University of Singapore recently shared with one of this paper's authors his frustration with a historic paper (Plotkin 1975):

> *The readability is not helped by the fact that there are lots of typos, e.g. page 134, Rule II 1: M = N should be M = M. The rule II 3 on page 136 is missing the subscript 1 above the bar. [personal communication, 6/4/2011]*

Once the reader understands a model, fixing such typos is straightforward. But during the initial struggle with the paper, flawed rules may pose seemingly insurmountable obstacles to the reader. In contrast, authors who have spent months or years exploring the intricacies of their model are prone to discount the significance of typos and small mistakes even if readers report extreme frustration.

Over the past decade, mechanized theorem proving has come into its own as one alternative to the paper-and-pencil approach (Aydemir et al. 2005). In this world, researchers "program" their models in formal languages, state theorems, and create machine-checked proofs. We consider this kind of theorem proving heavyweight, because it requires more explicit details than programming.

An alternative is to program models in functional languages such as Haskell: creating interpreters, typecheckers, etc. This approach provides important mechanical scrutiny, but the gap between the program and what appears in a paper's figures tends to make the task laborious and reduces the strength of the validation.



With these considerations in mind, PLT has developed Redex (Matthews et al. 2004; Felleisen et al. 2010), an executable domain-specific language for mechanizing semantic models. The philosophy of Redex is to treat semantic models as software artifacts just like plain software systems. As such, semantic models have a life cycle, and the life cycle idea for semantic models is similar to the one of software systems. Using Redex a semantics engineer formulates the syntax and semantics of the model; creates test suites;

$e ::= (e\ e\ \ldots)$
$\mid x$
$\mid (\lambda\ (x\ \ldots)\ e)$
$\mid$ call/cc
$\mid +$
$\mid$ *number*

```
(define-language Λc
  (e (e e ...)
     x
     (λ (x ...) e)
     call/cc
     +
     number)
  (x variable-not-otherwise-mentioned))
```

$e ::= \ldots. \mid (\mathsf{A}\ e)$
$v ::= (\lambda\ (x\ \ldots)\ e)$
$\mid$ call/cc
$\mid +$
$\mid$ *number*
$::= (v\ \ldots\ \ e\ \ \ldots)$
$\mid$

```
(define-extended-language
  Λc/red Λc
  (e .... (A e))
  (v (λ (x ...) e)
     call/cc
     +
     number)
  (E (v ... E e ...)
     hole))
```

Figure 1: λ-calculus plus `call/cc`

$$E[(\mathsf{A}\ e)] \longrightarrow eb \qquad [\text{abort}$$

$$E[(\ \text{all/}\quad v)] \longrightarrow E[(v\ (\lambda\ (x)\ (\mathsf{A}\ E[x])))] \quad [\ \text{all/}$$
where x eshr

$$E[((\lambda\ (x\ \ldots_1)\ e)\ v\ \ldots_1)] \longrightarrow E[e\{x{:=}v, \ldots\}] \qquad [\beta v$$

$$E[(+\ number\ \ldots)] \longrightarrow E[\Sigma[[number, \ldots]]] \qquad [+$$

```
(define red
  (reduction-relation
   Λc/red #:domain e
   (--> (in-hole E (A e))
        e
        "abort")
   (--> (in-hole E (call/cc v))
        (in-hole E (v (λ (x) (A (in-hole E x)))))
        (fresh x)
        "call/cc")
   (--> (in-hole E ((λ (x ..._1) e) v ..._1))
        (in-hole E (subst e (x v) ...))
        "βv")
   (--> (in-hole E (+ number ...))
        (in-hole E (Σ number ...))
        "+")))
```

Figure 2: The Λc reductions

runs random tests on conjectures; uses graphical tools for visualizing examples and debugging; and automatically renders the model as a PDF snippet.

It is our hypothesis that *small Redex efforts quickly pay off for the working semantics engineer.* To validate our hypothesis, we conducted two case studies, and this paper presents the results of these studies. The first shows how Redex helps test a language implementation with a language model. The second shows that the Redex methodology applies to a broad spectrum of contemporary research papers. Specifically, the authors encoded nine ICFP 2009 papers in Redex; equipped the models with unit tests; translated formal and informal claims into testable conjectures; and checked their validity. In the process, we found mistakes in all of the papers, including one whose essential result had been verified in Coq.

The next section reviews the Redex modeling language and tool suite. From there, the paper covers ten case studies. Our experience suggests lessons for the authors of semantic models as well as the designers of validation tools; we discuss these lessons in the paper's final sections, along with related work.

2. Welcome to Redex

Semantics engineers use the Redex language to write down the grammar, reductions, and metafunctions for calculi or transition systems. The language is a domain-specific language embedded in Racket. Redex programmers inherit the DrRacket IDE, a large standard library, and a large set of user-contributed libraries. The Redex toolkit covers a variety of tasks related to executing semantics definitions: a stepper for small-step operational semantics; inspectors for reduction graphs; a unit testing framework; a tool for randomized testing à la QuickCheck (Claessen and Hughes 2000); and automatic typesetting support.

From a linguistic perspective, Redex is a strict functional language with a powerful pattern matcher and domain-specific constructs supporting operational semantics. This section illustrates Redex with a model of the λ-calculus, extended with `call/cc`.

2.1 Grammars

The left-hand side of figure 1 shows the grammar of the language and the corresponding Redex code. The latter binds the Racket-level variable `Λc` to a Redex language, a series of non-terminals and alternatives. In this case, there are two non-terminals, *e* and *x*. The *e* non-terminal has six alternatives. The first, application expressions, uses an ellipsis to indicate repetition. In this case, the ellipsis amounts to insisting that each application expression consist of at least one sub-expression. Similarly, the third alternative uses an ellipsis to indicate that λ expressions can bind an arbitrary number of variables. The fourth and fifth alternatives are constants, call/cc and +, leaving *x* and *number*, two other non-terminals. The *number* non-terminal is built-in and matches arbitrary Racket numbers. The production for *x* uses the special keyword `variable-not-otherwise-mentioned`. It matches any symbol except call/cc, +, and λ because they are used as terminal symbols elsewhere in the grammar.

To give a reduction semantics to Λc, we add an alternative to *e* and define two extra non-terminals. The right-hand side of figure 1 shows both the mathematical extension and the Redex code.

The first position in a `define-extended-language` form names the new language and the second names to the to-be-extended language. Non-terminals appearing in the body of `define-extended-language` replace those of the same name in the old language, unless a `....` appears, in which case the non-terminal is extended. In this case, we extend *e* with the expression form $(\mathsf{A}\ e)$, which we use to give a reduction semantics for continuations.

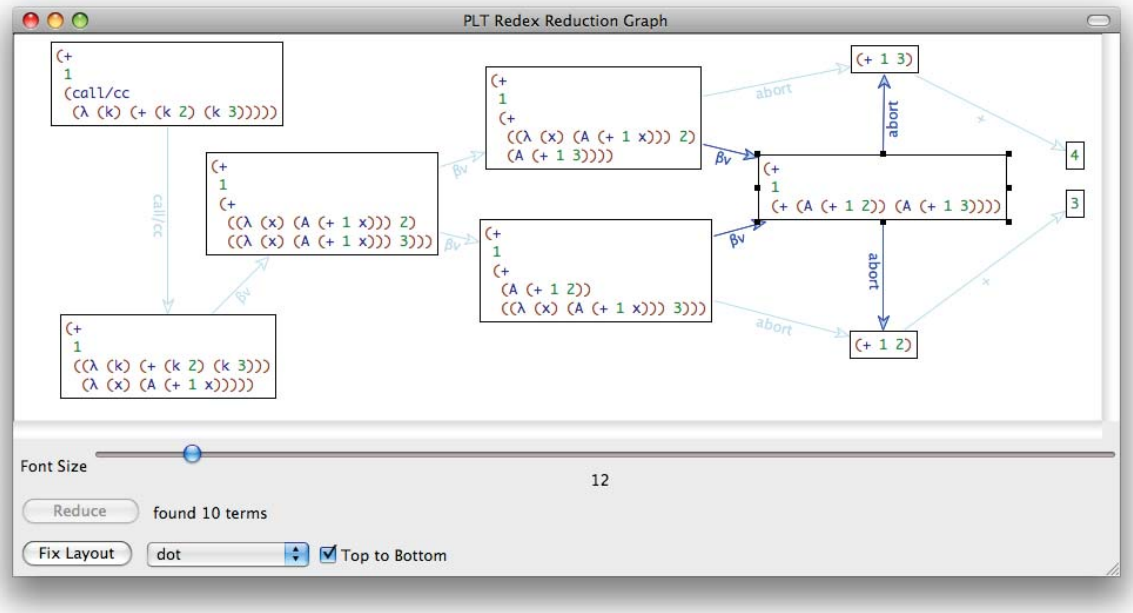


Figure 3: A screenshot of Redex's reduction visualizer

The other non-terminals, `v` and `E`, for values and evaluation contexts, respectively, are used to formulate standard reduction rules. The definition of `E` uses `hole`, a pattern matching construct that represents the hole in a context. Our running example uses two alternatives for evaluation contexts: the first mandates a left-to-right order of evaluation by insisting that evaluation can only take place to the right of values; the second says that a context can be a hole.

2.2 Reduction Relations and Metafunctions

Figure 2 contains the reductions for Λc on the left and the corresponding Redex source code on the right. A reduction relation is defined as a series of rules of the form

```
(--> pat_1 pat_2)
```

where any expression matching `pat_1` is transformed into `pat_2`. The `#:domain` keyword specifies a contract, in this case declaring that `red` relates terms matching the pattern `e`.

With `(in-hole E e)` a Redex programmer specifies a context decomposition, $E[e]$, meaning the first rule aborts the computation by dropping the context around A expressions.

The second rule of `red` rewrites (call/cc v) into an application of v to a function that behaves like a continuation. The `(fresh x)` annotation in the rule demands that the parameter of the new function does not appear anywhere else in the rewritten expression.

The left-hand side of the third rule uses ellipses with subscripts (`..._1`) to specify that the lengths of the two sequences must match, thus restricting the rule to applications without arity errors. The rule's right-hand side appeals to the metafunction `subst`, which Redex requires to be defined explicitly.

The `define-metafunction` keyword defines a metafunction; the first two positions specify a language and a contract, followed by the cases of the function, each enclosed in a pair of square brackets. The `subst` function recurs on a list of bindings, repeatedly applying a single-variable substitution function:

```
(define-metafunction Λc/red
  subst : e (x v) ... -> e
  [(subst e (x_1 v_1) (x_2 v_2) ...)
   (subst-1 x_1 v_1 (subst e (x_2 v_2) ...))]
  [(subst e) e])
```

The single-variable substitution function is defined as usual.[1]

The final rule appeals to a Σ metafunction. This metafunction exploits Redex's embedding in Racket:

```
(define-metafunction Λc/red
  Σ : number ... -> number
  [(Σ number ...)
   ,(foldr + 0 (term (number ...)))])
```

The right-hand side begins with an `unquote` (written as a comma), meaning that it is evaluated in Racket, and the Racket expression is expected to return a term that is the result of the metafunction. In this case, the function exploits the representation of Λc's numbers as Racket numbers to compute their sum. The expression `(term (number ...))` produces a list of the numbers supplied as arguments to Σ. In general, `term` behaves like `quote`, but also picks up the bindings of pattern variables (`number` in this case) and supports ellipses to indicate repetition.

Finally, Redex provides `apply-reduction-relation` to experiment with reduction relations. It accepts a relation and a term and returns a list of all contractions of the term:

```
> (apply-reduction-relation
   red (term (+ 1 (A (+ 2 3)))))
'((+ 2 3))
```

2.3 Exploring Examples

Redex provides visualization tools for exploring the behavior of examples. The `traces` function accepts a reduction relation and a term and shows the entire reduction graph of the term. To demonstrate the value of these tools, we adjust our reduction system to model an unspecified order of evaluation in the spirit of C:[2]

[1] In this case, it is an exact copy of the example model's substitution function from the Redex website: `http://redex.racket-lang.org/`.

[2] Scheme's unspecified order of evaluation is more sophisticated than C's, but Redex is up to the task (Matthews and Findler 2005; Sperber et al. 2007).

```
(define-extended-language
  any-which-way-Λc Λc/red
  (E (e ... E e ...)
     hole))
```

This extension replaces the E non-terminal entirely, allowing reductions to occur in any position inside an application expression.

Next, we use `extend-reduction-relation` to replace the language of the reduction relation (without adding any reductions):

```
(define any-which-way-red
  (extend-reduction-relation
   red any-which-way-Λc))
```

That is, this extension merely re-interprets the existing rules with the new definition of E.

This extended language does not satisfy the Church-Rosser property, as a quick experiment with `traces` shows:

```
> (traces any-which-way-red
          (term (+ 1 (call/cc
                      (λ (k)
                        (+ (k 2) (k 3)))))))
```

Figure 3 displays a screenshot of the resulting window. Each box contains a term that the original reduces to and the arrows are labeled with the reduction rule's name that connects the two terms. The arrows connected to the term underneath the mouse cursor are darkened to make them easier to pick out.

2.4 Randomized Testing

Redex's randomized testing support follows QuickCheck. A programmer writes down a property with `redex-check` (Klein and Findler 2009) and Redex generates instances of the property in an attempt to falsify it. Specifically,

```
(redex-check G n e)
```

tests the boolean-valued expression `e`, interpreted as a predicate universally quantified over `n`, by evaluating it at random terms generated from the non-terminal `n` of the grammar `G`.

For example, we can test the property that every expression in Λc is a value or reduces to another expression. To check whether an expression is a value, we use `redex-match`, which tests whether a particular term matches a given pattern; to check whether an expression reduces, we check whether `apply-reduction-relation`'s result is non-empty:

```
> (redex-check
   Λc/red e
   (or (redex-match Λc/red v (term e))
       (cons?
        (apply-reduction-relation
         red (term e)))))
counterexample found after 9 attempts:
S
```

Of course, there are a number of stuck states and Redex quickly finds a simple one, namely a free variable. If we add an explicit reduction to `error` as a way to signal an error for a free variable:

```
(--> (in-hole E x) error "free variable")
```

and then iteratively run the test above, fixing errors as they are discovered, `redex-check` eventually finds all of the (known) stuck states in the model.

2.5 Typesetting

Redex provides automatic typesetting support which transforms a language, reduction relation, metafunction, or a term into PostScript or PDF to be included in a paper. Indeed, all of the typeset versions of elements of the Λc model shown in this paper are generated automatically using Redex.

This example shows Redex's vanilla support for rendering a reduction relation:

```
> (render-reduction-relation red)
```

$E[(\mathsf{A}\ e)] \longrightarrow$ [abort
e

$E[(\ \mathsf{all}/\quad v)] \longrightarrow$ [all/
$E[(v\ (\lambda\ (x)\ (\mathsf{A}\ E[x])))]$
where x eshf

$E[((\lambda\ (x\ ..._1)\ e)\ v\ ..._1)] \longrightarrow$ [βv]
$E[\mathsf{subst}[[e, (x\ v), ...]]]$

$E[(+\ number\ ...)] \longrightarrow$ [+
$E[\Sigma[[number, ...]]]$

The main difference between this rendering of the reduction relation and the one shown on the left in figure 2 is that the rules are oriented vertically instead of horizontally. Adjusting the orientation is a matter of passing a flag to control the basic layout option. In addition, the substitution function is shown here using Redex's default typesetting for metafunctions, $\mathsf{subst}[[e, (x\ v), ...]]$. Redex also provides hooks for tuning the rendering of calls to metafunction, which may be used to render substitution in the conventional style, $e\{x{:=}v, ...\}$.

When a reduction relation or a metafunction escapes to Racket, Redex renders the Racket code in a monospaced font but with a pink background so it stands out:

```
> (render-metafunction Σ)
```

$\Sigma[[number, ...]] =$ `(foldr + 0` $(number\ ...)$

Redex programmers can then set hooks to adjust how such fragments are typeset.

3. Redex Models for Production Systems

Redex can help language designers validate their implementations against their specifications with low cost. To demonstrate this thesis, we conducted a case study using the model of delimited control by Flatt et al. (2007). Figure 4 shows the model's complete internal syntax, including forms left out of the original paper's presentation. At the time of the publication of that paper, the model's authors had implemented a Redex model,[3] built a thorough test suite, and mechanically generated their paper's figures from the Redex definitions. They did not, however, employ randomized testing; Redex had no built-in support for it at the time. This section explains how we revisited that model to see if randomized testing could find more issues in a well-tested model. It did: we found mistakes in both the implementation and specification of delimited control.

3.1 Randomized Testing in Redex

The obvious use of randomized testing is to check a paper's claims. Flatt et al. do not explicitly state any theorems, but all is not lost—they do imply that the model is a faithful abstraction of the production Racket implementation. We can therefore test the claim that the implementation produces the result predicted by the model:

```
(redex-check
 delim-cont-grammar e
 (equal? (model-eval (term e))
         (racket-eval (term e))))
```

In this claim, `model-eval` uses Redex to reduce its argument to a value and `racket-eval` evaluates the term via Racket.

[3] Available online: `http://www.cs.utah.edu/plt/delim-cont/`

```
 e ::= m | (wcm w m)
m ::= x | v | (e e ...) | (begin e e) | (    e e e) | (dw x e e e)
      | (if e e e) | (set! x e)
 v ::= (list v ...) | (λ (x ...) e) | (cont v E) | (comp E)
      | dynamic-wind | abort | current-marks%
      | cons | u | n | #f | #
      | zero? | print | + | first | rest%
 n ::= number
 u ::= call/cc | call/comp | call/cm
 w ::= ((v v) ...)
 E ::= W | W (dw x e E e)]
 W ::= M | (wcm w M)
 M ::= [] | (v ... W e ...) | (begin W e) | (    v W v)
      | (set! x W) | (if W e e)
 D ::= [] | E (dw x e [] e)]
```

Figure 4: The syntax of the delimited control model.

Unlike in QuickCheck, where users specify test generators for the data types they define, Redex derives naive test generators automatically from the language's grammar. In this case, the derived generator has two immediate problems. First, Redex's grammar specifications do not address variable binding. As a result, the generator often produces expressions with free variables, which Racket statically rejects. Second, the e non-terminal in figure 4 includes non-surface syntax such as (comp E) that Racket programmers cannot write directly. Since our goal is not to prove a proposition but to falsify it, we begin with simple solutions to these problems.

3.2 A Weak Attempt

One possibility is to discard test expressions that contain free variables and to avoid non-surface forms entirely:

```
(redex-check
 delim-cont-grammar e
 (with-handlers ([free-var-exn? (λ (_) #t)])
   (equal? (model-eval (term e))
           (racket-eval (term e))))
 #:prepare drop-non-surface)

 ; drop-non-surface : expr -> expr
 (define (drop-non-surface e) --)
```

This revision discards open expressions by catching unbound identifier errors with an exception handler that reports the test as a success. It eliminates uses of non-surface forms in the generated expression by rewriting them using `drop-non-surface`, which replaces non-surface expressions with one of their sub-expressions or a random constant if there are no sub-expressions.

This approach is naive, but it reveals three previously unknown errors, one in the portion of the semantics shown in the published paper and two in elided definitions:

1. The error visible in the paper's figures[4] is in the definition of evaluation contexts M, reproduced in figure 4. A prompt expression (% e e e) has three sub-expression: a tag used by the other control operators to identify the prompt, a body, and a handler expression that receives values thrown by abort expressions within the dynamic extent of the body. The expressions should be evaluated from left to right, but the definition of M lacks evaluation contexts corresponding to the first and third sub-expressions. For example, this omission causes evaluation of the following expression to get stuck:

   ```
   > (model-eval (% (+ 1 2) 3 (λ (x) x)))
   'stuck
   ```

2. The model defines function application with a rule like this one:

   ```
   (--> ((λ (x ...) e) v ...)
        (subst* (x ...) (v ...) e)
        "beta")
   ```

   Unlike the corresponding rule in section 2.2, the ellipses have no subscripts. Thus, the rule also applies to expressions with arity mismatches, e.g., ((λ () 1) 2). Reducing this expression with Redex raises a meta-level error because the formal parameters and actual arguments cannot be paired.

3. The reduction rule for zero? expressions may also raise a meta-level exception. The model's Redex encoding, which represents numbers as Racket numbers and primitive operators like + as Racket symbols, appeals to Racket's `zero?` function to reduce (zero? v) redexes. But the `zero?` function carries a contract that restricts its application to numbers, making reduction of the expression (zero? +), for example, raise a meta-error instead of producing #f.

3.3 Refining the Test Generator

Despite our initial success, there is good reason to explore more sophisticated test generation strategies. To start, in one sample of 10,000 expressions produced by Redex's naive generator, only 1,220 contain no free variables, and only 599 of those are not values. We can avoid discarding so many tests by supplying a function `close` that replaces unbound variables with random bound ones or constants when none are bound:

```
(redex-check
 delim-cont-grammar e
 (equal? (model-eval (term e))
         (racket-eval (term e)))
 #:prepare (compose close drop-non-surface))

; close : expr -> closed-expr
(define (close e) --)
```

For this particular model, though, we do not discover any new errors this way.

Redex's test coverage tool suggests another improvement, however. Executing one round of 10,000 random tests fails to exercise 20 of the 30 reduction rules even once. Three more rules, including the β_v rule shown in section 3.2, fire only a few times each.

To exercise these rules, the test generator must make several fortuitous choices. In the case of the β_v rule, the generator must first choose to place an application expression in a position that will ultimately be evaluated. Second, in the application's operator position, the generator must construct an expression that evaluates to a function. Third, in the operand positions, the generator must construct expressions that do not result in runtime errors or discard the continuation containing the application. Fourth, the generator must choose to construct the right number of operands.

We can encourage these choices by providing `redex-check` the hint that it should occasionally use the rules' left-hand sides instead of the more general pattern `e` as its basis for test generation. The left-hand side of the β_v rule, for example, directly addresses the second and fourth choices above.

Many of the patterns in the rules' left-hand sides, however, refer to non-surface forms, and so we must first replace the pass that removes non-surface expressions with one that transforms

[4] See p. 174 of the 2007 ICFP proceedings.

them into equivalent surface expressions. For example, expressions equivalent to comp values can be constructed from % (prompt), call/comp, and abort. We implement this transformation, as well as ones for the other non-surface forms for which it is possible (see section 3.4), using a function `transform-non-surface` and supply it to `redex-check`, along with the hint to use the `delim-cont-rules` reduction relation.

```
(redex-check
 delim-cont-grammar e
 (equal? (model-eval (term e))
         (racket-eval (term e)))
 #:prepare (compose close transform-non-surface)
 #:source delim-cont-rules)

; transform-non-surface : expr -> expr
(define (transform-non-surface e) --)
```

This technique finds six more previously unknown errors.

Two of these six are mistakes in the model made available with the paper, though they did not appear in the publication:

1. The model includes a semantics for continuation marks, a feature for associating name-value pairs with continuation frames (Clements et al. 2001). The expression (call/cm e_k e_v e_t) marks the active continuation frame with key e_k and value e_v then applies the thunk e_t. The expression (current-marks e_k e_p) collects all marks for key e_k on frames up to the nearest enclosing prompt tagged with e_p.

   The model's reduction rule for current-mar s appeals to a metafunction that traverses the delimited context to construct a list of its mark values. This metafunction, however, lacks a case for contexts of the form (if E e e), making mark collection undefined within the dynamic extent of the test position of if expressions. For example, evaluation of the following expression raises a meta-level error:

   ```
   (% 0
      (call/cm 1 #t
        (λ ()
          (if (first (current-marks 1 0))
              2
              3)))
      (λ (x) x))
   ```

2. The model's definition of capture-avoiding substitution is wrong. To perform the substitution subst[[x_1, e_1, (λ (x_2) e_2)]], the model takes care to rename x_2 to a variable not free in e_1, but it fails to avoid choosing x_1 or the free variables of e_2.

The remaining four errors are in the implementation of Racket (version 5.0.2). These errors eluded a hand-crafted test suite and years of production use, but randomized testing finds them quickly:

3. Continuation marks are not represented directly on continuation frames. Instead, a stack of marks is kept in parallel with the stack that represents the continuation. Delimited continuations therefore capture parts of the mark stack, and different slices of the stack involve different base offsets. While restoring part of a continuation to execute a dynamic-wind pre-/post-thunk, one of the offsets is installed incorrectly. The resulting crash would only happen for a pre-/post-thunk that is captured in a continuation that is itself captured as an extension of a composed continuation, possibly with a few more ingredients we have yet to identify.
4. This error is similar to the previous one. Like continuation marks, dynamic-wind frames are kept in a separate stack that is synchronized with the continuation stack. An offset connecting the two stacks is forced to an incorrect value when composing continuations in certain cases. The mistake produces a crash only after one more round of continuation capture and invocation.
5. Non-composable continuations store a prompt tag and, when they are invoked, the implementation checks that the current continuation includes a prompt with the same tag. Composable continuations come without a tag. The two kinds of continuations share much of the implementation infrastructure, however, and this shared implementation incorrectly stores and checks prompt tags for composable continuations.
6. This error is similar to the previous one. The implementation also performs a prompt-tag check after each the application of each dynamic-wind pre-thunk during the process of applying a non-composable continuation. For composable continuations, the implementations should not perform such prompt-tag checks, but once again, the shared implementation performs these checks for both kinds of continuations.

At the time of writing, we still do not fully understand the behavior of the test that discovered the first of these four errors, making the prospect of manually devising a test like it appear dismal. Fortunately, the repair was clear from the resulting core dump.

The implementation's hand-crafted test suite contains tests that get close to finding these errors, but the suite's author did not have the patience to construct tests of the necessary complexity. Patience aside, finding these errors seems to require a degree of uninhibited creativity that is difficult to achieve. Hanford (1970), one of the first to apply randomized testing to the implementation of programming languages, observes about his test generator for PL/1—dubbed "syntax machine"—that

> *[a]lthough as a writer of test cases, the syntax machine is certainly unintelligent, it is also uninhibited. It can test a [language] processor with many combinations that would not be thought of by a human test case writer.*

3.4 Unwelcome Errors

In addition to these nine errors, we found many more which we would have preferred to avoid—errors in the specification relating the model to its implementation and errors in the post-generation passes. We mention their discovery not as successes of randomized testing but as a reminder of its cost.

Formalizing the relationship between the model and its implementation with enough precision to test it is a non-trivial task. The primary challenge is to decide which non-surface expressions from figure 4 are well-formed. For example, the grammar includes continuation frames that contain two marks at a single key, but such configurations should not occur. Developing a specification that includes these invariants takes some effort. We used randomized testing to find expressions where violations of unknown invariants yield different behaviors in the model and implementation. There is no guarantee that this sort of randomized test-driven development results in a complete specification, but we are satisfied as long as the working draft avoids false positives in our tests.

The source of most unwelcome errors was our implementation of the passes that enforce well-formedness of non-surface expressions, transform well-formed ones into surface expressions, and remove free variables. Together, these passes comprise 259 non-comment, non-whitespace lines of code.

4. An Empirical Study of ICFP Papers

To improve our understanding of how lightweight metatheory mechanization can help authors with their papers, we used Redex to explore nine papers from the ICFP 2009 proceedings. The papers were chosen because we considered them suitable for mechanization in Redex, but some turned out to be challenges.

The nine papers include two which had already been mechanized. We chose two such papers not really expecting to find errors, but to see if we would learn something about Redex when implementing papers that already had a significant mechanized metatheory effort put into them.

We found mistakes in all nine papers, with less effort in each case than exhibited in section 3. We explain most of the errors we found below, in the order in which the papers appear in the 2009 proceedings. We omit some uninteresting errors common to multiple systems (e.g., confusing the particular object-language variable `x` with the meta-variable V that ranges over object-language variables). The authors of the papers we studied have confirmed the errors described here. The Redex models are available online: `www.eecs.northwestern.edu/~robby/lightweight-metatheory/`

4.1 *Safe functional reactive programming through dependent types* by Neil Sculthorpe and Henrik Nilsson

Sculthorpe and Nilsson (2009) define a functional reactive programming language embedded in Agda (Norell 2007). The embedded language's dependent type system rules out domain-specific errors such as loops with immediate feedback and uses of uninitialized signals. Its operational semantics, given as an Agda function defining discrete evaluation steps, carries a machine-checked proof of type safety since Agda accepts the function as total.

The paper does not show the Agda definition; it instead presents the semantics in the usual inference rules notation for big-step semantics. Encoding the paper's formulation in Redex revealed one error, introduced in the manual translation of the Agda code to nearly three full pages of figures. Specifically, the conclusion of the ϕ_1-DSW-EV rule in the paper's figure 6 applies to *switch* expressions; it should apply to *dswitch* expressions.

This paper has since been revised (Sculthorpe 2011).

4.2 *Causal commutative arrows and their optimization* by Hai Liu, Eric Cheng, and Paul Hudak

Liu et al. (2009) define a class of recursive arrows that they call *causal commutative arrows* (CCA) and show how they can be compiled into a single imperative loop. Our focus was the portion of the transformation that they describe formally, a procedure for computing an efficient normal form they call *causal commutative normal form* (CCNF).

The procedure takes the form of a normalization relation $\Downarrow$ that reduces expressions bottom-up using a relation $\mapsto$ based on the standard arrow axioms (Hughes 2000). For example, the normalization rule for sequential compositions $e_1 \ggg e_2$ normalizes the sub-expressions, reduces the result, then normalizes the contractum.

$$\frac{e_1 \Downarrow e_1' \qquad e_2 \Downarrow e_2' \qquad (e_1' \ggg e_2') \mapsto e \qquad e \Downarrow e'}{e_1 \ggg e_2 \Downarrow e'}$$

Using randomized testing to check whether $\Downarrow$ is indeed a function with the claimed domain and codomain found two problems:

1. In addition to arrow constructors, the language on which $\Downarrow$ is defined includes functions and pairs; consequently, some arrow-typed expression do not have arrow constructors at their roots. The proof in the paper's appendix mentions that such expressions must first be β-reduced, but there are no corresponding steps in the $\Downarrow$ definition.
2. Reduction via the $\mapsto$ relation creates arrows built from the *loopB* combinator, defined in terms of the primitive CCA constructors. To account for *loopB* expressions, the $\Downarrow$ relation includes the following rule:

$$\frac{f \Downarrow f' \quad \mathit{loopB}\ i\ f' \mapsto e \qquad e \Downarrow e'}{\mathit{loopB}\ i\ f \Downarrow e'}$$

   For some f, $\mathit{loopB}\ i\ f'$ is already in normal form. In these cases there is no e such that $\mathit{loopB}\ i\ f' \mapsto e$, leaving the rule's second premise unsatisfiable (and the implied procedure stuck).

This paper has since been revised (Liu et al. 2011).

4.3 *Partial memoization of concurrency and communication* by Lukasz Ziarek, KC Sivaramakrishnan, and Suresh Jagannathan

Ziarek et al. (2009) show how memoization can be applied in a concurrent language with synchronous message-passing primitives. To show that memoization preserves meaning, they define two evaluators for a concurrent language, one that uses memoization and one that does not. Encoding these systems in Redex exposed two mistakes:

1. The paper's theoretical result is a safety theorem guaranteeing that when memoized evaluation takes a state P to a state P', then non-memoized evaluation takes $\mathcal{T}[\![P]\!]$ to $\mathcal{T}[\![P']\!]$, where the meta-function $\mathcal{T}$ erases the extra structure used for memoization. As randomized testing quickly discovers, this theorem is false. It fails to exclude states in which the memo table incorrectly predicts the behavior of some function. The correspondence appears to hold for executions beginning with the empty table (the important case), but the proof's inductive structure requires a generalized claim about states with non-empty but well-formed tables. A proof typically gives this generalization explicitly, since well-formedness conditions for such accumulated data structures tend to be complex.
2. The non-memoizing evaluator operates on program states P taken from the following grammar, in which t ranges over thread identifiers and e ranges over expressions:

$$P ::= P \parallel P \mid t[e]$$

   Because a state P contain sat least one thread, the following communication rule cannot apply in the absence of a third thread:

$$\frac{P = P' \parallel t[E[\mathit{send}(l,v)]] \parallel t'[E'[\mathit{recv}(l)]]}{P \longmapsto P' \parallel t[E[\mathit{unit}]] \parallel t'[E'[v]]}$$

   The same problem exists with the memoizing evaluator.

4.4 *A concurrent ML library in Concurrent Haskell* by Avik Chaudhuri

Chaudhuri (2009) describes a way to implement the Concurrent ML primitives (Reppy 1999) in a language that supports only first-order message passing, such as Concurrent Haskell (Jones et al. 1996). He builds an abstract machine that abstracts the message-passing model common to Concurrent Haskell and other concurrent systems and then shows how programs using the Concurrent ML primitives may be compiled into terms in his abstract machine, while preserving safety, progress, and fairness.

We encoded this abstract machine, source language, and compiler in Redex. In addition to writing test cases by hand, we used randomized testing to check a weak variant of the paper's correctness theorem. Randomized testing did not produce any counterex-

amples to the theorem, but it did lead us to programs for which the abstract machine consumes unbounded resources where proper Concurrent ML implementations would not.

For example, consider the following source expression, in which `c` is a fresh channel:

```
select(in c, out c)
```

This expression permanently blocks any thread that evaluates it because `select` cannot perform either communication. In Concurrent ML, garbage collection reclaims this thread because no other thread can reach the channel; the abstract machine, on the other hand, performs infinitely many steps for this expression—effectively busy waiting for an event that cannot occur. This error also shows up in the released implementation of the Concurrent ML library for Concurrent Haskell based on the abstract machine.

4.5 *Automatically RESTful web applications: marking modular serializable continuations* by Jay McCarthy

McCarthy (2009) extends a technique for implementing first-class continuations via continuation marks (Pettyjohn et al. 2005), adding support for source programs that themselves use continuation marks. Despite a pencil-and-paper proof of correctness, a combination of manual and randomized testing found five errors in the translation's specification, as well as three errors in the semantics of its source and target languages:

1. The translation consists of four mutually recursive functions: one for translating values and expressions that would be values if not for a variable in some component, one for redexes, one for evaluation contexts, and a driver function that either defers to the values translation or decomposes the input and applies the translations for redexes and evaluation contexts. This schema relies on a unique decomposition lemma that turns out not to hold, due to four mistakes in the grammars for redexes and evaluation contexts.
2. The source and target languages are variants of A-normal form (Flanagan et al. 1993), but the translation of evaluation contexts inserts applications in a position that does not allow them. Adapting translation to preserve A-normal form seems to require abandoning the invariant that evaluation contexts translate to evaluation contexts rather than more general contexts, which the translation for continuation values assumes. In practice, there is no need to translate such values anyway, since they do not appear in the source text of realistic programs.
3. In translated programs, `call/cc` produces a procedure that discards the current continuation using `abort` then calls a function `resume` for rebuilding the captured continuation from a data representation of its frames. A mistake in the definition of `resume`, however, causes it to leave some frames out of the rebuilt continuation.
4. The translation's handling of continuation marks in the original program involves installing an additional mark on each frame. This mark holds a data structure that records all of the other marks on the associated frame. To maintain this cumulative mark, the translated program first fetches its current value using `c-w-i-c-m` ("call with immediate continuation mark"), which has the following signature:

   ```
   c-w-i-c-m: key (α -> β) α -> β
   ```

   This function examines the active frame's marks and calls the provided function with the value associated with the given key or the provided default value if there is no such mark. The translation's `c-w-i-c-m` call forgets the mandatory third argument.
5. The translation lacks recursive calls for two of the three positions inside the `(w-c-m e e e)` form, used for installing continuation marks.
6. Instead of including an explicit form for dereferencing store pointers, the source language semantics has two rules for each form that demands its operand. For example, in addition to the usual β_v rule, there is a rule that applies when the function position holds a pointer σ:

   $$\begin{array}{rll} \Sigma/E[(\sigma\ \overline{v})] & \longrightarrow_{SL} & \Sigma/E[e\overline{[x \to v]}] \\ & \text{where} & \Sigma(\sigma) \neq \lambda\ (\overline{x})\ e) \end{array}$$

   But with the usual definition of store-lookup (the author's intention), this strategy does not handle pointers to pointers to functions.
7. The source language semantics lacks a rule like the following, for indirect continuation application.

   $$\begin{array}{rll} \Sigma/E[(\sigma\ v)] & \longrightarrow_{SL} & \Sigma/E'[v] \\ & \text{where} & \Sigma(\sigma) = \kappa.E' \end{array}$$

8. The source and target languages provide a form `(c-c-m e ...)` for collecting continuation marks. This form is similar to the `current-marks` operator explained in section 3.3, but there are two differences. First, `c-c-m` has no prompt-tag operand, since the source and target languages do not provide delimited control. Second, `c-c-m` collects the marks for several keys at once. Its result should be a list of lists, in which the inner lists contain the marks on each continuation frame; as defined in the semantics, however, the marks for the final frame become the list's tail instead of its last element.

4.6 *Control-flow analysis of function calls and returns by abstract interpretation* by Jan Midtgaard and Thomas P. Jensen

Midtgaard and Jensen (2009) systematically derive a tail-call sensitive control-flow analysis using abstract interpretation and then prove that their analysis is equivalent to a CPS-based one from earlier work. We discovered two problems with the paper:

1. The CPS transformation's domain contains expressions with constants, but there is no case in the transformation functions to deal with the constants. This leads to a problem in lemma 5.1, which states that transforming a program to CPS and then transforming it back results in the original program. As stated, this lemma is only true for programs that contain no constants.
2. The paper defines $\equiv$ to be the least equivalence relation on expressions satisfying these two equations:

   `let x = t in s` $\equiv$ `s` `let x = t0 t1 in s` $\equiv$ `s`

   and the analysis result includes a mapping from representative elements of this equivalence class to the values that the corresponding expressions have at runtime. This definition of $\equiv$ breaks the equivalence of the direct-style and CPS analysis (theorem 5.1). Specifically, the direct-style analysis imprecisely predicts that `id2` might be returned by the term

   ```
   let W = fn N. (N N) in
     let id1 = fn x1. x1 in
       let id2 = fn x2. x2 in
         let J = (fn t. id2) id1
           let d = (W W) in id2
   ```

   but the CPS analysis correctly predicts that it never returns. The problem is, the equivalence relation equates the two occurrences of `id2` in the above program but should not.

   This paper has since been revised (Midtgaard and Jensen 2012).

4.7 *Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform* by Tiark Rompf, Ingo Maier, and Martin Odersky

Rompf et al. (2009) describe an implementation of delimited continuations for Scala. They define a type system that distinguishes expressions with control effects, allowing continuations to be implemented by a selective CPS transformation that leaves expressions in direct style when they do not reify their continuations.

As we discovered while encoding the system in Redex, the paper merely sketches the typing and transformation rules. A modest Redex model can close the gap between a sketch and a consistent description, and our model uncovered a significant omission in the paper's explanation. The definition of the transformation function $[\![.]\!]$ neglects necessary recursive calls on sub-expressions (e.g., on the operand of `shift`).

We did discover one inaccuracy not arising from the rules' informal nature. The transformation, which operates on expressions in A-normal form, distinguishes two classes of non-tail calls that reify their continuations—those where the expression e following the call also reifies its continuation (a behavior indicated in e's type) and those where e does not. In the latter case, the transformation has an optimization opportunity. In an attempt to exploit the opportunity, the definition of $[\![.]\!]$ mistakenly dispatches on the type of $[\![e]\!]$ instead of the type of e, causing it to apply the optimization even when it is unsound.[5]

4.8 *A Theory of typed coercions and its applications* by Nikhil Swamy, Michael Hicks, and Gavin M. Bierman

Swamy et al. (2009) define a proof system for validating particular program rewritings and give conditions under which various program-rewriting systems operate unambiguously. For all results except the ones on rewriting using polymorphic coercions, they provide Coq proof scripts.[6]

We discovered two problems with an example in the section explaining polymorphic coercions. First, one instantiation of a polymorphic coercion is missing. Second, the example is based on the assumption that the rewriting process will leave one particular expression alone when, in fact, it might be rewritten.

4.9 *Complete and decidable type inference for GADTs* by Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis

Schrijvers et al. (2009) define a type system for generalized algebraic datatypes (GADTs), giving both a declarative specification and a sound and complete inference algorithm. Encoding the algorithm in Redex uncovered three flaws in the paper's definition:

1. The type system that Schrijvers et al. consider most natural for GADTs is undecidable. Their key insight is that decidability can be recovered by designating sets of unification variables called *untouchables* that may not be unified to solve certain constraints. The rules for `let` expressions, typed-annotated `let` expressions, and individual `case` clauses introduce these variables, which stand for unknown types. The third of these three rules, however, designates the wrong variables as untouchable.
2. The rule for entire `case` expressions correctly insists that all of its clauses produce a result of the same type β. But instead of assigning the entire `case` the type β, the rule gives it the type α, a meta-variable that does not appear anywhere else in the rule though the notation $\overline{\alpha}$ does appear.
3. The constraint solving algorithm lacks a rule for arrow types.

4.10 Our Effort

§	Tests	LOC	Props
4.1	24	1196	1
4.2	10	849	5
4.3	32	1197	5
4.4	55	1445	4
4.5	105	1548	3
4.6	148	1159	8
4.7	97	1223	5
4.8	57	1335	3
4.9	67	1143	1
Mean	66	1233	3.88

Our case studies required two kinds of efforts. First, each investigator had to understand his assigned paper to a sufficient degree so that he could formulate a paper-and-pencil model. Second, the investigator had to implement the model in Redex. The adjacent table quantifies the second kind of effort. Each row shows the number of lines of code, the number of test cases, and the number of properties tested for each of the models in the above subsections. On the average, a model consists of 1,200 lines of code, including 66 tests and three or four claims.

5. Lessons Learned

Our experience suggests lessons for the authors of programming languages papers, for us as the developers of Redex, and for the developers of other validation tools.

5.1 Lessons for Authors

Redex supports mechanization in a form that accommodates time-pressed semantics engineers and still uncovers common errors. Although we do not have precise effort logs for the case study of section 4, we estimate that encoding and testing each model required less time than understanding the content of the paper. As our case studies show, lightweight mechanization reduces the number of mistakes in a model and thus increases its value as a communication vehicle.

Flaws aside, we would not have managed to understand these papers without their models. Prose is too imprecise and frequently too brief to build more than a superficial understanding. For example, one of the authors of the present paper would have rated himself an expert reviewer for the paper in section 4.5, having seen the semantics for continuation marks many times and having worked with continuation-based web servers. Despite this preparation, he failed to understand the paper's intuitive explanation of the system until he studied its formal model. In such cases, where the reader primarily relies on the model for explanation, typos—even ones obvious to experts in hindsight—can become significant barriers to communication.

Lightweight mechanization enables interactive exploration, expanding the means with which authors and readers communicate. In the case of every paper, we found that executing examples improved our understanding—even after we had already understood enough of the system to encode at least part of it in Redex. When we were unsure if we understood a definition or if its implications appeared problematic, we ran examples. Often the ones we choose would have been too tedious or too error-prone to work out by hand. Sometimes the experiment confirmed our hypothesis; other times it revealed a mistake in our reasoning. Either way, the exercise improved our understanding of the system.

[5] In an email exchange (Feb. 2, 2011 – Mar. 3, 2011), the paper's lead author stated that they did not intend their model as a precise description. He also explained that they meant for the $[\![.]\!]$ function to be applied by a driver function whose operation accounts for the absent recursive calls. The paper does not mention this driver. This author also reported that the Scala implementation does not make the same optimization mistake as the transformation sketch.

[6] Available online: `http://research.microsoft.com/~nswamy/papers/coercion-proofs.tgz`

5.2 Lessons for Redex

Our experience suggests that Redex is a mature technology but also highlights gaps in its ecosystem.

Redex offers little support for handling binding constructs in object languages. It provides a generic function for obtaining a fresh variable but no help in defining capture-avoiding substitution or α-equivalence. Three of the nine papers in section 4 require definitions of one these concepts, and definitions of these concepts facilitate testing in two other papers and the model of section 3. In one case (section 4.4), managing binding in Redex constituted a significant portion of the overall time spent studying the paper. Redex should benefit from a mechanism for dealing with binding, starting from the recently studied approaches (Gabbay and Pitts 2002; Lakin 2010; Pottier 2005; Sewell et al. 2010).

Next, Redex lacks direct support for non-algorithmic relations such as the coercion-insertion theory of Swamy et al. and the declarative typing rules of Schrijvers et al. When we modeled these systems, we were forced to escape to Redex's host language or to adopt an elaborate encoding, which we would not expect a casual Redex user to be comfortable with. Extending Redex with support for logic programming, as in Typol (Despeyroux 1984), Twelf (Pfenning and Schürmann 1999), αProlog (Cheney and Urban 2004), or αML (Lakin 2010) should solve this problem.

At present, Redex also provides no mechanism for specifying structural congruence. This gap complicates the encoding of transition rules such as those Ziarek et al. and Chaudhuri define on concurrent programs. We hope to adapt Maude's (Clavel et al. 2003) associative-commutative matching to Redex's notion of patterns.

Finally, while is often a boon that Redex's random test case generators require little programmer intervention, sometimes they are not as effective as they could be. The generator derived from the grammar in section 3, for example, requires substantial massaging to achieve high test coverage. This deficiency is especially pressing in the case of typed object languages, where the massaging code almost duplicates the specification of the type system.[7] The dynamic-monitoring technique behind Korat (Boyapati et al. 2002) may be effective in automatically constructing tests from the original specification. Alternatively, αProlog's counterexample-search strategies (Cheney and Momigliano 2007) are possibilities with the addition of more declarative support for binding specifications and inference rules.

5.3 Lessons for Developers of Other Tools

Last but not least, our case study suggests several lessons that should apply to all validation tools, regardless of how much they differ from Redex.

First, the lessons for authors concern developers too, since authors require tool support to apply the lessons. In particular, support for execution enables interactive exploration, benefiting authors and readers alike.

Second, tests complement proofs. We encountered five papers in which explicitly claimed theorems are false as stated. In three cases (section 4.2, section 4.6, and section 4.9), we could fix the problems; in the others (section 4.3 and section 4.5), we were unable to find and verify a fix in a modest time frame. In every case, though, rudimentary testing discovered errors missed with pencil-and-paper proofs.

Indeed, we claim that tests complement even machine-checked proofs. As one example, two of the POPLmark solutions that contain proofs of type soundness use call-by-name beta in violation of the specification (Crary and Gacek, personal communication). We believe unit testing would quickly reveal this error.

[7] See Klein et al. (2010, section 7) for another example.

	αML	αProlog	K	Ott	Redex	Ruler
Execution	✓	✓	✓	✓	✓	✓
Unit Tests			✓		✓	
Automated Tests		✓	✓		✓	
Typesetting			✓	✓	✓	✓
Binding	✓	✓	✓	✓		
Visualization			✓		✓	

Figure 5: A comparison of lightweight semantics engineering tools

Even better, one can sometimes test propositions that cannot be validated via proof. Our experience with the model of Racket's delimited control operators provides one example, as no formalization currently exists of the more than 230,000 lines of C and assembly in the Racket implementation. Testing also removes another obstacle to proof, the requirement that we first state the proposition of interest. Due to its exploratory nature, testing can inadvertently falsify unstated but desired propositions, e.g., that threads block without busy waiting (section 4.4). This is especially true for system-level and randomized testing. To some degree, the same is true of proving, but testing seems to be more effective at covering a broad space of system behaviors. Many other validation tools provide some level of support for executing examples without requiring an algorithm to be specified separately; αProlog and Isabelle go so far as to provide tools for automatically falsifying conjectures.

Third, mechanized typesetting avoids many transcription errors. Given the apparent frequency with which we observed typos in ICFP papers and their potential impact on communication, mechanically generating figures from a source subjected to *some* form of mechanical scrutiny seems justified. Ott (Sewell et al. 2010) and Isabelle (Nipkow et al. 2011) already support this workflow.

Fourth, example visualization aids debugging. We relied extensively on Redex's visualization features while investigating the flaws described in section 4, as well as the many more introduced by the manual process of translating figures to Redex. The features have been similarly useful in other efforts, e.g., the formalization of Typed Racket (Tobin-Hochstadt and Felleisen 2008, section 3.4). We conjecture that all validation tools would benefit from visualization components.

6. Related Work

The closest form of related work would be other studies that attempt to validate semantics engineering tools on published formal models, but we are unaware of any such studies. Accordingly, this section focuses on tools that could be used for such studies, large formal models that have been subjected to lightweight forms of validation, and studies of the validity of research results in general.

Other tools. The development of Redex draws inspiration from Alloy (Jackson 2002), a system designed to provide software engineers with a lightweight alternative to theorem proving. With Alloy, software engineers build models of software systems and explore them with mechanical support. Redex seeks to provide a similar experience to semantics engineers.

The Typol system for natural semantics supports a range of tools, providing execution by compilation to Prolog (Despeyroux 1984), a debugger and mechanized typesetting (Despeyroux 1988), and a bridge from lightweight to heavyweight validation (Terrasse 1995). These features and more survive in Redex and other contemporary tools.

Figure 5 provides a comparison between Redex and other modern lightweight semantics engineering tools. All of αML (Lakin 2010), αProlog (Cheney and Urban 2004), K (Rosu and Serbanuta

2010), Ott (Sewell et al. 2010), Redex, and Ruler (Dijkstra and Swierstra 2006) provide support for executing definitions, though in the case of Ott, the precise level of support depends on the particular proof assistant chosen as the backend. αProlog features an automated testing tool similar to `redex-check` but based on bounded-exhaustive search rather than randomized testing. Similarly, K can exploit's Maude's model checker to check predicates. K, Ott, Redex, and Ruler all support mechanized typesetting, but Redex's approach to fine-tuning the output differs—users write Racket code to transform Redex parse trees instead of annotating definitions with LaTeX snippets. αML, αProlog, K, and Ott provide the sort of binding support Redex lacks. Only Redex provides a library of domain-specific constructs for unit-testing and interactive visualization, but K users can write jUnit tests.

Thanks in part to the impetus of the POPLmark Challenge (Aydemir et al. 2005), semantics engineers increasingly use proof assistants (Nipkow et al. 2011; Norell 2007; Pfenning and Schürmann 1999; Slind and Norrish 2008; The Coq Development Team 2010) to validate semantic models. These tools have various levels of support for executing examples, but none share Redex's beginning-to-end support for the semantics engineering life cycle—yet.

Testing Language Definitions. Several groups report success with testing techniques where proof systems fail. For example, Fox (2003), Hardin et al. (2006), Sarkar et al. (2009), and Fox and Myreen (2010) check that they have defined correct models of various assembly and machine languages by comparing their models' answers to the answers produced by actual hardware or by off-the-shelf compilers. Ellison and Rosu (2011) employ similar techniques for C using K. Some of their efforts exploit randomized testing. Klein et al. (2010) also use a randomized technique to compare a model of the Racket virtual machine to the production implementation. The formal model of the R6RS (Sperber et al. 2007) helped catch bugs in the informal, prose specification.

Research Validity. This paper reveals mistakes in our own work and the work of our colleagues. While we did not discover any flaws that invalidate the essential contributions of any of the papers we studied, others have done so. Dwyer et al. (2006) examine several bug-finding systems and invalidate a number of published claims on lowering the search cost; their basic insight is that factors outside the control of an investigator—e.g., the search order for path-sensitive bug-finding tools—may heavily influence the performance of such tools. Similarly, Arcuri and Briand (2011) conduct "a systematic review of the use of randomized algorithms in selected software engineering venues in 2009 [and] show that randomized algorithms are used in a significant percentage of papers but that, in most cases, randomness is not properly accounted for. This casts doubts on the validity of most empirical results assessing randomized algorithms". Further afield, studies concerning the quality of research results are common in the biomedical community. Young et al. (2008), for example, write that "an empirical evaluation of the 49 most-cited papers on the effectiveness of medical interventions, published in highly visible journals in 1990–2004, showed that a quarter of the randomised trials and five of six non-randomised studies had already been contradicted or found to have been exaggerated by 2005."

7. Conclusion

Our validation project confirms the "lightweight mechanization" conjecture. Specifically it establishes Redex as an effective tool that can uncover mistakes in mathematical models of programming languages. The two case studies contribute two different insights.

With the survey of nine ICFP papers we validate the folklore claim that all mathematical papers contain mistakes. Our conclusion is *not* to blame the ICFP authors or reviewers for these mistakes but to suggest the routine use of lightweight tools to write such papers. Every mistake in a published model narrows the communication channel between authors and readers; conversely, we can widen this channel when we equip papers with executable lightweight models that readers can easily explore interactively.

With the case study of delimited continuations in production systems we illustrate how an implementor can benefit from the designers' lightweight model. Redex can help expose errors in an implementation, even a heavily-tested one, merely by testing the correspondence between it and a model. This aspect of semantics engineering is overlooked and deserves more attention, especially for large languages that evolve over many years.

Acknowledgments Thanks to Gavin Bierman, Avik Chaudhuri, Michael Hicks, Suresh Jagannathan, Thomas Jensen, Hai Liu, Jan Midtgaard, Tiark Rompf, Neil Sculthorpe, and Dimitrios Vytiniotis for patient and candid discussions of their work. Thanks also to Henrik Nilsson for helpful comments on a draft of this paper.

The authors gratefully acknowledge support for this research from the NSF, DARPA, and AFOSR.

Bibliography

Andrea Arcuri and Lionel C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proc. Intl. Conf. Soft. Eng.* , pp. 1–10, 2011.

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: the POPLMark Challenge. In *Proc. Intl. Conf. Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science volume 3603, pp. 50–65, 2005.

Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. In *Proc. Intl. Symp. Soft. Testing and Analysis*, pp. 123–133, 2002.

Avik Chaudhuri. A concurrent ML library in Concurrent Haskell. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 269–280, 2009.

James Cheney and Alberto Momigliano. Mechanized metatheory model-checking. In *Proc. Intl. Conf. Principles and Practice of Declarative Programming*, pp. 75–86, 2007.

James Cheney and Christian Urban. αProlog: a logic programming language with names, binding, and α-equivalence. In *Proc. Intl. Conf. Logic Programming*, Lecture Notes in Computer Science volume 3132, pp. 269–283, 2004.

Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 268–279, 2000.

Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. Maude 2.0 system. In *Proc. Intl. Conf. Rewriting Techniques and Applications*, Lecture Notes in Computer Science volume 2706, pp. 76–87, 2003.

John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *Proc. Euro. Symp. Programming*, pp. 320–334, 2001.

Thierry Despeyroux. Executable specification of static semantics. In *Proc. Intl. Symp. Semantics of Data Types*, Lecture Notes in Computer Science volume 173, pp. 215–233, 1984.

Thierry Despeyroux. Typol: a formalism to implement natural semantics. INRIA, Research Report No. 94, 1988.

Atze Dijkstra and S. Doaitse Swierstra. Ruler: programming Type rules. In *Proc. Intl. Symp. Functional and Logic Programming*, Lecture Notes in Computer Science volume 3945, pp. 30–46, 2006.

Matthew B. Dwyer, Suzette Person, and Sebastian G. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *Proc. ACM Symp. Foundations of Soft. Eng.* , pp. 92–104, 2006.

Chucky Ellison and Grigore Rosu. An Executable Formal Semantics of C with Applications. University of Illinois, http://hdl.handle.net/2142/25816, 2011.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2010.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 237–247, 1993.

Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding delimited and composable control to a production programming environment. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 165–176, 2007.

Anthony Fox. Formal specification and verification of ARM6. In *Proc. Intl. Conf. Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science volume 2758, pp. 25–40, 2003.

Anthony Fox and Magnus O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture . In *Proc. Intl. Conf. Interactive Theorem Proving*, Lecture Notes in Computer Science volume 6172, pp. 243–258, 2010.

Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing* 13(3–5), pp. 341–363, 2002.

Kenneth V. Hanford. Automatic generation of test cases. *IBM Systems Journal* 9(4), pp. 244–257, 1970.

David S. Hardin, Eric W. Smith, and William D. Young. A robust machine code proof framework for highly secure applications. In *Proc. Intl. Wksp. ACL2 Theorem Prover and its Applications*, pp. 11–20, 2006.

John Hughes. Generalizing monads to arrows. *Science of Computer Programming* 37(1–3), pp. 67–111, 2000.

Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Software Enginering and Methodology* 11(2), pp. 256–290, 2002.

Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proc. ACM Symp. Principles of Programming Languages*, pp. 295–308, 1996.

Casey Klein and Robert Bruce Findler. Randomized testing in PLT Redex. In *Proc. Scheme and Functional Programming*, pp. 26–36, 2009.

Casey Klein, Matthew Flatt, and Robert Bruce Findler. The Racket virtual machine and randomized testing. 2010. `http://plt.eecs.northwestern.edu/racket-machine/`

Matthew R. Lakin. An Executable Meta-Language for Inductive Definitions with Binders. PhD dissertation, University of Cambridge, 2010.

Hai Liu, Eric Cheng, and Paul Hudak. Causal commutative arrows and their optimization. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 35–46, 2009.

Hai Liu, Eric Cheng, and Paul Hudak. Causal commutative arrows. *J. Functional Programming* 21(4-5), pp. 467–496, 2011.

Jacob Matthews and Robert Bruce Findler. An operational semantics for R5RS Scheme. In *Proc. Scheme and Functional Programming*, pp. 157–165, 2005.

Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *Proc. Intl. Conf. Rewriting Techniques and Applications*, Lecture Notes in Computer Science volume 3091, pp. 301–311, 2004.

Jay McCarthy. Automatically RESTful web applications: marking modular serializable continuations. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 299–309, 2009.

Jan Midtgaard and Thomas P. Jensen. Control-flow analysis of function calls and returns by abstract interpretation. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 287–298, 2009.

Jan Midtgaard and Thomas P. Jensen. Control-flow analysis of function calls and returns by abstract interpretation. *Information and Computation*, 2012. `http://www.cs.au.dk/~jmi/ANF-CFA`

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle/HOL—A Proof Assistant for Higher-Order Logic. Springer Verlag, 2011.

Ulf Norell. Towards a Practical Programming Language Based on Dependent Type Theory. PhD dissertation, Chalmers University of Technology, 2007.

Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from generalized stack inspection. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 216–227, 2005.

Frank Pfenning and Carsten Schürmann. System description: Twelf—a meta-logical framework for deductive systems. In *Proc. Intl. Conf. Automated Deduction*, pp. 202–206, 1999.

Gordon D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science* 1(2), pp. 125–159, 1975.

François Pottier. An overview of Cαml. In *Proc. ACM SIGPLAN ML Wksp.* , Electronic Notes in Theoretical Computer Science volume 148, pp. 27–52, 2005.

John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 317–328, 2009.

Grigore Rosu and Traian Florin Serbanuta. An Overview of the K Semantic Framework. *J. Logic and Algebraic Programming* 79(6), pp. 397–434, 2010.

Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *Proc. ACM Symp. Principles of Programming Languages*, pp. 379–391, 2009.

Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for GADTs. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 341–352, 2009.

Neil Sculthorpe. Towards Safe and Efficient Functional Reactive Programming. Ph.D. dissertation, University of Nottingham, 2011.

Neil Sculthorpe and Henrik Nilsson. Safe functional reactive programming through dependent types. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 23–34, 2009.

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: effective tool support for the working semanticist. *J. Functional Programming* 20(1), pp. 71–122, 2010.

Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Proc. Intl. Conf. Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science volume 5170, pp. 28–32, 2008.

Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Richard Kelsey, William Clinger, Jonathan Rees, Robert Bruce Findler, and Jacob Matthews. Revised [6] report on the algorithmic language Scheme. Cambridge University Press, 2007.

Nikhil Swamy, Michael Hicks, and Gavin M. Bierman. A Theory of typed coercions and its applications. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 329–340, 2009.

Delphine Terrasse. Encoding natural semantics in Coq. In *Proc. Intl. Conf. Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science volume 936, pp. 230–244, 1995.

The Coq Development Team. The Coq Proof Assistant Reference Manual. Version 8.3, 2010. `http://coq.inria.fr/`

Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *Proc. ACM Symp. Principles of Programming Languages*, pp. 395–406, 2008.

Neal S. Young, John P.A. Ioannidis, and Omar Al-Ubaydli. Why Current Publication Practices May Distort Science. *PLoS Med* 5(10), pp. 1418–1422, 2008.

Lukasz Ziarek, KC Sivaramakrishnan, and Suresh Jagannathan. Partial memoization of concurrency and communication. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 161–172, 2009.