Grasp

A dynamic, context-oriented, fully pluggable term rewriting system

Eric Griffis

egriffis@cs.ucla.edu

Abstract

Programming language designers must routinely strike a compromise between flexibility, simplicity, and performance as competing goals determined by their creations' intended audiences. Historical programming language designs valued flexibility and performance over simplicity. We introduce a novel design methodology called *pluggable programming* that is simpler and less restrictive than existing methodologies centered around module- or object- based encapsulation. To showcase the flexibility inherent in our novel approach, we design and implement *Grasp*—to our knowledge, the first fully pluggable programming language.

1. Introduction

Good, cheap, fast—pick two. Software engineers routinely balance these competing objectives. As programming language designers, we may re-cast this trilemma: flexible, simple, performant—pick two. Historically, programming language designs valued flexibility and performance over simplicity. Some time around the late sixties, language designers began to realize that simplicity was of value [5] comparable to the other two qualities. The "simplicity in computing" revolution started around the time of IMP [13] and Smalltalk [9] and has continued to bear fruit [15] [12] [8] [10] [14] [1] [2]. We now understand simplicity as the desirable quality of a system which harmonizes generality and ease of use.

This report attempts to tip the balance even further toward flexibility and simplicity by pushing the limits of abstraction to maximize a quality we call *pluggability*. Though this word might evoke imagery of modules [2], our definition of pluggability transcends the module. In fact, our novel approach completely inverts the established software construction methodology. Our reasoning for abandoning decades of established tradition is thus: modules are agents of restriction; given a fully open and pluggable system, we can maximize flexibility without resorting to modules by simplifying program composition in a way that allows us to write arbitrarily complex programs in their simplest possible forms and then improve their performance when and where needed by composing them with more performant programs.

The cult of modularity is perhaps our greatest obstacle to full pluggability. Software built in traditional languages like C, C++, or Java require strong assumptions—such as knowledge of external bindings at write, compile, and run times—that, in many cases, do not scale well beyond a single program or library and ultimately restrict the sharing of functionality between dissimilar programs. Dynamic object-oriented languages like Smalltalk, Python, and Ruby provide features more amenable to program sharing. Perl has been very successful in this respect, as evidenced by its extensive CPAN library.

We *can* do better, though. These languages are designed, each to varying degrees, around a notion of encapsulation by restriction. We believe that Perl's technical success is a direct consequence of its cultural success. To justify this claim, we quote Larry Wall's daughter Heidi [16] on Perl's success as "the first postmodern" programming language:

> That's why IMP [Interactive Math Program, a math curriculum in which you sort of learn everything at once] is better for math students like me–we learn better when we can see the big picture, and how everything fits in. The old way of learning math never gave you any context.
>
> ⋮
>
> Look at the big picture. Don't focus in on two or three things to the exclusion of other things. Keep everything in context. Don't go out of your way to justify stuff that's obviously cool. Don't ridicule ideas merely because they're not the latest and greatest. Pick your own fashions. Don't let someone else tell you what you should like.

Setting any philosophical ideals aside, we must explain the mechanisms by which we intend to implement such lofty ambitions. Grasp is an extensible term rewriting platform with flexible syntax that provides at least two tools to the postmodern programmer. First, domain-specific syntax is easy to capture and transform in Grasp. Second, Grasp can translate abstract high level programs incrementally into concrete and locally optimized programs. These features advance the state of the art in programming methodology in subtle but tremendously valuable ways—no less than a generalization of object-oriented programming with support

for modern features like partial evaluation, declarative style, and much more.

Grasp expands upon the notion of Smalltalk [9] as a recursion on the notion of computer. We generalize the five major concepts of Smalltalk [7]—objects, messages, classes, instances, and methods—to four concepts: contexts, stages, patterns, and bindings. We define these terms in Section 5 and use them to discuss Grasp's features in Sections 6-8.

This report is organized as follows. Section 2 places Grasp within the bigger picture of existing work. Section 3 introduces the syntax and semantics of Grasp in an informal style. Section 4 introduces details necessary for understanding Grasp source programs. Section 5 explains the four major concepts of Grasp and how these concepts are combined to make programs. Section 6 provides simple example of Grasp in action.

2. Related Work

Operationally speaking, Grasp is a veritable mash-up of Church's untyped lambda calculus [3], Scheme [10], Smalltalk [7], and MetaML [14], though Grasp borrows liberally from the syntax of modern research-oriented functional languages like Haskell [8] and ML [12].

Theoretically speaking, Grasp is founded on various ideas gleaned from existing work on rewrite systems [4] in general and term rewriting systems (TRS) [11] in particular, although a formal theory for Grasp is a topic for future work. Various languages based on TRS exist, and systematic comparisons of Grasp with other TRS is another topic for future work.

Practically speaking, Grasp's utility overlaps significantly with OMeta [17]. OMeta extends a host language with a PEG-based DSL to process other languages. OMeta programs are grammars with semantic actions written in the host language. Grasp, on the other hand, is self-hosting and uses pattern matching instead of grammars to accomplish roughly the same tasks. The set of languages generated by Grasp patterns is, consequently, strictly smaller than the set of languages generated by OMeta grammars.

3. Overview of Grasp

3.1 Semantics

This section gives an overview of Grasp's semantics. A detailed informal semantics is the subject of Section 5. Appendix A provides a formal operational semantics.

Grasp can be characterized loosely as a statically scoped programming language. Each use of a *bound* variable is associated with a lexically apparent binding of that variable.

Grasp provides no built-in typing facilities. Fully pluggable [1] type checkers, inferencers, and coercers may all be written as Grasp programs and applied at will. We illustrate the power of this approach with a hypothetical type checker for explicitly typed programs. First, observe that a type checker is merely a program that consumes other programs. Our hypothetical checker can produce various useful behaviors. For instance, it can evaluate a successfully-checked program, return a type-erased copy of the input program, instrument typed expressions with run-time type checks, or simply report the results of the check.

Grasp bindings are objects in their own right. Bindings can be created dynamically, stored in data structures, returned as results, and so on. Arguments to Grasp bindings can be passed by value *or* by name, depending on the relationship between the applied binding and the active context, which means that the actual argument expressions may or may not be evaluated before the binding gains control. Furthermore, call-by-name semantics may be specified explicitly. See Section 5.4 for details.

One distinguishing feature of Grasp is that evaluation contexts, which in most other languages only operate implicitly, have "first-class" status as well as explicit activation syntax. Contexts are useful for implementing a wide variety of constructs, including closures, objects, and records.

A related distinguishing feature of Grasp is that unevaluate code fragments called *stages* also have "first-class" status. Stages are useful for directing evaluation of code fragments. Under an object-oriented programming style, there is no distinction between message passing and stage-directed evaluation. In fact, Grasp passes messages in the form of stages, which bear resemblance to Smalltalk blocks. We may view messages as blocks to be evaluated within the receiving context. This arrangement is strictly more general than in other object-oriented languages. For instance, according to the SmallTalk-80 reference [7],

> A message specifies which operation is desired, but not how that operation should be carried out. The *receiver*, the object to which the message was sent, determines how to carry out the requested operation. For example, addition is performed by sending a message to an object representing a number.

While this characterization of Smalltalk messages largely applies to Grasp in the abstract, the trailing example highlights an important distinction; namely, in Grasp, addition is performed by sending a message to a context that contains compatible notions of number *and* addition. More specifically, Grasp has no built-in notion of object, number, or arithmetic. Smalltalk numbers are objects which receive the "addition" message parameterized by the addend. In Grasp, numbers and arithmetic operators are just symbols and the entire arithmetic system is fully pluggable. An implementation in which numbers represent objects that receive simple parameterized messages is but one of many possibilities in Grasp.

Staying with the object-oriented perspective for now, another powerful feature of Grasp is that messages may contain unbound symbols. Suppose we have a context that implements arithmetic and another that implements a syntax for matrices. We can then evaluate all of the arithmetic expressions within a given matrix by simply staging the entire matrix within the arithmetic context. Assuming the arithmetic context does not bind the symbols in our matrix notation, those symbols are left intact in the result.

The most unusual aspect of Grasp semantics is its novel combination of staging—rather, stage nesting—and the induced "percolation" of free variables up successive stages. Grasp programs elicit behavior in the form of "closures"—combinations of contexts and stages. A stage is a code fragment guarded against evaluation by a simple annotation, and a context is a collection of declarations that govern how stages are evaluated. When combined, stages and contexts perform computations. Since Grasp closures may contain unbound symbols, they are not technically closed in the traditional sense.

3.2 Syntax

The grammar of Grasp generates a sublanguage of the language used for data. An important consequence of this simple, uniform representation is the susceptibility of Grasp programs and data to uniform treatment by other Grasp programs. The reader performs syntactic as well as lexical decomposition of the data it reads.

3.3 Notation and Terminology

Grasp operational semantics are described in a small-step style, but our unusual evaluation strategy occasionally leads to discussion of various degrees of partial evaluation. The symbol "$\longrightarrow$" should be read "steps to." The symbol "$\longrightarrow^*$" should be read "multi-steps to." The symbol "$\Downarrow$" should be read "evaluates to." For example,

$$2 + 3 \longrightarrow^* plus\ 2\ 3 \Downarrow 5$$

means that the term $2 + 3$ multi-steps to the term $plus\ 2\ 3$, which evaluates to the symbol 5.

3.4 Naming Conventions

Grasp often follows the usual Scheme naming conventions, such as suffixing predicates by "`?`" and infixing type-conversion operations with "`->`". By convention, "`#`" prefixes implementation-specific operations. For example, "`#load`" takes a module name and returns the program stored within the corresponding file. Our usage of "`#`" is, however, the only significant deviation from Scheme naming conventions. In addition to the Scheme convention, the prefix "`:`" often denotes a literal symbol for pattern matching.

4. Lexical Conventions

This section gives an informal account of some of the lexical conventions used in writing Grasp programs. For a formal syntax of Grasp, see Appendix A.

4.1 Symbols

Most atomic constructs allowed by other programming languages belong to a single class of symbols in Grasp. A *symbol* is any sequence of non-whitespace characters not reserved by the formal syntax. In particular, numbers are just symbols and any floating-point implementation must not use "`.`" as a decimal point because the character is reserved for staging annotations. We consider this particular limitation a weakness to be rectified in future versions of the language. Symbols are case-sensitive.

4.2 Whitespace and Comments

Whitespace characters include spaces, horizontal tabs, and newlines. Whitespace is used for improved readability and as necessary to separate symbols from each other, but is otherwise insignificant. Whitespace may not occur within a symbol.

A double dash (`--`) indicates the start of a comment. The comment continues to the end of the line on which the dashes appear. Comments are treated as whitespace by the reader.

```
-- The "fact" bindings compute the factorial of
-- a non-negative integer.

{ :fact 0 -> 1
; :fact n -> n * (‘fact (n - 1))
}
```

4.3 Other Notations

The following symbols are reserved by the formal syntax.

`( )` Parentheses are used purely for grouping, and are optional wherever grouping is implied by the lexical context. In particular, parentheses are not required around the outer-most level of a term.

`{ }` Braces denote the boundaries of an evaluation context.

`->` The arrow is used to separate a binding's key from its body. In source listing, the arrow is rendered as "`->`".

`;` The semicolon separates individual elements of an extension. See Section 5.4 for details.

`.` The dot serves two distinct but related roles. In a function definition, the dot separates the bound variable from the function body. Elsewhere, the dot is the staging annotation.

`‘` The backtick delays evaluation of a term. See Section 5.4 for details.

`\` The backslash denotes the start of a function.

5. Basic Concepts

5.1 Terms

Symbol are the simplest form, in that they are not compositions of other terms. A symbol that names a term is called a *variable* and is said to be *bound* to that term. The most fundamental of the variable binding constructs is the *binding*, as detailed in Section 5.3. Function application reduces directly to uncontained binding application.

Application, denoted by juxtaposition, associates terms in a left-associative manner. Extension, denoted by semicolon, associates terms in a right-associative manner. Extended terms are equivalent to the lists of Scheme. Generally speaking, compound terms evaluate from left to right.

Unlike most other languages, Grasp has no fixed notion of value. In Grasp, a *value* is any term that does not step. This arrangement empowers Grasp with unprecedented flexibility to combine programs—evaluation occurs around unfamiliar notation, which is left intact.

5.2 Patterns

Grasp patterns operate similarly to the patterns of functional languages like ML [12] and Haskell [8], except that pattern matching produces a labeled summary of the match attempt. Stages double as patterns when applied to other stages. The *pattern match*

$$.p\ .t$$

performs a structural comparison of the term p against the term t. If p does not match t, we get the symbol $mismatch$. Otherwise, we get the symbol $match$ applied to a context describing the match. For example,

$$.(x\ y)\ .(a\ b) \Downarrow match\ \{:x \to a; :y \to b\}$$

$$.:x\ .a \Downarrow mismatch$$

These two examples illustrate a special kind of pattern, the *literal* pattern. A literal pattern matches only the symbol following the colon. Another kind of special pattern is the *skip* pattern, denoted by "_". A skip pattern matches and ignores any structure.

5.3 Bindings

In conventional parlance, a binding maps a name to a piece of data. Our definition is strictly more general. The *binding*

$$b = p_b \to t_b$$

maps pattern p_b to term t_b.

Bindings are the primary vehicle for term rewriting. Uncontained bindings behave like macros—rewriting occurs before evaluation. When contained within a context, bindings behave like functions. See Section 5.4 for details.

If we apply b to a term t such that $.p_b\ .t \Downarrow match\ c$, then $b\ t \longrightarrow c\ t_b$. If p_b is a literal pattern, then b behaves like a conventional variable binding. If p_b is a symbol, then $b\ t \longrightarrow (:p_b \to t)\ t_b$, unless p_b is the skip symbol, in which case $b\ t \longrightarrow t_b$.

5.4 Contexts and Stages

Contexts are the fundamental construct for containment. The *context*

$$c = \{t\}$$

contains term t. Although we also use parentheses to delineate compound terms, contexts behave more like cell membranes in that they can pass information through their boundary. Contexts are, however, not entirely like cell membranes in that contexts allow unfamiliar information to pass through by default.

If t contains either a binding or an extension of bindings, then the bindings of t encode the evaluation strategy of c. We say that the bindings of t are *contained* by c or, conversely, that c contains the bindings of t.

Whenever we apply c to some stage $.t_1$, we force t_1 to evaluate. When t_1 can no longer evaluate on its own, any bindings contained in c are recursively matched against t_1. If c contains a matching binding, t_1 is transformed and then continues evaluation on its own. The evaluate-match-transform loop continues until t_1 no longer evaluates on its own or matches any binding contained in c, at which point we get back t_1 with single level of quotation stripped off. This arrangement allows us to think of contained bindings as functions and uncontained bindings as macros, because the presence or absence of context has a direct effect on the order in which evaluation and transformation occur.

As a final note, the current implementation applies contained bindings in order from left to right, although no two bindings within a given context may contain structurally identical keys.

6. Examples

In this section, we present a few small examples to highlight Grasp's flexibility. All of these examples run in the current interpreter as presented.

6.1 The Interpreter

Grasp is implemented as a package in Racket 5.2.1. To install the interpreter, unpack the `grasp-cs239.tar.gz` distribution tarball and point the `rack link` command at it:

```
raco link grasp-cs239/src/cs23
```

To run the interpreter, start racket and load the library:

```
cd grasp-cs239
racket
```

```
> (enter! cs239/untyped)
> (repl)
>>
```

The single-angle bracket is Racket's prompt, and the double-angle bracket is Grasp's prompt. To quit the Grasp interpreter, send an end-of-file (^D) character. The interpreter keeps an implicit context, and commands typed at the top level are evaluated as stages to which the implicit context is applied. The interpreter also supports the following top-level special commands:

$\#top$ displays the implicit context

$\#load$ replaces the implicit context with a file in the current working directory, e.g. $\#load\ Lists$ reads and evaluates the file `./Lists.grasp` and replaces the implicit context with the result.

$\#import$ composes the implicit context with a file in the current working directory, e.g. $\#import\ Lists$ reads and evaluates the file `./Lists.grasp` and extends the implicit context with the result.

$\#update$ composes the implicit context with the result of evaluation a given term, e.g. $\#update\ \{:foo \rightarrow 17\}$ extends the implicit context with a binding that maps symbol foo to symbol 17.

$\#trace$ toggles displaying of the individual steps in an evaluation

$\#exec$ replaces special symbols in the most recent result and re-evaluates it. Right now, the only special symbols are for arithmetic: $plus$, $minus$, $times$, $divide$, $less-than?$, and $equals?$. Beware: exec is currently very buggy.

6.2 Architecture-dependent Evaluation

Suppose we acquire a large and complex body of code that contains many fragments like this:

```
1 + (if x86 then (2 + 3) else (2 - 3))
```

We can store and recall this fragment into the interpreter.

```
>> #update {:f -> 1 + (if x86 then
                         (2 + 3) else (2 - 3))}
>> f
-- 1 + (if x86 then (2 + 3) else (2 - 3))
```

We can also store the fact that this computer is an x86.

```
>> #update {:x86 -> true}
>> x86
-- true
```

When we re-evaluate the program, this new fact is reflected.

```
>> f
-- 1 + (if true then (2 + 3) else (2 - 3))
```

To evaluate this term any further, we must implement the if-expression. First, we implement a generic symbol-based test-expression.

```
>> #update {:test -> (:true -> \x.\y.{} x;
                      :false -> \x.\y.{} y)}
```

This maps the symbol $test$ to a set of macros that rewrite their argument only once and then disappear. For example,

```
>> test true
-- \x.\y.{} x
```

When the test succeeds, we get back a function that consumes two arguments and does something the first. Test failure is similar. The empty context serves as the "bottom" value. Because the empty context does no rewriting on its own, it is useful for forcing stages. This means that the second and third arguments to the test-expression must be staged.

To tie this binding into the original program, we need to translate the if-expression syntax to test-expression syntax.

```
>> #update {:if x :then y :else z -> test x .y .z}
>> if true then 1 else 2
-- 1
>> if false then 1 else 2
-- 2
```

Now, the original program can make more progress.

```
>> f
-- 1 + (2 + 3)
```

The value f maps to has not actually changed.

```
>> #top
-- {:f -> 1 + (if x86 then (2 + 3) else (2 ...

>> #trace
>> f
f
1 + (if x86 then (2 + 3) else (2 - 3))
1 + (if true then (2 + 3) else (2 - 3))
1 + (test true .(2 + 3) .(2 - 3))
1 + ((:true -> \x.\y.{} x;:false -> ...
1 + ((\x.\y.{} x) .(2 + 3) .(2 - 3))
1 + ({:x -> .(2 + 3)} .(\y.{} x) .(2 - 3))
1 + ({:x -> .(2 + 3)} .(\y.{} .(2 + 3)) ...
1 + ((\y.{} .(2 + 3)) .(2 - 3))
1 + ({:y -> .(2 - 3)} .({} .(2 + 3)))
1 + ({:y -> .(2 - 3)} .(2 + 3))
1 + (2 + 3)
-- 1 + (2 + 3)
```

If we expect to run this program often, we can store the partially evaluated result.

```
>> #update {:fx86test -> 1 + (2 + 3)}
```

In order to evaluate this term any further, we need an implementation of arithmetic. This interpreter provides built-in prefix notation for addition ($plus$) and subtraction ($minus$), but we must explicitly fix the built-in symbols. First, let's add rules for converting from infix to prefix notation.

```
>> #update {x :+ y -> plus x y; x :- y -> minus x y}
```

Now, we can finish evaluating the original program. First, we recall the partially evaluated result.

```
>> fx86test
fx86test
1 + (2 + 3)
plus 1 (2 + 3)
plus 1 (plus 2 3)
-- plus 1 (plus 2 3)
```

Then, we `fix` the built-in symbols and continue evaluation.

```
>> #exec
#plus 1 (#plus 2 3)
#(plus 1) (#plus 2 3)
#(plus 1) (#(plus 2) 3)
#(plus 1) 5
6
-- 6
```

If our underlying system architecture changes, we should re-evaluate the original program.

```
>> #update {:x86 -> false}
>> f
f
1 + (if x86 then (2 + 3) else (2 - 3))
1 + (if false then (2 + 3) else (2 - 3))
1 + (test false .(2 + 3) .(2 - 3))
1 + ((:true -> \x.\y.{} x;:false -> ...
1 + ((\x.\y.{} y) .(2 + 3) .(2 - 3))
1 + ({:x -> .(2 + 3)} .(\y.{} y) .(2 - 3))
1 + ((\y.{} y) .(2 - 3))
1 + ({:y -> .(2 - 3)} .({} y))
1 + ({:y -> .(2 - 3)} .({} .(2 - 3)))
1 + ({:y -> .(2 - 3)} .(2 - 3))
1 + (2 - 3)
plus 1 (2 - 3)
plus 1 (minus 2 3)
-- plus 1 (minus 2 3)

>> #exec
#plus 1 (#minus 2 3)
#(plus 1) (#minus 2 3)
#(plus 1) (#(minus 2) 3)
#(plus 1) -1
0
-- 0
```

Notice that we saved several steps by keeping track of the pre-exec result. This concludes a simple demonstration of bindings, functions, and partial evaluation.

6.3 Propositional Logic

We also include simple programs for performing Boolean, list, and simple numeric operations. Note that the Numbers program is buggy because we have not yet worked out the kinks in $\#exec$. We also provide a program for converting logic statements into conjunctive normal form.

```
-- Determines the conjunctive normal form (CNF)
-- of a formula.
--
-- Formula syntax:
--   ~  negation
--   &  conjunction
--   |  disjunction
--   >> implication
--   <> equivalence
--
-- XXX: variable capture! ~ (B | C)

-- double negagive
{ :~ (:~ A) -> A
-- DeMorgan's laws
; :~ (A :& B) -> (~ A) | (~ B)
; :~ (A :| B) -> (~ A) & (~ B)
-- associativity
; A :& (B :& C) -> A & B & C
; A :| (B :| C) -> A | B | C
-- distributivity
; (A :& B) :| C -> (A | C) & (B | C)
; A :| (B :& C) -> (A | B) & (A | C)
-- implication
; A :>> B -> (~ A) | B
-- equivalence
; A :<> B -> (A >> B) & (B >> A)
}

-- literals
{ :literal? (_ :| _) -> false
; :literal? (_ :& _) -> false
; :literal? (:~ A) -> literal? A
; :literal? _ -> true
}
```

We first load the program

```
>> #load CNF
```

Now we can program in propositional logic.

```
>> a >> b
-- ~ a | b
>> a <> b
-- ~ a | b & (~ b | a)
>> a >> (b & c)
-- ~ a | b & (~ a | c)
>> a & (~ (a | b))
-- a & (~ a) & (~ b)
>> a | (~ (b & (~ c)))
-- a | (~ b) | c
>>
```

We can also evaluate logical statement.

```
>> #update {:a -> true; :b -> false}
>> a >> (b & c)
-- false
>> a <> (b & c)
-- false
>> a & (~ b)
-- true
>>
```

7. Grasp is a Platform

Grasp's vast problem space, broad vision, and early development history are analogous to those of Smalltalk-72 [9], differing primarily in terms of background and setting. Smalltalk implements a vision to, among other things, make

computer programming accessible to children, while Grasp implements the hopes and dreams of a programmer accumulated since childhood[1]. In this sense, Grasp quite literally aims to be a next-generation Smalltalk. Ultimately, we seek to expand the social dimension of computing to degrees not possible before widespread success of the Internet and social networking. More specifically, Grasp as a platform aims to replace basic collaborative functionality like the Web, email, and instant messaging, and includes plans for rich data sources like smart phones and tablets. One major goal is to fuse social and distributed computing in new ways to allow safe and responsible sharing of computing resources.

8. Future Work

The implementation is not yet robust enough for real work. Among other things, implementation-dependent $\#exec$ functionality is quite buggy, and patterns are limited to symbols not reserved by the core semantics, e.g. braces or equals signs can not be matched. We also need a more complete standard library with less tricky number semantics.

The design is also still brittle, especially with respect to contexts and stages—we are still working these details out. Perhaps a formal theory will lead to this. Most disconcerting are the various way to get in trouble with variable capture. For example, the following divergent propositional logic term has no obvious fix:

$$(\lambda x.x\ y)\ x$$

These are all short term goals. In the longer term, we plan to design and implement streams and networking. We also plan to work on a fully intuitionistic pluggable type system to give Grasp a pluggable static dynamic type system.

9. Conclusion

In this report, we have presented pluggable programming—a novel approach to software construction and evaluation that revolutionizes how we think about large programs with loosely-connected components. We demonstrated the utility of this approach with the Grasp programming language, and outlined a broad vision in which pluggable programming can enable people to create and share functionality as well as computing resources responsibly. Much work remains, but the fun has just begun.

[1] My obsession with programming languages began at age 9, when I discovered a GW-BASIC reference manual.

A. Formal Semantics

A.1 Core

$$
\begin{array}{rll}
t ::= & x & \text{symbol} \\
| & t\ t & \text{application} \\
| & t; t & \text{extension} \\
| & p \to t & \text{binding} \\
| & \{t\} & \text{context} \\
| & .t & \text{stage} \\
| & `t & \text{quote} \\
| & \lambda x.t & \text{function} \\
p ::= & \_ & \text{skip pattern} \\
| & `:x & \text{literal pattern} \\
| & x & \text{symbol pattern} \\
| & p\ p & \text{application pattern} \\
| & p; p & \text{extension pattern} \\
| & p \to p & \text{binding pattern} \\
| & \{p\} & \text{context pattern} \\
| & .p & \text{stage pattern} \\
| & `p & \text{quote pattern} \\
| & \lambda p.p & \text{function pattern}
\end{array}
$$

$\boxed{t \longrightarrow t}$

$$\text{(E-App1)}\ \frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}$$

$$\text{(E-App2)}\ \frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'}$$

$$\text{(E-Ext1)}\ \frac{t_1 \longrightarrow t_1'}{t_1; t_2 \longrightarrow t_1'; t_2}$$

$$\text{(E-Ext2)}\ \frac{t_2 \longrightarrow t_2'}{v_1; t_2 \longrightarrow v_1; t_2'}$$

$$\text{(E-FunApp)}\ \frac{}{(\lambda x_{11}.t_{12})\ v_2 \longrightarrow \{`:x_{11} \to v_2\}\ .t_{12}}$$

A.2 Binding

$$\text{(E-Bind)}\ \frac{\mu\ p_{k1}\ t_{k2}\ t_2 = t_2'}{(v_{i\langle k}; p_{k1} \to t_{k2}; v_{i\rangle k})\ t_2 \longrightarrow t_2'}$$

A.2.1 Pattern

$\boxed{t \longrightarrow t}$

$$\text{(E-SkipPat)}\ \frac{}{\_.t_2 \longrightarrow match\ \{\}}$$

$$\text{(E-LitPat)}\ \frac{x_1 = x_2}{.:x_1\ .x_2 \longrightarrow match\ \{\}}$$

$$\text{(E-SymPat)}\ \frac{}{.x_1\ .t_2 \longrightarrow match\ \{:x_1 \to t_2\}}$$

$$\text{(E-AppPat)}\ \frac{.p_1\ .t_3 \Downarrow match\ t_3' \qquad .p_2\ .t_4 \Downarrow match\ t_4'}{.(p_1\ p_2)\ .(t_3\ t_4) \longrightarrow match\ (t_3'\ t_4')}$$

$$\text{(E-ExtPat)}\ \frac{}{.(p_1; p_2)\ .(t_3; t_4) \longrightarrow .(p_1\ p_2)\ .(t_3\ t_4)}$$

$$\text{(E-BindPat)}\ \frac{}{.(p_1 \to p_2)\ .(p_3 \to t_4) \longrightarrow .(p_1\ p_2)\ .(p_3\ t_4)}$$

$$\text{(E-CtxPat1)}\ \frac{}{.\{p_1\}\ .\{t_2\} \longrightarrow .p_1\ .t_2}$$

$$\text{(E-CtxPat0)}\ \frac{}{.\{\}\ .\{\} \longrightarrow match\ \{\}}$$

$$\text{(E-StagePat)}\ \frac{}{..p_1\ ..t_2 \longrightarrow .p_1\ .t_2}$$

$$\text{(E-QuotePat)}\ \frac{}{.`p_1\ .`t_2 \longrightarrow .p_1\ .t_2}$$

$$\text{(E-FunPat)}\ \frac{}{.(\lambda p_1.p_2)\ .(\lambda x_3.t_4) \longrightarrow .(p_1\ p_2)\ .(x_3\ t_4)}$$

A.3 Context

$$\text{(E-Ctx)}\ \frac{t_i \longrightarrow t_i'}{\{t_i\} \longrightarrow \{t_i'\}}$$

$$\text{(E-ComMacro)}\ \frac{p_2 = p_{k1}}{\{v_{i\langle k}; p_{k1} \to t_{k2}; v_{i\rangle k}\}\ \{p_2 \to t_3\} \longrightarrow \{v_{i\langle k}; p_{k1} \to t_3; v_{i\rangle k}\}}$$

$$\text{(E-Com3)}\ \frac{}{\{v_i\}\ \{v_j\} \longrightarrow \{v_i\}\ \{v_{j=1}\}\ \{v_{j\rangle 1}\}}$$

$$\text{(E-Com2)}\ \frac{}{\{v_{i\le n}\}\ \{v_{n+1}\} \longrightarrow \{v_{i\le n+1}\}}$$

A.4 Closure

$$\boxed{t \longrightarrow t}$$

$$\text{(E-Step)}\ \frac{t_2 \longrightarrow t_2'}{\{v_i\}\ .t_2 \longrightarrow \{v_i\}\ .t_2'}$$

$$\text{(E-Loop)}\ \frac{v_i\ v_2 \longrightarrow v_2'}{\{v_i\}\ .v_2 \longrightarrow \{v_i\}\ .v_2'}$$

$$\text{(E-Ret)}\ \frac{v_i\ v_2 \not\longrightarrow}{\{v_i\}\ .v_2 \longrightarrow \rho\ v_2}$$

A.5 Substitution

Define $\mathrm{FV}(t)$, the set of *free variables* in term t, as follows:

$$\begin{array}{lcl}
\mathrm{FV}(x_1) & = & \{x_1\} \\
\mathrm{FV}(t_1\ t_2) & = & \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2) \\
\mathrm{FV}(t_1; t_2) & = & \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2) \\
\mathrm{FV}(t_1 \to t_2) & = & \mathrm{FV}(t_2) \setminus \mathrm{FV}(t_1) \\
\mathrm{FV}(\{t_1\}) & = & \mathrm{FV}(t_1) \\
\mathrm{FV}(\{\}) & = & \varnothing \\
\mathrm{FV}(.t_1) & = & \mathrm{FV}(t_1) \\
\mathrm{FV}(`t_1) & = & \varnothing \\
\mathrm{FV}(\lambda x_1.t_2) & = & \mathrm{FV}(t_2) \setminus \{x_1\}
\end{array}$$

Define $\mathrm{literals}(p)$, the *literals* of pattern p, as follows:

$$\begin{array}{lcl}
\mathrm{literals}(:x) & = & \{x\} \\
\mathrm{literals}(x) & = & \varnothing \\
\mathrm{literals}(p_1\ p_2) & = & \mathrm{literals}(p_1) \cup \mathrm{literals}(p_2) \\
\mathrm{literals}(p_1; p_2) & = & \mathrm{literals}(p_1) \cup \mathrm{literals}(p_2) \\
\mathrm{literals}(p_1 \to p_2) & = & \mathrm{literals}(p_1) \cup \mathrm{literals}(p_2) \\
\mathrm{literals}(\{\}) & = & \varnothing \\
\mathrm{literals}(\{p_1\}) & = & \mathrm{literals}(p_1) \\
\mathrm{literals}(.p_1) & = & \mathrm{literals}(p_1) \\
\mathrm{literals}(`t) & = & \varnothing \\
\mathrm{literals}(\lambda p_1.p_2) & = & \mathrm{literals}(p_1) \cup \mathrm{literals}(p_2)
\end{array}$$

Binding *expansion* is a partial function. The result is undefined if any recursive result is undefined.

$$\begin{array}{lcl}
\mu\ p_1\ t_2\ t_3 & = & \sigma\ t_3'\ t_2 \quad \text{if } .p_1\ .t_3 \Downarrow match\ t_3' \\
\mu\ p_1\ t_2\ (t_{31}\ t_{32}) & = & (\mu\ p_1\ t_2\ t_{31})\ (\mu\ p_1\ t_2\ t_{32}) \\
\mu\ p_1\ t_2\ (t_{31}; t_{32}) & = & (\mu\ p_1\ t_2\ t_{31}); (\mu\ p_1\ t_2\ t_{32}) \\
\mu\ p_1\ t_2\ (t_{31} \to t_{32}) & = & t_{31} \to (\mu\ p_1\ t_2\ t_{32}) \\
\mu\ \{\}\ t_2\ \{\} & = & t_2 \\
\mu\ p_1\ t_2\ \{t_{31}\} & = & \{\mu\ p_1\ t_2\ t_{31}\} \\
\mu\ p_1\ t_2\ .t_{31} & = & .(\mu\ p_1\ t_2\ t_{31}) \\
\mu\ p_1\ t_2\ (\lambda x_{31}.t_{32}) & = & \\
\multicolumn{3}{r}{\begin{cases} (\mu\ p_1\ t_2\ (\lambda x_\alpha.(\mu\ :x_{31}\ x_\alpha\ t_{32})) \\ \qquad \text{if } x_{31} \in \mathrm{literals}(p_1) \cup \mathrm{FV}(t_2) \\ \lambda x_{31}.(\mu\ p_1\ t_2\ t_{32}) \\ \qquad \text{if } x_{31} \notin \mathrm{literals}(p_1) \cup \mathrm{FV}(t_2) \end{cases}}
\end{array}$$

Define $\sigma\ \{:x \to v\}_i\ t$, the *substitution* of symbols $\{x\}_i$ by values $\{v\}_i$ in term t, as follows:

$$\begin{array}{l}
\sigma\ \{:x_i \to v_i\}_i\ x_2 = \begin{cases} x_2 & \text{if } x_2 \notin \{x_{i1}\}_i \\ v_{k2} & \text{if } x_2 = x_{k1} \end{cases} \\
\sigma\ \{:x \to t\}_i\ (t_2\ t_3) = (\sigma\ \{:x \to t\}_i\ t_2)\ (\sigma\ \{:x \to t\}_i\ t_3) \\
\sigma\ \{:x \to t\}_i\ (t_2; t_3) = (\sigma\ \{:x \to t\}_i\ t_2); (\sigma\ \{:x \to t\}_i\ t_3) \\
\sigma\ \{:x_i \to t_i\}_i\ (p_2 \to t_3) = \\
\quad p_2 \to (\sigma\ \{:x_{k1} \to t_{k2} \mid x_{k1} \neq p_2\}\ t_3) \\
\sigma\ \{:x_i \to t_i\}_i\ \{\} = \{\} \\
\sigma\ \{:x_i \to t_i\}_i\ \{t_j\}_j = \{\sigma\ \{:x_i \to t_i\}_i\ (t_j)_j\} \\
\sigma\ \{:x \to t\}_i\ .t_2 = .(\sigma\ \{:x \to t\}_i\ t_2) \\
\sigma\ \{:x \to t\}_i\ `t_2 = `t_2 \\
\sigma\ \{:x \to t\}_i\ (\lambda x_2.t_3) = \\
\quad \begin{cases} \sigma\ \{:x \to t\}_i\ (\lambda x_\alpha.\sigma\ \{:x_2 \to x_\alpha\}\ t_3) \\ \qquad \text{if } x_2 \in \{x_i\}_i \cup \mathrm{FV}(t_i)_i \\ \lambda x_2.(\sigma\ \{:x \to t\}_i\ t_3) \\ \qquad \text{if } x_2 \notin \mathrm{FV}(t_i)_i \end{cases}
\end{array}$$

Define $\rho\ t$, the *return* of term t, as follows:

$$
\begin{array}{lcl}
\rho\, x_1 & = & x_1 \\
\rho\, (t_1\ t_2) & = & (\rho\, t_1)\ (\rho\, t_2) \\
\rho\, (t_1; t_2) & = & (\rho\, t_1); (\rho\, t_2) \\
\rho\, (p_1 \to t_2) & = & p_1 \to t_2 \\
\rho\, \{t_i\} & = & \{t_i\} \\
\rho\, \{\} & = & \{\} \\
\rho\, .t_1 & = & .(\rho\, t_1) \\
\rho\, `t_1 & = & t_1 \\
\rho\, (\lambda x_1 . t_2) & = & \lambda x_1 . \rho\, t_2
\end{array}
$$

References

[1] G. Bracha. Pluggable type systems. In *OOPSLA workshop on revival of dynamic languages*. Citeseer, 2004.

[2] G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules as objects in newspeak. *ECOOP 2010–Object-Oriented Programming*, pages 405–428, 2010.

[3] A. Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[4] N. Dershowitz and J.-P. Jouannaud. *Rewrite systems*. Citeseer, 1989.

[5] E. W. Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[6] S. E. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: type-safe, generative, binding macros in macroml. In *ACM SIGPLAN Notices*, volume 36, pages 74–85. ACM, 2001.

[7] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[8] S. P. Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[9] A. C. Kay. The early history of smalltalk. In *History of programming languages—II*, pages 511–598. ACM, 1996.

[10] R. Kelsey, W. Clinger, J. Rees, H. Abelson, R. Dybvig, C. Haynes, G. Rozas, D. Bartley, R. Halstead, D. Oxley, et al. Revised report on the algorithmic language scheme. 1998.

[11] J. W. Klop, J. Klop, and M. Centrum. *Term rewriting systems*. Centre for Mathematics and Computer Science, 1990.

[12] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of Standard ML*. MIT press, 1997.

[13] P. Stephens. The imp language and compiler. *The Computer Journal*, 17(3):216–223, 1974.

[14] W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theoretical computer science*, 248 (1):211–242, 2000.

[15] D. Ungar and R. B. Smith. *Self: The power of simplicity*, volume 22. ACM, 1987.

[16] L. Wall. Perl, the first postmodern programming language. Linux World Conference, San Jose CA, 1999.

[17] A. Warth and I. Piumarta. Ometa: an object-oriented language for pattern matching. In *Proceedings of the 2007 symposium on Dynamic languages*, pages 11–19. ACM, 2007.