# A Visual Environment for Developing Context-Sensitive Term Rewriting Systems

Jacob Matthews[1], Robert Bruce Findler[1], Matthew Flatt[2], and Matthias Felleisen[3]

[1] University of Chicago {jacobm, robby}@cs.uchicago.edu
[2] University of Utah mflatt@cs.utah.edu
[3] Northeastern University matthias@ccs.neu.edu

**Abstract.** Over the past decade, researchers have found context-sensitive term-rewriting semantics to be powerful and expressive tools for modeling programming languages, particularly in establishing type soundness proofs. Unfortunately, developing such semantics is an error-prone activity. To address that problem, we have designed PLT Redex, an embedded domain-specific language that helps users interactively create and debug context-sensitive term-rewriting systems. We introduce the tool with a series of examples and discuss our experience using it in courses and developing an operational semantics for $R^5$RS Scheme.

## 1 Introduction

Since the late 1980s researchers have used context-sensitive term-rewriting systems as one of their primary tools for specifying the semantics of programming languages. They do so with good reason. Syntactic rewriting systems have proved to be simple, flexible, composable, and expressive abstractions for describing programming language behaviors. In particular, the rewriting approach to operational semantics described by Felleisen and Hieb [1] and popularized by Wright and Felleisen [2] is widely referenced and used.

Unfortunately, designing context-sensitive rewriting systems is subtle and error-prone. People often make mistakes in their rewriting rules that can be difficult to detect, much less correct. Researchers have begun to acknowledge and respond to this difficulty. Xiao, Sabry, and Ariola [3], for instance, developed a tool that verifies that a given context-sensitive term-rewriting system satisfies the unique evaluation context lemma. In the same spirit, we present PLT Redex, a domain-specific language for context-sensitive reduction systems embedded in PLT Scheme [4]. It allows its users to express rewriting rules in a convenient and precise way, to visualize the chains of reductions that their rules produce for particular terms, and to test subject reduction theorems. In section 2 of this paper we briefly explain context-sensitive term rewriting, in section 3 we introduce the rewrite language through a series of examples, and in section 4 we discuss how the language helps with subject reduction proofs. We discuss our experience in section 5, related work in section 6, and conclude with section 7.

## 2 Context-Sensitive Rewriting: A Brief Overview

In his seminal paper [5] on the relationship among abstract machines, interpreters and the $\lambda$-calculus, Plotkin shows that an evaluator specified via an abstract machine defines

$$
\begin{array}{lll}
e = (e\ e) \mid x \mid v & ((\lambda\ (x)\ e)\ v) \mapsto e[v/x] & (\beta_v) \\
v = (\lambda\ (x)\ e) \mid f & (f\ v) \mapsto \delta(f,v) & (\delta_v)
\end{array}
$$

**Plotkin**

$$\frac{e \mapsto e'}{e \rightarrow e'}$$

$$\frac{e \rightarrow e'}{(e\ e'') \rightarrow (e'\ e'')}$$

$$\frac{e \rightarrow e'}{(v\ e) \rightarrow (v\ e')}$$

**Felleisen/Hieb**

$E = [] \mid (v\ E) \mid (E\ e)$

if $e \mapsto e'$, then $E[e] \rightarrow E[e']$

**Fig. 1.** Specifying an evaluator for $\lambda_v$

the same function as an evaluator specified via a recursive interpreter. Furthermore, the standard reduction theorem for a $\lambda$-calculus generated from properly restricted reduction relations is also equivalent to this function. As Plotkin indicated, the latter definition is by far the most concise and the easiest to use for many proofs.

Figure 1 presents Plotkin's $\lambda_v$-calculus. The top portion defines expressions, values, and the two basic relations ($\beta_v$ and $\delta_v$). The rules below on the left are his specification of the strategy for applying those two basic rules in a leftmost-outermost manner.

In a 1989 paper, Felleisen and Hieb [1] develop an alternate presentation of Plotkin's $\lambda$-calculi. Like Plotkin, they use $\beta_v$ and $\delta_v$ as primitive rewriting rules. Instead of inference rules, however, they specify a set of evaluation contexts. Roughly speaking, an *evaluation context* is a term with a hole at the point where the next rewriting step must take place. Placing a term in the hole is equivalent to the textual substitution of the hole by the term [6]. The right side of the bottom half of figure 1 shows how to specify Plotkin's evaluator function with evaluation contexts.

While the two specifications of a call-by-value evaluator are similar at first glance, Felleisen and Hieb's is more suitable for extensions with non-functional constructs (assignment, exceptions, control, threads, etc). Figure 2 shows how easy it is to extend the system of figure 1 (right) with assignable variables. Each program is now a pair of a store and an expression. The bindings in the store introduce the mutable variables and bind free variables in the expression. When a dereference expression for a store variable appears in the hole of an evaluation context, it is replaced with its value. When an assignment with a value on the right-hand side appears in the hole, the let-bindings are modified to capture the effect of the assignment statement. The entire extension consists of three rules, with the original two rules included verbatim. Felleisen and Hieb also showed that this system can be turned into a conventional context-free calculus like the $\lambda$-calculus.

Context-sensitive term-rewriting systems are ideally suited for proving the type soundness of programming languages. Wright and Felleisen [2] showed how this works for imperative extensions of the $\lambda$-calculus and a large number of people have adapted the technique to other languages since then.

$$
\begin{aligned}
p &= ((\mathit{store}\ (x\ v)\ \ldots)\ e)\\
e &= \cdots\ \mathit{as\ before}\ \cdots \mid (\mathit{let}\ ((x\ e))\ e) \mid (\mathit{set!}\ x\ e)\\
P &= ((\mathit{store}\ (x\ v)\ \ldots)\ E)\\
E &= \cdots\ \mathit{as\ before}\ \cdots \mid (\mathit{let}\ ((x\ E))\ e) \mid (\mathit{set!}\ x\ E)
\end{aligned}
$$

$$
\begin{array}{l}
((\mathit{store}\ (x_1\ v_1)\ \ldots\ (x_2\ v_2)\ (x_3\ v_3)\ \ldots)\\
\ E[x_2]) \rightarrow\\
((\mathit{store}\ (x_1\ v_1)\ \ldots\ (x_2\ v_2)\ (x_3\ v_3)\ \ldots)\\
\ E[v_2])
\end{array}
\qquad
\begin{array}{l}
((\mathit{store}\ (x_1\ v_1)\ \ldots\ (x_2\ v_2)\ (x_3\ v_3)\ \ldots)\\
\ E[(\mathit{set!}\ x_2\ v_4)]) \rightarrow\\
((\mathit{store}\ (x_1\ v_1)\ \ldots\ (x_2\ v_4)\ (x_3\ v_3)\ \ldots)\\
\ E[v_4])
\end{array}
$$

$$
\begin{array}{l}
((\mathit{store}\ (x_1\ v_1)\ \ldots)\ E[(\mathit{let}\ ((x_2\ v_2))\ e)]) \rightarrow\\
((\mathit{store}\ (x_1\ v_1)\ \ldots\ (x_3\ v_2))\ E[e[x_3\ /\ x_2]])\\
\ \text{where } x_3 \text{ is fresh}
\end{array}
\qquad
\text{if } e \mapsto e', \text{ then } P[e] \rightarrow P[e']
$$

**Fig. 2.** Specifying an evaluator for $\lambda_S$

Not surprisingly, though, as researchers have modelled more and more complex languages with these systems, they have found it more and more difficult to model them accurately, as two of the authors discovered when they used it to specify the kernel of Java [7].

## 3 A Language for Specifying Context-Sensitive Rewriting

To manage this complexity, we have developed PLT Redex, a declarative domain-specific language for specifying context-sensitive rewriting systems. The language is embedded in MzScheme [4], an extension of $R^5$RS Scheme. MzScheme is particularly suitable for our purposes for two reasons. First, as an extension of Scheme, its basic form of data includes S-expressions and primitives for manipulating S-expressions as patterns. Roughly speaking, an S-expression is an abstract syntax tree representation of a syntactic term, making it a natural choice for manipulating program text. Second, embedding PLT Redex in MzScheme gives PLT Redex programmers a program development environment and extensive libraries for free.

The three key forms PLT Redex introduces are **language**, **red**, and *traces* (we typeset syntactic forms in bold and functions in italics). The first,

(**language** (*<non-terminal-name>* *<rhs-pattern>* ...) ...)

specifies a BNF grammar for a regular tree language. Each right-hand side is written in PLT Redex's pattern language (consisting of a mixture of concrete syntax elements and non-terminals, roughly speaking). With a language definition in place, the **red** form is used to define the reduction relation:

(**red** *<language-name>* *<lhs-pattern>* *<consequence>*)

Syntactically, it consists of three sub-expressions: a language to which the reduction applies, a source pattern specifying which terms the rule matches, and Scheme code that,

$$
\begin{aligned}
e &= v \mid (e\ e) \mid (+\ e\ e) \mid x \\
v &= (\lambda\ (x)\ e) \mid \mathit{number} \\
c &= [\,] \\
&\mid (v\ c) \mid (c\ e) \\
&\mid (+\ v\ c) \mid (+\ c\ e) \\
x &\in \mathit{Vars}
\end{aligned}
$$

```
(define λv
  (language (e v (e e) (+ e e) x)
            (v (lambda (x) e) number)
            (c hole
               (v c) (c e)
               (+ v c) (+ c e))
            (x (variable-except lambda +))))
```

$$
\begin{aligned}
&c[(+\ \ulcorner n_1 \urcorner\ \ulcorner n_2 \urcorner)] \\
&\rightarrow c[\ulcorner n_1 + n_2 \urcorner]
\end{aligned}
$$

```
(red λv (in-hole c_1 (+ number_1 number_2))
  (replace (term c_1) (term hole)
           (+ (term number_1) (term number_2))))
```

$$
\begin{aligned}
&c[(\lambda\ (x)\ e)\ v)] \\
&\rightarrow c[e[x/v]]
\end{aligned}
$$

```
(red λv (in-hole c_1 ((lambda (x_1) e_1) v_1))
  (replace (term c_1) (term hole)
           (substitute (term x_1)
                       (term v_1)
                       (term e_1)))))
```

**Fig. 3.** $\lambda_v$ semantics

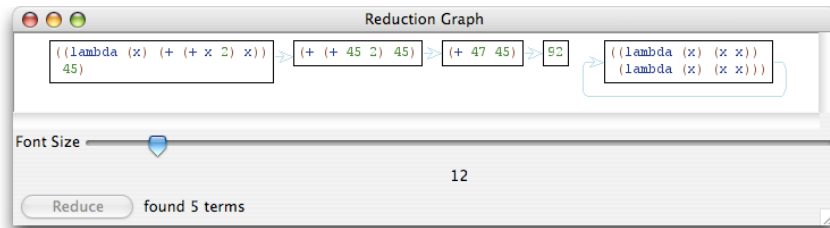



**Fig. 4.** Reduction of a simple $\lambda_v$ term and of $\Omega$

when evaluated, produces the resulting term as an S-expression. Finally, the function *traces* accepts a language, a list of reductions, and a term (in Scheme terms, an arbitrary S-expression). When invoked, it opens a window that shows the reduction graph of terms reachable from the initial term. All screenshots in this paper show the output of *traces*. The remainder of this section presents PLT Redex via a series of examples.

### 3.1 Example: $\lambda_v$

Our first example is Plotkin's call-by-value $\lambda$-calculus, extended with numbers and addition. Figure 3 shows its definition in Felleisen and Hieb's notation on the left, and in PLT Redex, on the right.

The $\lambda_v$ language consists of abstractions, numbers, applications, sums, and variable references, and has only two rewriting rules. As figure 3 shows, the traditional mathe-

matical notation translates directly into PLT Redex: each line in the BNF description of $\lambda_v$'s grammar becomes one line in **language**. The pattern (*variable-except lambda* $+$) matches any symbol except those listed (in this case, *lambda* and $+$).

The reduction rules also translate literally into uses of the **red** form. The first reduction rule defines the semantics of addition. The pattern in the second argument to **red** matches expressions where a syntactic term of the form ($+$ *number number*) is the next step to be performed. It also binds the pattern metavariables *c_1*, *number_1*, and *number_2* to the context and $+$'s operands, respectively. In general, pattern variables with underscores must match the non-terminal before the underscore.

The third subexpression of **red** constructs a new S-expression where the addition operation is replaced by the sum of the operands, using Scheme's $+$ operator (numeric constants in S-expressions are identical to the numbers they represent). The *in-hole* pattern is dual to the *replace* function. The former decomposes an expression into a context and a hole, and the latter composes an expression from a context and its hole's new content. The **term** form is PLT Redex's general-purpose tool for building S-expressions. Here we use it only to dereference pattern variables. The second reduction rule, $\beta_v$, uses the function *substitute* to perform capture-avoiding variable substitution.[4] Figure 4 shows a term that reduces to 92 on the left and a term that diverges on the right. Arrows are drawn from each term to the terms it can directly reduce to; the circular arrow attached to the $\Omega$ term indicates that it reduces to itself. In general, the *traces* function generates a user-specified number of terms and then waits until the "Reduce" button is clicked.

### 3.2 Example: $\lambda_S$

Figure 5 contains PLT Redex definitions for $\lambda_S$ in parallel to the definitions given in figure 2. The first rules uses an ellipses pattern to match a sequence of any length, including zero, whose elements match the pattern before the ellipses. In this case, the pattern used to match against the store is a common idiom, matching three instances of a pattern with ellipses after the first and the last. This idiom is used to select an interesting S-expression in a sequence; in this case it matches *x_i* to every variable in the store and *v_i* to the corresponding value. To restrict the scope of the match, we use the same pattern variable, *x_i*, in both the store and in the expression. This duplication constrains S-expressions matched in the two places to be structurally identical, and thus the variable in the store and the variable in the term must be the same.

In figure 5 we also see **term** used to construct construct large S-expressions rather than just to get the values of pattern metavariables. In addition to treating pattern variables specially, **term** also has special rules for commas and ellipses. The expression following a comma is evaluated as Scheme code and its result is placed into the S-expression at that point. Ellipses in a **term** expression are duals to the pattern ellipses. The pattern before an ellipsis in a pattern is matched against a sequence of S-expressions and the S-expression before an ellipsis in a **term** expression is expanded into a sequence

[4] Currently, *substitute* must be defined by the user using a more primitive built-in form called **subst** whose details we elide for space. We intend to eliminate this requirement in a future version of PLT Redex; see section 7.

```
(define λS
  (language
    (p ((store (x v) ...) e))
    (e ··· as before ···
      (let ((x e)) e)
      (set! x e))
    (x (variable-except
      lambda set! let))

   (PC ((store (x v) ...) EC))
   (EC ··· as before ···
      (set! x EC)
      (let ((x EC)) e)))

(red λS
    ((store (x_a v_a) ... (x_i v_i) (x_b v_b) ...)
     (in-hole EC_1 x_i))
    (term ((store (x_a v_a) ... (x_i v_i) (x_b v_b) ...)
       ,(replace (term EC_1) (term hole) (term v_i)))))
(red λS
    ((store (x_a v_a) ... (x_i v_old) (x_b v_b) ...)
     (in-hole EC_1 (set! x_i v_new)))
    (term ((store (x_a v_a) ... (x_i v_new) (x_b v_b) ...)
       ,(replace (term EC_1) (term hole) (term v_new)))))
(red λS
    ((store (x_a v_a) ...)
     (in-hole EC_1 (let ((x_i v_i)) e_1)))
    (let ((new-x (variable-not-in (term (x_a ...)) (term x_i))))
       (term ((store (x_a v_a) ... (,new-x v_i))
          ,(replace (term EC_1) (term hole)
                    (substitute (term x_i) new-x (term e_1)))))))
```

**Fig. 5.** $\lambda_S$ semantics

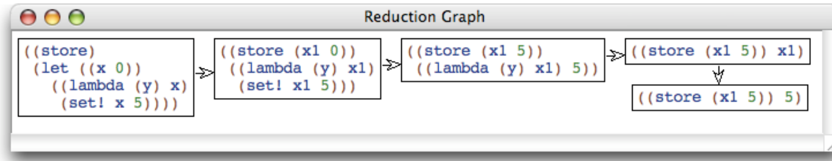



**Fig. 6.** Reduction of a simple $\lambda_S$ term

of S-expressions and spliced into its context.[5] Accordingly, the first rule produces a term whose store is identical to the store in the term it consumed.

The final rule also introduces another PLT Redex function, *variable-not-in*, which takes an arbitrary syntactic term and a variable name and produces a new variable whose name is similar to the input variable's name and that does not occur in the given term.

Figure 6 shows a sample reduction sequence in $\lambda_S$ using, in order, a *let* reduction, a *set!* reduction, a $\beta_v$ reduction, and a dereference reduction.

### 3.3 Example: Threaded $\lambda_S$

We can add concurrency to $\lambda_S$ with surprisingly few modifications. The language changes as shown in figure 7. A program still consists of a single store, but instead of just one expression it now contains one expression per thread. In addition, each reference to *EC* in the $\lambda_S$ reductions becomes *TC*. No other changes need to be made, and in particular no reduction rules need modification.

[5] With the exception of ellipsis and pattern variables, **term** is identical to Scheme's **quasiquote**.

(**define** $t$-$\lambda_S$
  (**language**
    (*p* ((*store* (*x v*) . . . ) (*threads e* . . . )))
    (*PC* ((*store* (*x v*) . . . ) *TC*))
    (*TC* (*threads e* . . . *EC e* . . . ))
    · · · *as before* · · · )

**Fig. 7.** Threaded $\lambda_S$

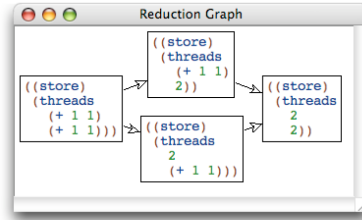
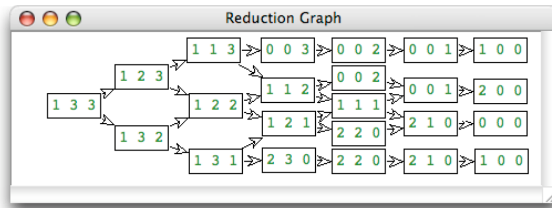



**Fig. 8.** Multiple reductions

((*store* (*x 1*))
  (*threads*
    (*set! x* (+ *x 1*))
    (*set! x* (+ *x* −*1*))))




On the left, a threaded $\lambda_S$ term and on the right, boxes containing $x$'s value and the number of subexpressions remaining in each thread

**Fig. 9.** Reduction summary using *traces*

To express non-determinism, PLT Redex's pattern language supports ambiguous patterns by finding *all possible* ways a pattern might match a term. Consider the *TC* evaluation context in figure 7, which uses the selection idiom described in section 3.2. Unlike that example, nothing restricts this selection to a particular thread, so PLT Redex produces multiple matches, one for each reducible thread. The *traces* window reflects this by displaying all of the reductions that apply to each term when constructing the reduction graph, as shown in figure 8.

Due to the possible interleaving of multiple threads, even simple expressions reduce many different ways and gaining insight from a thicket of terms can be difficult. Accordingly, *traces* has an optional extra argument that allows the user to provide an alternative view of the term that can express a summary or just the salient part of a term without affecting the underlying reduction sequence. Figure 9 shows an example summarized reduction sequence.

## 4 Subject Reduction

A widely used proof technique for establishing type soundness [2] is based on context-sensitive rewriting semantics. Each proof has a key subject-reduction lemma that guarantees that the type of a term before a reduction matches the type after the reduction.

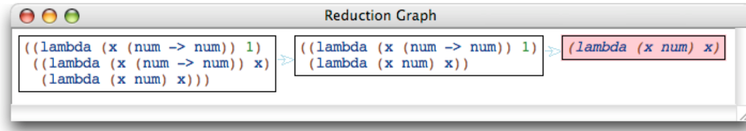



**Fig. 10.** Subject fails in second reduction

PLT Redex provides support for exploring and debugging such subject-reduction lemmas. The function *traces/predicate* allows the user to specify a predicate that implements the subject of the subject-reduction lemma. Then, *traces/predicate* highlights in red italics any terms that do not satisfy it.

As an example, figure 10 shows a reduction sequence for a term where the third step's type does not match the first step's. In this particular example, the user swapped the order of arguments to *substitute*, making the $\beta_v$ reduction incorrectly substitute the body of the function into the argument.

## 5 Experience Using PLT Redex

As the examples in section 3 suggest, PLT Redex is suitable for modeling a wide variety of languages. That suggestion has been borne out in practice as we have developed a reduction semantics for $R^5RS$ Scheme that captures the language in as much detail as the $R^5RS$ formal semantics does [8, section 7.2]. Our experience developing one facet of the semantics highlights the strength of PLT Redex. In evaluating a procedure call, the $R^5RS$ document deliberately leaves unspecified the order in which arguments are evaluated, but specifies that [8, section 4.1.3]

> *the effect of any concurrent evaluation of the operator and operand expressions is constrained to be consistent with* some *sequential order of evaluation. The order of evaluation may be chosen differently for each procedure call*

In the formal semantics section, the authors explain how they model this ambiguity:

> *[w]e mimic [the order of evaluation] by applying arbitrary permutations* permute *and* unpermute *. . . to the arguments in a call before and after they are evaluated. This is not quite right since it suggests, incorrectly, that the order of evaluation is constant throughout a program . . ..* [8, Section 7.2].

We realized that our rules, in contrast, can capture the intended semantics using nondeterminism to select the argument to reduce. Our initial, incorrect idea of how to capture this was to change the definition of expression evaluation contexts (otherwise similar to those in $\lambda_S$) so that they could occur on either side of an application:

$$(\textbf{language}\ (EC\ (e\ EC)\ (EC\ e))\ \cdots\ \textit{as before}\ \cdots))$$

```
(define r5                         (red r5 (in-hole PC_1 (e_1 inert_1))
  (language                              (replace (term PC_1) (term hole)
    (inert (mark v)                               (term ((mark e_1) inert_1))))
           e)                      (red r5 (in-hole PC_1 (inert_1 e_1))
    (EC ((mark EC) inert)                (replace (term PC_1) (term hole)
        (inert (mark EC))))                       (term (inert_1 (mark e_1)))))
    ··· as before ···)             (red r5 (in-hole c_1 ((mark (lambda (x_1) e_1)) (mark v_1)))
                                         (replace (term c_1)
                                                  (term hole)
                                                  (substitute (term x_1) (term v_1) (term e_1)))))
```

**Fig. 11.** Reduction rules for unspecified application order

When we visualized a few reductions using that modification using *traces*, we quickly spotted an error in our approach. We had accidentally introduced non-deterministic concurrency to the language. The term

```
(let ((x 1)) (((lambda (y) (lambda (z) x))
               (set! x (+ x 1)))
              (set! x (− x 1))))
```

should always reduce to *1*. Under our semantics, however, it could also reduce to *2* and *0*, just as the term in figure 9 does.

Experimenting with the faulty system gave us the insight into how to fix it. We realized that the choice of which term in an application should be ambiguous, but once the choice had been made, there should not be any further ambiguity. Accordingly, we introduced a mark in application expressions. The choice of where to place the mark is arbitrary, but once a mark is placed, evaluation under that mark must complete before the other subexpression of an application is evaluated.

Figure 11 shows the necessary revisions to $\lambda_S$ to support $R^5RS$-style procedure applications. We introduce the non-terminal *inert* to stand for terms where evaluation does not occur, *i.e.*, unmarked expressions or marked values. The top two reductions on the right-hand side of the figure non-deterministically introduce marks into applications. The evaluation contexts change to ensure that evaluation only occurs inside marked expressions, and application changes to expect marked procedures and arguments.

In addition to providing a graphical interface to reduction graphs, PLT Redex also provides a programatic interface to the reduction graphs as a consequence of its being embedded in PLT Scheme. This interface let us build large automatic test suites for the $R^5RS$ semantics system among others that we could run without having to call *traces* and produce visual output. We found these test cases to be invaluable during development, since changes to one section of our semantics often had effects on seemingly unrelated sections and inspecting visual output manually quickly became infeasible.

We gained additional experience by using PLT Redex as a pedagogical tool. The University of Utah's graduate-level course on programming languages introduces students to the formal specification of languages through context-sensitive rewriting. Stu-

dents model a toy arithmetic language, the pure $\lambda$-calculus, the call-by-value $\lambda$-calculus (including extensions for state and exceptions), typed $\lambda$-calculi, and a model of Java.

In the most recent offering of the course, we implemented many of the course's reduction systems using PLT Redex, and students used PLT Redex to explore specific evaluations. Naturally, concepts such as confluence and determinism stood out particularly well in the graphical presentation of reduction sequences. In the part of the course where we derive an interpreter for the $\lambda$-calculus through a series of "machines," PLT Redex was helpful in exposing the usefulness of each machine change.

As a final project, students implemented context-sensitive rewriting models from recent conference papers as PLT Redex programs. This exercise provided students with a much deeper understanding of the models than they would have gained from merely reading the paper. For a typical paper, students had to fill significant gaps in the formal content (*e.g.*, the figures with grammars and reduction rules). This experience suggests that paper authors could benefit from creating a machine-checked version of a model, which would help to ensure that all relevant details are included in a paper's formalism.

## 6 Related Work

Many researchers have implemented programs similar to our reduction tool. For example, Elan [9], Maude [10], and Stratego [11] all allow users to implement term-rewriting systems (and more), but are focused more on context-free term-rewriting. The ASF+SDF compiler [12] has strong connections to PLT Redex but is geared towards language implementation rather than exploration and so makes tradeoffs that do not suit the needs of lightweight debugging (but that make it a better tool for building efficient large-scale language implementations).

Our reduction tool is focused on context-sensitive rewriting and aims to help its users visualize and understand rewriting systems rather than employ them for some other purpose. The $\mathbf{in}^2$ graphical interpreter for interaction nets [13] also helps its users visualize sequences of reductions, but is tailored to a single language.

## 7 Conclusion and Future Work

Our own efforts to develop novel rewriting systems and to teach operational semantics based on term-rewriting to students have been aided greatly by having an automatic way to visualize rewriting systems. We are confident that PLT Redex can be useful to others for the same purposes.

Our implementation is reasonably efficient. The test suite for our "beginner" language semantics, a system with 14 nonterminals with 55 total productions and 52 reduction rules that models a reasonable purely-functional subset of Scheme intended for beginning programmers, runs 90 reductions in just over 2 seconds on our test machine. This represents a huge slowdown over the speed one would expect from a dedicated interpreter, but in practice seems quick enough to be useful.

We plan to extend PLT Redex to allow simple ways to express the binding structure of a language, which will allow us to synthesize capture-avoiding substitution rules

automatically. We also plan to add more support for the reduction rules commonly used in the literature, such as source patterns that match only if other patterns did not.

Our implementation of PLT Redex is available as an add-on package to DrScheme (http://www.drscheme.org). Choose DrScheme's File|Install .plt File menu item and supply this url: http://people.cs.uchicago.edu/%7Ejacobm/plt/pltredex.plt

**Acknowledgements.** We would like to thank Richard Cobbe, the students of the University of Utah's Spring 2003 CS 6520 class, and the anonymous reviewers of RTA 2004 for their helpful feedback on PLT Redex and this paper.

## References

1. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theoretical Computer Science (1992) 235–271
2. Wright, A., Felleisen, M.: A syntactic approach to type soundness. Information and Computation (1994) 38–94 First appeared as Technical Report TR160, Rice University, 1991.
3. Xiao, Y., Sabry, A., Ariola, Z.M.: From syntactic theories to interpreters: Automating the proof of unique decomposition. In: Higher-Order and Symbolic Computation. Volume 4. (1999) 387–409
4. Flatt, M.: PLT MzScheme: Language manual. Technical Report TR97-280, Rice University (1997) http://www.mzscheme.org/.
5. Plotkin, G.D.: Call-by-name, call-by-value and the $\lambda$-calculus. Theoretical Computer Science **1** (1975) 125–159
6. Barendregt, H.: The Lambda Calculus, Its Syntax and Semantics. Volume 103 of Studies in Logics and the Foundations of Mathematics. North Holland, Amsterdam (1981)
7. Flatt, M., Krishnamurthi, S., Felleisen, M.: A programmer's reduction semantics for classes and mixins. Formal Syntax and Semantics of Java **1523** (1999) 241–269 Preliminary version appeared in proceedings of *Principles of Programming Languages*, 1998. Revised version is Rice University technical report TR 97-293, June 1999.
8. Kelsey, R., Clinger, W., (Editors), J.R.: Revised$^5$ report of the algorithmic language Scheme. ACM SIGPLAN Notices **33** (1998) 26–76
9. Borovansky, P., Kirchner, C., Kirchner, H., Moreau, P.E., Vittek, M.: ELAN: A logical framework based on computational systems. In: Proc. of the First Int. Workshop on Rewriting Logic. Volume 4., Elsevier (1996)
10. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: Specification and programming in rewriting logic. Theoretical Computer Science (2001)
11. Visser, E.: Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In Middeldorp, A., ed.: Rewriting Techniques and Applications (RTA'01). Volume 2051 of Lecture Notes in Computer Science., Springer-Verlag (2001) 357–361
12. van den Brand, M.G.J., Heering, J., Klint, P., Oliver, P.A.: Compiling language definitions: The ASF+SDF compiler. ACM Transactions on Programming Languages and Systems **24** (2002) 334–368
13. Lippi, S.: in2: A Graphical Interpreter for Interaction Nets (system description). In Tison, S., ed.: Rewriting Techniques and Applications, 13th International Conference, RTA-02. LNCS 2378, Copenhagen, Denmark, Springer (2002) 380–384