Delphin: Functional Programming with Deductive Systems

Carsten Schürmann
Department of Computer Science
Yale University
New Haven, CT 06520
carsten@cs.yale.edu

Richard Fontana
Department of Computer Science
Yale University
New Haven, CT 06520
richard.fontana@yale.edu

Yu Liao *
Department of Computer Science
Yale University
New Haven, CT 06520
yu.liao@yale.edu

ABSTRACT

We present the design and implementation of the strict and pure functional programming language Delphin. Its novel and distinctive features include a two-level design that distinguishes cleanly between the tasks of representing data and programming with data. One level is the logical framework LF [5], serving as Delphin's data representation language. The other level is $\mathcal{T}_\omega^+$ [15], a type theory designed to support programming using pattern matching and recursion. The main contribution of this work is therefore Delphin, in which one can program with higher-order, dependently-typed data structures such as proofs and typing derivations in a natural and intuitive way.

1. INTRODUCTION

Data structures are among the most well-studied concepts in computer science. Structures such as lists, trees, graphs, and arrays, and many complex variations on them, along with the operations to manipulate these structures, have been designed and implemented, and researchers have analyzed the trade-offs between the compactness of representation and the speed at which operations can be performed.

But this foundational work on data structures has led to new questions related to the representation of other kinds of data. How might one represent a theorem, a specification, or a proof? How should one represent the fundamental concept of abstraction over values, which occurs when encoding functions and relations? What is the best abstract representation of an algorithm? More generally, how can one represent knowledge, or facts about the structure of knowledge (i.e., meta-knowledge)? Perhaps most importantly, how does one *manipulate* theorems, proofs, algorithms, knowledge, and meta-knowledge?

The naive approach to addressing these problems of representation is an encoding in which critical concepts like variables, substitutions, and environments are explicitly represented. Such efforts are in our opinion unacceptably low-level, however, and lead to complex, convoluted, and ultimately unreliable software systems. Since many modern applications, such as agents, avatars, security mechanisms, and theorem provers, depend critically on these concepts, we need better solutions to this important epistemological challenge.

A different approach to the problem of representation is an encoding in a *logical framework*, in which higher-order functions correspond to object-level variables, substitutions, and environments. A logical framework may be thought of as a formal meta-language allowing one to represent and reason about complex concepts, such as proofs, algorithms, abstract machines, operational semantics, execution traces, typing derivations, and in general all kinds of knowledge and meta-knowledge. The great benefit of logical frameworks is that they make such complicated representations as easy for programmers to work with as conventional data objects such as integers, rationals, lists, arrays, or trees. Many important concepts that are difficult and tedious to program explicitly, such as substitutions, environments, and contexts, are implicitly provided by the logical framework and remain hidden inside the representation.

Delphin, the functional programming language described in this paper, engages logical framework technology to model closely the true meaning of concepts. The wide variety of built-in features provided by the LF logical framework, which serves as Delphin's data representation level, leads to elegant encodings despite the intrinsic conceptual complexity of the underlying knowledge

*This research is funded in part by NFS under grant CCR-0133502.

and meta-knowledge being represented. Delphin programmers can take advantage of LF's simple and direct encodings of, for example, proofs, typing derivations, and execution traces, and thereby concentrate their programming efforts on the functionality that can be achieved with these representations, such as type inference algorithms, compilers and theorem provers.

This paper is organized as follows. In Section 2, we discuss logical frameworks in general, and we describe LF as the logical framework used by Delphin. The Delphin language itself is described in Section 3. The type-theoretic underpinnings of Delphin are sketched in Section 4. Section 5 comments briefly on the implementation, following which we give an overview of related work in Section 6. We conclude the paper in Section 7.

2. LOGICAL FRAMEWORKS

Most functional programming languages offer programmers a means of introducing user-defined datatypes. In ML and Haskell, for example, we can write:

SML

```
datatype 'a list
  = nil
  | cons of 'a * 'a list
```

Haskell

```
data List a
  = Nil
  | Cons a (List a)
```

Both datatypes represent lists: finite lists in the ML case, and potentially infinite lists in the Haskell case. But both datatypes have something else in common. In each case, the language used for representing data is the same type-theoretic language that is used for programming with data objects. For example, in ML there is only one product space `'a * 'a list`, and there is only one function space, and each of these spaces can be used either for representation or for programming.

This feature of conventional functional programming languages gives rise to certain problems. Consider the issue of how to decide whether two functions are equivalent. Since it is so difficult to devise, for example, a syntactic criterion that determines observational equivalence, neither ML nor Haskell attempts to decide equality on functions at all. For datatypes that do not include functions, the equality relation implemented in these languages is based on extensional equality. ML uses equality types to signify datatypes that come with implicit extensional equality relations, while Haskell uses type classes.

On the other hand, it is also widely recognized that a single type system for both representation and programming tasks brings many advantages. For example, infinite lists, or streams, are naturally encoded in ML as datatypes; a function is used to delay the evaluation of the tail of a stream. Functions, such as continuations, can also be conveniently contained within datatypes in SML and in Haskell.

The principal design goal of the programming language Delphin is to carefully divide responsibility between two type systems, even though this means losing some of the aforementioned advantages present in conventional functional languages. One type system is the logical framework LF [5], whose sole purpose is to represent, and *not* to compute. The other type system is $\mathcal{T}_\omega^+$ whose objects are programs that compute but which do not represent.

We thus distinguish between functionality and data, just as one naturally distinguishes between reasoning and the objects that are being reasoned about. The interface between the two type systems is kept simple, with a few key characteristics. Delphin programs operate on and manipulate objects of the logical framework. Delphin therefore provides mechanisms for analyzing, destructing, reconstructing, storing, and printing objects. Programs are written in an ML-like style and are defined by pattern matching.

A logical framework must guarantee that the underlying concepts being represented in the framework are represented *adequately*. For example, the correctness of a Delphin program that infers types and typing derivations of expressions of some calculus rests in the first place on an adequate encoding of expressions and typing derivations.

Of the many available logical frameworks, we have chosen LF to serve as Delphin's data representation language. LF is expressive enough to represent many concepts in computer science, logic, and formal methods elegantly, and the conciseness, adequacy, and efficiency of LF representations make them superior to standard datatypes. The following example shows how a fragment of Mini-ML can be expressed in LF. To conserve space we only discuss the fragment containing natural numbers, functions, and recursion, but the example scales to other constructs as well.

Types	τ	::=	$\mathbf{bool} \mid \tau_1 \Rightarrow \tau_2$
Expressions	e	::=	$x \mid e_1 @ e_2 \mid \mathbf{fn}\ x : \tau . e \mid \mathbf{z} \mid \mathbf{s}\ e$
			$\mid (\mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow\ e_1 \mid \mathbf{s}\ x \Rightarrow\ e_2)$
			$\mid \mathbf{rec}\ x.e$
Contexts	Δ	::=	$\cdot \mid \Delta, x : \tau$

The typing rules for Mini-ML are depicted in Figure 1. Expressions and inference rules have very natural encodings in the LF logical framework. LF extends the simply-typed λ-calculus with dependent types. We write $\Gamma \vdash_\Sigma M : A$ for the LF typing judgment, where Γ is the LF context, M is an object and A is the object's type. In Section 4 we give a brief overview of the type theory behind LF.

For Delphin, the critical feature provided by LF is that every LF object has a canonical (β-normal η-long) form, for which we write $\Gamma \vdash_\Sigma M \Uparrow A$. The existence of canonical forms for LF data objects provides Delphin with a notion of structural equality, which permits Delphin functions to be defined by pattern matching. We take

$$\frac{\Delta(x) = \tau}{\Delta \vdash x : \tau}\ \mathsf{of\_var} \qquad \frac{\Delta, x : \tau_1 \vdash e : \tau_2}{\Delta \vdash \mathbf{fn}\ x : \tau_1.e : \tau_1 \Rightarrow \tau_2}\ \mathsf{of\_fn}$$

$$\frac{\Delta \vdash e_1 : \tau_2 \Rightarrow \tau_1 \qquad \Delta \vdash e_2 : \tau_2}{\Delta \vdash e_1 @ e_2 : \tau_1}\ \mathsf{of\_app}$$

$$\frac{}{\Delta \vdash \mathbf{z} : \mathbf{nat}}\ \mathsf{of\_z} \qquad \frac{\Delta \vdash e : \mathbf{nat}}{\Delta \vdash \mathbf{s}\ e : \mathbf{nat}}\ \mathsf{of\_s}$$

$$\frac{\Delta \vdash e : \mathbf{nat} \qquad \Delta \vdash e_1 : \tau \qquad \Delta, x : \mathbf{nat} \vdash e_2 : \tau}{\Delta \vdash \mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow\ e_1 \mid \mathbf{s}\ x \Rightarrow\ e_2 : \tau}\ \mathsf{of\_case}$$

$$\frac{\Delta, x : \tau \vdash e : \tau}{\Delta \vdash \mathbf{fix}\ x : \tau.e : \tau}\ \mathsf{of\_fix}$$

Figure 1: Inference rules for Mini-ML

$\beta\eta$-conversion as the notion of definitional equality [5, 3]. Dependent functions that map objects of type A_1 to A_2 are written as $\Pi x : A_1.\, A_2$. Following standard terminology, types that are indexed by other LF objects are referred to as type families.

In the LF logical framework we represent judgments as types and derivations as objects. For example, the encoding of a derivation of $\mathcal{D}$ of typing judgment $\Delta \vdash e : \tau$ is captured by the definition of the type family "of":

$$\ulcorner \mathcal{D} :: \Delta \vdash e : \tau \urcorner = \ulcorner \Delta \urcorner \vdash_\Sigma \ulcorner \mathcal{D} \urcorner \Uparrow \mathrm{of}\ \ulcorner e \urcorner\ \ulcorner \tau \urcorner \quad (1)$$

where we write $\ulcorner \cdot \urcorner$ for the representation function, and Σ for the LF signature that captures the representation of each Mini-ML type, each language construct and each individual typing rule. The particular signature for our version of Mini-ML [7] is standard and is depicted in Figure 2.

Following standard practice [9], in this presentation we omit all implicit Π-abstractions from the types of "of_fn", "of_app", and "of_fix". The absence of a representation of the inference rule "of_var" should be noted. It is not necessary to represent it explicitly, because Δ is represented by Γ, and therefore the variable lookup rule for Mini-ML corresponds directly to the variable lookup rule for LF.

From a functional programming perspective the three LF type families "tp", "exp", and "of" can be regarded as datatype declarations. Each constant can be seen as a constructor of the type family that is named in the head of the constant's type. For better readability, the constants in the Mini-ML encoding above are grouped in a way that clarifies this view.

2.1 Regular Worlds

When working with complex encodings, such as the encodings of Mini-ML expressions and typing derivations given above, we need a guarantee that the encoding

```
tp : type.
arr : tp → tp → tp.
nat : tp.

exp : type.
z : exp.
s : exp → exp.
case : exp → exp → (exp → exp) → exp.
app : exp → exp → exp.
fn : tp → (exp → exp) → exp.
fix : (exp → exp) → exp.

of : exp → tp → type.
of_z : of z nat.
of_s : of E nat → of (s E) nat.
of_case : of E nat → of E1 T
      → (Πx:exp. of x T → of (E2 x) T)
      → of (case E E1 E2) T.
of_fn : (Πx:exp. of x T1 → of (E x) T2)
      → of (fn T1 E) (arr T1 T2).
of_app : of E1 (arr T2 T1) → of E2 T2
      → of (app E1 E2) T1.
of_fix : (Πx:exp. of x T → of (E x) T)
      → of (fix E) T.
```

Figure 2: Mini-ML

is *adequate*. For example, Mini-ML expressions must be guaranteed to be in one-to-one correspondence with canonical LF objects of type "exp", and Mini-ML typing derivations must be guaranteed to be in one-to-one correspondence with canonical LF objects of type "of $\ulcorner e \urcorner$ $\ulcorner t \urcorner$". Adequacy is proved by induction. Proof of adequacy in the Mini-ML example is important because it ensures that LF objects of type "exp" that are returned by a program actually make sense and can be interpreted as Mini-ML expressions. If this were not the case, the outcome of our programs could not be trusted. Consider the following straightforward encoding of "exp" as a Haskell datatype:

```
data Exp
  = Z
  | S Exp
  | Case Exp Exp (Exp -> Exp)
  | Fn (Exp -> Exp)
  | App Exp Exp
  | Fix (Exp -> Exp)
```

This encoding is not adequate; for example,

```
Fn (\x -> case x of App _ _ -> Fn (\x -> x)
                    _       -> Fix (\x -> x))
```

has type "exp", but it does not correspond to any Mini-ML expression. In a higher-order encoding, in which LF functions occur as arguments to constants, special care must be taken in formulating the induction hypothesis

of the adequacy theorem. Hypothetical judgments, such as the typing judgment given in the previous section, are encoded as higher-order functions. Consequently, the formulation of the adequacy theorem must establish a connection between free variables (or parameters) in an LF encoding and hypotheses of the form $x : \tau$ in a Mini-ML context Δ. Adequacy is thus a property of *open* LF objects, i.e., objects that contain free variables, and not merely a property of *closed* objects.

The following observation regarding the LF encoding of Mini-ML provides further illustration of the open nature of LF objects. Any Delphin program that recurses on subexpressions of Mini-ML expressions **fn** $x.e$ and **fix** $x.e$, for example, must traverse a λ-binder, which introduces new parameters into the context. If a Delphin program recurses on subderivations of typing derivations ending in of_fn or of_fix it must traverse two λ-binders. The type inference problem for Mini-ML provides an example. For each Mini-ML expression e with free variables among Δ, we would like to write a function that computes the type τ and a derivation $\mathcal{D} :: \Delta \vdash e : \tau$, if they exist. In Delphin this can be expressed directly, without the need to define auxiliary datatypes, such as lists, and auxiliary functions that access such datatypes. It is necessary, however, to allow the Delphin function to operate on open terms.

Consider the case $e =$ **fn** $x.e'$, which is represented in LF as "fn $\ulcorner\tau_1\urcorner$ $(\lambda x : \text{exp}.\ulcorner e'\urcorner)$", where $\ulcorner x\urcorner = x$. A program that computes the type of e and the corresponding typing derivation must be applied to e' to determine its type and derivation first. In order to do this it must recurse on e'. However, this would seem to be impossible in a programming language which represents e' as an LF function.

The solution to this basic problem lies in freeing the concept of datatypes from the misconception that they must be static in nature with only a finite number of fixed constructors. On the contrary, with some precautions which we will explain below, datatypes can be considered free, open-ended, and extendible by dynamically adding new data constructors at runtime. We should be able to extend the datatype "exp" by a new constructor "x:exp" on the fly. This would enable us to write a Delphin program that can recurse on e', by recursing on e' x rather than e' alone.

This idea requires us to take some precautions. Delphin functions are defined by pattern matching, and pattern matching is not possible if the set of object constructors is too flexible and not fixed. Patterns define the shape of objects, using constructors of the types of these objects. There is, however, a way out of this seeming dilemma [15]. As long as the overall structure of the dynamic extensions of a datatype is known a priori, one can use a special *block variable* $\underline{x}$ to range over those extensions. For example, in the Mini-ML type inference problem, the Delphin program might have to traverse several binders of the form $\lambda x : exp.\, e'x$, and with every traversal it introduces a new parameter. After n iterations, objects of type "exp" may consist of $n+6$ different constants, i.e., "z", "s", "case", "fn", "app", "fix" ,"x_1 : exp", ..., "x_n:exp". A block variable $\underline{x}$ may be used to capture all n parameter cases at once. Dependencies that are introduced by dependent types complicate this scenario slightly, as we discuss below.

One might expect that these dynamic extensions are local to the datatype, which can be viewed as consisting of the traditional static part and the dynamic part. But this is not the case. Interestingly [14, 15], one can use these extensions to express such properties as: "Every newly-introduced parameter "x:exp" is well typed, expressed by the related assumption "u:of x t" for some type "t : tp". We will see this in the next section when we show how to program in Delphin (see also Example 1 below). How, what, and when new parameters are introduced depends on the functionality that is implemented in the Delphin program and not merely on the datatypes manipulated by the program.

The foregoing observation leads to the concept of *worlds* which capture these dynamic extensions. Clearly, every ML and Haskell program is defined in the empty world, because neither ML nor Haskell datatypes can be extended dynamically. Delphin functions can be defined in arbitrary worlds defined beforehand by the programmer, and the implementation can then take advantage of them.

DEFINITION 1 (BLOCKS).

$$B ::= L : \begin{array}{l}\text{some } (y_1 : A'_1 ... y_n : A'_n) \\ \text{block } (x_1 : A_1 ... x_m : A_m)\end{array}$$

A block B describes m constructors $x_i : A_i$ that may be introduced during evaluation of a Delphin program for some instantiation of all n implicitly existentially quantified assumptions $y_j : A'_j$. L is the name or label of a block.

In other examples, such as the polymorphic version of Mini-ML in which type variables are represented by means of the LF context, a variety of different parameter blocks may arise during evaluation. What all these contexts have in common, however, is that they are regularly formed. During evaluation Delphin programs introduce additional blocks of parameters into the LF context, which grows to the right. Upon return those parameters will be discarded, so that any Delphin program halts in the same context in which it has been started. *Regular worlds* are used to describe *any* possible context that may arise.

DEFINITION 2 (WORLDS). $\Phi ::= B \mid \Phi + \Phi \mid \Phi^*$

One can intuitively think of worlds as regular expressions with blocks B as terminal symbols, where $+$ describes alternatives and * describes repetition.

$$\frac{\Gamma_0 \vdash \sigma : \Gamma_1 \qquad \Gamma_0 \vdash \Gamma \equiv_\alpha [\sigma]\Gamma_2}{\Gamma_0 \vdash \Gamma \in \mathcal{L}(L : \text{some } \Gamma_1 \text{ block } \Gamma_2)}\ \textsf{block}$$

$$\frac{}{\Gamma_0 \vdash \cdot \in \mathcal{L}(\Phi^*)}\ \textsf{empty}$$

$$\frac{\Gamma_0 \vdash \Gamma_1 \in \mathcal{L}(\Phi) \qquad \Gamma_0, \Gamma_1 \vdash \Gamma_2 \in \mathcal{L}(\Phi^*)}{\Gamma_0 \vdash \Gamma_1, \Gamma_2 \in \mathcal{L}(\Phi^*)}\ \textsf{unfold}$$

$$\frac{\Gamma_0 \vdash \Gamma \in \mathcal{L}(\Phi_1)}{\Gamma_0 \vdash \Gamma \in \mathcal{L}(\Phi_1 + \Phi_2)}\ \textsf{left} \qquad \frac{\Gamma_0 \vdash \Gamma \in \mathcal{L}(\Phi_2)}{\Gamma_0 \vdash \Gamma \in \mathcal{L}(\Phi_1 + \Phi_2)}\ \textsf{right}$$

Figure 3: Regular contexts generated by Φ.

EXAMPLE 1. *The Delphin function that implements a type inference algorithm for Mini-ML is defined in a world* $\Phi = B^*$ *where*

$$B = L : \text{some } (t : \text{tp}) \text{ block } (x : \text{exp}, u : \text{of } x\ t).$$

2.2 Adequacy

Adequacy, which refers to the existence of a bijection between informal deductions and objects in the type theory, is a property that must be established individually for every datatype in LF and for every world in which instances will be used. Adequacy theorems always lie outside of the type theory and are proved by induction on the structure of the informal deductions for one direction and on the canonical form deductions for the other. Therefore, the adequacy theorem quantifies over all possible contexts Γ that may be encountered during evaluation, written as $\Gamma \in \mathcal{L}(\Phi)$. The rules defining the set of contexts $\mathcal{L}(\Phi)$ are given in Figure 3.

The block "L : some Γ_1 block Γ_2" satisfies the invariant that Γ_1, Γ_2 form a valid context. In context Γ_0, the set of regular contexts $\mathcal{L}(L : \text{some } \Gamma_1 \text{ block } \Gamma_2)$ consists of all α-variants of the block Γ_2, where free variables declared in Γ_1 have been instantiated by objects (summarized as substitution σ) valid in Γ_0. We write $[\sigma]\Gamma_2$ for a context under a substitution. That σ is valid is enforced by the first premiss $\Gamma_0 \vdash \sigma : \Gamma_1$ of the "block" rule, whose definition we omit. The second premiss of the same rule is the standard α-conversion congruence $\Gamma_0 \vdash \Gamma_1 \equiv_\alpha \Gamma_2$, which permits tacit variable renaming on regular contexts. Consequently, without loss of generality all contexts in $\mathcal{L}(\Phi)$ are valid.

In this section we show that Mini-ML's encoding is adequate with respect to the world Φ from Example 1. Informally, the adequacy theorem states that the representation function $\ulcorner \cdot \urcorner$ used in Equation (1) is a bijection. Contexts, terms, types, and typing derivations must be embedded adequately, and $\Gamma \in \mathcal{L}(\Phi)$ must stand in one-to-one correspondence to them.

THEOREM 1 (ADEQUACY). *Let* Φ *be defined as in Example 1. For all contexts* Γ*, s.t.* $\cdot \vdash \Gamma \in \mathcal{L}(\Phi)$*, there exists a* Δ*, s.t.* $\Gamma = \ulcorner \Delta \urcorner$*, and vice versa.*

PROOF. By induction on the derivation of $\cdot \vdash \Gamma \in \mathcal{L}(\Phi)$ in one direction and Δ in the other direction. □

THEOREM 2 (ADEQUACY). *Let* Φ *be defined as in Example 1, Mini-ML typing context* Δ*, and* $\cdot \vdash \Gamma \in \mathcal{L}(\Phi)$*, s.t.* $\ulcorner \Gamma \urcorner = \Delta$.

1. *For all objects* $\Gamma \vdash M \Uparrow \text{tp}$ *there exists a Mini-ML type* τ*, s.t.* $\ulcorner \tau \urcorner = M$*, and vice versa.*
2. *For all objects* $\Gamma \vdash M \Uparrow \text{exp}$ *there exists a Mini-ML expression* e*, s.t.* $\ulcorner e \urcorner = M$ *with free variables among* Δ*, and vice versa.*
3. *For all objects* $\Gamma \vdash M \Uparrow \text{of} \ulcorner e \urcorner \ulcorner \tau \urcorner$ *there exists a Mini-ML typing derivation* $\mathcal{D} :: \Delta \vdash e : \tau$*, s.t.* $\ulcorner \mathcal{D} \urcorner = M$*, and vice versa.*

PROOF. By structural induction on the canonicity derivations in one direction, τ, e, and $\mathcal{D} :: \text{of} \ulcorner e \urcorner \ulcorner \tau \urcorner$ in the other direction, using Theorem 1. □

3. DELPHIN

Delphin is a strict functional programming language which is designed to allow programming with datatypes that consist of a fixed set of constructors along with dynamic extensions of these datatypes valid in some world. The core language that is presented in this paper has been implemented and can be accessed through [17]. Delphin's syntax is inspired by that of Standard ML of New Jersey. Delphin permits function definition by pattern matching and recursion. Its datatypes are essentially LF types, and the objects manipulated by Delphin programs are LF objects. Delphin's implementation also includes a type checker and an interpreter. Delphin's type system is deceptively simple, since it only provides type constructors for function and product spaces. Although these constructors provide dependent types, there is no mechanism that would allow programmers to define their own Delphin types. The current design does not allow programs to be polymorphic, but we plan to investigate the issue of polymorphism in future work.

3.1 Datatypes

Delphin's datatype declarations are LF signatures, which include declarations of constructors for type families and worlds. The Twelf system [10] is an implementation of the logical framework LF which is designed to facilitate developing, implementing, experimenting with, and verifying properties about deductive systems, such as the Mini-ML type system in the example given above. In fact, Twelf is an extraordinarily useful and effective tool for engineering, developing, and debugging representations of data, and we have accordingly chosen to use and

```
eval :: all {e:exp} exists {v:exp} true

fun eval z = <z, <>>
  | eval (s E) =
    let
      val <V, <>> = eval E
    in
      (s V)
    end
  | eval (case E E1 E2) =
    (case (eval E)
       of <z, <>>    => eval E1
        | <s V, <>>  => eval (E2 V))
  | eval (fn T E) = <fn T E, <>>
  | eval (app E1 E2) =
    let
      val <fn T E1', <>> = eval E1
      val <V2, <>> = eval E2
    in
      eval (E1' V2)
    end
  | eval (fix E) =
      eval (E (fix E))

val <D, <>> = eval (app (fn nat [x] x) z)
```

Figure 4: A Mini-ML evaluator.

parse Twelf signatures as Delphin datatypes. Our implementation therefore takes direct advantage of Twelf technology, and does not provide a separate mechanism for defining datatypes. Figure 2, for example, contains the content of a file that is already in Twelf format.

Programmers are free to extend datatypes dynamically during evaluation as long as these extensions conform to the rules stipulated by the world in which a function is defined. We say that *a function cannot leave the world in which it lives during evaluation.* The idea of worlds is not new; it was introduced in [14], studied as a means of defining recursive functions in [15], and applied to reasoning by induction in [16]. Worlds have also been implemented in the Twelf system [10]. The block from Example 1 is declared in Twelf as follows:

```
%block L : some {T:tp}
           block {x:exp} {u:of x T}.
```

3.2 Language Features

Delphin programs consist of variable declarations, value definitions, and function definitions. Local function definitions are also possible. Consider for example an evaluator for Mini-ML programs, which is presented in Figure 4.

The first instruction declares the variable `eval` to be of type $\forall e : \text{exp}.\, \exists v : \text{exp}.\, \top$. In the text we write types in mathematical notation; in Delphin source code types are given in corresponding ASCII notation. $\forall$ stands for the Delphin-level dependent function space, and $\exists$ stands for the Delphin-level dependent product space; these should not be confused with the LF-level Π. $\top$ corresponds to the `unit` type of ML.

The next instruction defines the program `eval` by pattern matching; it closely resembles an ML function declaration. `<z, <>>` in the first case of `eval` illustrates the syntax for pairs. `<>` is `()` in ML, and has type $\top$. When programming an evaluator for Mini-ML programs there is no need to define a notation for substitution or environments. These concepts are provided by LF implicitly; the programmer can take advantage of them by simply applying $\texttt{E}_2$ to `V` in the second case of `case`, $\texttt{E}_1'$ to $\texttt{V}_2$ in the `app` case, and `E` to `(fix E)` in the `fix` case. Therefore, juxtaposition can have one of two meanings. Depending on where it occurs, it is either an LF-level application or a Delphin-level application.

The last instruction in Figure 4 is a value definition that employs pattern matching. It demonstrates how to call the evaluator, here using the simple ML expression $(\textbf{fn}\ x : \textbf{nat}.x)\ \textbf{z}$.

As second example, we consider the type inference problem used to motivate this work, whose implementation in Delphin is depicted in the two programs in Figures 5 and 6. The first program checks whether two Mini-ML types are equivalent. The second program infers the type of a Mini-ML expression together with the typing derivation.

Clearly, in order to infer the type of a Mini-ML expression, `infer` must recurse under a λ-binder to infer the type of the body of the function. This means that `infer` cannot live in the empty world, and neither can `check`, because it may be called from `infer` *after* new parameters have been introduced. This observation is reflected in the types of the two functions: both are declared to live in the world generated by the block labeled `L`. This signifies that both `check` and `infer` may be executed in any context $\Gamma \in \mathcal{L}(B^*)$, where $B = L : \text{some}\ (t : \text{tp})\ \text{block}\ (x : \text{exp}, u : \text{of}\ x\ t)$.

Therefore, `infer` may be called with one of these dynamically introduced parameters. This possibility is covered by the first case:

```
infer #L_X = <#L_T, <#L_U, <>>>
```

`#L` stands for a variable that ranges over blocks of variables, or instances of blocks. In mathematical notation we would write $\underline{x}$ for `#L`. `_T` is a projection of the existentially quantified variable `T` from block `L`, and `_X` and `_U` project the two parameters, respectively. This case can be read as follows: "If parameter x is encountered, return its type and the appropriate typing derivation."

Parameters are correspondingly introduced through the `new` command provided by Delphin. Its usage is demonstrated by the `fn` case of the `infer` function (and simi-

larly in the `case` and `fix` cases).

```
val <T, <P, <>>>  =
  new
    {x:exp}{u:of x T1}
  in
    infer (E x)
  end
```

Here `E` is an LF function of type `exp` → `exp` and $\mathtt{T}_1$ is a Mini-ML type. Without worlds, `infer` could not be defined, because its argument has type `exp` and therefore it could not recurse on `E`. But with `new` the LF context can be extended, provided that the extension does not violate the world in which `infer` lives. Indeed it does not, because t is simply being instantiated to $\mathtt{T}_1$ in Example 1. This solves the problem because in the extended context `E x` has type `exp`, and the evaluation may proceed.

Type-theoretically speaking, `new` is executed in a situation in which $\Gamma \vdash E : \text{exp} \to \text{exp}$ and $\Gamma \vdash T_1 : \text{tp}$, where $\Gamma \in \mathcal{L}(B^*)$. Then two new parameters are introduced, and `infer` is called recursively, with the argument

$$\Gamma, x : \text{exp}, u : \text{of}\ T_1 \vdash E\ x : \text{exp}$$

returning a value of type $\exists T : \text{tp}.\, \exists P : \text{of}\ (E\ x)\ T.\, \top$.

$$\begin{array}{lll} \Gamma, x : \text{exp}, u : \text{of}\ x\ T_1 & \vdash & T : \text{tp} \\ \Gamma, x : \text{exp}, u : \text{of}\ x\ T_1 & \vdash & P : \text{of}\ (E\ x)\ T \end{array}$$

The binding, however, must take place in the unextended context, Γ, and not in $\Gamma, x : \text{exp}, u : \text{of}\ x\ T_1$. This problem is easily solved by an operation called abstraction and is implemented using reasoning within LF while taking advantage of the following strengthening properties:

Lemma 1 (Strengthening).
If $\Gamma, x : \text{exp}, u : \text{of}\ x\ t \vdash M : \text{tp}$ *then* $\Gamma \vdash M : \text{tp}$.

The result of abstraction is therefore

$$\begin{array}{lll} \Gamma & \vdash & T : \text{tp} \\ \Gamma & \vdash & P : \Pi x : \text{exp}.\, \text{of}\ x\ T_1 \to \text{of}\ (E\ x)\ T \end{array}$$

and these are the declarations used in the bodies of the `case`, `fn`, and `fix` cases. P has a functional type, which, by the adequacy theorem (Theorem 2), corresponds to a hypothetical derivation. Thus, it may be passed as an argument to "of_fn". The arguments for the other two cases that use `new` are, respectively, "of_fix" and "of_case". Note, that in the `case`-case `T` must be `nat`, which is directly incorporated into the pattern matching.

A necessary and sufficient criterion that implies strengthening theorems for all type families has been developed elsewhere [18]. It is effectively computable, and it follows from a static analysis of the dependency relation

```
check :: world (L) all {t1:tp} all {t2:tp} true

fun check nat nat = <>
  | check (arr T1 T2) (arr T1' T2') =
    let
      val _ = check T1 T1'
      val _ = check T2 T2'
    in
      <>
    end
```

Figure 5: Equivalence of types

among objects of different types. From this criterion, it is a simple exercise to construct the abstraction operation used above [15]. We give a formal description of abstraction in Definition 3 in Section 4.

The ability to use `new` to introduce new parameters in accordance with the current world is a novel concept in functional programming languages. In this respect, Delphin differs significantly from standard functional programming languages like ML and Haskell.

3.3 World Checking

Delphin enforces the consistency of worlds. A Delphin program f must not call g unless the world in which g lives is at least as large as that in which f lives. Which world a function lives in is part of its type, and the relation "as least as large as" means that all blocks in the world of the caller must occur in the world of the callee.

Lemma 2 (Block consistency).
If $\Gamma \in \mathcal{L}(\Phi^*)$ *then* $\Gamma \in \mathcal{L}((\Phi + B)^*)$ *for any block* B.

Just as importantly, the parameters that are introduced by a `new` statement must be checked for consistency with one of the blocks defined by the world. In Delphin, parameters of several blocks can be introduced using one `new` statement. Unless the parameters are shuffled, Delphin can decide the validity of such a parameter block.

When a program is loaded into Delphin, world information is inferred for every function call and is then checked for consistency. World-checking is an operation orthogonal to type checking. If world-check errors are detected Delphin will abort the loading process and report the error to the programmer.

3.4 Type Checking

Delphin programmers are allowed to omit a significant portion of reconstructible information from LF objects. LF objects carry so much redundancy that is possible to infer types even when arguments are omitted, provided that those arguments occur elsewhere in the type of an object or type constant [9]. See also the presentation of Delphin datatypes in Figure 2, in which all implicit

```
infer :: world (L) all {e:exp}
           exists {t:tp} exists {D:of e t} true

fun infer #L_X = <#L_T, <#L_U, <>>>
  | infer z = <nat, <of_z, <>>>
  | infer (s E) =
    let
      val <nat, <P, <>>>  = infer E
    in
      <nat, <of_s P, <>>>
    end
  | infer (case E E1 E2) =
    let
      val <nat, <D, <>>> = infer E
      val <T1, <D1, <>>> = infer E1
      val <T2, <D2, <>>> =
        new
          {x:exp}{u:of x nat}
        in
          infer (E2 x)
        end
      val _ = check T1 T2
    in
      <T1, <of_case D D1 D2, <>>>
    end
  | infer (fn T1 E) =
     let
      val <T, <P, <>>>  =
        new
          {x:exp}{u:of x T1}
        in
          infer (E x)
        end
    in
      <T, <of_fn P, <>>>
    end
  | infer (app E1 E2) =
    let
      val <arr T2 T1, <D1, <>>> = infer E1
      val <T2', <D2, <>>> = infer E2
      val _ = check T2 T2'
    in
      <T1, <of_app D1 D2, <>>>
    end
  | infer (fix T1 E) =
     let
      val <T, <P, <>>>  =
        new
          {x:exp}{u:of x T1}
        in
          infer (E x)
        end
      val _ = check T1 T
    in
      <T, <of_fn P, <>>>
    end
```

Figure 6: A Mini-ML type inference algorithm.

```
count :: world (L) all {e:exp} all {t:tp}
           all {P : of e t} exists {N : exp} true
fun count #L_X #L_T #L_U = <s z, <>>
  | count z nat of_z = <z, <>>
  | count (s E) nat (of_s P) = count E nat P
  | count (case E E1 E2) T (of_case P P1 P2) =
    let
      val <N, <>> = count E nat P
      val <N1,<>> = count E1 T P1
      val <N2,<>> =
        new
          {X:exp} {U:of X nat}
        in
          count (E2 X) T (P2 X U)
        end
      val <N3, <>> = add N N1
    in
      add N3 N2
    end
  | count (fn T1 E) (arr T1 T2) (of_fn P)=
    new
      {X:exp} {U:of x T1}
    in
      count (E x) T2 (P X U)
    end
  | count (app E1 E2) T2 (of_app D1 D2) =
    let
      val <N1, <>> = count E1 _ D1
      val <N2, <>> = count E2 _ D2
    in
      add N1 N2
    end
  | count (fix T E) =
    new
      {X:exp} {U:of x T}
    in
      count (E x) T (P X U)
    end
```

Figure 7: Counting Typing Hypotheses.

Π abstractions have been omitted. Delphin in essence inherits Twelf's type inference algorithm, which enables one to write the compact programs in Figures 4-6.

It follows, however, that in general any function that is defined by cases over objects with implicit arguments may contain many more variables than the ones explicitly provided by the programmer. Examples in which such issues arise include Delphin programs that analyze cases over typing derivations, such as the program given in Figure 7, which computes the number of hypotheses over the derivation $\mathcal{D} :: \Delta \vdash e : \tau$. We assume that there is a function that adds two numbers of type `add` $\in \forall n_1 : \text{exp}. \forall n_2 : \text{exp}. \exists n_3 : \text{exp}. \top$.

Type checking with dependent Σ types can be quite challenging, because principal types do not always exist. Consider for example the possible types of

$$
\begin{array}{lll}
\text{Formulas} & F ::= & \forall x : A.\, F \mid \forall \underline{x} : (L;\sigma).\, F \mid F_1 \supset F_2 \\
& & \mid \exists x : A.\, F \mid \exists \underline{x} : (L;\sigma).\, F \mid F_1 \wedge F_2 \mid \top \\
\text{Programs} & P ::= & \Lambda x : A.P \mid \Lambda \underline{x} : (L;\sigma).P \mid \Lambda \mathbf{x} \in F.P \\
& & \mid \mathbf{x} \mid \langle M; P\rangle \mid \langle \underline{x}; P\rangle \mid \langle P_1; P_2\rangle \mid \langle\rangle \mid P\ M \\
& & \mid \nu P \mid P\,\underline{x} \mid P_1\ P_2 \mid \mathtt{case}\ \Omega \mid \mu \mathbf{x} \in F.P \\
\text{Cases} & \Omega ::= & . \mid \Omega, (\Psi \rhd \sigma \mapsto P) \\
\text{Contexts} & \Psi ::= & \cdot \mid \Psi, x : A \mid \Psi, \underline{x} : (L;\sigma) \mid \Psi, \mathbf{x} \in F
\end{array}
$$

Figure 8: Delphin type and program syntax

`<z, <of_z, <of_z, <>>>>`:

$$
\begin{array}{l}
\exists x : \text{exp}.\, \exists u : \text{of}\ x\ \text{nat}.\, \exists v : \text{of}\ x\ \text{nat}.\, \top \\
\exists x : \text{exp}.\, \exists u : \text{of}\ x\ \text{nat}.\, \exists v : \text{of}\ z\ \text{nat}.\, \top \\
\exists x : \text{exp}.\, \exists u : \text{of}\ z\ \text{nat}.\, \exists v : \text{of}\ x\ \text{nat}.\, \top \\
\exists x : \text{exp}.\, \exists u : \text{of}\ z\ \text{nat}.\, \exists v : \text{of}\ z\ \text{nat}.\, \top
\end{array}
$$

Which one of those types to pick is a complicated type inference problem that one might be able to solve using bidirectional type inference techniques. When Delphin cannot determine the principal type of a program, it reports an error and requests that the user provide the correct type explicitly.

The type inference and type checking algorithms in Delphin are decidable, and their formal foundation is given in Section 4. In future work we plan to implement a coverage checker and a termination checker [13, 11].

4. TYPE-THEORETIC FOUNDATION

The LF and $\mathcal{T}_\omega^+$ type theories are well-understood [5, 14]. In this section we aim to provide a glimpse of the underlying theory of Delphin, its type system, its operational semantics, and its meta properties. For an in-depth discussion the reader is invited to consult the literature.

4.1 Type Theory $\mathcal{T}_\omega^+$

The implementation of Delphin is based on $\mathcal{T}_\omega^+$. The syntactic categories are presented in Figure 8. The class of formulas represents the class of types of Delphin programs, which we have already discussed in Section 3. $\mathcal{T}_\omega^+$ provides three distinct concepts of variables. First, there are LF variables $x : A$ that range over LF objects of LF type A. Second, there are block variables $\underline{x} : (L;\sigma)$, used to extend Delphin datatypes dynamically. Third, there are program variables $\mathbf{x} \in F$ that bind Delphin programs, such as `check` and `infer`. All three kinds of variables can be declared in a $\mathcal{T}_\omega^+$ context Ψ for which we write Γ if it is free of program variables.

In a slight departure from the standard formulation of LF, we allow block variables to occur in LF contexts and enrich the otherwise standard set of inference rules by obj_proj. The resulting calculus for LF is depicted in Figure 9. The object $\pi_y(\underline{x})$ describes the projection of component y from block $\underline{x}$.

To conserve space, this paper does not give the syntactic desugaring function that maps the external (ASCII) representation of Delphin programs into $\mathcal{T}_\omega^+$. One version of this function is explained in detail in [15]. We use $\Lambda x : A.\, P$, $\Lambda \underline{x} : (L;\sigma).\, P$ and $\Lambda \mathbf{x} \in F.\, P$ for functional abstraction, and $\langle -; -\rangle$ as programs of the existential and conjunction formulas.

For purposes of functional programming, the main novel features of $\mathcal{T}_\omega^+$ include its support for recursive programming, pattern matching, and the dynamic extension of datatypes. Recursion is expressed in terms of "μ", pattern matching by case analysis "case", and the extension of datatypes by "ν". In general, patterns in Delphin are non-linear, functional, and dependently typed. Therefore, traditional pattern matching techniques are inapplicable. Ω represents the list of cases. One might wonder why there is no case subject defined for "case". Each substitution in every case of Ω matches against the entire environment, as we will explain below.

4.2 Type System

Figure 10 gives the set of typing rules which define the following two $\mathcal{T}_\omega^+$ typing judgments:

$$
\begin{array}{ll}
\text{Valid programs} & \Psi \vdash_{\Sigma,\Phi} P \in F \\
\text{Valid cases} & \Psi \vdash_{\Sigma,\Phi} \Omega \in F
\end{array}
$$

Many of the typing rules are standard. There are a few nonstandard rules, however, which deserve explanation. $\forall$I_block, $\forall$E_block are the introduction and elimination rules for $\forall \underline{x} : (L;\sigma).\, F$, providing the type of programs that introduce or rename dynamic datatype extensions. Even more nonstandard is the new rule, which internalizes dynamically introduced extensions of datatypes by means of abstraction, as explained in the discussion of the `infer` program. The notation (block L)$[\sigma]$ refers to a context of x_i's one obtains by substitution σ for the y_i's in the block labeled L (see Definition 1).

The formal definition of abstraction takes advantage of the subordination relation $A_1 \prec A_2$ capturing the essence of strengthening lemmas, which means that no object of type A_1 can occur in an object of type A_2.

DEFINITION 3 (ABSTRACTION). *1. Type-level abstraction:* $\text{abs}\ \Gamma.\, A_2 =$

$$
\begin{array}{ll}
A_2 & \text{if } \Gamma = \cdot \\
\text{abs}\ \Gamma'.\, A_2 & \text{if } \Gamma = x : A_1, \Gamma' \text{ and } A_1 \not\prec A_2 \\
\Pi x : A_1.\, (\text{abs}\ \Gamma'.\, A_2) & \text{if } \Gamma = x : A_1, \Gamma' \text{ and } A_1 \prec A_2
\end{array}
$$

2. Object-level abstraction: Let M be well-typed of type A_2. $\text{abs}\ \Gamma.\, M =$

$$
\begin{array}{ll}
M & \text{if } \Gamma = \cdot \\
\text{abs}\ \Gamma'.\, M & \text{if } \Gamma = x : A_1, \Gamma' \text{ and } A_1 \not\prec A_2 \\
\lambda x : A_1.\, (\text{abs}\ \Gamma'.\, M) & \text{if } \Gamma = x : A_1, \Gamma' \text{ and } A_1 \prec A_2
\end{array}
$$

3. Object-level application: Let M be well-typed of

$$\frac{\Psi \vdash_{\Sigma;\Phi} A : \text{type} \quad \Psi, x : A \vdash_{\Sigma;\Phi} P \in F}{\Psi \vdash_{\Sigma;\Phi} \Lambda x : A.\, P \in \forall x : A.\, F}\ \forall\text{I\_LF} \qquad \frac{\Psi \vdash_{\Sigma;\Phi} P \in \forall x : A.\, F \quad \Psi \vdash_{\Sigma} M : A}{\Psi \vdash_{\Sigma;\Phi} P\ M \in F[M/x]}\ \forall\text{E\_LF}$$

$$\frac{\Psi, \underline{x} : (L;\sigma) \vdash_{\Sigma;\Phi} P \in F}{\Psi \vdash_{\Sigma;\Phi} \Lambda \underline{x} : (L;\sigma).\, P \in \forall \underline{x} : (L;\sigma).\, F}\ \forall\text{I\_block} \qquad \frac{\Psi \vdash_{\Sigma;\Phi} P \in \forall \underline{x} : (L;\sigma).\, F \quad \Psi(\underline{y}) = (L;\sigma)}{\Psi \vdash_{\Sigma;\Phi} P\ \underline{y} \in F[\underline{y}/\underline{x}]}\ \forall\text{E\_block}$$

$$\frac{\Psi, \mathbf{x} \in F_1 \vdash_{\Sigma;\Phi} P \in F_2}{\Psi \vdash_{\Sigma;\Phi} \Lambda \mathbf{x} \in F_1.\, P \in F_1 \supset F_2}\ \forall\text{I\_delphin} \qquad \frac{\Psi \vdash_{\Sigma;\Phi} P_1 \in F_2 \supset F_1 \quad \Psi \vdash_{\Sigma;\Phi} P_2 \in F_2}{\Psi \vdash_{\Sigma;\Phi} P_1\ P_2 \in F_1}\ \forall\text{E\_delphin}$$

$$\frac{\Psi \vdash_{\Sigma} M : A \quad \Psi \vdash_{\Sigma;\Phi} P \in F[M/x]}{\Psi \vdash_{\Sigma;\Phi} \langle M; P\rangle \in \exists x : A.\, F}\ \exists\text{I\_LF} \qquad \frac{\Psi(\underline{y}) = (L;\sigma) \quad \Psi \vdash_{\Sigma;\Phi} P \in F[\underline{y}/\underline{x}]}{\Psi \vdash_{\Sigma;\Phi} \langle \underline{y}; P\rangle \in \exists \underline{x} : (L;\sigma).F}\ \exists\text{I\_block}$$

$$\frac{\Psi \vdash_{\Sigma;\Phi} P_1 \in F_1 \quad \Psi \vdash_{\Sigma;\Phi} P_2 \in F_2}{\Psi \vdash_{\Sigma;\Phi} \langle P_1; P_2\rangle \in F_1 \wedge F_2}\ \wedge\text{I} \qquad \frac{\Psi(\mathbf{x}) = F}{\Psi \vdash_{\Sigma;\Phi} \mathbf{x} \in F}\ \text{var} \qquad \frac{}{\Psi \vdash_{\Sigma;\Phi} \langle\rangle \in \top}\ \text{true}$$

$$\frac{\Psi, x \in F \vdash_{\Sigma;\Phi} P \in F}{\Psi \vdash_{\Sigma;\Phi} \mu x \in F.\, P \in F}\ \text{rec} \qquad \frac{\Psi \vdash_{\Sigma;\Phi} \Omega \in F}{\Psi \vdash_{\Sigma;\Phi} \text{case}\ \Omega \in F}\ \text{case} \qquad \frac{\Psi \vdash_{\Sigma;\Phi} P_1 \in F_1 \quad \Psi, \mathbf{x} \in F_1 \vdash_{\Sigma;\Phi} P_2 \in F_2}{\Psi \vdash_{\Sigma;\Phi} \text{let}\ \mathbf{x} = P_1\ \text{in}\ P_2 \in F_2}\ \text{let}$$

$$\frac{\Psi \vdash_{\Sigma;\Phi} P \in \forall \underline{x} : (L;\sigma).\, F \quad \text{abs}\ ((\text{block}\ L)[\sigma]).\, F = F'}{\Psi \vdash_{\sigma;\Phi} \nu P \in F'}\ \text{new}$$

$$\frac{}{\Psi \vdash_{\Sigma;\Phi} \cdot \in F}\ \text{empty} \qquad \frac{\Psi_1 \vdash_{\Sigma;\Phi} \Omega \in F \quad \Psi_2 \vdash_{\Sigma;\Phi} \sigma \in \Psi_1 \quad \Psi_2 \vdash_{\Sigma;\Phi} P \in F[\sigma]}{\Psi_1 \vdash_{\Sigma;\Phi} \Omega, (\Psi_2 \triangleright \sigma \mapsto P) \in F}\ \text{cons}$$

Figure 9: Delphin Typing Rules.

type $\text{abs}\ \Gamma.\, A_2.\ M\ \Gamma =$

$$\begin{array}{ll} M & \textit{if}\ \Gamma = \cdot \\ M\ \Gamma' & \textit{if}\ \Gamma = x : A_1, \Gamma'\ \textit{and}\ A_1 \not\prec A_2 \\ (M\ x)\ \Gamma' & \textit{if}\ \Gamma = x : A_1, \Gamma'\ \textit{and}\ A_1 \prec A_2 \end{array}$$

4. *Meta-level abstraction: Let F be a well-formed formula.* $\text{abs}\ \Gamma.\, F =$

$$\begin{array}{ll} \top & \textit{if}\ F = \top \\ \forall x : (\text{abs}\ \Gamma.\, A).\text{abs}\ \Gamma.\, F'[x\ \Gamma/x] & \textit{if}\ F = \forall x : A.F' \\ \exists x : (\text{abs}\ \Gamma.\, A).\text{abs}\ \Gamma.\, F'[x\ \Gamma/x] & \textit{if}\ F = \exists x : A.F' \\ (\text{abs}\ \Gamma.\, F_1) \supset (\text{abs}\ \Gamma.\, F_2) & \textit{if}\ F = F_1 \supset F_2 \\ (\text{abs}\ \Gamma.\, F_1) \wedge (\text{abs}\ \Gamma.\, F_2) & \textit{if}\ F = F_1 \wedge F_2 \end{array}$$

The cons rule is also noteworthy; it expresses the requirement that in every case the substitution must match the environment.

4.3 Operational Semantics

Delphin has a call-by-value operational semantics whose rules are given in Figure 11. The evaluation judgment $\Gamma; \eta \vdash P \hookrightarrow V$ relates the program P to be evaluated with the outcome of the evaluation V. Γ is a pure LF context that represents the context of all dynamic extensions of datatypes. Thus it contains block variables exclusively. η is an environment. Values include closures.

$$\begin{array}{rl} V ::= & \{\eta; \Lambda x : A.P\} \mid \{\eta; \Lambda \underline{x} : (L;\sigma).P\} \\ & \mid \{\eta; \Lambda \mathbf{x} \in F.P\} \mid \langle M; V\rangle \mid \langle \underline{x}; V\rangle \mid \langle V_1; V_2\rangle \mid \langle\rangle \end{array}$$

As with the typing rules, the rules for the operational semantics are for the most part standard, but a few of the rules are unusual. ev_Λ_delphin, ev_app_delphin, and ev_new introduce, retract, and abstract the dynamic extension of datatypes and ev_case, ev_yes, and ev_no govern case analysis. The second premiss of ev_yes stipulates the existence of a refined environment η' which can only be determined during runtime using higher-order matching.

4.4 Meta-theory

Delphin's operational semantics ensures that the result of a computation that is begun in a regularly formed world is well-defined in the same world when the computation halts. During evaluation new block variables may be dynamically introduced, but they will be discharged by the time the computation terminates. Delphin's operational semantics is type preserving.

THEOREM 3 (TYPE-PRESERVATION). *If $\Psi \vdash P \in F$ and $\Gamma; \eta \vdash P \hookrightarrow V$ and η is an environment that assigns objects in Γ to Ψ, then $\Gamma \vdash V \in F[\eta]$.*

PROOF. By induction on the structure of the evaluation derivation. □

5. IMPLEMENTATION

Delphin is implemented in Standard ML of New Jersey. The implementation consists of four modules: a parser, an elaborator, a type checker, and an interpreter. Delphin employs the Twelf internal representation for objects manipulated by Delphin programs. Twelf itself provides many of the tools needed for parsing Twelf objects, such as objects representing typing derivations, type reconstruction, type checking, and subordination. Delphin's top-level loop executes the following tasks in turn: parsing, elaboration, world checking, type checking, and evaluation. The implementation is currently in prototype stage. We plan to release the first version of Delphin in the upcoming weeks.

6. RELATED WORK

Our work involves the design of a novel functional programming language which can be used to compute with data structures of significant complexity, especially those employing binding operators, such as proofs, typing derivations, and computation traces. Such structures are commonly used in applications like proof-carrying code [8, 1] and typed compilation [6].

The languages DML [19] and Cayenne [2] differ from Delphin in that they extend existing functional programming languages, SML and Haskell respectively, by introducing dependent types; moreover, they are motivated by goals that are quite different from those which have inspired the Delphin project. DML's enrichment of ML with dependent types makes it possible to capture more program invariants, which may in turn facilitate program error detection or compiler optimization. Cayenne combines dependent types and first class types, thus making more programs typeable. These languages differ radically from Delphin in their structural design. Delphin is a two-tiered language. Its upper layer, a recursive function space used for computation, is entirely separate from its lower LF layer, which is used for data representation. By contrast, DML and Cayenne introduce dependent types directly into the type system of the host language. DML only uses restricted dependent types; type index objects are drawn from a constraint domain which is much less powerful than LF's λ^{Π} type system. DML and Cayenne also differ from Delphin with respect to the data structures that can be easily supported by the language. Because dependent types in DML and Cayenne are introduced for typing purposes only, their data structures are the same as those provided by the respective host languages. Thus it is still very cumbersome to program with complex data structures such as those which represent proofs or typing derivations. Delphin is specifically designed to support programs that can easily represent and operate upon such complex data structures.

FreshML [12, 4] is an ML-like metalanguage for programming with data structures that may involve variable binding. Like Delphin, FreshML supports recursive function definitions and pattern matching over its data structures. However, FreshML merely promotes object-level renaming to the meta-level, through a set-theoretic interpretation of name abstraction. Object-level substitution must therefore be implemented for each object language separately. This contrasts sharply with approach adopted by Delphin, in which renaming and substitution are provided entirely "for free" at the meta-level. The principal criticism of higher-order encodings made by the FreshML authors is that it is difficult to integrate it with the ability to define recursive functions, but this is precisely the problem that Delphin has been designed to address. Delphin's two-level design allows programming with recursion and pattern matching to coexist with elegant higher-order encodings in LF.

7. CONCLUSION

We have presented Delphin, a functional programming language that builds upon logical framework technology. The principal novel feature of Delphin is its strict separation between the representation of data and the programs that manipulate such data. The underlying data representation employs the LF logical framework, in which every encoded object has a canonical form. Delphin's use of LF permits programmers to use dependent types and higher-order functions to express complex data structures such as typing derivations, proofs, program transformations, and computation traces. In traditional functional programming languages, programmers can express such concepts only by implementing auxiliary data structures to represent contexts and substitutions, which are provided implicitly and "for free" by LF. By using LF as the data representation language, Delphin enables programmers to compute with these complex data structures as easily as programmers can manipulate conventional data structures in mainstream functional programming languages.

Delphin programming is designed to resemble ML programming. However, many features available in ML and other conventional functional programming languages, such as exceptions and state, are not yet available in Delphin. Also the standard constraint domains, such as integers, reals, doubles, strings, and Booleans, are not yet implemented. Constraint domains of this sort, however, are part of the Twelf system, and we therefore do not foresee any fundamental difficulties in adding them to Delphin. In future research we intend to extend Delphin to provide such features.

8. ACKNOWLEDGMENTS

We would like to thank Henrik Nilsson for many helpful discussions which proved to be directly relevant to the implementation of Delphin.

9. REFERENCES

[1] A. W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 247–258, Boston, USA, June 2001.

[2] L. Augustsson. Cayenne - a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.

[3] T. Coquand. An algorithm for testing conversion in type theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.

[4] M. Gabbay and A. Pitts. A new approach to abstract syntax involving binders. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 214–224, Trento, Italy, July 1999. IEEE Computer Society Press.

[5] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, Jan. 1993.

[6] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, California, 1995.

[7] S. Michaylov and F. Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, Jan. 1991. Springer-Verlag LNAI 596.

[8] G. C. Necula. Proof-carrying code. In N. D. Jones, editor, *Conference Record of the 24th Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, Jan. 1997. ACM Press.

[9] F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[10] F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[11] B. Pientka. Termination and reduction checking for higher-order logic programs. In *IJCAR*, pages 401–415, 2001.

[12] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction, MPC2000, Proceedings, Ponte de Lima, Portugal, July 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.

[13] E. Rohwedder and F. Pfenning. Mode and termination checking for higher-order logic programs. In H. R. Nielson, editor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, Apr. 1996. Springer-Verlag LNCS 1058.

[14] C. Schürmann. *Automating the Meta-Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, 2000. CMU-CS-00-146.

[15] C. Schürmann. Recursion for higher-order encodings. In L. Fribourg, editor, *Proceedings of the Conference on Computer Science Logic (CSL 2001)*, pages 585–599, Paris, France, August 2001. Springer Verlag LNCS 2142.

[16] C. Schürmann. A type-theoretic approach to induction with higher-order encodings. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence and Reasoning(LPAR 2001)*, pages 266–281, Havana, Cuba, 2001. Springer Verlag LNAI 2250.

[17] C. Schürmann, R. Fontana, and Y. Liao. The Delphin website: `http://www.cs.yale.edu/~carsten/delphin`, 2002.

[18] R. Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 1999. Forthcoming.

[19] H. Xi and F. Pfenning. Dependent types in practical programming. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 214–227, New York, NY, 1999.

$$\frac{\Sigma(c) = A}{\Gamma \vdash_{\Sigma;\Phi} c : A}\ \textsf{obj\_con} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash_{\Sigma;\Phi} x : A}\ \textsf{obj\_var} \qquad \frac{\Gamma(\underline{y}) = (L;\sigma) \qquad (\text{block } L)[\sigma](x) = A}{\Gamma \vdash_{\Sigma;\Phi} \pi_x(\underline{y}) : A}\ \textsf{obj\_proj}$$

$$\frac{\Gamma \vdash_{\Sigma;\Phi} A_1 : \text{type} \qquad \Gamma, x : A_1 \vdash_{\Sigma;\Phi} M : A_2}{\Gamma \vdash_{\Sigma;\Phi} \lambda x : A_1.\, M : \Pi x : A_1.\, A_2}\ \textsf{obj\_lam} \qquad \frac{\Gamma \vdash_{\Sigma;\Phi} M_1 : \Pi x : A_2.\, A_1 \qquad \Gamma \vdash_{\Sigma;\Phi} M_2 : A_2}{\Gamma \vdash_{\Sigma;\Phi} M_1\ M_2 : A_1[M_2/x]}\ \textsf{obj\_app}$$

$$\frac{\Sigma(a) = K}{\Gamma \vdash_{\Sigma;\Phi} a : K}\ \textsf{tp\_const}$$

$$\frac{\Gamma \vdash_{\Sigma;\Phi} A_1 : \text{type} \qquad \Gamma, x : A_1 \vdash_{\Sigma;\Phi} A_2 : \text{type}}{\Gamma \vdash_{\Sigma;\Phi} \Pi x : A_1.\, A_2 : \text{type}}\ \textsf{tp\_pi} \qquad \frac{\Gamma \vdash_{\Sigma;\Phi} A_1 : \Pi x : A_2.\, K \qquad \Gamma \vdash_{\Sigma;\Phi} M : A_2}{\Gamma \vdash_{\Sigma;\Phi} A_1\ M : K[M/x]}\ \textsf{tp\_app}$$

$$\frac{}{\Gamma \vdash_{\Sigma;\Phi} \text{type} : \text{ kind}}\ \textsf{kd\_type} \qquad \frac{\Gamma \vdash_{\Sigma;\Phi} A : \text{type} \qquad \Gamma, x : A \vdash_{\Sigma;\Phi} K : \text{ kind}}{\Gamma \vdash_{\Sigma;\Phi} \Pi x : A.K \text{ kind}}\ \textsf{kd\_pi}$$

Figure 10: LF Typing Rules.

$$\frac{}{\Gamma;\eta \vdash \mathbf{x} \hookrightarrow \eta(\mathbf{x})}\ \textsf{ev\_var} \qquad \frac{}{\Gamma;\eta \vdash \langle\rangle \hookrightarrow \langle\rangle}\ \textsf{ev\_unit} \qquad \frac{\Gamma;\eta \vdash P_1 \hookrightarrow V_1 \qquad \Gamma;\eta, V_1/\mathbf{x} \vdash P_2 \hookrightarrow V}{\Gamma;\eta \vdash \text{let } \mathbf{x} = P_1 \text{ in } P_2 \hookrightarrow V}\ \textsf{ev\_let}$$

$$\frac{}{\Gamma;\eta \vdash \Lambda x : A.\, P \hookrightarrow \{\eta; \Lambda x : A.\, P\}}\ \textsf{ev\_}\Lambda\textsf{\_LF} \qquad \frac{\Gamma;\eta \vdash P \hookrightarrow \{\eta'; \Lambda x : A.\, P'\} \qquad \Gamma;\eta', M[\eta]/x \vdash P' \hookrightarrow V}{\Gamma;\eta \vdash P\ M \hookrightarrow V}\ \textsf{ev\_app\_LF}$$

$$\frac{}{\Gamma;\eta \vdash \Lambda\mathbf{x} \in F.\, P \hookrightarrow \{\eta; \Lambda\mathbf{x} \in F.\, P\}}\ \textsf{ev\_}\Lambda\textsf{\_delphin}$$

$$\frac{\Gamma;\eta \vdash P_1 \hookrightarrow \{\eta'; \Lambda\mathbf{x} \in F.\, P_1'\} \qquad \Gamma;\eta \vdash P_2 \hookrightarrow V_2 \qquad \Gamma;\eta', V_2/\mathbf{x} \vdash P_1' \hookrightarrow V}{\Gamma;\eta \vdash P_1\ P_2 \hookrightarrow V}\ \textsf{ev\_app\_delphin}$$

$$\frac{}{\Gamma;\eta \vdash \Lambda\underline{x} \in (L;\sigma).\, P \hookrightarrow \{\eta; \Lambda\underline{x} \in (L;\sigma).\, P\}}\ \textsf{ev\_}\Lambda\textsf{\_block}$$

$$\frac{\Gamma, \underline{x} : (L;\sigma);\eta \vdash P \hookrightarrow \Lambda\underline{x} \in (L;\sigma).\, P' \qquad \Gamma;\eta \vdash [\underline{y}/\underline{x}]P' \hookrightarrow V}{\Gamma;\eta \vdash P\ \underline{y} \hookrightarrow V}\ \textsf{ev\_app\_block}$$

$$\frac{\Gamma;\eta \vdash P \hookrightarrow \Lambda\underline{x} \in (L;\sigma).\, P' \qquad \Gamma, \underline{x} : (L;\sigma);\eta, \underline{x}/\underline{x} \vdash P' \hookrightarrow V' \qquad \text{abs } (\text{block } L)[\sigma].\, V' = V}{\Gamma;\eta \vdash \nu P \hookrightarrow V}\ \textsf{ev\_new}$$

$$\frac{\Gamma;\eta \vdash P_1 \hookrightarrow V_1 \qquad \Gamma;\eta \vdash P_2 \hookrightarrow V_2}{\Gamma;\eta \vdash \langle P_1; P_2\rangle \hookrightarrow \langle V_1; V_2\rangle}\ \textsf{ev\_pair} \qquad \frac{\Gamma;\eta \vdash P \hookrightarrow V}{\Gamma;\eta \vdash \langle M; P\rangle \hookrightarrow \langle M[\eta]; V\rangle}\ \textsf{ev\_inx}$$

$$\frac{\Gamma;\eta, \mu\mathbf{x} \in F.\, P/\mathbf{x} \vdash P \hookrightarrow V}{\Gamma;\eta \vdash \mu\mathbf{x} \in F.\, P \hookrightarrow V}\ \textsf{ev\_rec} \qquad \frac{\Gamma \vdash \eta \sim \Omega \hookrightarrow V}{\Gamma;\eta \vdash \text{case } \Omega \hookrightarrow V}\ \textsf{ev\_case}$$

$$\frac{\Gamma;\eta \vdash P[\eta'] \hookrightarrow V \qquad \psi \circ \eta' = \eta}{\Gamma;\eta \vdash \eta \sim (\Omega, (\Psi \triangleright \psi \mapsto P)) \hookrightarrow V}\ \textsf{ev\_yes} \qquad \frac{\Gamma;\eta \vdash \eta \sim \Omega \hookrightarrow V}{\Gamma;\eta \vdash \eta \sim (\Omega, (\Psi \triangleright \psi \mapsto P)) \hookrightarrow V}\ \textsf{ev\_no}$$

Figure 11: Delphin Operational Semantics.