

Towards a Practical Programming Language Based on the Polymorphic Lambda Calculus

Extended Abstract

Peter Lee, Mark Leone, Spiro Michaylov, and Frank Pfenning

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890, U.S.A.

1 Introduction

The value of polymorphism in programming languages has been demonstrated by languages such as ML [19]. Recent efficient implementations of ML have shown that a language with implicit polymorphism can be practical [1]. The core of ML's type system is limited, however, by the fact that only instances of polymorphic functions may be passed as arguments to other functions, but not the polymorphic functions themselves. This has serious practical consequences, particularly for modular programming. For this reason, ML incorporates the generic **let** construct which cannot be defined in terms of λ -abstraction and other extensions to the basic Hindley-Milner type system [12, 18], such as the module system [16]. More powerful type systems allowing explicit quantification over types, such as the second-order polymorphic λ -calculus [11, 27] (which we refer to as F_2), are free of such limitations. Thus, it is natural to ask whether they too can form the basis for practical programming languages.

A number of efforts in language design have sought to accomplish this by augmenting or adapting existing languages and systems with aspects of F_2 . In the design of the language FX-89 [13], for example, ML's type system is essentially extended with "open" and "close" constructs that provide a means of converting between explicitly quantified and generic types. Other design efforts which also extend a basic type system with some form of type abstraction include Pebble [5] and Russell [9].

We are taking an alternative approach in which a core calculus with a type system possessing the desired characteristics (in this case the ability to abstract and quantify over types explicitly) is extended to a practical programming language. The extensions must be made carefully so that the "essence" of the core calculus and its beneficial theoretical properties are preserved. The core calculus used as our starting point is F_ω , the ω -order polymorphic λ -calculus [11] (highlights of which we will briefly describe later). One of the attractive properties of pure F_ω is that there is a simple and natural logic (higher-order logic) and a natural type theory (the Calculus of Constructions) in which we can develop or reason about programs (see [15] and [8, 21], respectively). Systems like

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 implementation of the Calculus of Constructions [10] tend to emphasize program construction by a process akin to constructive theorem proving, while the underlying programming language (in this case F_ω) is rather underdeveloped and impractical. Our aim is to make the underlying language more practical and manageable.

Though in many respects similar to our effort, Quest [6] is much more ambitious in the range of constructs it includes, such as subtypes and bounded quantification. This makes it much more difficult to reason about and compile, and, at present, no logic for specification or reasoning about Quest programs has been developed.

Extending F_ω to a practical language introduces a number of challenging problems in both design and implementation. For example, data types and case expressions are definable within the core calculus [25], which means that these features can in principle be defined as syntactic sugar. However, definitions of data types, especially at higher order, are quite complex, and hence it is difficult to carry out such extensions. Furthermore, the problem of type reconstruction (the problem of filling in omitted type information) is known to be only semi-decidable [2, 22], while little is known about the related problem of type inference. Also, there are theoretical results indicating that a functional implementation of datatypes leads to inherent inefficiencies [20, 7]. Thus, a large part of our research is devoted to addressing these problems.

In this paper we report on our work in progress, which has thus far resulted in the design and prototype implementation of pure LEAP¹, a conservative (semantically equivalent) extension of F_ω . Some aspects of LEAP are discussed in [24, 26, 25]. One way to view our work is that it provides a target language for program development techniques using higher-order logic or the Calculus of Constructions, yet is readable, and can be efficiently compiled. The result of compilation currently is Common Lisp code. However, our long range plan is to design an extended LEAP language, which is no longer a conservative extension of F_ω but includes unrestricted recursion and side-effects. Such a language would no longer allow formal reasoning about its programs in such a simple and direct way, but its run-time implementation problems are much better understood. Extended LEAP would serve as the target language for compilation of pure LEAP.

The remainder of the paper is organized as follows. We begin with a brief introduction to the polymorphic λ -calculus. Then we outline the implementation of pure LEAP, including the structure of the implementation, issues of type reconstruction and type-argument synthesis, inductively defined types, and other conservative extensions. Finally we discuss the problems and strategies of compiling pure LEAP to Common Lisp code.

2 Explicit Polymorphism

This section serves two purposes. First, it provides a cursory review of the polymorphic λ -calculus. This calculus was chosen as the core of LEAP because its type system allows explicit quantification over types. There are a number of thorough treatments of variants of this calculus in the literature, for instance [11] and [27]. Here we will give only a few highlights, concentrating on the well-known connections to programming languages. Our second purpose is to introduce the syntax of LEAP as accepted by our prototype implementation.

Let us begin by defining the second-order polymorphic λ -calculus, using LEAP's syntax. In the following grammar, we use the metavariables E for expressions, x for variables, T for types, and A

¹ A Language with Eval And Polymorphism.

for type variables.

Definition 1 F_2 , the Second-order Polymorphic λ -Calculus (in LEAP syntax).

$$E ::= x \mid \text{lam } x:T. E \mid (E E') \mid \text{Lam } A. E \mid E [T]$$

$$T ::= A \mid T \rightarrow T' \mid \text{Del } A. T$$

In this calculus, polymorphism is introduced by explicitly quantifying type variables. This is unlike ML, where type abstraction is always implicit. So, for example, in Standard ML the polymorphic identity function is written as follows

```
<SML> val id = fn x => x;
      val id = fn : 'a -> 'a
```

whereas in F_2 , it is expressed like this:

```
<F2> define id = Lam A. lam x:A. x;
      id : Del A. A -> A
```

The function `id` accepts a type argument, `A`, as specified by the type abstraction `Lam`. Type abstractions give rise to quantified types denoted by Reynolds' Δ operator, written as `Del` (and oftentimes appearing in the literature as \forall).

As is well known, these two versions of `id` are not quite the same. In ML the following fails to type-check:

```
<SML> (fn f => (f 1 , f true)) id;
      Error: f true : type mismatch.
```

despite the fact that `id` is polymorphic, whereas in F_2 , because of the explicit quantification in the type of `id`, we can write²

```
<F2> (lam f:(Del A. A->A). pair [Nat] [Bool] (f [Nat] 1) (f [Bool] true)) id;
```

where the type arguments to function `f` are enclosed in square brackets. The problem in ML is that the type variables in polymorphic arguments to functions can be instantiated only once, whereas in F_2 the instantiation of type variables can be controlled by the application of type abstractions to different types in different contexts.

The additional expressive power in F_2 comes, it seems, at the expense of a considerable amount of notational baggage. We will see later that one of our principles in the design of LEAP has been to alleviate this by incorporating type reconstruction as well as by judicious use of syntactic sugar.

In Standard ML this inability to abstract over polymorphic functions is circumvented somewhat by providing a generic `let` construct. So, this example can be written in ML as follows:

```
<SML> let val f = fn x => x in (f 1 , f true) end;
```

However, this means that the “argument” `f` must be known in advance. The function

²In an environment where `Nat`, `Bool`, `id`, `pair`, and `1` are appropriately defined.

```
<SML> fn f => (f 1 , f true);
```

still is not typable.

Extending beyond F_2 to F_ω , the notion of *kinds* is introduced. In the following grammar, the metavariable K ranges over kinds.

Definition 2 F_ω , the ω -order Polymorphic Typed λ -Calculus (in LEAP syntax).

$$\begin{aligned} E & ::= x \mid \text{lam } x:T. E \mid (E E') \mid \text{Lam } A. E \mid E [T] \\ T & ::= A \mid T \rightarrow T' \mid \text{Del } A:K. T \mid \text{LAM } A:K. T \mid T T' \\ K & ::= * \mid K \rightarrow K' \end{aligned}$$

Kinds are related to types in the same way that types are related to expression terms. All of the types in the F_2 fragment of F_ω are of kind “type,” which in LEAP is written as “*.” In addition to types of kind *, F_ω also allows “higher-order” types, which are essentially functions over types and kinds. For instance, “type functions” that map types to types have kind $* \rightarrow *$. This feature allows one to give closed term definitions for inductive data types [4, 25, 26]. For example, in ML the list type constructor can be defined as follows:

```
<SML> datatype 'a list =
  nil of 'a list |
  cons of 'a * 'a list;
```

On the other hand, it turns out that a Curried form of this data type can be defined in F_ω in closed form:

```
<Fω> Define List = LAM C:* . Del A:*->* .
  (Del B:* . A B) -> (Del B:* . B -> A B -> A B) -> (A C);
<Fω> define nil = Lam C:* . Lam A:*->* .
  lam n:(Del B:* . A B) . lam c:(Del B:* . B -> A B -> A B) .
  n [C];
<Fω> define cons = Lam C:* . lam x:A1 . lam l:List A1 . Lam A:*->* .
  lam n:(Del B:* . A B) . lam c:(Del B:* . B -> A B -> A B) .
  c [C] x (l [A] n c);
```

List has kind $* \rightarrow *$; so, for example, List Nat (*i.e.*, List applied to type Nat) gives the type of lists of natural numbers.

Again, the syntax in the F_ω definitions is exceedingly cumbersome, and their form quite obscure. However, the fact that inductive data types can be defined within this calculus suggests that an ML-style “datatype” definition facility could be incorporated into our LEAP language as a purely syntactic extension. This is the approach we have taken in LEAP.

3 The Implementation of Pure LEAP

The implementation of a programming language based on F_ω poses a number of interesting challenges. Although the type system is extremely powerful, the amount of explicit type information present in even the simplest of programs can be overwhelming. F_ω has no built-in data types or constants, so it can be laborious to express even simple arithmetic, not to mention the problem of

compiling it efficiently. Our approach in the design of LEAP has been to add features, such as a facility for defining inductive types, to F_ω by making conservative, syntactic extensions. This has resulted in an expressive and syntactically tractable language which is also semantically equivalent to F_ω .

We now present highlights of the design and implementation of pure LEAP, an extension of F_ω with global definitions, type reconstruction, type argument synthesis, kind inference, inductive datatype definitions, and function definition by primitive recursion.

3.1 Representing F_2

As in the previous section, we begin with the F_2 -based fragment of pure LEAP. The first matter to consider is the representation of the abstract syntax of LEAP programs. We chose a representation based on *higher-order abstract syntax* [23] in which all the name binding operators of the language are represented by λ -abstractions. This simplifies many of the implementation problems, since we can now take advantage of higher-order unification as used in λ Prolog to do type reconstruction. Also, issues such as renaming of bound variables are handled by λ Prolog (which is our primary implementation language, Common Lisp is the other), thus allowing us to formulate the various implementation components of LEAP very concisely and quickly.

$$\begin{aligned} E & ::= x \mid (\mathbf{lam} \ T \ (\lambda x. E)) \mid (\mathbf{app} \ E \ E') \mid (\mathbf{Lam} \ (\lambda A. E)) \mid (\mathbf{App} \ E \ T) \\ T & ::= A \mid (\mathbf{fun} \ T \ T') \mid (\mathbf{Del} \ (\lambda A. T)) \end{aligned}$$

Here, the abstract syntax operators are represented as constants, such as app, fun, or lam. Syntactic operators which bind variables have λ -abstractions as subterms. These λ -abstractions are in the abstract syntax metalanguage, and are *not* abstractions in F_2 . One important consequence of this basic representation is that *variables* in the metalanguage (λ Prolog) represent variables in the F_2 .

As an example, consider the closed-form definition of the natural numbers in F_2 :

```
Define Nat = Del A . A -> (A -> A) -> A;
define zero = Lam A . lam z:A . lam s:A->A . z;
define succ = lam n:Nat . Lam A . lam z:A . lam s:A->A . s (n [A] z s);
```

The higher-order abstract syntax representation for these is as follows:

```
Nat ≡ (Del (λA. (fun A (fun (fun A A) A))))
Zero ≡ (Lam (λA. (lam A (λz. (lam (fun A A) (λs. z))))))
Succ ≡ (lam Nat (λn. (Lam (λA. (lam A (λz. (lam (fun A A)
    λs. (app s (app (app (App n A) z) s))))))))))
```

3.2 Type Reconstruction

Now, with a higher-order abstract syntax representation for expressions and types, higher-order unification can be used to implement type reconstruction. The basic idea is to allow types to be omitted from λ -abstractions and from applications of polymorphic functions to types. In other words, the syntax given in Definition 1 is extended with the following productions:

$$E ::= \dots \mid \mathbf{lam} \ x \ . \ E \mid E \ []$$

The first production indicates that the type of a λ -variable can now be omitted. The second production shows that type arguments can now be left out, although a “placeholder” indicating that a type needs to be filled in by the reconstruction algorithm, still must be supplied.

The reconstruction algorithm we use is the one given in [22] in the language λ Prolog [17]. The algorithm is based on the unification of higher-order abstract syntax terms, which can conveniently be implemented in λ Prolog, since the latter is based on higher-order unification.

In the LEAP implementation, a predicate `typeof` written in λ Prolog reconstructs missing types in partially-typed expressions. For example, the λ Prolog clause for determining the type of an application is something like:

```
typeof (app E1 E2) T2 :- typeof E1 (fun T1 T2), typeof E2 T1.
```

That is to say, the type of E_1 applied to E_2 is T_2 if the type of E_1 is $T_1 \rightarrow T_2$ and the type of E_2 is T_1 . The type of a `lam`-abstraction is determined by the following clause:³

```
typeof (lam T1 E) (fun T1 T2) :-
  pi ( $\lambda$  x. (typeofvar x T1 => typeof (E x) T2)).
```

This clause can be read as “the type of the abstraction ‘`lam x : T1. E`’ is $T_1 \rightarrow T_2$ if for arbitrary x of type T_1 , the type of E is T_2 .” The (higher-order) unification of the construction `(E x)` ensures that the E is abstracted by a variable x of the appropriate type.

Type reconstruction for F_2 and F_ω is known to be undecidable [2, 22], but in practice our λ Prolog programs behaves well. It is not as restrictive as the algorithms in [3, 13] and we have found that the additional freedom is often useful. The main price we pay in practice is not undecidability or even efficiency, but the fact that principal type schemas no longer exist. This means we will have to go back to the user and ask for more type information in case types can be restored in several, essentially different ways (self-application as in [22] is an example of this phenomenon).

3.3 Type-argument Synthesis

Type reconstruction dramatically simplifies programming in LEAP, but it is still necessary to use `Lam` for type abstractions and write empty-bracket “placeholders” to indicate where a type argument should be reconstructed. If these restrictions are removed, we are left the problem of *type inference*. Unfortunately, it is not known whether type inference for F_ω is decidable, or if an effective procedure for type inference exists. Furthermore, even if a (semi-)decision procedure were to be found, it is not clear that it would be useful in practice.

In LEAP, we have adopted a simple syntactic extension that eliminates most of the empty type applications. When a variable is defined (either in a top-level definition or bound in a `lam`-abstraction), the programmer is permitted to specify the number of type applications that should be synthesized for each occurrence of that variable. For example, recall our earlier definitions of `nil` and `cons`:

```
define nil* = Lam C:* . ...
define cons* = Lam C:* . ...
```

³A λ Prolog goal of the form “`pi (λ X. subgoal)`” can be thought of as universal quantification: it succeeds if the subgoal succeeds for an *arbitrary* value of X . A goal of the form “`assumption => subgoal`” succeeds if the subgoal succeeds under the specified assumption.

Here, we have written single asterisks after the identifiers, indicating that in each case, one type argument is to be synthesized. So, for example, we can write `cons* [Nat] 1 (nil* [Nat])` or, using the type synthesis feature: `cons 1 nil`. The fact that `nil` and `cons` are defined with single asterisk suffixes means that occurrences of these variables without asterisks are to have synthesized type arguments. In other words, the above “macro expands” into `cons* [] 1 (nil* [])` at which point the type reconstruction phase of the LEAP implementation fills in the omitted type information.

This simple scheme has a number of advantages over previous approaches. In FX-89 [13], control over the “duality” between generic polymorphism and explicit quantification is provided by “open” and “close” primitives. For example, in FX-89 it is possible to write:

```
<FX-89> (lambda (f :  $\forall a. a \rightarrow a$ ) (open f) (pair (f 1) (f true)))
```

The “open” operation converts the explicitly quantified type for `f` to an ML-style generic type. The “close” operation performs the opposite conversion. Without syntactic sugar, one could express this program as:

```
< $F_\omega$ > lam f : Del A. A -> A. pair [Nat] [Bool] (f [Nat] 1) (f [Bool] true);
```

But in LEAP, with type reconstruction and type-argument synthesis, we can write:

```
<LEAP> lam f*. pair (f 1) (f true);
```

where we assume that `pair` was defined with a two-asterisk suffix (*i.e.*, as `pair** = ...`).

3.4 Extensions to F_ω

Text describing the extension of the representation from F_2 to F_ω and the description of “kind inference” has been omitted from this extended abstract.

3.5 Inductively Defined Types

Another important syntactic extension to F_ω that we have made for Pure LEAP is a mechanism for defining *inductive types*. Many useful data types can be defined inductively, such as the natural numbers, tuples, polymorphic lists and trees, and even F_ω programs [24]. Such a definition mechanism is crucial to LEAP’s practicality because F_ω has no primitive types or built-in constants.

An inductive type definition consists of a type name, a list of constructor names, and their types. For example, natural numbers and polymorphic lists can be defined in LEAP as follows:

```
indtype Nat:* with                                indtype List:*->* with
  zero : Nat                                       nil* : Del A. List A
  succ : Nat -> Nat;                               cons* : Del A. A -> List A -> List A;
```

There are two restrictions on the form of inductive type definitions: the type being defined must be the result type of each constructor, and it may only occur *positively* in the types of the constructor arguments (see [25] or [26] for more details).

Interestingly, inductive type definitions are simply an elaborate form of syntactic sugar because any inductively defined type can be represented by a closed type in F_ω , and its constructors can be

represented by closed terms in F_ω . Although there are many ways an inductively defined type can be represented, we have chosen one in which a term with an inductive type can serve as an *iterator*. For example, a desirable property of the Church numeral \bar{n} is that $\bar{n} f = f^n$ (ie, f composed with itself n times).

Defining the closed form of an inductive type is straightforward. Suppose T is the type being defined and T_i is the type of constructor i . The closed form of T is:

$$\text{LAM } A_1 : K_1 \dots \text{LAM } A_n : K_n . \text{Del } A : K . \\ ([A/T]T_1) \rightarrow \dots \rightarrow ([A/T]T_n) \rightarrow (A A_1 \dots A_n)$$

where n is the number of type arguments T should take (as determined by its kind K) and where $[A/T]T_i$ is the result of substituting A for free occurrences of T in the type of constructor i . Hence, the closed forms of **Nat** and **List** are:

$$\text{Nat} = \text{Del } A : * . A \rightarrow (A \rightarrow A) \rightarrow A \\ \text{List} = \text{LAM } A1 : * . \text{Del } A : * \rightarrow * . \\ (\text{Del } B : * . A B) \rightarrow (\text{Del } B : * . B \rightarrow A B \rightarrow A B) \rightarrow (A A1)$$

Extracting the closed term definitions for the constructors⁴ is not as straightforward, but still constitutes a purely syntactic expansion. A complete description of the method used to expand inductive datatype definitions into these closed-form expressions is given in [25].

3.6 Case Expressions and Inductive Definitions

Text describing the syntactic sugar allowing definition by cases and primitive recursion has been omitted from this extended abstract.

4 Compiling LEAP

Two important properties of F_ω have a major impact on the implementation of the run-time aspects of pure LEAP. First, once a program has been determined to be well-typed, the type information is not needed to execute it. Second, as a result of the strong normalization theorem [11], LEAP may be implemented using eager evaluation, since no well-typed term can lead to an infinite reduction sequence. It should be noted that the compilation techniques described in this section are only applicable to pure LEAP — in particular, they are incompatible with general recursion and side-effects. Our current prototype implementation compiles LEAP code to Common Lisp, although in principle any one of a number of languages (such as Scheme) could be used as a “ λ -calculus engine”. In fact, Scheme would have been the most appropriate language to use, but Common Lisp happens to be an integral part of our environment, and thus more convenient in this case.

Two problems arise from the low-level pure LEAP representation of data and functions. First, if a term is represented as a function, it is not desirable to print it as such. Second, this representation of terms, and the inherited representation of functions on them, such as constructors and, even more so, destructors, is inherently inefficient.

The first problem is not very difficult to solve. The basic idea is that any term is represented as a function of a number of variables, each corresponding to one of the constructors from which it

⁴Examples **zero**, **succ**, **nil**, and **cons** are given earlier.

may be built. If this function is applied to functions that serve as the constructors, a useful Lisp term will result. For example, the functional representation of any natural number can be applied to Lisp expressions '0 and #'1+ resulting in a Lisp number, and that of any list can be applied to the Lisp expressions 'nil and #'(lambda (h) #'(lambda (t) (cons h t))), resulting in a Lisp list. Of course, as we discuss below, it is not trivial to determine that a particular term is a natural number, list, etc. In particular, the type information associated with a term is used extensively.

By the time the actual LEAP compiler deals with a program, all the syntactic sugar has been expanded away, and all we are left with is an F_ω term. This means that a lot of information has been obscured from the compiler. An analogous situation is that of Scheme compilers, where a **do** loop is typically expanded into a tail recursion in an early stage, and later the compiler needs to recognize that instances of tail recursion can be compiled as efficiently as if they were **do** loops. In LEAP, this obscuring of information introduces a major complication, in that the functions which operate on the representations of inductive types tend to be inherently very inefficient. For example, the **succ** constructor for natural numbers, if just compiled directly, would count up to a number before adding one to it.

Destructors are even worse [7, 20] because in general they need to be implemented using primitive recursion. This makes obtaining the predecessor of a natural number or the tail of a list extremely inefficient. We have developed a general technique for recognizing instances of constructors and some other functions on inductive types and translating many of them to more primitive functions. To date the compiler does this for natural numbers and lists only. To show how this can be done we present a compilation function $\langle \rangle$ from LEAP terms and their types to Common Lisp terms – types are available to the compiler as a result of the type reconstruction phase. We assume the definitions of Lisp functions **internat** and **iterlist** which turn numbers and lists into their respective iterators. For example, **internat** applied to n , f and x iterates f n times over x .

$$\begin{aligned}
\langle x, T \rangle &= x \\
\langle \text{lam } x : T. E, T \rightarrow T' \rangle &= \#'(lambda (x) \langle E, T' \rangle) \\
\langle E E', T' \rangle &= (\text{funcall} \langle E, T \rightarrow T' \rangle \langle E', T \rangle) \\
\langle \text{Lam } A. E, \text{Del } A. A \rightarrow (A \rightarrow A) \rightarrow A \rangle &= \\
&\quad (\text{funcall} (\text{funcall} \langle E, A \rightarrow (A \rightarrow A) \rightarrow A \rangle '0) #'1+) \\
\langle \text{Lam } A. E, \text{Del } A. A \rightarrow (T \rightarrow A \rightarrow A) \rightarrow A \rangle &= \\
&\quad (\text{funcall} (\text{funcall} \langle E, A \rightarrow (T \rightarrow A \rightarrow A) \rightarrow A \rangle 'nil) \\
&\quad \quad \#'(lambda (h) #'(lambda (t) (cons h t)))) \\
\langle \text{Lam } A. E, \text{Del } A. T \rangle &= \langle E, T \rangle \\
&\quad \text{if neither of the preceding two rules apply} \\
\langle E [T], T \rightarrow (T \rightarrow T) \rightarrow T \rangle &= \text{internat} \langle E, \text{Del } A. A \rightarrow (A \rightarrow A) \rightarrow A \rangle \\
&\quad \text{if } E \text{ has the type } \text{Del } A. A \rightarrow (A \rightarrow A) \rightarrow A \\
\langle E [T], T \rightarrow (T' \rightarrow T \rightarrow T) \rightarrow T \rangle &= \text{iterlist} \langle E, \text{Del } A. A \rightarrow (T' \rightarrow A \rightarrow A) \rightarrow A \rangle \\
&\quad \text{if } E \text{ has the type } \text{Del } A. A \rightarrow (T' \rightarrow A \rightarrow A) \rightarrow A \\
\langle E [T], [T/A]T' \rangle &= \langle E, \text{Del } A. T' \rangle \\
&\quad \text{if neither of the preceding two rules apply}
\end{aligned}$$

In addition to the above, we have made some progress on dealing with primitive recursion using a partial evaluation technique, but the results are only preliminary, and will not be presented here.

5 Nonconservative Extensions

Text describing exploration of language extension by unlimited recursion and side-effects has been omitted from this extended abstract.

6 Conclusions

We have succeeded in making conservative extensions to the ω -order polymorphic λ -calculus, resulting in the design and implementation of pure LEAP. The extensions include type reconstruction, type-argument synthesis, and inductive datatype definitions. We believe that such syntactic extensions are important in achieving a practical programming language, and we think it is important that they can be made without disturbing the theoretical properties of the F_ω core calculus.

In carrying out the implementation of LEAP, we have found that the Ergo Support System (ESS) development tools [14] aided us greatly, allowing us to rapidly prototype many of our ideas. In particular, the use of a higher-order representation of LEAP terms and types in the ESS implementation of λ Prolog has permitted the use of higher-order unification to achieve type reconstruction. Furthermore, the semi-decidability of type reconstruction seems not to be too much of a hindrance in our (admittedly small-scale) experience with LEAP programs. Our work in the efficient compilation of LEAP data constructors is in a preliminary stage, but early results are encouraging.

Finally, it seems clear that to make LEAP a truly practical programming language, a number of nonconservative extensions (such as general recursion and side-effects) will have to be added. This extended language will raise somewhat different implementation problems, and in many ways the adoption of such extensions simplifies the implementation problem. In fact, the back end of the New Jersey SML compiler could be used for compilation. However, the efficient implementation of pure LEAP as described here remains an important goal for us, because it is this language that has a simple logical system for developing and reasoning about programs. In the long run, our compiler will translate programs in pure LEAP into the extended language which is compiled with standard technology. Thus the compiler will act as an intermediary between a system for verified program development (such as the Calculus of Constructions) and an efficient, functional programming language (such as SML).

References

- [1] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages, Austin*, pages 293–302. ACM Press, January 1989.
- [2] Hans-J. Boehm. Partial polymorphic type inference is undecidable. In *26th Annual Symposium on Foundations of Computer Science*, pages 339–345. IEEE, October 1985.
- [3] Hans-J. Boehm. Type inference in the presence of type abstraction. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation, Portland, Oregon*, pages 192–206. ACM Press, June 1989.
- [4] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed Λ -programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.
- [5] Rod Burstall and Butler Lampson. A kernel language for abstract data types and modules. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, pages 1–50. Springer-Verlag, 1984.
- [6] Luca Cardelli. Typeful programming. Technical report, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, February 1989.
- [7] Loïc Colson. About primitive recursive algorithms. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Proceedings of the 16th International Colloquium on Automata, Languages and Programming, Stresa, Italy*, pages 194–206. Springer-Verlag LNCS 372, July 1989.

- [8] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.
- [9] James Donahue and Alan Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.
- [10] Project Formel. The Calculus of Constructions. INRIA-ENS, July 1989. Documentation and User's Guide, Version 4.10.
- [11] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.
- [12] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.
- [13] James W. O'Toole Jr. and David K. Gifford. Type reconstruction with first-class polymorphic values. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation, Portland, Oregon*, pages 207–217. ACM Press, June 1989.
- [14] Peter Lee, Frank Pfenning, Gene Rollins, and William Scherlis. The Ergo Support System: An integrated set of tools for prototyping integrated environments. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 25–34. ACM Press, November 1988. Also available as Ergo Report 88–054, School of Computer Science, Carnegie Mellon University, Pittsburgh.
- [15] Daniel Leivant. Contracting proofs to programs. In P. Odifreddi, editor, *Logic and Computer Science*. Academic Press, 1990. To appear.
- [16] David B. MacQueen. Modules for Standard ML. *Polymorphism Newsletter*, 2(2), October 1985.
- [17] Dale A. Miller and Gopalan Nadathur. Higher-order logic programming. In *Proceedings of the Third International Conference on Logic Programming*. Springer Verlag, July 1986.
- [18] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.
- [19] 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.
- [20] Michel Parigot. On the representation of data in lambda-calculus. Draft, 1988.
- [21] Christine Paulin-Mohring. Extracting F_ω programs from proofs in the calculus of constructions. In *Sixteenth Annual Symposium on Principles of Programming Languages*, pages 89–104. ACM Press, January 1989.
- [22] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, Snowbird, Utah*, pages 153–163. ACM Press, July 1988. Also available as Ergo Report 88–048, School of Computer Science, Carnegie Mellon University, Pittsburgh.
- [23] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation, Atlanta, Georgia*, pages 199–208. ACM Press, June 1988. Available as Ergo Report 88–036, School of Computer Science, Carnegie Mellon University, Pittsburgh.
- [24] Frank Pfenning and Peter Lee. Metacircularity in the polymorphic lambda-calculus. *Theoretical Computer Science*, 1990. To appear. A preliminary version appeared in *TAPSOFT '89, Proceedings of the International Joint Conference on Theory and Practice in Software Development, Barcelona, Spain*, pages 345–359, Springer-Verlag LNCS 352, March 1989.

- [25] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Proceedings of the Fifth Conference on the Mathematical Foundations of Programming Semantics*. Springer Verlag LNCS, March 1989. To appear. Available as Ergo Report 88-069, School of Computer Science, Carnegie Mellon University, Pittsburgh.
- [26] Benjamin Pierce, Scott Dietzen, and Spiro Michaylov. Programming in higher-order typed lambda-calculi. Technical Report CMU-CS-89-111, Carnegie Mellon University, Pittsburgh, Pennsylvania, March 1989.
- [27] 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.