LEAP: A Language with Eval And Polymorphism

Frank Pfenning and Peter Lee

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Abstract

We describe the core of a new strongly-typed functional programming language called LEAP, a "Language with Eval And Polymorphism." Pure LEAP is an extension of the ω-order polymorphic λ-calculus (F_ω) by global definitions that allows the representation of programs and the definition of versions of `reify`, `reflect`, and `eval` for all of F_ω. Pure LEAP is therefore highly reflexive *and* strongly typed. We believe that Pure LEAP can be extended to a practical and efficient metalanguage in the ML tradition. At present we are experimenting with a prototype implementation of Pure LEAP.

1 Introduction

In this paper we describe the core of a new strongly-typed functional programming language called LEAP, a "Language with Eval And Polymorphism." Our initial motivation came from the problem of finding a strongly-typed language suitable for use as a metalanguage for manipulating programs, proofs, and other similar symbolic data. The language ML [11] seemed to satisfy many of our criteria, but was not powerful enough to serve as its own metalanguage in a natural way. (We discuss what we mean by "natural" in Section 2.)

This then led us to the question, first posed by Reynolds in [17], of whether strongly-typed languages admit metacircular interpreters. Conventional wisdom seemed to indicate that the answer was "No." Our answer is "Almost." After a brief review of F_ω in Section 3, we explain this answer in Sections 4 and 5 by giving a construction reminiscent of the *reflective tower* of Smith [18,19]. Wand and Friedman's analysis of the reflective tower [3,22] emphasizes *reification*, the translation from programs to data, and *reflection*, the translation from data to programs, as central concepts. In the setting of a strongly-typed functional language, we have found elegant and concise definitions of reification and reflection.

Somewhat unexpectedly for us, the "tower" begins with an interpreter for the second-order polymorphic λ-calculus (F_2) (see Girard [5,6] and Reynolds [16]) written in the third-order polymorphic λ-calculus (F_3). This does not easily extend to higher orders—only

This research was supported in part by the Office of Naval Research under contract N00014-84-K-0415 and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 5404, monitored by the Office of Naval Research under the same contract.

the addition of global definitions with polymorphic kinds to F_ω allowed us to extend the construction. The result is a core language called Pure LEAP which is strong enough to allow the definition of reification and reflection functions for all of F_ω.

These theoretical results lead us to ask whether LEAP can be usefully extended while still preserving this ability to build a reflective tower. This is, in fact, possible, and we describe several such extensions in Section 6. First we extend LEAP with inductive data type definitions and primitive recursion (a conservative extension), and then we briefly sketch out extensions involving references, exceptions, and general recursion (no longer conservative, but preserving the reflection property as before).

We claim that LEAP can be the core of a practical language in which efficient (meta-)programs can be written. To test its practicality, we are presently experimenting with a prototype implementation of LEAP as well as designing a full language around it.

2 Reflection, Reflexivity, and Static Typing

The idea of reflection in untyped programming languages can be found in both the early and the recent literature. In [17] Reynolds gives a metacircular interpreter for a simple, untyped functional language within itself. This was pursued further by Steele and Sussman [20] and others. In fact, writing metacircular interpreters has long been a standard practice in LISP [8,9]. Smith, in [18,19], introduces the notion of the *reflective tower*, illustrating it in the language 3-LISP. Friedman and Wand give their own analysis of the reflective tower in [4] and [3,22], isolating *reification* and *reflection* as key concepts.

This paper reports on our attempt to model reification and reflection in a strongly-typed language. Our results may be summarized as follows: (1) The third-order polymorphic λ-calculus (F_3) is powerful enough to represent programs written in the second-order polymorphic λ-calculus (F_2) and also the functions `reify` and `reflect`. This allows the definition of `eval` for F_2 in F_3. (2) If one extends F_ω by allowing polymorphic kinds (forming the Pure LEAP language), then one can define `reify` and `reflect` for all of F_ω, thus falling just short of a complete metacircular interpreter for all of LEAP. (3) The analogue of the structure of the reflective tower emerges when one considers the restriction of Pure LEAP to types of order n (LEAP_n). Then LEAP_{n+1} is powerful enough to allow the definition of F_n for $n \geq 2$. (4) We conjecture that it is impossible to define `reify` and `reflect` for the simply typed λ-calculus in F_2, that is, the tower begins with an interpreter for F_2 in F_3.

There are two representation "tricks" that make reflection possible in Pure LEAP. The first is to dispense entirely with the environments that play such a crucial role in previous work on metacircular interpreters and reflective towers. This trick seems necessary, since environments bind variables of different type, and therefore cannot be typed consistently. Instead, one uses continuations to reify (represent) λ-abstraction. As a result we obtain a reification mechanism similar to the Lisp `quote` operator, but in which all variables are antiquoted (and hence captured in the current environment) at the time they are reified. (Actually, reification is more akin to the `backquote` operator, since `backquote` is typically used in Lisp to create program data structures containing captured variables.) Hence the

environments of, for example, 3-LISP are implicitly carried by the reified data structures. (This is described in greater detail in Section 4.)

The second trick is the solution to the technical challenge of dealing with inductively defined data types with polymorphic constructors. This problem had been addressed in the literature (see [1] and [11] for two different approaches) only for the case where types are guaranteed to be uniform over any given element of the inductive type (such as lists: a list of type α has sublists only of type α). Programs do not have this uniformity property, since programs of type α can have subprograms of arbitrary type.

2.1 Reflexivity of languages

We are concerned not only with the ability of a language to form a reflective tower, but also with how easily and naturally this construction can be expressed. We call this the *reflexivity* of the language. We will not attempt to give a formal definition for when a language is reflexive. Instead we will try to give some informal criteria for judging the degree of reflexivity of a language, the basic one being the ability of a language to serve as its own metalanguage. This by itself does not seem enough, since then every Turing-complete language would be reflexive. In addition, we would like to require that the language/metalanguage relationship is "natural." When is this relationship "natural"? We think the answers to the following questions provide some hints when evaluating the degree of reflexivity of a language.

- How redundant is the definition of a metacircular interpreter? In a highly reflexive language, the metacircular interpreter should be simple and direct. The more that features of the object language can be implemented by using the corresponding features of the metalanguage, the more reflexive the language. We call this phenomenon *inheritance* of object language features from the metalanguage. Typical examples of features for which inheritance might be desirable are evaluation order (*e.g.*, call-by-value *vs.* call-by-name) and, as we shall see, static type-checking.
- How much of the metalanguage can be interpreted by the metacircular interpreter? Ideally, the metalanguage and object language should coincide.
- Can we define the functions `reify` and `reflect` in addition to `eval`? That is, can we coerce data into programs and vice versa?
- How well can object language syntax and metalanguage syntax be integrated? We will mainly ignore this issue: with the aid of good syntactic tools one should always be able to achieve a reasonably smooth integration of metalanguage and object language.

2.2 Inheritance of metalanguage features

We believe that the concept of inheritance is important when considering the relationship of a metalanguage to its object language. Inheritance (though not under this name) was already considered by Reynolds [17]. The following examples should help to illustrate the concept.

- An ML interpreter written in ML would likely be highly redundant, since type inference would have to be reimplemented explicitly. In other words, it seems that ML type inference cannot be inherited, in part because because of the complexity of the data type of programs, and also because of the "generic" nature of the ML **let** construct. Our solution to the generic **let** problem is discussed in Section 6.1.2.
- An interpreter written for a dynamically-scoped LISP will also be redundant, since environments must be represented and manipulated explicitly by the interpreter. The notion of variable binding cannot be inherited and must be programmed explicitly. However, many other features such as automatic storage management clearly are inherited in a typical metacircular LISP interpreter. However, our results for LEAP indicate that a statically-scoped LISP could use closures in the metainterpreter instead of environments.
- An interpreter for (pure) Prolog without cut written in Prolog is not very redundant, in particular since unification can be inherited. Other properties, such as whether search should be conducted in depth-first or breadth-first order can also be inherited. Prolog with cut is less reflexive, since the notion of cut must be implemented explicitly and cannot be inherited.
- In the LEAP language, type inference and variable binding mechanisms will both be inherited. Evaluation order will also be inherited, thus making LEAP very reflexive. It should be noted that this is not so important for the pure language, since it has the strong normalization property (see Theorem 3).

As one can see from the examples, reflexivity is elusive. Care must be taken when extending a language in order not to lose too much reflexivity. The reflexivity of pure Prolog, for instance, seems to be diminished by the addition of a cut operator. In other cases, the reflexivity of a language can be enhanced through strengthening. For example, we shall see that the addition of explicit polymorphism to the simply-typed λ-calculus results in a highly reflexive language.

Languages that have a strong degree of reflexivity seem in some way to distill the essence of a computational paradigm into a pure form. We believe that language designers should pay attention to the issue of reflexivity, in particular when designing a language for use as a metalanguage. We hope to demonstrate this principle in the following sections as we describe Pure LEAP, a highly reflexive language based on the ω-order polymorphic λ-calculus.

3 The ω-Order Polymorphic λ-Calculus

In [5,6], Girard defines a powerful extension to Church's simply typed λ-calculus [2] and goes on to give a constructive proof of strong normalization for his system. A fragment of Girard's calculus was independently discovered by Reynolds [16] who introduced abstraction on type variables and application of functions to types in order to define explicitly polymorphic functions. Reynolds' calculus is known as the second-order polymorphic λ-calculus.

Here we consider the ω-order polymorphic λ-calculus, which is an extension of Reynolds' system but only a fragment of Girard's system (since it omits existentially quantified types). Our presentation of the calculus contains three distinct syntactic categories: *kinds*, *types*, and *terms*.

Since our calculus is higher-order, we have, in addition to types of terms, functions from types to types, *etc.* We will call every such object a *type*. The subset of these that are first-order, or, equivalently, of kind "Type," can actually be the type of a term. These and other properties of the calculus are summarized at the end of this section. Following Girard, we will write F_n for the language of the nth-order polymorphic λ-calculus, and F_ω for the union over all finite orders.

The language should properly be parameterized over a signature for type constructors and term constants. Since the pure language contains no such constants or constructors, we will abbreviate the presentation. We use K, K' for kinds, $\alpha, \beta, \ldots$ for types and type variables, θ for type variables, $M, N, \ldots$ for terms, and $x, y, \ldots$ for variables.

Definition 1 *The syntactic categories of* kind, type, *and* term *are defined inductively by*

$$\begin{array}{lrcl} \textit{Kinds} & K & ::= & \text{Type} \mid K \to K' \\ \textit{Types} & \alpha & ::= & \theta \mid \lambda\theta{:}K \,.\, \alpha \mid \alpha\,\beta \mid \alpha \Rightarrow \beta \mid \Delta\theta{:}K \,.\, \alpha \\ \textit{Terms} & M & ::= & x \mid \lambda x{:}\alpha \,.\, M \mid M\,N \mid \Lambda\theta{:}K \,.\, M \mid M\,[\alpha] \end{array}$$

We will not give the formal type inference system for this language here, but merely explain it informally. A more formal development can be found in [14]. The λ symbol is used to construct functions that can be applied to a term, yielding a term, and also to build functions that can be applied to a type, yielding a type. The symbol Λ constructs functions that can be applied to types, yielding a term. Such a function will have a Δ type. The order of a term in this calculus is determined by what kind of abstractions over types are allowed: we obtain the second-order polymorphic λ-calculus (F_2) if we allow abstractions only over type variables of kind Type; we obtain F_3 if we allow abstractions over type variables of kinds Type $\to \ldots \to$ Type; *etc.* We use "$M \in \alpha$" to indicate that term M has type α, and "$\alpha \in K$" to indicate that α has kind K. We use Γ to stand for contexts, which uniquely assign kinds to type variables and types to term variables. We will omit empty contexts.

In the second-order fragment F_2 of F_ω, one can explicitly define common data types and operations on them, such as natural numbers ($\text{int} \equiv \Delta\theta \,.\, \theta \Rightarrow (\theta \Rightarrow \theta) \Rightarrow \theta$), products, disjoint sum, and lists ($\text{list} \equiv \lambda\alpha \,.\, \Delta\theta \,.\, (\alpha \Rightarrow \theta \Rightarrow \theta) \Rightarrow \theta \Rightarrow \theta$). For a good exposition see Reynolds [15] or Böhm [1]. We will give an alternative way of defining some of these data types in Section 6.1.3.

Next we define the *judgments* of the inference system that allow us to find valid types for terms and kinds for types.

Definition 2 *The judgments we use to define when a term is well-typed are:*

$\vdash \Gamma$ context	Γ *is a valid context*
$\vdash K \in$ kind	K *is a valid kind*
$\Gamma \vdash \alpha \in K$	α *has kind* K
$\Gamma \vdash u \in \alpha$	u *has type* α

The inference rules used to establish the validity of types, terms, or contexts can be found in [14]. We will regard α-convertible types and terms (with binders λ, Λ, and Δ) to be equal. Thus we will ignore the issues of variable renaming and name clashes.

In the inference rules of the polymorphic λ-calculus, we will allow conversion between $\beta\eta$-equivalent types. We define β and η conversions of types as is usually done on terms. For example, a β-redex has the form $(\lambda\theta{:}K\,.\,\alpha)\,\gamma$.

In the conversions for terms we now also include the β-conversion of type applications, $(\Lambda\alpha\,.\,M)\,[\beta] \underset{\beta}{=} (\beta/\alpha)M$ and the η-conversion, $(\Lambda\alpha\,.\,M\,[\alpha]) \underset{\eta}{=} M$, α not free in M, of type abstractions. We write $M \underset{\lambda}{=} N$ if M is $\beta\eta$-equivalent to N in this extended sense.

During the remainder of the paper, we will make use of some fundamental properties of the calculus whose proofs can be found elsewhere (see, for example, [5]) or follow immediately from known results. We state here only a few of them.

Theorem 3 [Girard] (Basic properties of F_ω)

1. *If* $\Gamma \vdash M \in \alpha$ *then* $\Gamma \vdash \alpha \in \mathrm{Type}$.
2. *If* $\Gamma \vdash \alpha \in K$ *then* α *has a unique* $\beta\eta$*-normal form.*
3. *If* $\Gamma \vdash M \in \alpha$ *then* M *has a unique* $\beta\eta$*-normal form.*
4. $\Gamma \vdash M \in \alpha$ *is decidable.*

4 Pure LEAP

In order to be able to give a finitary definition of `reify` and `reflect` at all levels of F_ω, we need to allow global definition of types and functions with free variables ranging over kinds. Such variables are generic in the same way that some type variables are generic in ML (see Milner [12]). We will use the concrete syntax:

$$\begin{array}{lcll} \theta & \equiv & \beta & \text{global definition of } \theta \text{ to stand for } \beta \\ x & \equiv & M & \text{global definition of } x \text{ to stand for } M \end{array}$$

for global definitions of types and terms, respectively. This addition to F_ω is benign in the sense that given any term M to be type-checked and evaluated in a given global context, we can find an equivalent term N in F_ω itself. N is obtained from M simply by expanding the definitions from the context. This is also how type-checking and evaluation for Pure LEAP are defined. Later, if the language is extended to allow side-effects, and a commitment to call-by-value is made, evaluation must be reconsidered. In Pure LEAP, every term will have a unique normal form, so the issue of a call-by-value or call-by-name semantics does not arise.

5 Reflection in LEAP

We now describe the representation of programs in Pure LEAP, and present our definitions of `reify`, `reflect`, and `eval`.

5.1 Representation of programs

When attempting to build a reflexive language, the first concern must be the ability to represent programs in the language as data. Two approaches seem plausible: to build in a new special data type for programs, or to use combinations of existing built-in data types to represent programs. Since we would like (at the outset) to keep our language as pure as possible, we will follow the latter approach.

Perhaps the best way to understand this construction is in terms of *inductively defined types*. An inductively defined type is given by a list of its "constructors" and their types. This is an extension of the **datatype** construction in ML, since constructors may be explicitly polymorphic. It is shown in [13] (extending ideas of Böhm & Berarducci [1]) that these types do not require an addition to the core language, since inductively defined types are *representable* by closed types. With this in mind, we can now present a specification of the type of programs:

indtype π : Type $\Rightarrow$ Type **with**
  `rep` : $\Delta\alpha$:Type . $\alpha \Rightarrow \pi\,\alpha$
  `lam` : $\Delta\alpha$:Type . $\Delta\beta$:Type . $(\alpha \Rightarrow \pi\,\beta) \Rightarrow \pi\,(\alpha \Rightarrow \beta)$
  `app` : $\Delta\alpha$:Type . $\Delta\beta$:Type . $\pi\,(\alpha \Rightarrow \beta) \Rightarrow \pi\,\alpha \Rightarrow \pi\,\beta$
  `typlam` : $\Delta\alpha$:Type $\rightarrow$ Type . $(\Delta\beta$:Type . $\pi\,(\alpha\,\beta)) \Rightarrow \pi\,(\Delta\beta$:Type . $(\alpha\,\beta))$
  `typapp` : $\Delta\alpha$:Type $\rightarrow$ Type . $\pi\,(\Delta\beta$:Type . $(\alpha\,\beta)) \Rightarrow \Delta\beta$:Type . $\pi\,(\alpha\,\beta)$
  end

The basic problem is to be able to explicitly define a function π from types to types, such that $\pi\alpha$ is a type representing programs of type α. The usual, well-known approach for defining inductive data types in the second-order polymorphic λ-calculus (see [1,15]) fails, but we do not have a proof that such a representation is impossible. The data types that have been shown to be representable in F_2 either have constructors that are not polymorphic (such as $\mathsf{int} \equiv \Delta\alpha\,.\,\alpha \Rightarrow (\alpha \Rightarrow \alpha) \Rightarrow \alpha$, which has constructors 0:int and succ:int $\Rightarrow$ int), or have the property that the type variables in the constructor are uniform over the whole data type (such as $\mathsf{list} \equiv \lambda\alpha\,.\,\Delta\theta\,.\,(\alpha \Rightarrow \theta \Rightarrow \theta) \Rightarrow \theta \Rightarrow \theta$ with constructors cons:$\Delta\theta\,.\,\theta \Rightarrow \mathsf{list}\,\theta \Rightarrow \mathsf{list}\,\theta$ and nil:$\Delta\theta\,.\,\mathsf{list}\,\theta$). This allows the definitions of the constructors to be uniform over this type variable.

An attempt at a straightforward extension of this approach to the case of a data type of programs fails, since a program of type β may have components of type $\alpha \Rightarrow \beta$ and α, and thus in fact of arbitrary type.

This problem disappears when one goes to the third-order polymorphic λ-calculus, since in it one can explicitly use a function from types to types that maps the type of the

components to the type of a term. We will begin the formalization of these ideas by giving an F_3 encoding of F_2 programs. Each line is annotated with a corresponding constructor function that is defined below. We use Θ for a bound variable of kind Type $\to$ Type, that is, for a function from types to types.

$$
\begin{array}{lll}
\pi \equiv & \lambda\gamma\,.\,\Delta\Theta{:}\mathrm{Type}\to\mathrm{Type}\,. & \\
& (\Delta\alpha\,.\,\alpha\Rightarrow\Theta\,\alpha)\Rightarrow & \texttt{(* rep *)} \\
& (\Delta\alpha\,\Delta\beta\,.\,(\alpha\Rightarrow\Theta\,\beta)\Rightarrow\Theta\,(\alpha\Rightarrow\beta))\Rightarrow & \texttt{(* lam *)} \\
& (\Delta\alpha\,\Delta\beta\,.\,\Theta\,(\alpha\Rightarrow\beta)\Rightarrow\Theta\,\alpha\Rightarrow\Theta\,\beta))\Rightarrow & \texttt{(* app *)} \\
& (\Delta\alpha{:}\mathrm{Type}\to\mathrm{Type}\,.\,(\Delta\beta\,.\,\Theta\,(\alpha\,\beta))\Rightarrow\Theta\,(\Delta\beta\,.\,\alpha\,\beta))\Rightarrow & \texttt{(* typlam *)} \\
& (\Delta\alpha{:}\mathrm{Type}\to\mathrm{Type}\,.\,\Theta\,(\Delta\beta\,.\,\alpha\,\beta)\Rightarrow(\Delta\beta\,.\,\Theta\,(\alpha\,\beta)))\Rightarrow & \texttt{(* typapp *)} \\
& \Rightarrow\Theta\,\gamma &
\end{array}
$$

This is a special case of a very general transformation from an inductive definition of a data type into an encoding into F_ω described in [13]. The definitions of the constructors in this encoding can be found in Figure 1.

$$
\begin{array}{lll}
\texttt{rep} & : & \Delta\alpha\,.\,\alpha\Rightarrow\pi\,\alpha \\
\texttt{rep} & \equiv & \Lambda\alpha\,\lambda x{:}\alpha\,. \\
& & \Lambda\Theta\,\lambda \mathit{rep}\,\lambda \mathit{lam}\,\lambda \mathit{app}\,\lambda \mathit{typlam}\,\lambda \mathit{typapp}\,. \\
& & \mathit{rep}\,[\alpha]\,x \\
 & & \\
\texttt{lam} & : & \Delta\alpha\,\Delta\beta\,.\,(\alpha\Rightarrow\pi\,\beta)\Rightarrow\pi\,(\alpha\Rightarrow\beta) \\
\texttt{lam} & \equiv & \Lambda\alpha\,\Lambda\beta\,\lambda f{:}\alpha\Rightarrow\pi\,\beta\,. \\
& & \Lambda\Theta\,\lambda \mathit{rep}\,\lambda \mathit{lam}\,\lambda \mathit{app}\,\lambda \mathit{typlam}\,\lambda \mathit{typapp}\,. \\
& & \mathit{lam}\,[\alpha]\,[\beta]\,(\lambda x{:}\alpha\,.\,f\,x\,[\Theta]\,\mathit{rep}\,\mathit{lam}\,\mathit{app}\,\mathit{typlam}\,\mathit{typapp}) \\
 & & \\
\texttt{app} & : & \Delta\alpha\,\Delta\beta\,.\,\pi\,(\alpha\Rightarrow\beta)\Rightarrow\pi\,\alpha\Rightarrow\pi\,\beta \\
\texttt{app} & \equiv & \Lambda\alpha\,\Lambda\beta\,\lambda x{:}\pi(\alpha\Rightarrow\beta)\,\lambda y{:}\pi\alpha\,. \\
& & \Lambda\Theta\,\lambda \mathit{rep}\,\lambda \mathit{lam}\,\lambda \mathit{app}\,\lambda \mathit{typlam}\,\lambda \mathit{typapp}\,. \\
& & \mathit{app}\,[\alpha]\,[\beta]\,(x\,[\Theta]\,\mathit{rep}\,\mathit{lam}\,\mathit{app}\,\mathit{typlam}\,\mathit{typapp})\,(y\,[\Theta]\,\mathit{rep}\,\mathit{lam}\,\mathit{app}\,\mathit{typlam}\,\mathit{typapp}) \\
 & & \\
\texttt{typlam} & : & \Delta\alpha{:}\mathrm{Type}\to\mathrm{Type}\,.\,(\Delta\beta\,.\,\pi\,(\alpha\,\beta))\Rightarrow\pi\,(\Delta\beta\,.\,\alpha\,\beta) \\
\texttt{typlam} & \equiv & \Lambda\alpha{:}\mathrm{Type}\to\mathrm{Type}\,\lambda f{:}\Delta\beta\,.\,\pi\,(\alpha\,\beta)\,. \\
& & \Lambda\Theta\,\lambda \mathit{rep}\,\lambda \mathit{lam}\,\lambda \mathit{app}\,\lambda \mathit{typlam}\,\lambda \mathit{typapp}\,. \\
& & \mathit{typlam}\,[\alpha]\,(\Lambda\beta\,.\,f\,[\beta][\Theta]\,\mathit{rep}\,\mathit{lam}\,\mathit{app}\,\mathit{typlam}\,\mathit{typapp}) \\
 & & \\
\texttt{typapp} & : & \Delta\alpha{:}\mathrm{Type}\to\mathrm{Type}\,.\,\pi\,(\Delta\beta\,.\,\alpha\,\beta)\Rightarrow(\Delta\beta\,.\,\pi\,(\alpha\,\beta)) \\
\texttt{typapp} & \equiv & \Lambda\alpha{:}\mathrm{Type}\to\mathrm{Type}\,\lambda f{:}\pi\,(\Delta\beta\,.\,\alpha\,\beta)\,\Lambda\beta\,. \\
& & \Lambda\Theta\,\lambda \mathit{rep}\,\lambda \mathit{lam}\,\lambda \mathit{app}\,\lambda \mathit{typlam}\,\lambda \mathit{typapp}\,. \\
& & \mathit{typapp}\,[\alpha]\,(f\,[\Theta]\,\mathit{rep}\,\mathit{lam}\,\mathit{app}\,\mathit{typlam}\,\mathit{typapp})\,[\beta]
\end{array}
$$

Figure 1: Definition of program constructors for F_2 in F_3.

Several things should be noted in this definition:

1. Representations of programs are not unique. That is, any program M in normal form can be represented as $\mathtt{rep}\,[\alpha]\,M$ (α the type of M), but it also has a representation in terms of `lam`, `app`, `typlam`, `typapp`, and `rep`, where `rep` is applied only to variables.

2. The `rep` constructor can not be eliminated, since it is crucial in order to convert bound variables into their representations. We do not see a simple way of fixing this by changing the type of the `lam` constructor to $\Delta\alpha\ \Delta\beta\ .\ (\pi\,\alpha \Rightarrow \pi\,\beta) \Rightarrow \pi\,(\alpha \Rightarrow \beta)$, since that seems to preclude a representation of `lam`.

5.2 Reification and reflection

In the definition and theorems below we will omit contexts. They can be filled in easily.

Definition 4 (Program representation) *Let M be a term of F_2. We define the* standard representation $\overline{M}$ *of M in F_3 inductively as follows:*

If $x \in \alpha$ *then*	$\overline{x}$	$=$	$\mathtt{rep}\,[\alpha]\,x$
If $\lambda x{:}\alpha\ .\ M \in \alpha \Rightarrow \beta$ *then*	$\overline{\lambda x{:}\alpha\ .\ M}$	$=$	$\mathtt{lam}\,[\alpha]\,[\beta]\,(\lambda x{:}\alpha\ .\ \overline{M})$
If $M \in \alpha \Rightarrow \beta$ *and* $N \in \alpha$ *then*	$\overline{M\,N}$	$=$	$\mathtt{app}\,[\alpha]\,[\beta]\,\overline{M}\,\overline{N}$
If $\Lambda\beta\ .\ M \in \Delta\beta\ .\ \alpha\,\beta$ *then*	$\overline{\Lambda\beta\ .\ M}$	$=$	$\mathtt{typlam}\,[\alpha]\,(\Lambda\beta\ .\ \overline{M})$
If $M\,[\beta] \in \alpha\,\beta$ *then*	$\overline{M\,[\beta]}$	$=$	$\mathtt{typapp}\,[\alpha]\,\overline{M}\,[\beta]$

*We define the relation "*represents*" inductively like the standard representation, except that* $\mathtt{rep}\,[\alpha]\,M$ *(which is not the standard representation of any term unless M is a variable) is defined as representing M.*

The following theorem shows that this is a proper representation function, but the crucial property will of course be that evaluation is definable over this representation (see Theorem 8).

Theorem 5 (Soundness of program representation) *Let $N \in \alpha$. Then $\overline{N} \in \pi\,\alpha$.*

Proof: By a simple induction on the structure of N.

□

Conjecture 6 (Faithfulness of program representation) *Let N be a term of type $\pi\,\alpha$. Then there is an $M \in \alpha$ such that N represents M.*

It should be noted that this conjecture is not critical for the further development of program representation and evaluation in the remainder of this paper. Should it turn out that there are terms of type $\pi\,\alpha$ which are not the representation of programs of type α, the representation of the functions defined below are still correct on terms that *are* representation of programs, and will again produce representations of programs.

5.3 The definition of `reflect`

The crucial step in the definition of `eval` is the definition of `reflect`, which maps the representation of a term of type α into a term of type α, that is, `reflect`: $\Delta\alpha\ .\ \pi\,\alpha \Rightarrow \alpha$. Such a function will have to do some form of evaluation, since normal-form terms of type $\pi\,\alpha$ can represent terms of type α that are not in normal form.

Let us first present the function in the form of an *iterative definition* (see [1] for a discussion of iterative definitions in F_2 and [13] for a generalization that encompasses F_ω).

$$\begin{array}{lcl}
\texttt{reflect}\,[\alpha]\,(\texttt{rep}\,[\alpha]\,x) & = & x \\
\texttt{reflect}\,[\alpha \Rightarrow \beta]\,(\texttt{lam}\,[\alpha]\,[\beta]\,x) & = & \lambda y{:}\alpha\ .\ \texttt{reflect}\,[\beta]\,(x\,y) \\
\texttt{reflect}\,[\beta]\,(\texttt{app}\,[\alpha]\,[\beta]\,x\,y) & = & (\texttt{reflect}\,[\alpha \Rightarrow \beta]\,x)\,(\texttt{reflect}\,[\alpha]\,y) \\
\texttt{reflect}\,[\Delta\beta{:}\text{Type}\ .\ \alpha\,\beta]\,(\texttt{typlam}\,[\alpha]\,x) & = & \Delta\beta{:}\text{Type}\ .\ \texttt{reflect}\,[\alpha\,\beta]\,(x\,[\beta]) \\
\texttt{reflect}\,[\alpha\,\beta]\,(\texttt{typapp}\,[\alpha]\,x\,[\beta]) & = & \texttt{reflect}\,[\Delta\beta{:}\text{Type}\ .\ \alpha\,\beta]\,x\,[\beta]
\end{array}$$

Note that x and y are object language variables ranging over terms, and that α and β are object language type variables. These variables are essentially bound over the body of the iterative definition.

Iteratively defined functions over inductively defined types turn out to be representable in F_ω. In this case the explicit definition of `reflect` is surprisingly simple. This explicit definition highlights the fact that a program is represented as its own iteration function—iteration is achieved by applying the representation of a program to each of the cases from an iterative definition. Let $\texttt{id} \equiv \Delta\alpha\ \lambda x{:}\alpha\ .\ x$ be the polymorphic identity. Then we get in this case:

$$\begin{array}{lcl}
\texttt{reflect} & : & \Delta\gamma\ .\ \pi\,\gamma \Rightarrow \gamma \\
\texttt{reflect} & \equiv & \Lambda\gamma\ \lambda p{:}\pi\,\gamma\ . \\
 & & \quad p\,[\lambda\delta\ .\ \delta] \\
 & & \quad (\Lambda\alpha\ .\ \texttt{id}\,[\alpha]) \\
 & & \quad (\Lambda\alpha\ \Lambda\beta\ .\ \texttt{id}\,[\alpha \Rightarrow \beta]) \\
 & & \quad (\Lambda\alpha\ \Lambda\beta\ .\ \texttt{id}\,[\alpha \Rightarrow \beta]) \\
 & & \quad (\Lambda\alpha{:}\text{Type} \to \text{Type}\ .\ \texttt{id}\,[\Delta\beta\ .\ \alpha\,\beta]) \\
 & & \quad (\Lambda\alpha{:}\text{Type} \to \text{Type}\ .\ \texttt{id}\,[\Delta\beta\ .\ \alpha\,\beta])
\end{array}$$

Theorem 7 (Correctness of `reflect`) *Let $N \in \pi\,\alpha$ be some (not necessarily standard) representation of the term M. Then* $\texttt{reflect}\,N \underset{\lambda}{=} M$.

Proof: By induction on the normal form of N in terms of the constructors of π.

□

5.4 The definitions of `reify` and `eval`

Given the definition of `reflect`, it is a simple matter to give the definition of $\texttt{eval}{:}\pi\,\alpha \Rightarrow \pi\,\alpha$. Intuitively, `eval` should take the representation of a term and return a representation of its normal form. This is achieved simply by composing reflection with representation. This

definition (given formally below) will *not* return the standard representation of the normal form of the term, but rather exploit the fact that every normal form term M can be represented as $\mathtt{rep}\,M$.

$$\begin{array}{rcl} \mathtt{reify} & : & \Delta\alpha\,.\,\alpha \Rightarrow \pi\,\alpha \\ \mathtt{reify} & \equiv & \mathtt{rep} \\ \mathtt{eval} & : & \Delta\alpha\,.\,\pi\,\alpha \Rightarrow \pi\,\alpha \\ \mathtt{eval} & \equiv & \Lambda\alpha\,\lambda x{:}\pi\,\alpha\,.\,\mathtt{reify}\,[\alpha]\,(\mathtt{reflect}\,[\alpha]\,x) \end{array}$$

Theorem 8 (Correctness of `eval`) *Let $N \in \pi\,\alpha$ be some (not necessarily standard) representation of the term M. Then* $\mathtt{eval}\,[\alpha]\,N \in \pi\,\alpha$ *is a representation of the normal form of M.*

We do not have a simple and intuitive characterization of exactly which functions are definable over the given representation of programs. In particular, we do not know whether the apparently simpler one-step outermost β-reduction is representable. The problem is that the first argument to `lam` expects a function of type $\alpha \Rightarrow \pi\,\beta$, not of type $\pi\,\alpha \Rightarrow \pi\,\beta$. One-step call-by-value reduction is an example of another function (beside evaluation) that *is* definable, that is, we can evaluate the argument to a top-level β-redex and then perform one outermost reduction.

5.5 Generalizing to higher types

We will now generalize the definition of π to allow representation of programs in F_ω. Note that a term representing a program in F_n will be in F_{n+1}.

$$\begin{array}{rcll} \pi & \equiv & \lambda\gamma\,.\,\Delta\Theta{:}\mathrm{Type} \to \mathrm{Type}\,. & \\ & & (\Delta\alpha\,.\,\alpha \Rightarrow \Theta\,\alpha) \Rightarrow & (*\ \mathtt{rep}\ *) \\ & & (\Delta\alpha\,\Delta\beta\,.\,(\alpha \Rightarrow \Theta\,\beta) \Rightarrow \Theta\,(\alpha \Rightarrow \beta)) \Rightarrow & (*\ \mathtt{lam}\ *) \\ & & (\Delta\alpha\,\Delta\beta\,.\,\Theta\,(\alpha \Rightarrow \beta) \Rightarrow \Theta\,\alpha \Rightarrow \Theta\,\beta)) \Rightarrow & (*\ \mathtt{app}\ *) \\ & & (\Delta\alpha{:}K \to \mathrm{Type}\,.\,(\Delta\beta{:}K\,.\,\Theta\,(\alpha\,\beta)) \Rightarrow \Theta\,(\Delta\beta{:}K\,.\,\alpha\,\beta)) \Rightarrow & (*\ \mathtt{typlam}\ *) \\ & & (\Delta\alpha{:}K' \to \mathrm{Type}\,.\,\Theta\,(\Delta\beta{:}K'\,.\,\alpha\,\beta) \Rightarrow (\Delta\beta{:}K'\,.\,\Theta\,(\alpha\,\beta))) \Rightarrow & (*\ \mathtt{typapp}\ *) \\ & & \Rightarrow \Theta\,\gamma & \end{array}$$

This definition and the corresponding definitions of the constructor functions are now parameterized over the kinds K and K'. Since definitions with $\equiv$ are viewed as global, these kind variables are generic and may be instantiated differently at different occurrences of π. This is a part of the language where full reflexivity fails, since $\equiv$ cannot be represented in LEAP.

6 Extending Pure LEAP to LEAP

We now turn our attention to extending Pure LEAP to the full "LEAP core language." Our goal here is to incorporate useful features of functional languages while adhering to the

principle of reflexivity. Specifically, in order to arrive at full LEAP, we make extensions in two phases: first those which can be defined entirely within Pure LEAP and hence constitute only conservative, *syntactic* extensions, and then the nonconservative, *semantic* extensions to Pure LEAP which still preserve reflexivity.

6.1 Syntactic Extensions

We begin with a brief description of the syntactic extensions.

6.1.1 Partial type inference

Explicit polymorphism makes Pure LEAP impractically verbose; a type inference system for the language is essential. *Partial type inference* allows the types of bound variables and the type arguments to terms to be omitted, but type abstractions and placeholders for type arguments (denoted by $[\,]$) must be supplied. For example, self-application may be written as $\lambda x\,.\,(x\,[\,]\,x)$, but *not* as $\lambda x\,.\,(x\,x)$. Partial type inference would type-check the former, but not the latter.

In [14], Pfenning shows that the partial type inference problem for F_ω (and hence LEAP) is undecidable, but also gives a complete semi-decision procedure based on higher-order unification. More extensive experiments are necessary in order to gauge the practicality of this algorithm. Our current prototype uses a λProlog [10] implementation of this algorithm, with very encouraging preliminary results.

6.1.2 Generic polymorphism and the * syntax

In the λ-calculus, the construction $\texttt{let}\ x = N\ \texttt{in}\ M$ is taken as an abbreviation for $(\lambda x\,.\,M)N$. The enhanced legibility of the shorthand is due to the lexical proximity of the x and N. In this form, the `let` construct can be carried over into LEAP in unadulterated form.

However, in ML the `let` construct is a convenient and critically important device for establishing *generic polymorphism*.

Thus, for example,

$$\texttt{let}\ f = \lambda x\,.\,x\ \texttt{in}\ (f\,1, f\,\texttt{true})$$

in ML is type-correct, since $\lambda x\,.\,x$ has principle type $\alpha \Rightarrow \alpha$ for a type variable α and this type variable may be instantiated differently at different occurrences of f in the scope of the binding on f (and is thus called *generic*). Hence `let` cannot be treated merely as syntactic sugar, since the expanded version of the example above,

$$(\lambda f\,.\,(f\,1, f\,\texttt{true}))\,(\lambda x\,.\,x)$$

is not type-correct.

This genericity reduces reflexivity since it seems to be impossible for type-checking with generic type variables to be inherited. We are left, then, with the problem of recovering the programming convenience of ML's `let` without destroying the reflexivity of the language.

The solution we propose introduces additional verbosity over ML, which fortunately can be "sweetened" with some syntactic sugar.

We would rewrite the example above in LEAP as follows:

$$\texttt{let } f* = \Delta\alpha\,.\,\lambda x{:}\alpha\,.\,x \texttt{ in } (f\,1, f\,\texttt{true})$$

Here the "starred" identifier, $f*$, is defined in the body of the `let` term. The single star is a purely syntactic, macro-like feature which in this case specifies that occurrences of the variable f (without the star) are to be macro-expanded into the term $f*[\,]$.

We adopt this as a general syntactic feature of LEAP so that whenever $x*\ldots*$ is defined, in-scope occurrences of x appearing without a type argument are automatically expanded to $x*\ldots*[\,]\ldots[\,]$, where the number of *'s matches the number of [ ]'s. This essentially "syntactifies" generic polymorphism without giving up much expressive convenience (and still preserving reflexivity). The additional verbosity over ML occurs at the place where a polymorphic function is defined, since type abstractions must be made explicit. However, functions are typically used much more often than defined, and so this overhead does not seem an undue burden.

Taking the example of `eval` and the π-constructors from the previous section, we can replace `eval` with `eval*`, `rep` with `rep*`, `lam` with `lam**`, and so on, in order to make `eval` and the π-constructors to appear "generically" polymorphic.

6.1.3 Primitive recursion and inductively-defined data types

In [15], Reynolds gives several examples of encodings of inductively-defined data types in the second-order polymorphic λ-calculus. Among the examples are integers, lists, and trees. Nonrecursive data types such as the unit type, pairs, and disjoint sums can also be encoded in a similar manner as special cases of the general encoding. These encodings require only the second order, and can be transferred directly into Pure LEAP. Our encoding of the the type of program representations, π, is an example of such an encoding that seems to require functions from types to types, *i.e.*, the third-order polymorphic λ-calculus.

For a practical language, such encodings are much too unwieldy. Hence, we make a syntactic extension to Pure LEAP which provides a sublanguage for inductively-defined type specifications. An example of such a specification appears in Section 5.1, where we define the type π using this syntactic extension. A full discussion of the definition of primitive recursion and inductively-defined data types in Pure LEAP is given in another paper [13].

6.2 Semantic Extensions

Several features found in languages such as Standard ML can not be defined simply through syntactic extension of Pure LEAP. These include general recursion, polymorphic assignable references, and polymorphic exceptions (or `call/cc`).

In all three cases, it appears to be possible to incorporate these features into the language by adding new constants which embody the desired semantics. Having chosen the constants,

it remains for us only to verify that reflexivity is not violated by the extensions. For polymorphic references and exceptions, we have found that the explicit polymorphism in Pure LEAP with suitable restrictions which can be easily checked, provide an extra degree of control which eliminates the need for "weak" [7] or "imperative" [21] type variables.

7 Conclusions

As we stated in the introduction, our original goal was to design an practical, statically-typed language suitable for use as a metalanguage for manipulating programs, proofs, and other similar symbolic data. What we have attained is Pure LEAP, a statically-typed language core which admits the definition of a metacircular interpreter for a large language fragment in a natural and direct way. This language is based on the ω-order polymorphic λ-calculus of Girard, extended by global definitions and some syntactic sugar. In what ways does Pure LEAP satisfy our original goal? In other words, how well does Pure LEAP serve as a metalanguage?

Of course, without a serious implementation we can only speculate on this question, but almost any argument that might be made for ML as a metalanguage can also be made for LEAP. In addition, Pure LEAP is able to represent and manipulate data (*e.g.*, programs in object languages) with richer type structures than is possible in ML. How useful this added power is in practice will require much further investigation and experience with the language.

Other issues to be studied further include the exact extent of the language, in particular with respect to additions such as references, exceptions, recursion, and so on. We have done some preliminary work along these lines, and have some evidence that such extensions will not destroy the reflexivity of the language. Another issue is the efficient implementation of LEAP. Work here is presently underway, with a simple implementation based on λProlog currently operational. One of the main challenges appears to be devising efficient implementation strategies for inductively-defined data types.

We hope to have more to report as the design and implementation of a full language around Pure LEAP proceeds.

Acknowledgements

The authors would like to thank Christine Paulin-Mohring for pointing out a problem in a purported "proof" of Conjecture 6, and also Ken Cline, Scott Dietzen, Spiro Michaylov, and Benjamin Pierce for many helpful discussions about Pure LEAP.

References

[1] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[2] Alonzo Church. *The Calculi of Lambda-Conversion.* Princeton University Press, Princeton, New Jersey, 1941.

[3] Mitchell D. Wand and Daniel P. Friedman. The mystery of the tower revealed: a non-reflective description of the reflective tower. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming, Cambridge*, pages 198–307, ACM, August 1986.

[4] Daniel P. Friedman and Mitchell Wand. Reification: reflection without metaphysics. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 348–355, ACM Press, August 1984.

[5] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordere supérieur.* PhD thesis, Université Paris VII, 1972.

[6] Jean-Yves Girard. Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92, North-Holland Publishing Co., Amsterdam, London, 1971.

[7] David B. MacQueen. References and weak polymorphism. 1988. Standard ML of New Jersey compiler release notes.

[8] John McCarthy. History of LISP. *ACM SIGPLAN Notices*, 13(8):217–223, August 1978.

[9] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, 1962.

[10] Dale A. Miller and Gopalan Nadathur. Higher-order logic programming. In *Proceedings of the Third International Conference on Logic Programming*, Springer Verlag, July 1986.

[11] Robin Milner. The Standard ML core language. *Polymorphism*, II(2), October 1985. Also Technical Report ECS-LFCS-86-2, University of Edinburgh, Edinburgh, Scotland, March 1986.

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

[13] Frank Pfenning. *Inductively Defined Types in the Calculus of Constructions.* Ergo Report 88–069, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1988.

[14] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, ACM Press, July 1988.

[15] John Reynolds. Three approaches to type structure. In Hartmut Ehrig, Christiane Floyd, Maurice Nivat, and James Thatcher, editors, *Mathematical Foundations of Software Development*, pages 97–138, Springer-Verlag LNCS 185, March 1985.

[16] John Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, Springer-Verlag LNCS 19, New York, 1974.

[17] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740, ACM, New York, 1972.

[18] Brian Cantwell Smith. *Reflection and Semantics in a Procedural Language.* Technical Report MIT-LCS-TR-272, Massachusetts Institute of Technology, Cambridge, Massachusetts, January 1982.

[19] Brian Cantwell Smith. Reflection and semantics in Lisp. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City*, pages 23–35, ACM, January 1984.

[20] Guy Steele and G. Sussman. *The Art of the Interpreter, or, The Modularity Complex (Parts Zero, One, and Two).* Artificial Intelligence Laboratory Memo AIM-453, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1978.

[21] Mads Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, Department of Computer Science, Edinburgh University, 1987.

[22] Mitchell D. Wand and Daniel P. Friedman. The mystery of the tower revealed: a nonreflective description of the reflective tower. *Lisp and Symbolic Computation*, 1(1):11–38, June 1988.