Theory and Implementation of a Functional Programming Language

Ari Lamstein *

October 2000

Abstract

The goal of this research is to design and implement a small functional programming language that incorporates some of the features that arise from the theoretical study of programming language semantics. We begin with the study of the λ-calculus, an idealized mathematical language. We present the language PPL, a strongly typed, call-by-name language which supports recursion and polymorphism. We describe a compiler of PPL into a low-level assembly language. The compilation is based on an abstract machine interpretation and includes a type inference algorithm.

1 Introduction

The foundations of functional programming languages lie in the untyped λ-calculus, which was developed by Church in 1932. It was developed with two mathematical goals in mind [1]:

1. To develop a general theory of functions.
2. To use that theory as a foundation for mathematics.

While the attempts at (2) were not successful, the λ-calculus developed as a rich theory of computation [1]. It is well-known that Turing's notion of computability is equivalent to λ-definability; that is, the class of computable functions can be encoded in the λ-calculus [4]. As we will see, it is this fact coupled with its simple semantics that leads us to use the λ-calculus as a model for functional programming languages.

The principal difference between functional programming languages and their imperative counterparts (such as C or Java) is how programs written in them are viewed. Programs in imperative languages are viewed as a sequence of statements that operate by altering the content of the memory. Programs in functional languages, however, are viewed as expressions; their evaluation corresponds to the program's execution.

*This research was conducted under the supervision of Peter Selinger at the University of Michigan. The author can be contacted via email at ari@lamstein.com.

An effect of this is the strength of variable bindings. In functional languages such bindings are permanent, as they are in common mathematical notation. To say "let x = 5;" in a functional language such as ML means that one is forbidden to later assign another value to that variable. In an imperative language, however, variable bindings are considered "mutable," as updating memory is a statement, and not an expression.

In a purely functional language an expression yields the same result each time it is evaluated. Statements in an imperative language, however, can behave differently at different times they are executed. This is because the statements can have "side effects", such as reading and writing to memory.

Thus, constructs such as for and while loops are seldom used in functional languages, as the *expressions* in the loop body have the same result each time they are executed. Such loops are only useful in the presence of side effects, where one can alter a physical property of the computer, such as the screen or an address of memory.

By minimizing side effects, one uses a higher-level approach when writing algorithms. Rather than concerning oneself with how the computation is carried out, we instead consider a description of the result being computed. For example, consider writing the factorial function in ML:

```
let fac x = if (x=0) then 1 else (x * (fac (x-1)));
```

Here we just state the definition of the function. Now consider implementing the same function in an imperative style in C:

```
int fac (int x) {
  int result=1;

  while (x>0) {
    result = result*x;
    x = x - 1;
  }

return result;
}
```

Here we concern ourselves with how the actual computation is carried out: we allocate memory for a new variable, repeatedly change its value, and then return that final value. In ML we leave the process of allocating and updating memory to the compiler.

The primary benefit of imperative languages is speed. By directly manipulating memory and relying less on recursion, programs in imperative languages are often able to perform the same task in less time. But as computers become faster, this advantage is meaningful for a smaller number of programs. Also, compiler optimizations for functional languages sometimes produce code which is comparable to that obtained from imperative languages.

The primary benefits of functional languages are the clarity and small size of code needed to write an algorithm. Clarity is achieved by freeing the programmer from the low-level implementation of computation. One is rarely faced with the arduous task that is common when reading and writing code which is written in imperative languages: namely, keeping track of both the value and meaning of several variables.

Functional languages place no more restrictions on functions than on variables. One can pass functions to other functions, store them in data structures, or return them as the result of a function. One can even pass functions with an arbitrary number of their parameters instantiated.

The notion of passing a function with an arbitrary number of its parameters instantiated is related to *curried notation*. As an example, consider the function `plus`, which one normally thinks of as taking a pair and returning a value; i.e. a function with the signature

$$\texttt{plus:}\ \ (\texttt{int * int}) \to \texttt{int}.$$

However, one can also think of addition as the application of two functions, each of which takes one argument. This yields the signature

$$\texttt{plus:}\ \ \texttt{int} \to (\texttt{int} \to \texttt{int}).$$

Here, for example, the result of providing 3 to `plus` is a function which takes an integer and adds 3 to it. Thus, $(\texttt{plus}\ 3)5 = 8$ and $(\texttt{plus}\ 10)5 = 15$. This transformation of a function which takes n arguments to the application of n functions, each of which takes one argument, is called *currying*. It is even possible to apply `plus` to just one argument, and pass the result (which is a function) as an argument to another function. This property allows one to create very robust libraries. For example, consider the library function `map` in ML. This function has the type signature

$$\texttt{map}: (\alpha \to \beta) \to (\alpha\ \texttt{list}) \to (\beta\ \texttt{list}).$$

It takes two arguments, a function f from α to β, and a list of elements of type α. It returns a list of elements of type β, which is obtained by applying f to each element of the list. Here we use α and β to represent the fact that the map function is polymorphic, allowing one to substitute arbitrary types for α and β.

The above properties make many tasks, such as those involving list manipulation, relatively easy. As an example, suppose one wanted to increase the value of each element in a list of integers `intlist` by 3. One could do this by simply writing

```
let list' = List.map (plus 3) intlist,
```

where plus is the curried addition function from above. This example is representative of how algorithms in functional languages are generally written: by the application of one or more functions to a data structure. This is a level of modularization which is not possible without higher order functions.

2 The Untyped λ-calculus

2.1 The Language

We now introduce the fundamental definitions of the λ-calculus. We refer the reader to [1, 8, 3] for a more comprehensive treatment.

Let $\mathcal{V}$ be a countably infinite set of variables (written $x, y, z, \ldots$). The class of λ-terms consists of words constructed from the following alphabet:

- *variables:* $x, y, z \ldots$
- *lambda abstractor:* λ
- *parentheses:* $($, $)$
- *period:* .

Definition. The class Λ of λ-terms is the least class satisfying the following:

1. if x is a variable, then $x \in \Lambda$,
2. if $M, N \in \Lambda$, then $(MN) \in \Lambda$,
3. if $M \in \Lambda$, then $(\lambda x.M) \in \Lambda$.

Terms of the form (MN) represent the application of the function M to the argument N. Terms of the form $(\lambda x.M)$ are called λ-*abstractions*, and denote the function that maps x to M. From the above definition, we can see that the following are λ-terms.

$$\begin{array}{l} x \\ (xx) \\ (\lambda y.(xx)) \\ ((\lambda y.(xx))(\lambda x.(xx))) \end{array}$$

We will drop unnecessary parentheses from λ-terms by using the following conventions. Application associates to the left, and so MNP stands for $((MN)P)$. We write $\lambda x_1 x_2 \ldots x_n.M$ for $\lambda x_1.(\lambda x_2 \ldots (\lambda x_n.M) \ldots)$

Application binds stronger than λ-abstraction, and so $\lambda x.MN$ means $\lambda x.(MN)$ and not $(\lambda x.M)N$. Thus, the right-hand side of a λ-abstraction extends as far to the right as possible.

The operations which we will perform on λ-terms depend on the distinction between free and bound variables. Informally, in the term $\lambda x.M$ the variable x is said to be *bound*, and the subterm M is called the *scope* of the binding. A variable is called *free* if it is not bound.

Definition. The set of free variables of a λ-term is defined as follows:

$$\begin{array}{lcl} FV(x) & = & x \\ FV(\lambda x.M) & = & FV(M) \setminus \{x\} \\ FV(MN) & = & FV(M) \cup FV(N) \end{array}$$

Note that a variable may appear both free and bound in a term, such as x in $(\lambda x.x)x$.

One can think of bound variables as local variables. The name of a bound variable is not significant, as both $\lambda x.x$ and $\lambda y.y$ denote the same function. We will henceforth identify λ-terms which differ only by a renaming of bound variables. Such terms are called α-equivalent.

Definition. The terms $M, N \in \Lambda$ are said to be α-equivalent, written $M =_\alpha N$, if M and N are equal except possibly for the renaming of bound variables.

Thus, $\lambda x.xyz =_\alpha \lambda w.wyz$ but $\lambda x.xyz \neq_\alpha \lambda x.xwz$ as y is not bound in $\lambda x.xyz$. Another example is $\lambda x.xyz \neq_\alpha \lambda y.yyz$. Here the second term has a bound variable (the last y) which occurs free in the first term!

2.2 Substitution

Our primary interest with λ-terms will be computation, which we view as the evaluation of expressions. The first notion of evaluation we consider is that of β-reduction.

The term $\lambda x.M$ is an expression for the function which maps x to the term M. Here x may occur in M. In the context of programming languages, x is called the *parameter* of M. If the function $\lambda x.M$ is applied to the value N, then the result is M, with all occurrences of x replaced by N.

Suppose that f is the addition operation, written in prefix form, so that $f\,2\,3$ evaluates to 5. Then $\lambda x.fxx$ is the function which maps x to $x + x$, thus the "times two" function. If we evaluate $(\lambda x.fxx)3$, we obtain $f\,3\,3$, or 6.

We write $M[x := N]$ for the result of replacing the free variable x by N in the term M. This is the *substitution* of N for x in M, and is formally defined below.

Definition. The result of substituting N for the free occurrences of x in M, written $M[x := N]$, is defined as follows.

1. $x[x := N] = N$
2. $y[x := N] = y, \quad$ if $x \neq y$
3. $(\lambda y.M_1)[x := N] = \lambda y.(M_1[x := N]), \quad$ provided $x \neq y$ and $y \notin FV(N)$.
4. $(M_1M_2)[x := N] = (M_1[x := N])(M_2[x := N])$

The reason for the provision in (3) is that if we were to make the following substitution blindly:

$$\lambda x.xyz[y := x] = \lambda x.xxz$$

we would "gain" a bound variable. But this goes against the spirit of substitution, as the name of a bound variable should not affect the result of the substitution.

We solve this problem by introducing a new step in the substitution process. Because a name clash arises when we rename y as a bound variable, we first rename x as a variable which does not occur in $\lambda x.xyz$. This new term is α-equivalent to the original one. Thus, the above substitution can be carried out as follows:

$$\lambda x.xyz[y := x] =_\alpha \lambda k.kyz[y := x] = \lambda k.kxz$$

2.3 β-Reduction

We define β-reduction with the idea of substitution and function application in mind.

We call a term of the form $(\lambda x.N)P$ a β-*redex* and say that it *reduces* to $N[x := P]$. We say that a term $M \in \Lambda$ β-reduces to a term M' in one step, written $M \to_\beta M'$, if M' is obtained from M by reducing a single redex that is a subterm of M. Formally, $\to_\beta$ is the smallest relation on terms such that

$$(\lambda x.N)P \to_\beta N[x := P] \qquad \frac{M \to_\beta M'}{MN \to_\beta M'N}$$

$$\frac{N \to_\beta N'}{MN \to_\beta MN'} \qquad \frac{M \to_\beta M'}{\lambda x.M \to_\beta \lambda x.M'}$$

Definition. If $M \in \Lambda$ has no subterms of the form $(\lambda x.N)P$, then M is said to be in *β-normal form.*

Note that a normal form cannot be β-reduced any further.

We write $\to_\beta^*$ for the reflexive transitive closure of $\to_\beta$. Furthermore, we say that if $M \to_\beta^* M'$ and M' is in normal form, then M' is the *result* of β-reducing (or evaluating) M.

Recall our premise that the λ-calculus can be viewed as a prototype for functional programming languages. We now give examples of β-reduction which support this premise.

Example 2.1 (Identity). The term $\lambda x.x$ is the identity function.

Proof. This claim is only true if $(\lambda x.x)M \to_\beta^* M$ for all $M \in \Lambda$.

$$(\lambda x.x)M \to_\beta x[x := M] = M \quad \square$$

Example 2.2 (Boolean Values, Conditional). The λ-terms $\lambda x.(\lambda y.x)$ and $\lambda x.(\lambda y.y)$ correspond to the boolean values true and false, respectively. Let B be a boolean value. Then BMN is the conditional, meaning "if B is true then evaluate M, else evaluate N."

Recall that $BMN = (BM)N$ and means "first evaluate the application of B to M; apply that result to N".

Proof. We first show that $(\lambda x.(\lambda y.x))MN \to_\beta^* M$ for all terms M and N:

$$\begin{aligned}
&(\lambda x.(\lambda y.x))MN \\
&\to_\beta ((\lambda y.x)[x := M])N = (\lambda y.M)N \\
&\to_\beta M[y := N] = M
\end{aligned}$$

We must also show that $(\lambda x.(\lambda y.y))MN \to_\beta^* N$ for all terms M and N.

$$\begin{aligned}
&(\lambda x.(\lambda y.y))MN \\
&\to_\beta ((\lambda y.y)[x := M])N = (\lambda y.y)N \\
&\to_\beta y[y := N] = N \quad \square
\end{aligned}$$

2.4 The Simply-Typed λ-Calculus

There are two main reasons for introducing types into a language: efficiency and safety. An example of efficiency occurs when one declares an array of 1,000 items. How can the computer know how much room to allocate for each item? If each item is to be an integer, then one would want 1,000 consecutive words; if each item is to be a double, then one would want 2,000 consecutive words. However, allocating memory "on the fly", or allocating more memory than is necessary, is an inefficient solution. One declares the type of an array as an aid to the compiler.

Safety is another reason to introduce types. As an example, consider attempting to add anything other than two numbers. This results in undefined behavior, and so should not be allowed. A similar situation occurs when the types of parameters given to a function do not match those the function expects to receive. By providing a *type-checking algorithm*, the compiler can ensure that no such errors occur.

We present a type system for the λ-calculus which is known as the *simply-typed* system. This name distinguishes it from more powerful systems, such as polymorphic ones.

Definition. Let $\mathcal{T}$ be a countably infinite set of type variables (written $\alpha, \beta, \dots$). The set of type expressions (written $A, B, \dots$) is given by the grammar:

$$\text{Types } A, B ::= \alpha \;\Big|\; \text{int} \;\Big|\; \text{bool} \;\Big|\; A \to B$$

Thus, types are built from type variables via the $\to$ constructor.

Recall that $\lambda x.M$ is a function which maps x to M. We recognize this by assigning λ-terms of this form the type $A \to B$, where x is of type A and M is of type B. We introduce a notation to discuss types. The foundation of this notation is an *assumption*, which is written $x : A$ and read "x is of type A".

Definition. A *context* is a list of assumptions $x_1 : A_1, \dots, x_n : A_n$, such that no variable appears more than once. We denote contexts by Γ. *Typing judgments* are written $\Gamma \vdash M : A$, where Γ is a context, M is a term and A is a type. Judgments are read "in the context Γ, M is of type A."

¿From these definitions we are able to make *typing rules*, which denote an if-then relationship between judgments. For example, the rule

$$\frac{x : A, \Gamma \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}$$

is read "if the assumption $x : A$ and context Γ make M of type B, then Γ makes $\lambda x.M$ of type $A \to B$."

The simply-typed λ-calculus introduces a new restriction on function application. Recall that in the untyped λ-calculus function applications of the form MN can be done between any M and N. However, once one takes the consideration of types into account, such applications are only logical if M is of type $A \to B$ and N is of type A. That is, M must be of function type and N must be of the same type as M's domain.

Table 1: Typing rules for the simply-typed λ-calculus

$$\frac{}{x : A, \Gamma \vdash x : A} \qquad \frac{x : A, \Gamma \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

The result of this application is a term of type B. Thus, we have the following judgment regarding application:

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

The complete typing rules for the simply-typed λ-calculus are showing in Table 1.

A *type derivation* is the derivation of a typing judgment by repeated use of the typing rules.

Definition. We say that a term M in the simply-typed λ-calculus is *well-typed* if there exists some typing derivation with $\Gamma \vdash M : A$ as its conclusion for some Γ and A.

The types that we have presented in this system are "implicit". This means that terms are not annotated explicitly with their types; types are instead treated as a property of the terms, and it is possible for a term to have several types. For example, the identity function has the type $A \to A$ for any type A. One can even consider $\alpha \to \alpha$ to be the *most general type* of this term, and the types $A \to A$ and $(B \to C) \to (B \to C)$ to be *instances* of it. As we will see in Section 3.3, these concepts provide the foundation for type inference algorithms.

We now give examples of terms and their types.

Example 2.3 (The Type of $\lambda x.(\lambda y.x)$). Recall from Example 2.2 that $\lambda x.(\lambda y.x)$ represents the boolean value true. We wish to ascertain whether or not this term is well-typed. We first apply the rule for abstractions:

$$\frac{x : A, \Gamma \vdash \lambda y.x : B}{\Gamma \vdash \lambda x.(\lambda y.x) : A \to B}$$

Since $\lambda x.(\lambda y.x)$ is an abstraction, it must have the type $A \to B$ for some types A and B. But this will only hold if $\lambda y.x$ has type B under the assumptions $x : A$ and Γ. But as $\lambda y.x$ is a function, and the x in $\lambda y.x$ is bound to the outermost λ, the type B must in fact be $C \to A$ for some type C.

$$\cfrac{\cfrac{y : C, x : A, \Gamma \vdash x : A}{x : A, \Gamma \vdash \lambda y.x : C \to A}}{\Gamma \vdash \lambda x.(\lambda y.x) : A \to (C \to A)}$$

Thus, $\lambda y.x$ has the type $C \to A$ if, under the assumption $y : C$, we have $x : A, \Gamma \vdash x : A$. As all of the subterms of $\lambda x.(\lambda y.x)$ can be assigned a type, the term is well-typed. $\square$

Example 2.4 (An Untypable Term). Not all terms are well-typed; an example is the term xx. Recall that for applications MN the term M must have type $A \to B$ while the term N must have type A. But for $M = N = x$, this is impossible, as a variable may not have more than one type in the same expression. □

2.5 Summary

We can now list three ways in which the untyped λ-calculus relates to functional programming:

1. Functions can be passed as arguments to other functions. One example is

$$(\lambda x.xy)(\lambda z.z) \to_\beta xy[x := \lambda z.z] = (\lambda z.z)y \to_\beta z[z := y] = y.$$

2. Functions can be the results of functions or programs. One example is

$$(\lambda z.z)(\lambda x.xy) \to_\beta z[z := \lambda x.xy] = \lambda x.xy.$$

The result is a function which takes one argument.

3. Functions can be passed (or returned) with some of their arguments already instantiated. For example, the λ-term

$$(\lambda r.rs)((\lambda xy.xy)z)$$

passes the function $\lambda xy.xy$, with the variable x already mapped to z, to the function $\lambda r.rs$.

3 The Language PPL

3.1 Definition of the Language

We now introduce the small functional language PPL. We denote PPL expressions by M or N. Their abstract syntax is given by the following grammar:

$$\begin{array}{llll} \text{PPL Terms} & M, N & ::= & x \mid n \mid \text{true} \mid \text{false} \mid \lambda x.M \mid MN \\ & & & \mid \text{let } x = M \text{ in } N \mid \text{let rec } x = M \text{ in } N \\ & & & \mid () \mid (M_1, \ldots, M_n) \mid \text{pi}_n^j\, M \\ & & & \mid \text{in}_n^j\, M \mid \text{case } M \text{ of in}_n^1\, x_1 \Rightarrow N_1 \mid \ldots \mid \text{in}_n^n\, x_n \Rightarrow N_n \end{array}$$

We give a brief explanation of these terms. The term x denotes a variable and the term n denotes an integer constant. The terms true and false are the boolean values.

The term $\lambda x.M$ represents the function that maps x to M. The term MN denotes the application of the function M to the argument N.

The term (let $x = M$ in N) is a *binding*, defining x to be an abbreviation for M inside the term N. The term (let rec $x = M$ in N) is similar to (let $x = M$ in N), except it also allows x to occur recursively in M.

The term () represents the unique 0-tuple. The PPL term $(M_1, \ldots, M_n)$ represents an n-tuple of PPL terms, where $n \geq 2$. The term $\text{pi}_n^j\, M$ is the j^{th} projection of the n-tuple M. The term $\text{in}_n^j\, M$ is a *tagged term*; it denotes the injection of M into the j^{th} component of n disjoint sets.

(case M of $\text{in}_n^1\, x_1 \Rightarrow N_1 \mid \ldots \mid \text{in}_n^n\, x_n \Rightarrow N_n$) represents a case distinction on the tag of M. If M evaluates to a tagged term $\text{in}_n^j\, P$, then evaluate N_j, where x_j is replaced by P.

We now list the ways in which a variable occurs bound in PPL:

1. x occurs bound in the term $\lambda x.M$. The scope of x is M.
2. x occurs bound in the term (let $x = M$ in N). Here the scope of x is N.
3. x occurs bound in the term (let rec $x = M$ in N). Here the scope of x is both M and N.
4. In the term

$$\text{case } M \text{ of in}_n^1\, x_1 \Rightarrow N_1 \mid \ldots \mid \text{in}_n^n\, x_n \Rightarrow N_n,$$

the variables $x_1, \ldots, x_n$ occur bound in the terms $N_1, \ldots, N_n$, respectively.

All other occurrences of variables in PPL terms are free.

We define α-equivalence and substitution for PPL terms by analogy with the λ-calculus. We omit the formal definitions here. As before, we identify terms up to α-equivalence.

3.2 Call-by-Name Reduction

For the evaluation of PPL terms we adopt the *call-by-name (CBN)* strategy. The rules for this evaluation strategy are given in Table 2.

Definition. The PPL terms x, n, true, false, $\lambda x.M$, (), $(M_1, \ldots, M_n)$ and $\text{in}_n^j\, M$ are in *CBN normal form.*

Note that a CBN normal form, like a β-normal form, does not reduce any further. There are terms, however, which have no reduction but are not normal forms (e.g. $M = \text{pi}_n^j$ true). We say that such terms cause *run-time errors*, and we write $M \to_n$ error.

The difference between CBN and β-reduction lies in when one reduces. In β-reduction one reduces whenever possible. But in CBN reduction one never reduces inside an abstraction, injection, let, let rec, tuple, or the right-hand-side of a case. Also, in the term MN we reduce only M, and not N.

This last rule is the reason this semantics is called call-by-name (as opposed to call-by-value). In passing N as a parameter to $\lambda x.M$, we substitute N as an unevaluated term into the body of the function, rather than first ascertaining its value.

Table 2: Evaluation rules for PPL

(1)	$(\lambda x.M)N \to_n M[x := N]$
(2)	$\dfrac{M \to_n M'}{MN \to_n M'N}$
(3)	$\text{let } x = M \text{ in } N \to_n N[x := M]$
(4)	$\text{let rec } x = M \text{ in } N \to_n N[x := \text{let rec } x = M \text{ in } M]$
(5)	$\text{pi}_n^i(M_1, \ldots, M_n) \to_n M_i$
(6)	$\dfrac{M \to_n M'}{\text{pi}_n^i\, M \to_n \text{pi}_n^i\, M'}$
(7)	$(\text{case in}_n^i\, M \text{ of in}_n^1\, x_1 \Rightarrow N_1 \mid \ldots \mid \text{in}_n^n\, x_n \Rightarrow N_n) \to_n N_i[x_i := M]$
(8)	$\dfrac{M \to_n M'}{\text{case } M \text{ of} \ldots \to_n \text{case } M' \text{ of} \ldots}$

In the presence of predefined function symbols there will be additional evaluation rules and normal forms. For instance, if we add a basic function plus we need to add the following rules:

$$\frac{M \to_n M'}{\text{plus}\, MN \to_n \text{plus}\, M'N} \qquad \frac{N \to_n N'}{\text{plus}\, m\, N \to_n \text{plus}\, m\, N'} \qquad \frac{}{\text{plus}\, m\, n \to_n m + n}$$

The meaning of these rules is as follows. Given the expression $(\text{plus}\, MN)$, repeatedly reduce M until it is in normal form (an integer). Then reduce N until it is in normal form (an integer). When both arguments are in normal form, add them.

We also need to add terms of the form (plus) and $(\text{plus}\, M)$ to the definition of CBN normal form, as they are functions which are both legal and should not be evaluated further. The situation for other function symbols is analogous.

We use $\to_n^*$ to represent the reflexive transitive closure of $\to_n$. We write $M \Downarrow M'$ if $M \to_n^* M'$ and M' is a normal form. We write $M \Downarrow \text{error}$ if there exists M' such that $M \to_n^* M'$ and $M' \to_n \text{error}$.

The reduction rules have the following interesting property:

Lemma 3.1. *The evaluation rules for PPL are* deterministic. *That is, for all PPL terms M there exists at most one M' such that $M \to_n M'$.*

Proof. The proof is by induction on the structure of M.

Suppose $M = x$, n, $\lambda x.N$, true, false, $()$, $(M_1, \ldots, M_n)$, or in_n^j. Then M is a normal form and does not reduce any further. The claim follows trivially.

In all other cases, suppose $M \to_n M'$.

If $M = NP$ then M' is determined by the structure of N. If N is a normal form then M can only reduce by rule (1) from Table 2. Thus, M' is uniquely determined by N and P. If N is not a normal form, then M can only reduce by rule (2). Thus, $M' = N'P$, where $N \to_n N'$. By the induction hypothesis, N' is uniquely determined by N. It follows that M' is uniquely determined as well.

If $M = (\text{let } x = N \text{ in } P)$, then M can only reduce by rule (3). Thus, M' is uniquely determined. Similarly, if $M = (\text{let rec } x = N \text{ in } P)$ then the only rule that applies to M is (4). In this case M' is uniquely determined as well.

If $M = \text{pi}_n^i\, N$ then M' is determined by the structure of N. If N is a normal form then M can only reduce by rule (5). Thus, M' is uniquely determined by N. If N is not a normal form, then M can only reduce by rule (6). Thus, $M' = \text{pi}_n^i\, N'$, where $N \to_n N'$. By the induction hypothesis, N' is uniquely determined by N. Thus, M' is uniquely determined as well.

If $M = (\text{case } N \text{ of} \ldots)$, then M' is determined by the structure of N. If N is a normal form then M can only reduce by rule (7). Thus, M' is uniquely determined by N. If N is not a normal form, then M can only reduce by rule (8). Thus, $M \to_n M'$, where $M' = (\text{case } N' \text{ of} \ldots)$. By the induction hypothesis, N' is uniquely determined by N. Thus, M' is uniquely determined as well. $\square$

It follows from the lemma and the above remarks that for any term M, there exists at most one $M' \in (\text{Terms} \cup \text{error})$ such that $M \Downarrow M'$.

3.3 Types In PPL

We introduce a type system for PPL which is analogous to the one defined for the simply-typed λ-calculus in Section 2.4.

Definition. Let $\mathcal{T}$ be a countably infinite set of *type variables* (written $\alpha, \beta, \ldots$). The set of type expressions for PPL is given by the grammar:

$$A, B ::= \alpha \;\big|\; \text{int} \;\big|\; \text{bool} \;\big|\; A \to B \;\big|\; A_1 * \ldots * A_n \;\big|\; 1 \;\big|\; A_1 + \ldots + A_n \;\big|\; 0,$$

where $n \geq 2$.

The type $A \to B$ is the type of functions which map values of type A to values of type B. The type $A_1 * \ldots * A_n$ is the product of the types A_1 through A_n. Its elements are n-tuples of the form $(M_1, \ldots, M_n)$, where each M_i is of type A_i. The type $A_1 + \ldots + A_n$ is the disjoint union of the types A_1 through A_n. Its elements are tagged terms of the form $\text{in}_n^j\, M$, where M is of type A_j. The type 0 is the empty type. The type 1 is the unit type; it has the unique element $()$.

The typing rules for simply-typed PPL are given in Table 3.

Note that some PPL expressions are well-typed in the empty context. For example, true is always of type bool, and similarly for false. As with the simply-typed λ-calculus, typing is *implicit*. This translates to programmers not needing to state the type of a variable or function; he need only use it consistently.

Table 3: The typing rules for PPL

$$\frac{}{x : A, \Gamma \vdash x : A}$$

$$\frac{n \in \mathbb{N}}{\Gamma \vdash n : \text{int}}$$

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}}$$

$$\frac{}{\Gamma \vdash \text{false} : \text{bool}}$$

$$\frac{x : A, \Gamma \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}$$

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$\frac{\Gamma \vdash M : A \qquad x : A, \Gamma \vdash N : B}{\Gamma \vdash \text{let } x = M \text{ in } N : B}$$

$$\frac{x : A, \Gamma \vdash M : A \qquad x : A, \Gamma \vdash N : B}{\Gamma \vdash \text{let rec } x = M \text{ in } N : B}$$

$$\frac{\Gamma \vdash M : A_1 * \ldots * A_n}{\Gamma \vdash \text{pi}_n^i\, M : A_i}$$

$$\frac{}{\Gamma \vdash () : 1}$$

$$\frac{\Gamma \vdash M_1 : A_1 \qquad \ldots \qquad \Gamma \vdash M_n : A_n}{\Gamma \vdash (M_1, \ldots, M_n) : A_1 * \ldots * A_n}$$

$$\frac{\Gamma \vdash M : A_i}{\Gamma \vdash \text{in}_n^i\, M : A_1 + \ldots + A_n}$$

$$\frac{\Gamma \vdash M : A_1 + \ldots + A_n \qquad x_i : A_i, \Gamma \vdash N_i : C \quad (\text{for } 1 \leq i \leq n)}{\Gamma \vdash \text{case } M \text{ of in}_n^1\, x_1 \Rightarrow N_1 \mid \ldots \mid \text{in}_n^n\, x_n \Rightarrow N_n : C}$$

In the presence of predefined function symbols, there will be additional typing rules giving the type of each such symbol. For example,

$$\frac{}{\Gamma \vdash \text{plus} : \text{int} \to \text{int} \to \text{int}}$$

is the typing rule for plus.

3.4 The polymorphic type system

We now extend the type system with a form of polymorphism known as ML-polymorphism. The polymorphic type system rests on the notion of *type schemas*, which are expressions of the form

$$\forall \alpha_1 \ldots \forall \alpha_n . A.$$

Here $\alpha_1, \ldots, \alpha_n$ are type variables, $n \geq 0$, and A is a type expression. Notice that quantifiers can occur only at the top level in a type schema; there are no quantifiers inside A. The type variables $\alpha_1, \ldots, \alpha_n$ are *bound* in the above type schema, and we identify type schemas up to the renaming of bound variables. We sometimes denote type schemas by $S, T, \ldots$.

We write $A[\alpha_1 := A_1, \ldots, \alpha_n := A_n]$ for the result of simultaneously substituting the type expressions $A_1, \ldots, A_n$ for the type variables $\alpha_1, \ldots, \alpha_n$, respectively, in the type expression A. We say that the type expression B is a *generic instance* of the type schema $\forall \alpha_1 \ldots \forall \alpha_n . A$ if $B = A[\alpha_1 := A_1, \ldots, \alpha_n := A_n]$ for some $A_1, \ldots, A_n$.

Before stating the polymorphic typing rules, we first generalize the definition of a context Γ. A *polymorphic context* is a list $x_1 : S_1, \ldots, x_n : S_n$ of pairs of a variable and a type schema. The old definition of a context is a special case of this newer one, since a simple type (in the sense of the simply-typed system) is just a type schema with zero quantifiers. A *polymorphic typing judgment* is an expression of the form $\Gamma \vdash M : A$, where Γ is a polymorphic context, M is a PPL term, and A is a type expression. It is important to note that quantifiers may occur in Γ, but not in A.

The rules for the polymorphic type system are identical to those for the simple type system, except for the rules for variables, "let", and "let rec". These modified rules are shown in Table 4. In the rules for "let" and "let rec", $\alpha_1, \ldots, \alpha_n$ are type variables that are not free in Γ.

This type of polymorphism was first described for the programming language ML by Milner and Damas [2]. An important feature of this type system is that it is *decidable*. That is, a compiler can decide whether or not a given program is typable in this system. We describe such an algorithm for PPL programs in Section 4.

3.5 Type Soundness

We now formally state the relationship between PPL's type system and the CBN reduction strategy. Before proceeding, we first prove a property about the substitution of PPL terms.

Table 4: The polymorphic typing rules for PPL

(1)	$\dfrac{}{x : S, \Gamma \vdash x : A}$, where A is a generic instance of S	
(2)	$\dfrac{\Gamma \vdash M : A \qquad x : \forall\alpha_1 \ldots \forall\alpha_n.A, \Gamma \vdash N : B}{\Gamma \vdash \text{let } x = M \text{ in } N : B}$,	where $\alpha_1, \ldots, \alpha_n$ not free in Γ
(3)	$\dfrac{x : A, \Gamma \vdash M : A \qquad x : \forall\alpha_1 \ldots \forall\alpha_n.A, \Gamma \vdash N : B}{\Gamma \vdash \text{let rec } x = M \text{ in } N : B}$,	where $\alpha_1, \ldots, \alpha_n$ not free in Γ

Lemma 3.2 (Substitution). *If $x : \forall\alpha_1 \ldots \forall\alpha_n.B, \Gamma \vdash N : A$ and $\Gamma \vdash P : B$, where $\alpha_1, \ldots, \alpha_n \notin FV(\Gamma)$, then $\Gamma \vdash N[x := P] : A$.*

Proof. The proof is by induction on N.

Suppose that $N = x$. We have assumed that $x : \forall\alpha_1 \ldots \forall\alpha_n.B, \Gamma \vdash x : A$. From rule (1) in Table 4, it follows that A is a generic instance of $\forall\alpha_1 \ldots \alpha_n.B$. Thus, $A = B[\alpha_1 := A_1, \ldots, \alpha_n := A_n]$ for some type expressions $A_1, \ldots, A_n$.

We have also assumed that $\Gamma \vdash P : B$. Since $\alpha_1, \ldots, \alpha_n$ are not free in Γ, we can replace all free occurences of $\alpha_1, \ldots, \alpha_n$ by the type expressions $A_1, \ldots, A_n$ in the derivation of $\Gamma \vdash P : B$. The result is a derivation of $\Gamma \vdash P : A$. Thus, $\Gamma \vdash x[x := P] : A$ as desired.

Suppose that $N = n, \text{true}, \text{false}, ()$ or y, where y is a variable and $y \neq x$. Then $x \notin FV(N)$, and so $N = N[x := P]$. The claim follows.

Suppose that $N = \lambda y.M$ and $N : \forall\alpha_1 \ldots \forall\alpha_n.B, \Gamma \vdash x : A$. Without loss of generality, $y \neq x$. From Table 3 we know that $A = C \to D$ and $y : C, x : \forall\alpha_1 \ldots \forall\alpha_n.B, \Gamma \vdash M : D$ for some type expressions C and D. We want to show that $\Gamma \vdash (\lambda y.M)[x := P] : C \to D$. But $(\lambda y.M)[x := P] = \lambda y.(M[x := P])$, and we know that $y : C, \Gamma \vdash M[x := P] : D$ from the induction hypothesis. Thus, $\Gamma \vdash (\lambda y.M)[x := P] : C \to D$, as desired. The remaining cases are similar to this. □

Lemma 3.3 (Subject Reduction). *If $\Gamma \vdash M : A$ and $M \to_n M'$, then $\Gamma \vdash M' : A$.*

Proof. The proof is by induction on the derivation of $M \to_n M'$. There is a case for each rule in Table 2.

Suppose that $M = (\lambda x.N)P$ and $M' = N[x := P]$. By assumption, $\Gamma \vdash (\lambda x.N)P : A$. From the typing rules in Table 3, it follows that $x : B, \Gamma \vdash N : A$ and $\Gamma \vdash P : B$ for some type expression B. It follows from Lemma 3.2 that $\Gamma \vdash N[x := P] : A$.

Suppose that $M = NP$ and $M' = N'P$, where $N \to_n N'$. From the rules in Table 3, we can tell that $\Gamma \vdash N : B \to A$ and $\Gamma \vdash P : B$ for some type expression B. From the induction hypothesis we know that $\Gamma \vdash N' : B \to A$. It follows that $\Gamma \vdash M' : A$.

Suppose that $M = (\text{let } x = P \text{ in } N)$ and $M' = N[x := P]$. By assumption, $\Gamma \vdash (\text{let } x = P \text{ in } N) : A$. From the polymorphic typing rules in Table 4, we know that $\Gamma \vdash P : B$ and $x : \forall\alpha_1 \ldots \forall\alpha_n.B, \Gamma \vdash N : A$, for some $\alpha_1, \ldots, \alpha_n$ not free in Γ. It follows from Lemma 3.2 that $\Gamma \vdash N[x := P] : A$.

Suppose that $M = (\text{let rec } x = P \text{ in } N)$ and $M' = N[x := (\text{let rec } x = P \text{ in } P)]$. By assumption, $\Gamma \vdash (\text{let rec } x = P \text{ in } N) : A$. From the typing rules in Table 4, we can tell that $x : B, \Gamma \vdash P : B$ and $x : \forall\alpha_1 \ldots \forall\alpha_n.B, \Gamma \vdash N : A$, for some $\alpha_1, \ldots, \alpha_n$ not free in Γ. Also from Table 4, we know that $\Gamma \vdash (\text{let rec } x = P \text{ in } P) : B$. It follows from Lemma 3.2 that $\Gamma \vdash N[x := (\text{let rec } x = P \text{ in } P)] : A$.

The remaining cases are similar to the ones proved above. □

Lemma 3.4 (Correctness). *If $\vdash M : A$ then $M \not\to_n$ error.*

Proof. The proof is by induction on M.

Suppose that $\vdash M : A$ and M is not a normal form. We must show that $M \to_n M'$ for some M'. Since M is not a normal form, we only need to consider the cases where M is an application, a "let" or "let rec" expression, a projection or a case distinction.

Suppose that $M = NP$. By assumption, $\vdash M : A$. From the typing rules in Table 3 we know that $\vdash N : B \to A$ and $\vdash P : B$ for some type expression B. If N is not a normal form, we have $N \to_n N'$ for some N' by the induction hypothesis. In this case, M reduces by rule (2) in Table 2. If N is a normal form, however, it follows from Table 3 that N must be a variable or a λ-abstraction. But since $\vdash N : B \to A$ in the empty context, N cannot be a variable. Thus, $N = \lambda x.Q$ for some variable x and term Q. It follows that $M \to_n Q[x := P]$.

Suppose that $M = (\text{let } x = P \text{ in } N)$. Then M can be reduced directly from rule 3 in Table 2. The case for $M = (\text{let rec } x = P \text{ in } N)$ is similar to this.

Suppose that $M = \text{pi}_n^i\, N$. By assumption, $\vdash M : A$. If N is not a normal form, we have $N \to_n N'$ for some N' by the induction hypothesis. In this case, M reduces by rule (6) in Table 2. If N is a normal form, however, then from the rules in Table 3 we know that $\vdash N : A_1 * \ldots * A_n$ and $A_i = A$. Table 3 also tells us that the only normal forms of type $A_1 * \ldots * A_n$ are variables and n-tuples. But since $\vdash N : A_1 * \ldots * A_n$ in the empty context, N cannot be a variable. Thus, $N = (N_1, \ldots, N_n)$. It follows that $M \to_n N_i$.

Suppose that $M = (\text{case } N \text{ of in}_n^1\, x_1 \Rightarrow N_1 \mid \ldots \mid \text{in}_n^n\, x_n \Rightarrow N_n)$. Recall that $\vdash M : A$. From the rules in Table 3 we know that $\vdash N : A_1 + \ldots + A_n$. If N is not a normal form, we have $N \to_n N'$ for some N' by the induction hypothesis. In this case, M reduces by rule (8) in Table 2. If N is a normal form, however, then Table 3 tells us that N must be a variable or injection. But since $\vdash N : A_1 + \ldots + A_n$ in the empty context, N cannot be a variable. Thus, $N = \text{in}_n^j\, P$ for some j. It follows that $M \to_n N_i[x_i := M]$. □

Proposition 3.5 (Type Soundness). *If $\vdash M : A$ then $M \not\to_c^*$ error.*

Proof. Suppose that $\vdash M : A$ and $M \to_c^* M'$. By repeated application of Lemma 3.3, we know that $\vdash M' : A$. From Lemma 3.4 we have $M' \not\to_n$ error. Therefore $M \not\to_c^*$ error. □

4 Type Inference

Recall that PPL is both a polymorphic and a functional language. In Section 1 we demonstrated that this combination allows one to create very robust higher-order functions. However, these functions raise a problem of their own: explicitly annotating their types is an error-prone endeavor.

For example, consider the `exists2` function in ML. This function takes a list a of type α and a list b of type β, as well as a function of type $\alpha \to \beta \to$ bool. The function returns true if $f\, a_i\, b_i =$ true for any a_i and b_i, where a_i is the i^{th} element of list a. It returns false otherwise. The type signature of this function is

$$(\alpha \to \beta \to \texttt{bool}) \to \alpha\,\texttt{list} \to \beta\,\texttt{list} \to \texttt{bool}.$$

Now consider writing a function `xorexists2` which, given two lists a and b, and two functions f_1 and f_2, returns true if $(\texttt{exists2}\; f\; a\; b)$ holds for exactly one of $f = f_1$ and $f = f_2$. The code for this function is very succinct:

```
let xorexists2 f1 f2 a b = xor (exists2 f1 a b)
                               (exists2 f2 a b),
```

where `xor` is the exclusive or function. But this function generates the following signature:

$$(\alpha \to \beta \to \texttt{bool}) \to (\alpha \to \beta \to \texttt{bool}) \to \alpha\;\texttt{list} \to \beta\;\texttt{list} \to \texttt{bool}$$

The signature for this function is as long as (and more opaque than) the function itself! Writing such signatures is something which one would like to avoid if possible.

In Section 2.4 we discussed the importance of a compiler knowing the type signature of functions. We have just shown, however, that it is not always desirable to have the programmer explicitly declare the signature of her functions. The ideal situation appears to involve the compiler *inferring* the types of the variables and functions in a program. This would give us both the safety which type-checking algorithms provide as well as free us from writing extensive annotations. There is also the hope that the compiler can infer more general types than the programmer can.

4.1 Unification

We begin our discussion of polymorphic type inference with a few definitions.

Definition. A *type substitution* is a function σ from variables to type expressions. If A is a type expression, then we write $\sigma(A)$ for the result of applying σ to each type variable in A. If σ and τ are type substitutions, we define their *composition* $\tau \circ \sigma$ to be the type substitution which maps each type variable α to $\tau(\sigma(\alpha))$. A *unifier* of two type expressions A and B is a substitution σ such that $\sigma(A) = \sigma(B)$.

Here are some examples of unifiers:

1. Let $A = \alpha \to \beta$ and $B = \beta \to \gamma$. Then a unifier for A and B is $\sigma = \{\alpha \mapsto \alpha, \beta \mapsto \alpha, \gamma \mapsto \alpha\}$. Verify that $\sigma(A) = \sigma(B) = \alpha \to \alpha$.

2. Let $A = (\beta * \beta) \to \beta$ and $B = \alpha \to (\gamma * \gamma)$. Then a unifier for A and B is $\sigma = \{\alpha \mapsto ((\gamma * \gamma) * (\gamma * \gamma)), \beta \mapsto (\gamma * \gamma), \gamma \mapsto \gamma\}$. Verify that $\sigma(A) = \sigma(B) = ((\gamma * \gamma) * (\gamma * \gamma)) \to (\gamma * \gamma)$.

3. Let $A = \alpha$ and $B = \alpha \to \beta$. Then A and B do not have a unifier. A proof of this can be done by contradiction. Suppose σ is a unifier of A and B. Then $\sigma(\alpha) = C$ and $\sigma(\beta) = D$ for some type expressions C and D. But $\sigma(A) = C$ while $\sigma(B) = C \to D$. Thus, $\sigma(A) \neq \sigma(B)$.

4. Let $A = \alpha * \beta$ and $B = \gamma \to \delta$. Then A and B do not have a unifier. A proof of this can be done by contradiction. Suppose σ is a unifier of A and B. Then $\sigma(\alpha) = C, \sigma(\beta) = D, \sigma(\gamma) = E$ and $\sigma(\delta) = F$ for some type expressions C, D, E and F. But $\sigma(A) = C*D$ while $\sigma(B) = E \to F$. Thus, $\sigma(A) \neq \sigma(B)$.

As the above examples show, not all terms can be unified.

If σ and σ' are type substitutions, then we say that σ is *more general* than σ' if there exists a substitution τ such that $\sigma' = \tau \circ \sigma$. A unifier σ of A and B is called a *most general unifier* of A and B if it is more general than any other unifier of A and B.

An interesting property of most general unifiers is that if two type expression have any unifier, then they also have a most general one. Furthermore, the most general unifier can be computed efficiently by a unification algorithm which was originally published by Robinson [9].

4.2 Polymorphic Type Inference

Let S be a type schema. We write $\sigma(S)$ for the result of applying σ to each *free* type variable in S. Here we rename bound variables as necessary to avoid name clashes, as was done for λ-terms in Section 2.2. If Γ is a context, we write $\sigma(\Gamma)$ for the application of σ to each type schema in Γ.

The behavior of the type inference algorithm is as follows: when given a PPL program M, it returns the most general type A such that $M : A$ is well-typed. If no such type A exists, the algorithm fails.

More generally, we provide an algorithm for the following problem: given a typing judgment $\Gamma \vdash M : A$, find a substitution σ such that $\sigma(\Gamma) \vdash M : \sigma(A)$ is well-typed according to the polymorphic typing rules. If such a substitution exists, return the most general one. Otherwise, fail. This algorithm for polymorphic type inference was first discovered by Milner [7] and was later refined by Damas and Milner [2].

The algorithm works by recursion on the structure of the term M. In each case, the algorithm corresponds to a "bottom-up" reading of the typing rules in Tables 3 and 4. Given a typing judgment $\Gamma \vdash M : A$, we find the substitution σ as follows. Here we list the cases for $M =$ true, false, n, x, $\lambda x.N$, NP, (let $x = N$ in P) and (let rec $x = N$ in P). The other cases proceed by similar reasoning. If any of the mentioned unifiers do not exist, the algorithm fails.

1. If $M =$ true or false, then let σ be the most general unifier of A and bool.

2. If $M = n, n \in \mathbb{N}$, then let σ be the most general unifier of A and int.

3. If $M = x$, find the assumption $x : S$ in Γ. Let B be a fresh generic instance of the schema S, and let σ be the most general unifier of A and B.

4. If $M = \lambda x.N$, let α and β be fresh type variables. Applying the algorithm recursively, find the most general substitution τ such that τ makes $\Gamma, x : \alpha \vdash N : \beta$ well-typed. Let τ' be the most general unifier of A and $\alpha \to \beta$. Let $\sigma = \tau' \circ \tau$.

5. If $M = NP$, let β be a fresh type variable. Applying the algorithm recursively, find the most general substitution τ such that τ makes $\Gamma \vdash N : \beta \to A$ well-typed. Let $\Gamma' = \tau(\Gamma)$ and $B' = \tau(\beta)$. Let τ' be the most general substitution that makes $\Gamma' \vdash P : B'$ well-typed. Let $\sigma = \tau' \circ \tau$.

6. If $M = (\text{let } x = N \text{ in } P)$, let γ be a fresh type variable. Applying the algorithm recursively, find the most general substitution τ such that τ makes $\Gamma \vdash N : \gamma$ well-typed. Let $\Gamma' = \tau(\Gamma)$, $A' = \tau(A)$, $C' = \tau(\gamma)$ and let $\beta_1, \ldots, \beta_n$ be the type variables which are free in C' but not in Γ'. Let τ' be the most general substitution that makes $x : \forall \beta_1 \ldots \beta_n.C', \Gamma' \vdash P : A'$ well-typed. Let $\sigma = \tau' \circ \tau$.

7. If $M = (\text{let rec } x = N \text{ in } P)$, let γ be a fresh type variable. Applying the algorithm recursively, find the most general substitution τ such that τ makes $x : \gamma, \Gamma \vdash N : \gamma$ well-typed. Let $\Gamma' = \tau(\Gamma)$, $A' = \tau(A)$, $C' = \tau(\gamma)$ and let $\beta_1, \ldots, \beta_n$ be the type variables which are free in C' but not in Γ'. Let τ' be the most general substitution that makes $x : \forall \beta_1 \ldots \beta_n.C', \Gamma' \vdash P : A'$ well-typed. Let $\sigma = \tau' \circ \tau$.

5 Abstract Machine Interpretation

5.1 Definition of the Abstract Machine

We now discuss a method of implementing the language PPL. As a functional language, we wish to be able to pass functions as parameters to other functions with some (possibly none) of their parameters instantiated. The syntax of an earlier example of β-reduction leads us to an idea of how to accomplish this:

$$(\lambda r.rs)((\lambda xy.xy)z) \to_\beta rs[r := (\lambda xy.xy)z] = ((\lambda xy.xy)z)s.$$

That is, when given the function $\lambda r.rs$ we evaluate the term rs *while also keeping track* of what its free variables map to.

Our implementation of PPL is based on the above idea. But rather than directly keep track of what the variables map to, we instead use a level of indirection, and keep a list of pointers (addresses) to what the variables map to. We first describe the implementation in terms of an abstract machine, in the style of Krivine [5].

Definition. Let $\mathcal{A}$ be a countable set of addresses (written $a_1, a_2, \ldots$). A *term closure* is a pair $\{M, \sigma\}$, where M is a PPL term, σ is a partial function from variables to $\mathcal{A}$, and $FV(M) \subseteq \text{dom}(\sigma)$. A *match closure* is a pair $\{(\text{in}_n^1\, x_1 \Rightarrow N_1 \mid \ldots \mid \text{in}_n^n\, x_n \Rightarrow N_n), \sigma\}$ with analogous definitions and provisions for σ.

A *heap* is a partial function from addresses to closures. A *stack* is a list whose elements are addresses, integers or labels.

A *state* of the abstract machine is a triple $\langle \{M, \sigma\}, h, \kappa \rangle$, where $\{M, \sigma\}$ is a closure, h is a heap and κ is a stack.

We write nil for the empty stack and $t :: \kappa$ for the stack with topmost element t and remaining elements κ. We write $\sigma(x \mapsto a)$ for the partial function which is identical to σ except that it also maps x to a. We use a similar notation for heaps and write $|\kappa|$ for the size of the stack κ.

The transition relation of the abstract machine is denoted by $\to_m$. We write

$$\langle \{M, \sigma\}, h, \kappa \rangle \to_m \langle \{M', \sigma'\}, h', \kappa' \rangle$$

if the abstract machine can go from the state $\langle \{M, \sigma\}, h, \kappa \rangle$ to the state $\langle \{M', \sigma'\}, h', \kappa' \rangle$ in a single step.

We write $\langle \{M, \sigma\}, h, \kappa \rangle \to_m \text{halt}(s)$ when the abstract machine halts with result s. The possible results are:

$$s ::= n \;\Big|\; \text{true} \;\Big|\; \text{false} \;\Big|\; \text{“fn”} \;\Big|\; \text{“()”} \;\Big|\; \text{“}n\text{-tuple”} \;\Big|\; \text{“in}j/n\text{”},$$

where n and j are integers.

Table 5 contains the list of the abstract machine rules for all PPL terms. Here we explain the rules for $\lambda x.M$, MN and x.

The rules for evaluating a λ-abstraction $\lambda x.M$ are:

$$\begin{array}{l} \langle \{\lambda x.M, \sigma\}, h, a :: k \rangle \to_m \langle \{M, \sigma(x \mapsto a)\}, h, \kappa \rangle. \\ \langle \{\lambda x.M, \sigma\}, h, \text{nil} \rangle \to_m \text{halt}(\text{“fn”}). \end{array}$$

In order to evaluate $\lambda x.M$, the machine pops the address for a closure from the stack and evaluates M under σ plus the additional binding $(x \mapsto a)$. Here a is the address of x's closure. If the stack is empty, however, the program halts. In this case the result of the program is a function. Thus, abstractions can be thought of as pops from the stack.

The rules for evaluating an application MN are:

$$\begin{array}{ll} \langle \{MN, \sigma\}, h, \kappa \rangle \to_m \langle \{M, \sigma\}, h(a \mapsto \{N, \sigma\}), a :: \kappa \rangle, & \text{where } a \text{ is fresh.} \\ \langle \{Mx, \sigma\}, h, \kappa \rangle \to_m \langle \{M, \sigma\}, h, a :: \kappa \rangle, & \text{where } \sigma(x) = a. \end{array}$$

Recall that we are using the call-by-name reduction strategy. The abstract machine implements this by not evaluating N before the function call. Since N may be evaluated later, however, we must save the mapping of its variables. We accomplish this by building a closure for N on the heap and pushing its address onto the stack. Thus, applications can be thought of as pushes onto the stack. In the case where the argument is a variable x, we use an optimization: instead of building a new closure for x, we use the already existing closure $\sigma(x)$.

The rule for evaluating a variable is:

$$\langle \{x, \sigma\}, h, \kappa \rangle \to_m \langle \{M, \tau\}, h, \kappa \rangle, \quad \text{where } \sigma(x) = a \text{ and } h(a) = \{M, \tau\}.$$

Variables in PPL only have meaning in the context of a closure. Evaluating a variable can be thought of as a jump to its closure.

Table 5: Abstract machine rules for PPL

- $\langle\{x,\sigma\},h,\kappa\rangle \to_m \langle\{M,\tau\},h,\kappa\rangle$, where $\sigma(x)=a$, and $h(a)=\{M,\tau\}$.
- $\langle\{Mx,\sigma\},h,\kappa\rangle \to_m \langle\{M,\sigma\},h,a :: \kappa\rangle$, where $\sigma(x)=a$.
- $\langle\{MN,\sigma\},h,\kappa\rangle \to_m \langle\{M,\sigma\},h(a\mapsto\{N,\sigma\}),a :: \kappa\rangle$, where N is not a variable and a is fresh.
- $\langle\{\lambda x.M,\sigma\},h,a :: \kappa\rangle \to_m \langle\{M,\sigma(x\mapsto a)\},h,\kappa\rangle$.
- $\langle\{\lambda x.M,\sigma\},h,\text{nil}\rangle \to_m \text{halt}(\text{“fn”})$.
- $\langle\{\text{let } x=M \text{ in } N,\sigma\},h,\kappa\rangle \to_m \langle\{N,\sigma(x\mapsto a)\},h(a\mapsto\{M,\sigma\}),\kappa\rangle$, where a is fresh.
- $\langle\{\text{let rec } x=M \text{ in } N,\sigma\},h,\kappa\rangle \to_m \langle\{N,\sigma(x\mapsto a)\},h(a\mapsto\{M,\sigma(x\mapsto a)\}),\kappa\rangle$, where a is fresh.
- $\langle\{n,\sigma\},h,\text{nil}\rangle \to_m \text{halt}(n)$.
- $\langle\{\text{true},\sigma\},h,\text{nil}\rangle \to_m \text{halt}(\text{true})$.
- $\langle\{\text{false},\sigma\},h,\text{nil}\rangle \to_m \text{halt}(\text{false})$.
- $\langle\{(),\sigma\},h,\text{nil}\rangle \to_m \text{halt}(\text{“()”})$.
- $\langle\{(M_1,\dots,M_n),\sigma\},h,\text{nil}\rangle \to_m \text{halt}(\text{“}n\text{-tuple”})$.
- $\langle\{(M_1,\dots,M_n),\sigma\},h,j :: \kappa\rangle \to_m \langle\{M_j,\sigma\},h,\kappa\rangle$, if $1\le j\le n$.
- $\langle\{\text{pi}_n^j\, M,\sigma\},h,\kappa\rangle \to_m \langle\{M,\sigma\},h,j :: \kappa\rangle$.
- $\langle\{\text{in}_n^j\, M,\sigma\},h,\text{nil}\rangle \to_m \text{halt}(\text{“in}j/n\text{”})$.
- $\langle\{\text{in}_n^j\, M,\sigma\},h,a :: \kappa\rangle \to_m \langle\{\lambda x_j.N_j,\tau\},h(b\mapsto\{M,\sigma\}),b :: \kappa\rangle$, where $h(a)=\{\text{in}_n^1\, x_1 \Rightarrow N_1 \mid \dots \mid \text{in}_n^n\, x_n \Rightarrow N_n,\tau\}$ and b is fresh.
- $\langle\{\text{case } M \text{ of in}_n^1\, x_1 \Rightarrow N_1 \mid \dots \mid \text{in}_n^n\, x_n \Rightarrow N_n,\sigma\},h,\kappa\rangle \to_m \langle\{M,\sigma\},h(a\mapsto\{\text{in}_n^1\, x_1 \Rightarrow N_1 \mid \dots \mid \text{in}_n^n\, x_n \Rightarrow N_n,\sigma\}),a :: \kappa\rangle$, where a is fresh.

Table 6: Abstract machine rules for “plus”

1. $\langle\{\text{plus},\sigma\},h,\kappa\rangle \to_m \text{halt}(\text{“fn”})$, where $|\kappa|<2$.
2. $\langle\{\text{plus},\sigma\},h,a :: b :: \kappa\rangle \to_m \langle\{M,\tau\},h,\text{plus}_1 :: b :: \kappa\rangle$, where $h(a)=\{M,\tau\}$.
3. $\langle\{m,\sigma\},h,\text{plus}_1 :: b :: \kappa\rangle \to_m \langle\{N,\tau\},h,\text{plus}_2 :: m :: \kappa\rangle$, where $h(b)=\{N,\tau\}$.
4. $\langle\{n,\sigma\},h,\text{plus}_2 :: m :: \kappa\rangle \to_m \langle\{m+n,\sigma\},h,\kappa\rangle$.

In addition to the rules shown in Table 5, there are also rules for each basic function symbol; a list of all basic function symbols can be found in Section 7. We give the rules for “plus” in Table 6.

Rule 1 allows “plus” and “plus M” to be legal PPL programs. Both of these expression are functions and cannot be evaluated further.

One need not pass a number to “plus”; one can also pass a term which evaluates to a number. Rule 2 states that if $\{\text{plus},\sigma\}$ is to be evaluated with the addresses a and b as the topmost elements of the stack, the machine pops a from the stack, pushes the label “plus_1” onto it, and evaluates the closure which a points to. Let the result of evaluating M be m. At this point Rule 3 applies, and the machine pops the label “plus_1” and the address b from the stack and pushes m and the label “plus_2” onto it. The machine then evaluates the closure which b points to. Let the result of evaluating N be n. At this point Rule 4 applies. The machine then pops two elements from the stack and computes $m+n$.

5.2 Properties of the Abstract Machine

The transition rules for the abstract machine given in Table 5 are *deterministic*. That is, for each state there is at most one successor or “halt” state.

We write $\langle\{M,\sigma\},h,\kappa\rangle \to_m$ error when $\langle\{M,\sigma\},h,\kappa\rangle$ cannot legally halt or move to another state. We write $\to_m^*$ for the reflexive transitive closure of $\to_m$.

Examples of terms M for which $\langle\{M,\sigma\},h,\kappa\rangle \to_m^*$ error are $M = \text{pi}_2^1$ false and $M = 5$ true. The reader should verify that these terms result in an error, and also that they are not well-typed. In fact, we can state the relationship between the abstract machine and CBN-reduction more generally as follows:

Proposition 5.1. *Let M be a closed PPL program, and let σ and h be arbitrary. Then*

1. $\langle\{M,\sigma\},h,\text{nil}\rangle \to_m^* \text{halt}(n)$ iff $M \to_c^* n$.
2. $\langle\{M,\sigma\},h,\text{nil}\rangle \to_m^* \text{halt}(\text{true})$ iff $M \to_c^* \text{true}$.
3. $\langle\{M,\sigma\},h,\text{nil}\rangle \to_m^* \text{halt}(\text{false})$ iff $M \to_c^* \text{false}$.

4. $\langle\{M,\sigma\},h,\text{nil}\rangle \to_m^* \text{halt}(\text{"fn"})$ iff $M \to_c^* \lambda x.N$, for some N (or, in the presence of basic functions, $M \to_c^* M'$ for some other normal form M' of function type, such as (plus N)).

5. $\langle\{M,\sigma\},h,\text{nil}\rangle \to_m^* \text{halt}(\text{"()"})$ iff $M \to_c^* ()$.

6. $\langle\{M,\sigma\},h,\text{nil}\rangle \to_m^* \text{halt}(\text{"}n\text{-tuple"})$ iff $M \to_c^* (N_1,\dots,N_n)$ for some $N_1,\dots,N_n$.

7. $\langle\{M,\sigma\},h,\text{nil}\rangle \to_m^* \text{halt}(\text{"in}j/n\text{"})$ iff $M \to_c^* \text{in}_n^j\, N$, for some j, n and N.

8. $\langle\{M,\sigma\},h,\text{nil}\rangle \to_m^* \text{error}$ iff $M \to_c^* \text{error}$.

Corollary 5.2. *If* $\Gamma \vdash M : A$ *then* $\langle\{M,\sigma\},h,nil\rangle \not\to_m^*$ *error.*

Proof. From Proposition 3.5 we know that $\Gamma \vdash M : A$ implies $M \not\to_c^*$ error. From Proposition 5.1 we know that $\langle\{M,\sigma\},h,\text{nil}\rangle \to_m^*$ error if and only if $M \to_c^*$ error. Thus, $\langle\{M,\sigma\},h,\kappa\rangle \not\to_m^*$ error. □

6 A Compiler for PPL

6.1 An Idealized Assembly Language

In this section we describe an idealized assembly language which will be the target language of the compiler. Its chief differences to actual assembly languages are that we assume an infinite number of registers, and that we have a built-in opcode for dynamic memory allocation. (This is usually realized by an operating system call). We denote registers by $V, C, r_1, r_2, \dots$. Each register can hold an integer or a pointer. There is a special register SS which holds the current stack size.

Memory locations can also be denoted by symbolic *labels*, which we denote by $l_1, l_2, \dots$. An *l-value* (assignable value) is either a register or a memory location $[r,n]$ specified by a register r and an integer offset n. The expression $[r,n]$ refers to the memory location n words after that pointed to by the register r. A *symbolic value* is either an l-value or a *literal value*. A literal value is either an integer constant, written $\#n$, or a label l.

The assembly language has the following opcodes. Here v ranges over values, lv over l-values, and s ranges over results (as defined for the abstract machine above).

Opcode	Meaning
JUMP v	Jump to address v
EXIT s	Exit and print string s
EXITINT v	Exit with integer result v
EXITBOOL v	Exit with boolean result false ($v = 0$) or true ($v \neq 0$).
LOAD lv, v	Store the value v in location lv
POP lv	Pop a value from the stack and store it in lv
PUSH v	Push the value v onto the stack
ALLOC lv, v	Allocate v words of memory and store a pointer to the first one in lv
CMP v_1, v_2	Compare the two values, and remember the result
BE v	If CMP results in equal, jump to location v
BG v	If CMP results in greater than, jump to location v
BL v	If CMP results in less than, jump to location v
BNE v	If CMP results in not equal, jump to location v
BGE v	If CMP results in greater than or equal jump to location v
BLE v	If CMP results in less than or equal to, jump to location v

There are also opcodes to calculate basic functions, such as ADD lv, v (add v to lv) and MULT lv, v (multiply lv by v).

6.2 The Translation of PPL Terms

We now explain how PPL terms are represented in the assembly language. An address is represented as a pointer, and integers are represented as themselves. Boolean values are also represented as integers, where false $= 0$ and true $= 1$. The heap is represented as allocated memory, and the stack of the abstract machine is represented by the stack in the assembly language. A PPL term is represented as assembly code.

A closure $\{M,\sigma\}$ (where $FV(M) = x_1,\dots,x_k$) is represented by a data structure that occupies $k+1$ words of memory. At offset 0 is a pointer to the code for M, and at offsets 1 through k are pointers to the values for $x_1,\dots,x_k$, respectively.

A match closure $\{(\text{in}_n^1\, x_1 \Rightarrow N_1 \mid \dots \mid \text{in}_n^n\, x_n \Rightarrow N_n),\sigma\}$ (where the free variables of the matching are $x_1,\dots,x_k$) is represented as a data structure that occupies $k+n$ words of memory. At offsets 0 through $(n-1)$ are pointers to the code for $\lambda x_i.N_i$, where $1 \le i \le n$. At offsets n through $(n+k-1)$ are pointers to the values for $x_1,\dots,x_k$, respectively.

A term closure is invoked by putting a pointer to the closure into the special register C, and then jumping to the address at offset 0. Similarly, the j^{th} branch of a match closure is invoked by putting a pointer to the match closure into C and then jumping to the address at offset $j-1$.

If an integer or a boolean value has been computed, the convention is to put the value in the special register V and then jump to the address on top of the stack. If the stack is empty, then V is the final result of the program.

The translation of a PPL term M is defined relative to a *symbol table* s, which is a function that maps the free variables of M to symbolic values. The notation $[\![M]\!]_s$

means "the assembly code for the PPL expression M, where the value of each free variable x is found in $s(x)$."

The definition of $[\![M]\!]_s$ for all PPL terms M is given in the Appendix. Compare these definitions with the abstract machine rules for PPL terms given in Table 5. Here we explain the definition of $[\![\lambda x.M]\!]_s$.

$[\![\lambda x.M]\!]_s =$

	CMP	$SS, \#0$
	BG	l
	EXIT	"fn"
l :	POP	r
	$[\![M]\!]_{s(x\mapsto r)}$	

where l is a new label and r is a new register.

Recall that if the stack is empty then $\lambda x.M$ is the result of the program, and the program returns "fn". If the stack is not empty, however, the machine pops the address of a closure from the stack and evaluates M under the current symbol table plus the mapping of x to the address just popped.

Note that all assembly code consists of a number of labeled segments, each of which ends in a JUMP or an EXIT.

The code for the basic functions is more complex than that of the basic PPL terms. We give the assembly code for plus in Table 7. Its relation to the abstract machine rules is as follows.

The code for "plus" checks to see if there are less than two elements on the stack; if so, it halts with the result "fn". Thus, "plus" tests if rule 1 from Table 6 applies.

If rule 1 does not apply then rule 2 does; rule 2 is realized at the label plus_0. Here the machine pops the address of the closure for the first argument to the register C. It then pushes the label plus_1 onto the stack and invokes the closure pointed to by C. The label plus_1 functions as the return address of a subroutine call; the machine will jump to the label plus_1 after it is done evaluating the first closure. By convention, the value which the closure evaluates to is now stored in the register V.

Now the machine pops the address of the closure for the second argument to plus to the register C, and saves the value of V and the return address plus_2 on the stack. Then it invokes the closure at C. The reader should verify that this behavior corresponds to Rule 3.

When the second closure has been evaluated, the value of the second argument is in the register V, and the value of the first argument is on top of the stack. The code pops the topmost element from the stack, adds the two arguments, and puts the result in V. This behavior corresponds to Rule 4. Finally, the code follows the standard convention for integer results by jumping to the return address on top of the stack, or halting if the stack is empty.

Table 7: The assembly code for plus

$[\![\text{plus}]\!]_s =$

	CMP	$SS, \#2$
	BGE	plus_0
	EXIT	"fn"
plus_0 :	POP	C
	PUSH	plus_1
	JUMP	$[C, 0]$
plus_1 :	POP	C
	PUSH	V
	PUSH	plus_2
	JUMP	$[C, 0]$
plus_2 :	POP	A
	ADD	V, A
	CMP	$SS, \#1$
	BGE	plus_3
	EXITINT	V
plus_3 :	POP	r
	JUMP	r

7 A Guide to the Implementation

7.1 Introduction

A prototypical implementation of PPL is available on the world-wide web at [6]. It is written in Objective Caml. The main component of the implementation is the executable program `ppl`, which reads a PPL program and outputs compiled pseudo assembly code. For reasons of portability, the "assembly code" generated by `ppl` is actually realized as a set of pre-processor macros in the C language; thus, the output of `ppl` can be compiled by any C-compiler on the target machine.

The concrete syntax for PPL is slightly different than the syntax introduced in the paper. For example, we write `pi1/2` instead of pi_2^1, as sub- and superscripts are not supported in ASCII. Our implementation reads `\` as λ, thus `\x.x` is the identity function. Our implementation also supports some syntactic sugar. For example, `\x y z.M` is interpreted as `\x.(\y.(\z.M))`. One can also write `let fun x y z = M` instead of `let fun = \x.\y.\z.M`. Both of these expressions represent the function which takes x, y and z as variables and maps them to the term M.

Our implementation is also lenient with match statements in case distinctions. We allow cases to occur in an arbitrary order, with some cases duplicated or missing. In such cases our compiler issues a warning, and if a missing case is encountered at runtime, the program exits with an error message. We have also added several basic functions to the language. We list them together with their type signatures in Table 8.

Table 8: The Basic Functions of PPL

Function	Type Signature
plus	int → int → int
minus	int → int → int
mult	int → int → int
divide	int → int → int
mod	int → int → int
greater	int → int → bool
geq	int → int → bool
less	int → int → bool
leq	int → int → bool
equal	int → int → bool
neq	int → int → bool
and	bool → bool → bool
or	bool → bool → bool
not	bool → bool
if	$\forall\alpha$.bool → $\alpha \to \alpha \to \alpha$

The implementation of "and" and "or" are *lazy* - the second argument is only evaluated if necessary. Similarly, in the term "if M N Q", only one of N or Q is evaluated. The other basic functions evaluate all of their arguments.

7.2 User Manual

We now present information necessary to use our PPL compiler. There are several flags which affect the compiler's behavior, and their behavior is summarized in Table 9. The default behavior of the compiler is realized by typing `ppl filename`, where `filename` is a file which contains a PPL program. The compiler will read the program, write its most general type to the terminal, and write its assembly code translation to a file.

The `--parse` flag will cause the compiler to read and type-check a PPL program but not compile it. The `--reduce` and `--step` flags cause the compiler to apply CBN reduction to the input term. In `--reduce` mode, the final normal form is printed, whereas in `--step` mode, the entire reduction sequence is printed, one term per line. The `--typeinfo mode` flag alters the amount of type information that is displayed to the terminal. `mode` must be one of `none`, `all`, `top`, `let` or an integer nesting depth n. The compiler then gives type information on, respectively, no variables, all variables, all variables defined on the top-level, variables defined only by let and letrec constructs, and variables defined up to a depth of n. The `--untyped` flag causes the compiler to not type-check the PPL program. The `--optimize` flag will cause the compiler to β-reduce certain redexes before compilation, yielding more efficient assembly code.

Table 9: Command line options for the compiler

Flag	Effect
(no options)	The default behavior. Read input from file, print most general type to terminal, and write compiled program to file.
`--parse, -p`	Do not compile; parse and type-check only.
`--step, -s`	Print CBN reduction sequence.
`--reduce, -r`	Reduce term to CBN normal form.
`--untyped, -u`	Omit type-checking.
`--optimize, -z`	Create optimized compiled code.
`--typeinfo mode, -i mode`	Print additional type information depending on `mode`.
`--term term`	Use `term` as input.
`--stdin`	Read input from terminal.
`--stdout`	Write output to terminal.
`--output filename, -o filename`	Write output to specified file.
`--help, -h`	Print help message and exit.
`--version, -v`	Print version info and exit.

The `--term`, `--stdin`, `--stdout`, and `--output filename` flags affect where the compiler looks for input and where it writes the output. When using the `--term term` flag one may find it useful to enclose `term` in quotation marks. This will prevent shell substitutions.

The `--help` flag provides a list and brief description of all PPL options. The `--version` flag gives information on which version of the compiler is in use.

There is also a graphical user interface for PPL called `ppli`. It is written in Tcl/Tk. In Unix, it can be accessed by typing `wish ppli` at the command line. `ppli` provides a window to type a PPL program or load a program from a file, together with various buttons for compiling, reducing, and stepping through the reduction sequence of a term.

7.3 Example Programs

Our compiler for PPL comes with several example programs. The file `programs.txt`, also available at [6], contains a list and description of each program.

8 Future Work

We have implemented PPL as a self-contained example of a functional programming language. It is intended as a basis for experimenting with improved implementation techniques, as well as new language features. Features that we would like to add in the future include improved source level optimization, closure optimization, general recursive types, non-local control features in the style of the $\lambda\mu$-calculus, and a run-time system with a proper garbage collector. We are also interested in extending automatic type inference to recursive types, and the interaction of lazy evaluation and continuations.

9 Appendix: Assembly Code For PPL Terms

$[\![x]\!]_s =$

	LOAD	$C, s(x)$
	JUMP	$[C, 0]$

$[\![\lambda x.M]\!]_s =$

	CMP	$SS, \#1$
	BGE	l
	EXIT	"fn"
$l:$	POP	r
	$[\![M]\!]_{s(x \mapsto r)}$	

(where l is a new label and r is a new register.)

$[\![Mx]\!]_s =$

	PUSH	$s(x)$
	$[\![M]\!]_s$	

$[\![MN]\!]_s =$

	; build closure for N	
	ALLOC	$r, \#(n+1)$
	LOAD	$[r, 0], l$
	LOAD	$[r, 1], s(x_1)$
	...	
	LOAD	$[r, n], s(x_n)$
	PUSH	r
	$[\![M]\!]_s$	
$l:$	$[\![N]\!]_{(x_1 \mapsto [C,1], \dots, x_n \mapsto [C,n])}$	

(where l is a new label, r is a new register, N is not a variable, and $FV(N) = \{x_1, \dots, x_n\}$.)

$[\![\text{let } x = M \text{ in } N]\!]_s =$

	; build closure for M	
	ALLOC	$r, \#(n+1)$
	LOAD	$[r, 0], l$
	LOAD	$[r, 1], s(x_1)$
	...	
	LOAD	$[r, n], s(x_n)$
	$[\![N]\!]_{s(x \mapsto r)}$	
$l:$	$[\![M]\!]_{(x_1 \mapsto [C,1], \dots, x_n \mapsto [C,n])}$	

(where l is a new label, r is a new register, and $FV(M) = \{x_1, \dots, x_n\}$.)

$[\![\text{let rec } x = M \text{ in } N]\!]_s =$

	; build closure for M	
	ALLOC	$r, \#(n+1)$
	LOAD	$[r, 0], l$
	LOAD	$[r, 1], s'(x_1)$
	...	
	LOAD	$[r, n], s'(x_n)$
	$[\![N]\!]_{s'}$	
$l:$	$[\![M]\!]_{(x_1 \mapsto [C,1], \dots, x_n \mapsto [C,n])}$	

(where l is a new label, r is a new register, $FV(M) = \{x_1, \dots, x_n\}$, and $s' = s(x \mapsto r)$.)

$[\![n]\!]_s =$

	LOAD	$V, \#n$
	CMP	$SS, \#1$
	BGE	l
	EXITINT	V
$l:$	POP	r
	JUMP	r

(where l is a new label and r is a new register.)

$[\![\text{true}]\!]_s =$

	LOAD	$V, \#1$
	CMP	$SS, \#1$
	BGE	l
	EXITBOOL	V
$l:$	POP	r
	JUMP	r

(where l is a new label and r is a new register.)

$[\![\text{false}]\!]_s =$

	LOAD	$V, \#0$
	CMP	$SS, \#1$
	BGE	l
	EXITBOOL	V
$l:$	POP	r
	JUMP	r

(where l is a new label and r is a new register.)

$[\![()]\!]_s =$

```
    EXIT    "()"
```

$[\![(M_1,\dots,M_n)]\!]_s =$

```
      CMP     SS, #1
      BGE     l1
      EXIT    "n-tuple"
l1 :  POP     r
      CMP     r, #1
      BNE     l2
      [[M1]]s
l2 :  CMP     r, #2
      BNE     l3
      [[M2]]s
l3 :  ...
      ...
ln :  [[Mn]]s
```

(where $n \geq 2$, $l_1,\dots,l_n$ are new labels, and r is a new register.)

$[\![\mathrm{pi}_n^j\, M]\!]_s =$

```
      PUSH    #j
      [[M]]s
```

$[\![\mathrm{in}_n^j\, M]\!]_s =$

```
      CMP     SS, #1
      BGE     l1
      EXIT    "inj/n"
l1 :  ; build closure for M
      ALLOC   r, #(n + 1)
      LOAD    [r, 0], l2
      LOAD    [r, 1], s(x1)
      ...
      LOAD    [r, n], s(xn)
      POP     C
      PUSH    r
      ; invoke jth branch of match closure
      JUMP    [C, j - 1]
l2 :  [[M]](x1 ↦ [C,1], ..., xn ↦ [C,n])
```

(where l_1, l_2 are new labels, r is a new register, and $FV(M) = \{x_1,\dots,x_n\}$.)

$[\![\text{case } M \text{ of } \mathrm{in}_k^1\, y_1 \Rightarrow N_1 \mid \dots \mid \mathrm{in}_k^k\, y_k \Rightarrow N_k]\!]_s =$

```
      ; build match closure
      ALLOC   r, #(k + n)
      LOAD    [r, 0], l1
      ...
      LOAD    [r, k - 1], lk
      LOAD    [r, k], s(x1)
      ...
      LOAD    [r, k + n - 1], s(xn)
      PUSH    r
      [[M]]s
l1 :  [[λy1.N1]](x1 ↦ [C,k], ..., xn ↦ [C,k+n-1])
      ...
lk :  [[λyk.Nk]](x1 ↦ [C,k], ..., xn ↦ [C,k+n-1])
```

(where $l_1,\dots,l_k$ are new labels, r is a new register, and $FV(\mathrm{in}_k^1\, y_1 \Rightarrow N_1 \mid \dots \mid \mathrm{in}_k^k\, y_k \Rightarrow N_k) = \{x_1,\dots,x_n\}$.)

References

[1] H. P. Barendregt. *The Lambda Calculus, its Syntax and Semantics*. North Holland, 2nd edition, 1984.

[2] L. Damas and R. Milner. Principal type-schemes for functional programs. In *9th Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[3] C. Hankin. *Lambda Calculi: A Guide For Computer Scientists*. Oxford University Press, 1994.

[4] S. C. Kleene. λ-definability and recursiveness. *Duke Mathematical Journal*, 2:340–353, 1936.

[5] J.-L. Krivine. Un interpreteur du lambda-calcul. Draft, available from ftp://ftp.logique.jussieu.fr/pub/distrib/interprt.dvi, 1996.

[6] A. Lamstein and P. Selinger. Implementation of the programming language PPL, Oct. 2000. Available from http://theory.stanford.edu/˜selinger/ppl/.

[7] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[8] G. E. Revesz. *Lambda-Calculus, Combinators, and Functional Programming*. Tracts in Theoretical Computer Science 4. Cambridge University Press, 1998.

[9] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12:23–41, 1965.