Practical Programming with Higher-Order Encodings and Dependent Types*

Adam Poswolsky[1] and Carsten Schürmann[2]

[1] Yale University poswolsky@cs.yale.edu
[2] IT University of Copenhagen carsten@itu.dk

Abstract. Higher-order abstract syntax (HOAS) refers to the technique of representing variables of an object-language using variables of a meta-language. The standard first-order alternatives force the programmer to deal with superficial concerns such as substitutions, whose implementation is often routine, tedious, and error-prone. In this paper, we describe the underlying calculus of Delphin. Delphin is a fully implemented functional-programming language supporting reasoning over higher-order encodings and dependent types, while maintaining the benefits of HOAS. More specifically, just as representations utilizing HOAS free the programmer from concerns of handling explicit contexts and substitutions, our system permits programming over such encodings without making these constructs explicit, leading to concise and elegant programs. To this end our system distinguishes bindings of variables intended for instantiation from those that will remain uninstantiated, utilizing a variation of Miller and Tiu's ∇-quantifier [4].

1 Introduction

Logical frameworks are meta-languages used to represent information. Any system supporting the declaration of custom datatypes is providing a framework for representing information. Church's simply typed λ-calculus is arguably the first logical framework that supports higher-order encodings, which means that binding constructs of the object language (the information modeled) are expressed in terms of the binding constructs of the λ-calculus. This deceptively simple idea allows for encodings of complex data structures without having to worry about the representation of variables, renamings, or substitutions that are prevalent in logic derivations, typing derivations, operational semantics, and more.

The logical framework LF [3] is essentially an extention of Church's λ-calculus with dependent types and signatures. A signature contains a collection of constants used to construct objects of different types, also known as datatype constructors. Dependent types and type families (type level constants that need to be indexed by objects) can capture invariants about representations that are impossible with just simple types. A list can be indexed by its length. An expression can be indexed by its type. An evaluation relation can be represented as a type indexed by two expressions, its input and output. The list goes on.

* This research has been funded by NSF grants CCR-0325808 and CCR-0133502.

Neither the simply typed λ-calculus nor LF are suitable for programming. Neither framework permits the definition of recursive functions by cases. They are logical frameworks, whose sole purpose is the representation of syntax modulo variable renaming and substitution. Furthermore, we must be careful when adding anything to LF. For example, the addition of case analysis would inevitably lead to *exotic* terms, i.e. typeable terms that do not correspond to any concrete term in the object-language being encoded. The existence of such exotic terms would eliminate the main benefits of higher-order encodings.

Thus, the first challenge of designing a calculus of recursive functions supporting higher-order encodings is to cleanly separate the two function spaces for representation and computation. Our Delphin calculus defines a computation level supporting function definition by case analysis and recursion without extending the representation level LF. Therefore, all of LF's representational features and properties are preserved.

The second challenge of designing our calculus is supporting recursion under representation level (LF) functions. We solve this problem by distinguishing between two methods of variable binding. The *function type constructor* $\forall$ (or $\supset$ when non-dependent) binds variables that are intended for instantiation, which means that computation is delayed until application. Additionally, we provide a *newness type constructor* ∇ to bind variables that will always remain *uninstantiated* and hence computation will not be delayed. The introduction form of ∇ is the ν (pronounced *new*) construct, $\nu x.\ e$, where x can occur free in e. Evaluation of e occurs while the binding x remains uninstantiated. Therefore, for the scope of e, the variable x behaves as a constant in the signature, which we will henceforth call a parameter. One may view ν as a method of dynamically extending the signature.

The Delphin calculus distinguishes between *parameters* (extensions of the signature) and *objects* (built from constants and parameters). The type $A^{\#}$ refers to a parameter of type A. Intuitively, the type $A^{\#}$ is best viewed as a subtype of A. Although all parameters of type A do have type A, the converse does not necessarily hold.

The presence of parameters introduce concerns with respect to case analysis. When performing case analysis over a type, we cannot only consider the constants declared in the signature, but we must also consider parameters. To this end, Delphin permits a versatile definition of cases. Pattern variables of type $A^{\#}$ will be used to capture these additional cases.

Our ∇-type constructor is related to Miller and Tiu's ∇-quantifier [4], where they distinguish between eigenvariables intended for instantiation from those representing scoped constants. In their logic, the formula $(\forall x.\ \forall y.\ \tau(x,\ y)) \supset \forall z.\ \tau(z,\ z)$ is provable, whereas $(\nabla x.\ \nabla y.\ \tau(x,\ y)) \supset \nabla z.\ \tau(z,\ z)$ is not. Similarly, the Delphin type $(\forall x.\ \forall y.\ \tau(x,\ y)) \supset \forall z.\ \tau(z,\ z)$ is inhabited by $\lambda f.\ \lambda z.\ f\ z\ z$. However, the type $(\nabla x.\ \nabla y.\ \tau(x,\ y)) \supset \nabla z.\ \tau(z,\ z)$ is in general not inhabited because nothing might be known about $\tau(z,\ z)$.

In this paper we describe our calculus of recursive functions and its implementation in the Delphin programming language. Delphin is available for download

Types	$A, B ::= a \mid A\ M \mid \Pi x{:}A.\ B$		Signature	$\Sigma ::= \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A$
Objects	$M, N ::= x \mid c \mid M\ N \mid \lambda x{:}A.\ N$		Context	$\Gamma ::= \cdot \mid \Gamma, x{:}A$
Kinds	$K ::= \mathit{type} \mid \Pi x{:}A.\ K$			

Fig. 1. The logical framework LF

at `http://www.cs.yale.edu/~delphin`. We begin this paper with an overview of the logical framework LF in Section 2. We motivate the Delphin language in Section 3, and provide examples in Section 4. We discuss its static semantics in Section 5 followed by the operational semantics in Section 6. Next, we present some meta-theoretical results in Section 7. An advanced example with combinator transformations is given in Section 8. We briefly discuss some implementation details in Section 9. Finally, we describe related work in Section 10 before we conclude and assess results in Section 11.

2 Logical Framework LF

The Edinburgh logical framework [3], or LF, is a meta-language for representing deductive systems defined by judgments and inference rules. Its most prevalent features include dependent types and the support for the higher-order encodings of syntax and hypothetical judgments.

We present the syntactic categories of LF in Figure 1. Function types assign names to their arguments in $\Pi x{:}A.\ B$. We write $A \to B$ as syntactic sugar when x does not occur in B. Types may be indexed by objects and we provide the construct $A\ M$ to represent such types. We write x for variables while a and c are type and object constants (or constructors), respectively. We often refer to a as a type family. These constants are provided in a fixed collection called the signature. The functional programmer may interpret the signature as the collection of datatype declarations.

In the presence of dependencies, not all types are valid. The *kind* system of LF acts as a type system for types. We write $\Gamma \vdash^{\mathsf{lf}} M : A$ for valid objects and $\Gamma \vdash^{\mathsf{lf}} A : K$ for valid types, in a context Γ that assigns types to variables. The typing and kinding rules of LF are standard [3] and are omitted here in the interest of brevity. All LF judgments enjoy the usual weakening and substitution properties on their respective contexts, but exchange is only permitted in limited form due to dependencies. We take $\equiv_{\alpha\beta\eta}$ as the underlying notion of definitional equality between LF-terms. Terms in β-normal η-long form are also called canonical forms.

Theorem 1 (Canonical forms). *Every well-typed object $\Gamma \vdash^{\mathsf{lf}} M : A$ possesses a unique canonical form (modulo α-renaming) $\Gamma \vdash^{\mathsf{lf}} N : A$, such that $M \equiv_{\alpha\beta\eta} N$.*

Encodings consist of a signature and a representation function, which maps elements from our domain of discourse into canonical forms in our logical framework. We say that an encoding is *adequate* if the representation function ($\ulcorner - \urcorner$)

is a compositional bijection (one that commutes with substitution). We next present examples of a few adequate encodings. We write the signature to the right of the representation function.

Example 1 (Natural numbers).

$$\begin{array}{rcl} \ulcorner 0 \urcorner & = & \mathrm{z} \\ \ulcorner n+1 \urcorner & = & \mathrm{s}\, \ulcorner n \urcorner \end{array} \qquad \begin{array}{ll} \mathrm{nat} & : \mathit{type} \\ \mathrm{z} & : \mathrm{nat} \\ \mathrm{s} & : \mathrm{nat} \rightarrow \mathrm{nat} \end{array}$$

Example 2 (Expressions). As another example, we choose the standard language of untyped λ-terms $t ::= x \mid \textbf{lam}\; x.\, t \mid t_1 @ t_2$. The encoding $\ulcorner t \urcorner$ is as follows:

$$\begin{array}{lcl} \ulcorner x \urcorner & = & x \\ \ulcorner \textbf{lam}\; x.\, t \urcorner & = & \mathrm{lam}\; (\lambda x{:}\mathrm{exp}.\, \ulcorner t \urcorner) \\ \ulcorner t_1 @ t_2 \urcorner & = & \mathrm{app}\; \ulcorner t_1 \urcorner\, \ulcorner t_2 \urcorner \end{array} \qquad \begin{array}{l} \mathrm{exp} : \mathit{type} \\ \mathrm{lam} : (\mathrm{exp} \rightarrow \mathrm{exp}) \rightarrow \mathrm{exp} \\ \mathrm{app} : \mathrm{exp} \rightarrow \mathrm{exp} \rightarrow \mathrm{exp} \end{array}$$

In this example, we represent object-level variables x by LF variables x of type exp, which is recorded in the type of lam. As a result, we get substitution for free: $\ulcorner [t_1/x] t_2 \urcorner = [\ulcorner t_1 \urcorner / x] \ulcorner t_2 \urcorner$.

Example 3 (Natural deduction calculus). Let $A, B ::= A \Rightarrow B \mid p$ be the language of formulas. We will use $\Rightarrow$ as an infix operator below. We write $\mathcal{E} :: \vdash A$ if $\mathcal{E}$ is a derivation in the natural deduction calculus. Natural deduction derivations $\mathcal{E} :: \vdash A$ are encoded in LF as $\ulcorner \mathcal{E} \urcorner : \mathrm{nd}\; \ulcorner A \urcorner$, whose signature is given below.

$$\dfrac{\begin{array}{c} \dfrac{}{\vdash A}\,u \\ \vdots \\ \vdash B \end{array}}{\vdash A \Rightarrow B}\,\mathsf{impi} \qquad \dfrac{\vdash A \qquad \vdash A \Rightarrow B}{\vdash B}\,\mathsf{impe} \qquad \begin{array}{l} \mathrm{o} : \mathit{type} \\ \Rightarrow : \mathrm{o} \rightarrow \mathrm{o} \rightarrow \mathrm{o} \\ \\ \mathrm{nd} : \mathrm{o} \rightarrow \mathit{type} \\ \mathrm{impi} : (\mathrm{nd}\; A \rightarrow \mathrm{nd}\; B) \rightarrow \mathrm{nd}\; (A \Rightarrow B) \\ \mathrm{impe} : \mathrm{nd}\; (A \Rightarrow B) \rightarrow \mathrm{nd}\; A \rightarrow \mathrm{nd}\; B. \end{array}$$

We omit the leading Πs from the types when they are inferable. This is, for example, common practice in Twelf. The logical framework LF draws its representational strength from the existence of canonical forms, providing an induction principle that allows us to prove adequacy.

3 Delphin Calculus

The Delphin calculus is specifically designed for programming with (higher-order) LF encodings. It distinguishes between two levels: computational and representational. Its most prominent feature is its newness type constructor ∇, which binds uninstantiable parameters introduced by our ν construct. Figure 2 summarizes all syntactic categories of the Delphin calculus.

We use δ to distinguish between *representational types* A, *parameters* $A^{\#}$, and *computational types* τ.[3] Representational types A are the LF types defined

[3] In the corresponding technical report [6] we also allow for computation-level parameters $\tau^{\#}$, which we omit here for the sake of simplicity.

Types	δ	$::= \tau \mid A \mid A^{\#}$
Computational Types	τ, σ	$::= \top \mid \forall \alpha{\in}\delta.\ \tau \mid \exists \alpha{\in}\delta.\ \tau \mid \nabla x{\in}A^{\#}.\ \tau$
Variables	α	$::= x \mid u$
Expressions	e, f	$::= \alpha \mid M \mid \text{unit} \mid e\ f \mid (e,\ f) \mid \nu x{\in}A^{\#}.\ e \mid e \backslash x$
		$\mid \mu u{\in}\tau.\ e \mid \text{fn}\ (c_1 \mid \ldots \mid c_n)$
Cases	c	$::= \epsilon\alpha{\in}\delta.\ c \mid \nu x{\in}A^{\#}.\ c \mid c\backslash x \mid e \mapsto f$

Fig. 2. Syntactic Definitions of Delphin

in Section 2. We write $A^{\#}$ to denote parameters of type A. Through this distinction we strengthen pattern matching as well as permit functions that range over parameters. It is best to view $A^{\#}$ as a subtype of A. We also distinguish representation-level and computation-level variables by x and u, respectively. Computational types are constructed from four type constructors: the unit type constructor $\top$, the function type constructor $\forall$, the product type constructor $\exists$, and the *newness* type constructor ∇.

Computational types τ disallow computing anything of LF type A. This is necessary as LF types may depend on objects of type A, and we chose to disallow dependencies on computation-level expressions. This separation ensures that the only objects of type A are LF terms M. Although computation cannot result in an object of type A, it may result in an object of type $\exists x{\in}A.\ \top$. We abbreviate this type as $\langle A \rangle$ and summarize all abbreviations in Figure 3.

Since $\forall$ and $\exists$ range over δ, they each provide three respective function and pairing constructs– over A, τ, and $A^{\#}$. For example, a function of type $\forall x{\in}\text{o}.\ \langle \text{nd}\ x \rangle$ computes natural deduction derivations for any formula. In contrast, a function of type $\forall x{\in}\text{o}^{\#}.\ \langle \text{nd}\ x \rangle$ only works on parameters.

As already stated, functions may range over any type δ. We write $\delta \supset \tau$ for $\forall \alpha{\in}\delta.\ \tau$ when α does not occur in τ, which will always be the case when δ is a τ. We define values of Delphin functions as a list of cases fn $(c_1 \mid \ldots \mid c_n)$, which means that we do not introduce an explicit computation-level λ-term. This technique allows us to avoid aliasing of bound variables, which significantly simplifies the presentation of our calculus in the presence of dependent types.

We write a single case as $e \mapsto f$ where e is the *pattern* and f is the *body*. Patterns may contain pattern variables, which are explicitly declared. We use ϵ to declare pattern variables of any type representing objects or parameters. For example, fn $\epsilon u{\in}\tau.\ u \mapsto u$ encodes the identity function on type τ. Multiple cases are captured via alternation, $c_1 \mid c_2$, and $\cdot$ stands for an empty list of cases. A Delphin level λ-binder $\lambda\alpha{\in}\delta.\ e$ may thus be expressed as fn $\epsilon\alpha{\in}\delta.\ \alpha \mapsto e$.

Function application is call-by-value and is written as $e\ f$. During computation, e is expected to yield a set of cases c, of which one that matches the argument is selected and executed. During the matching process, ϵ-bound pattern variables are appropriately instantiated.

The Delphin type for dependent pairs is denoted by $\exists \alpha{\in}\delta.\ \tau$, and its values are pairs of the form $(e,\ f)$, where both e and f are values. We write $\delta \star \tau$ when

$$\begin{array}{rcl} \delta \supset \tau &=& \forall \alpha{\in}\delta.\ \tau \\ \delta \star \tau &=& \exists \alpha{\in}\delta.\ \tau \\ \langle A \rangle &=& \exists x{\in}A.\ \top \\ \langle M \rangle &=& (M, \text{unit}) \\ \lambda\alpha{\in}\delta.\ e &=& \text{fn}\ \epsilon\alpha{\in}\delta.\ \alpha \mapsto e \\ \text{case}\ e\ \text{of}\ cs &=& (\text{fn}\ cs)\ e \end{array} \qquad \begin{array}{l} \text{let}\ (\alpha{\in}\delta,\ u{\in}\tau) = e\ \text{in}\ f \\ \qquad = \text{case}\ e\ \text{of}\ \epsilon\alpha{\in}\delta.\ \epsilon u{\in}\tau.\ (\alpha,\ u) \mapsto f \\ \text{let}\ \langle x \rangle = e\ \text{in}\ f \\ \qquad = \text{case}\ e\ \text{of}\ \epsilon x{\in}A.\ \langle x \rangle \mapsto f \\ \textbf{let}\ \alpha = e\ \text{in}\ f \\ \qquad = (\lambda\alpha{\in}\delta.\ f)\ e \end{array}$$

Fig. 3. Abbreviations

α does not occur in τ, which will always be the case when δ is a τ. Pairs are eliminated via case analysis.

Delphin's newness type constructor is written as $\nabla x{\in}A^{\#}.\ \tau$ and the corresponding values are $\nu x{\in}A^{\#}.\ e$, where e is a value. In Section 6 we will see that a term $\nu x{\in}A^{\#}.\ e$ will *always* evaluate to a term $\nu x{\in}A^{\#}.\ e'$. In other words, evaluation in an extended signature results in values in the same extended signature. Just as ν dynamically extends the signature, the ∇-type is eliminated via $e\backslash x$, which dynamically shrinks the signature to its form before x was introduced.

One may perform case analysis over a ∇-type. This gives us a way to translate between values of the ∇-type and LF's Π-type. For example, we can utilize case analysis to convert between the value $\langle \lambda x.\ M\ x \rangle$ and $\nu x.\ \langle M\ x \rangle$. A Delphin function that would convert the former into the latter would have type $\langle \Pi x : A.\ B \rangle \supset \nabla x{\in}A^{\#}.\ \langle B \rangle$ and be written as fn $\epsilon y{\in}(\Pi x : A.\ B).\ \langle y \rangle \mapsto \nu x{\in}A^{\#}.\ \langle y\ x \rangle$.

Conversely, a function of type $\nabla x{\in}A^{\#}.\ \langle B \rangle \supset \langle \Pi x : A.\ B \rangle$ can be written as fn $\epsilon y{\in}(\Pi x : A.\ B).\ (\nu x{\in}A^{\#}.\ \langle y\ x \rangle) \mapsto \langle y \rangle$. Notice that the pattern is $\langle y\ x \rangle$, illustrating an example of higher-order matching. Just as we introduced the ∇-type to reason over higher-order encodings, we can employ higher-order matching to get rid of it again.

We also remark that we have $\nu\alpha.\ c$ and $c\backslash\alpha$ over cases, which have a similar meaning to their counterparts over expressions. By allowing these constructs to range over cases, we add further flexibility in what we can express with pattern (ϵ-bound) variables. For example, this is useful in implementing *exchange* properties as well as the properties that will be proved in Lemma 1.

Finally, we turn to the usual recursion operator $\mu u{\in}\tau.\ e$. Note that μ can only recurse on Delphin computational types τ and not on LF types A.

4 Examples

We illustrate Delphin with a few examples building on the encodings of natural numbers and expressions given in Section 2.

Example 4 (Addition). The function plus adds two natural numbers.

$$\begin{array}{l} \mu\mathsf{plus}{\in}\langle \text{nat} \rangle \supset \langle \text{nat} \rangle \supset \langle \text{nat} \rangle. \\ \qquad \text{fn}\ \langle \text{z} \rangle \qquad\qquad\quad \mapsto \text{fn}\ \epsilon M{\in}\text{nat}.\langle M \rangle \mapsto \langle M \rangle \\ \qquad \mid \epsilon N{\in}\text{nat}.\langle \text{s}\ N \rangle \mapsto \text{fn}\ \epsilon M{\in}\text{nat}.\langle M \rangle \mapsto \text{let}\ \langle x \rangle = (\mathsf{plus}\ \langle N \rangle\ \langle M \rangle)\ \text{in}\ \langle \text{s}\ x \rangle \end{array}$$

Example 5 (Interpreter).

```
$\mu\mathsf{eval} \in\langle\text{exp}\rangle \supset \langle\text{exp}\rangle.$
    fn $\epsilon E_1{\in}\text{exp}.\ \epsilon E_2{\in}\text{exp}.\ \langle\text{app } E_1\ E_2\rangle$
              $\mapsto$ case ($\mathsf{eval}\ \langle E_1\rangle,\ \mathsf{eval}\ \langle E_2\rangle$) of
                  $\epsilon F{\in}\text{exp} \to \text{exp}.\ \epsilon V{\in}\text{exp}.\ (\langle\text{lam } F\rangle,\ \langle V\rangle) \mapsto \mathsf{eval}\ \langle F\ V\rangle$
     | $\epsilon E{\in}\text{exp} \to \text{exp}.\ \langle\text{lam } E\rangle\ \ \mapsto \langle\text{lam } E\rangle$
```

Example 6 (Beta Reduction). We can reduce redices under λ-binders.

```
$\mu\mathsf{evalBeta} \in\langle\text{exp}\rangle \supset \langle\text{exp}\rangle.$
    fn $\epsilon E_1{\in}\text{exp}.\ \epsilon E_2{\in}\text{exp}.\ \langle\text{app } E_1\ E_2\rangle$
              $\mapsto$ case ($\mathsf{evalBeta}\ \langle E_1\rangle,\ \mathsf{evalBeta}\ \langle E_2\rangle$) of
                   $\epsilon F{\in}\text{exp} \to \text{exp}.\ \epsilon V{\in}\text{exp}.\ (\langle\text{lam } F\rangle,\ \langle V\rangle) \mapsto \mathsf{evalBeta}\ \langle F\ V\rangle$
                 | $\epsilon x{\in}\text{exp}^{\#}.\ \epsilon V{\in}\text{exp}.\ (\langle x\rangle,\ \langle V\rangle) \mapsto \langle\text{app } x\ V\rangle$
     | $\epsilon E{\in}\text{exp} \to \text{exp}.\ \langle\text{lam } E\rangle$
              $\mapsto$ case ($\nu x{\in}\text{exp}^{\#}.\ \mathsf{evalBeta}\ \langle E\ x\rangle$) of
                   $\epsilon E'{\in}\text{exp} \to \text{exp}.(\nu x{\in}\text{exp}^{\#}.\ \langle E'\ x\rangle) \mapsto \langle\text{lam } E'\rangle$
     | $\epsilon x{\in}\text{exp}^{\#}.\ \langle x\rangle \mapsto \langle x\rangle$
```

The $\langle\text{lam } E\rangle$ case illustrates how we handle higher-order terms. Since E is of functional type, we create a parameter x to continue computation with $(E\ x)$ under ν. The term $\nu x{\in}\text{exp}^{\#}.\ \mathsf{evalBeta}\ \langle \text{E}\ x\rangle$ has type $\nabla x{\in}\text{exp}^{\#}.\ \langle\text{exp}\rangle$. Although the introduction of parameters is easy, eliminating them is more difficult. We do this by case analysis, by first stipulating the existence of an E' of functional type and then match against $\langle E'\ x\rangle$. This illustrates an example of higher-order matching. The parameter x cannot escape its scope because E' was declared outside of the scope of x. This lack of dependency is reflected by the lexical scoping in the Delphin code above: the pattern variable $\epsilon E'$ is declared to the left of νx.

Finally, the base case is required for completeness. New parameters are introduced in the lam case and we specify here that they reduce to themselves.

Example 7 (Variable Counting). For the final example in this section, we write a function that counts the number of variable occurrences in untyped λ-terms. For example, the number of variables in $\ulcorner \textbf{lam}\ x.\ x@(\textbf{lam}\ y.\ x@y)\urcorner$ is $\ulcorner 3\urcorner$.

```
$\mu\mathsf{cntvar}{\in}\langle\text{exp}\rangle \supset \langle\text{nat}\rangle.$
   fn $\epsilon E_1{\in}\text{exp}.\epsilon E_2{\in}\text{exp}.\langle\text{app } E_1\ E_2\rangle$   $\mapsto \mathsf{plus}\ (\mathsf{cntvar}\ \langle E_1\rangle)\ (\mathsf{cntvar}\ \langle E_2\rangle)$
     | $\epsilon E{\in}(\text{exp} \to \text{exp}).\langle\text{lam } E\rangle$         $\mapsto$ case ($\nu x{\in}\text{exp}^{\#}.\ \mathsf{cntvar}\ \langle \text{E}\ x\rangle$) of
                                                  $\epsilon N{\in}\text{nat}.(\nu x{\in}\text{exp}^{\#}.\ \langle N\rangle) \mapsto \langle N\rangle$
     | $\epsilon x{\in}\text{exp}^{\#}.\ \langle x\rangle$                     $\mapsto \langle\text{s z}\rangle$
```

We explain the $\langle\text{lam } E\rangle$ case. Since E is of functional type, we create a parameter $x{:}\text{exp}^{\#}$ and recurse on $\langle E\ x\rangle$. From the very definition of natural numbers in Example 1, we deduce that it is impossible for the result to depend on x and express this by matching against $\langle N\rangle$ instead of $\langle N'\ x\rangle$. Note that if it was possible for x to occur in the result then this case would only match, during runtime, in situations where the x did not occur free in the result. Therefore, if the programmer leaves out essential cases then it is possible to get stuck, corresponding to a *match non-exhaustive* error, just as in ML.

5 Static Semantics

Before presenting the typing rules, the role of context deserves special attention.

$$\text{Contexts } \Omega ::= \cdot \mid \Omega, \alpha{\in}\delta \mid \Omega, x\overset{\triangledown}{\in}A^{\#}$$

A Delphin context, Ω, serves two purposes. Besides assigning types to variables, it also distinguishes between variables intended for instantiation from uninstantiable parameters. We write $\alpha{\in}\delta$ to express variables α that will be instantiated, such as pattern variables. Alternatively, we write $x\overset{\triangledown}{\in}A^{\#}$ to store information about uninstantiable parameters, introduced by ν. The distinction between $x{\in}A^{\#}$ and $x\overset{\triangledown}{\in}A^{\#}$ is highlighted by comparing $\lambda x{\in}A^{\#}.\ e$ and $\nu x{\in}A^{\#}.\ e$. The first binds a parameter that is intended for instantiation while the latter will remain uninstantiated. We do not allow reorderings of Ω because of dependencies. Additionally, we assume all declarations in Ω to be uniquely named, and we achieve this goal by tacitly renaming variables. During the actual execution of Delphin programs, Ω only contains declarations of the latter form, which one may interpret as an extension to the signature. In comparison, computation in ML always occurs with a fixed signature.

Definition 1 (Casting). *In order to employ LF typing, we define $\|\Omega\|$ as casting of a context Ω, which throws out all declarations $u{\in}\tau$ and converts $x{\in}A$, $x{\in}A^{\#}$, and $x\overset{\triangledown}{\in}A^{\#}$ all into $x{:}A$, yielding an LF context Γ.*

5.1 Type System

In the presence of dependencies, not all types are valid. We write $\Omega \vdash \delta$ wff for valid types and Ω ctx for valid contexts, but omit both judgments here due to space considerations. We write $\Omega \vdash e \in \delta$ for the central derivability judgment, which we present in Figure 4. Note that the rules have implicit premises using the validity judgments to ensure that the context and all types are well-formed. We make these explicit in the corresponding Technical Report [6].

The variable rules $\tau\mathsf{var}$ and $\mathsf{var}^{\#}$ allow one to use assumptions in the context of types τ and $A^{\#}$, respectively. The only term of type $A^{\#}$ is a variable x.

The rule isLF is the only rule for type A and stipulates that in order for an expression M to be an LF term, we must be able to type it using the LF typing judgment under $\|\Omega\|$ (Definition 1).

The rest of the rules deal with computational types τ. Function types are introduced via cases c. The introduction rule impI expresses that all branches must have the same type. Note that we allow for an empty list of cases which may be used to write a function over an empty type. Functions are eliminated

$$\frac{(u{\in}\tau)\text{ in }\Omega}{\Omega \vdash u \in \tau}\,\mathsf{\tau var} \qquad \frac{((x{\in}A^{\#})\text{ or }(x\overset{\triangledown}{\in}A^{\#}))\text{ in }\Omega}{\Omega \vdash x \in A^{\#}}\,\mathsf{var}^{\#} \qquad \frac{\|\Omega\| \vdash^{\mathrm{lf}} M : A}{\Omega \vdash M \in A}\,\mathsf{isLF}$$

$$\frac{i \geq 0, \text{For all } i, \Omega \vdash c_i \in \tau}{\Omega \vdash \text{fn } (c_1 \mid \ldots \mid c_n) \in \tau}\,\mathsf{impI} \qquad \frac{\Omega \vdash e \in \forall\alpha{\in}\delta.\ \tau \qquad \Omega \vdash f \in \delta}{\Omega \vdash e\ f \in \tau[f/\alpha]}\,\mathsf{impE}$$

$$\frac{\Omega, x\overset{\triangledown}{\in}A^{\#} \vdash e \in \tau}{\Omega \vdash \nu x{\in}A^{\#}.\ e \in \nabla x{\in}A^{\#}.\ \tau}\,\mathsf{new} \qquad \frac{\Omega \vdash e \in \nabla x'{\in}A^{\#}.\ \tau}{\Omega, x\overset{\triangledown}{\in}A^{\#}, \Omega_2 \vdash e\backslash x \in \tau[x/x']}\,\mathsf{pop}$$

$$\frac{\Omega \vdash e \in \delta \qquad \Omega \vdash f \in \tau[e/\alpha]}{\Omega \vdash (e,\ f) \in \exists\alpha{\in}\delta.\ \tau}\,\mathsf{pairI} \qquad \frac{\Omega, u{\in}\tau \vdash e \in \tau}{\Omega \vdash \mu u{\in}\tau.\ e \in \tau}\,\mathsf{fix} \qquad \frac{}{\Omega \vdash \text{unit} \in \top}\,\mathsf{top}$$

. . . .

$$\frac{\Omega, \alpha{\in}\delta \vdash c \in \tau}{\Omega \vdash \epsilon\alpha{\in}\delta.\ c \in \tau}\,\mathsf{cEps} \qquad \frac{\Omega \vdash e \in \delta \qquad \Omega \vdash f \in \tau[e/\alpha]}{\Omega \vdash e \mapsto f \in \forall\alpha{\in}\delta.\ \tau}\,\mathsf{cMatch}$$

$$\frac{\Omega, x\overset{\triangledown}{\in}A^{\#} \vdash c \in \tau}{\Omega \vdash \nu x{\in}A^{\#}.\ c \in \nabla x{\in}A^{\#}.\ \tau}\,\mathsf{cNew} \qquad \frac{\Omega \vdash c \in \nabla x'{\in}A^{\#}.\ \tau}{\Omega, x\overset{\triangledown}{\in}A^{\#}, \Omega_2 \vdash c\backslash x \in \tau[x/x']}\,\mathsf{cPop}$$

Fig. 4. Delphin Typing Rules

through application with impE. The elimination refines τ under a substitution $[f/\alpha]$ replacing all occurrences of α by f. Formally, we use simultaneous substitutions but refer the interested reader to the corresponding technical report [6] for details. If δ is a computational-type σ, then α cannot occur free in τ and this substitution will be vacuous.

Cases contain explicit pattern variables, which are simply added to the context in cEps. The actual function type is introduced in cMatch illustrating that functions are defined via case analysis. In the branch $e \mapsto f$, e is the pattern and f is the body. The type of f is refined by its pattern via a substitution $\tau[e/\alpha]$. This expresses how different bodies may have different types, all depending on their corresponding pattern. As we define functions by cases, we do not need to refine the context Ω. Additionally, our distinction between computation-level and representation-level types ensures that this substitution is always defined. Finally, we also have a ν and $c\backslash x$ construct over cases, via cNew and cPop. These have similar semantics to their counterparts on expressions, discussed next.

The introduction form of ∇ is called new. As discussed in Section 3, the type $\nabla x{\in}A^{\#}.\ \tau$ declares $x\overset{\triangledown}{\in}A^{\#}$ as a new parameter. The expression $\nu x{\in}A^{\#}.\ e$ evaluates e where the parameter x can occur free. Previously, our examples have shown how to utilize higher-order matching via case-analysis to eliminate these types. However, the elimination rule pop eliminates a ∇-type via an application-like construction, $e\backslash x$, which shifts computation of e to occur without the uninstantiable parameter x. If $\Omega \vdash e \in \nabla x'{\in}A^{\#}.\ \tau$, then x' is a fresh uninstantiable parameter with respect to the context Ω. Therefore, in an extended context $\Omega, x\overset{\triangledown}{\in}A^{\#}, \Omega_2$, we can substitute x for x' and yield a term of type $\tau[x/x']$. The following lemma illustrates examples where this is useful.

Lemma 1. *The following types are inhabited.*

1. $\nabla x{\in}A^{\#}.\ (\tau \supset \sigma) \supset (\nabla x{\in}A^{\#}.\ \tau \supset \nabla x{\in}A^{\#}.\ \sigma)$
2. $(\nabla x{\in}A^{\#}.\ \tau \supset \nabla x{\in}A^{\#}.\ \sigma) \supset \nabla x{\in}A^{\#}.\ (\tau \supset \sigma)$
3. $\nabla x{\in}A^{\#}.\ (\tau \star \sigma) \supset (\nabla x{\in}A^{\#}.\ \tau \star \nabla x{\in}A^{\#}.\ \sigma)$
4. $(\nabla x{\in}A^{\#}.\ \tau \star \nabla x{\in}A^{\#}.\ \sigma) \supset \nabla x{\in}A^{\#}.\ (\tau \star \sigma)$

Proof. We only show 1 and 2, the other 2 cases are straightforward.

1. $\lambda u_1{\in}\nabla x{\in}A^{\#}.\ (\tau \supset \sigma).\ \lambda u_2{\in}(\nabla x{\in}A^{\#}.\ \tau).\ \nu x{\in}A^{\#}.\ (u_1\backslash x)\ (u_2\backslash x)$
2. $\lambda u_1{\in}(\nabla x{\in}A^{\#}.\ \tau \supset \nabla x{\in}A^{\#}.\ \sigma).$
   $\text{fn}\ \epsilon E{\in}(\nabla x{\in}A^{\#}.\ \tau).\ \nu x{\in}A^{\#}.\ ((E\backslash x) \mapsto (u_1\ E)\backslash x)$

Finally, pairs are introduced via pairI and eliminated using case analysis. The typing rules for recursion (fix) and unit (top) are standard.

6 Operational Semantics

Definition 2 (Values). *The set of values of are:*

$$\textit{Values: } v ::= \text{unit} \mid \text{fn}\ (c_1 \mid \ldots \mid c_n) \mid \nu x{\in}A^{\#}.\ v \mid (v_1,\ v_2) \mid M$$

As usual for a call-by-value language, functions are considered values. A newness term $\nu x{\in}A^{\#}.\ v$ is a value only if its body is a value, which is achieved via evaluation under the ν-construct. LF terms M are the only values (and expressions) of type A, and pairs are considered values only if their components are values. Therefore, $\langle M\rangle$ is the only value of type $\langle A\rangle$ (Figure 3).

We present the small-step operational semantics, $\Omega \vdash e \rightarrow f$, in Figure 5. The first rule illustrates that the evaluation of $\nu x{\in}A^{\#}.\ e$ simply evaluates e under the context extended with x. The declaration is marked as $x\overset{\triangledown}{\in}A^{\#}$ as this represents an extension to the signature. Evaluation under ν drives our ability to reason under LF λ-binders. Additionally, we evaluate $e'\backslash x$ by first evaluating e' down to $\nu x'{\in}A^{\#}.\ e$ and then substitute x for x'. Therefore, we see that $e'\backslash x$ behaves much like an application.

The small-step operational semantics for cases, $\Omega \vdash c \rightarrow c'$, is also shown in Figure 5. The first rule non-deterministically instantiates the pattern variables. In our implementation we delay this choice and instantiate them by unification during pattern matching, which is discussed briefly in Section 9. The next three rules allow us to work with ν over cases, which is the same for the ν over expressions. We provide a rule to reduce the pattern of a case branch, which can be any arbitrary expression. In Section 4 we discussed how a program could get

$$\dfrac{\Omega, x\overset{\triangledown}{\in}A^{\#} \vdash e \to f}{\Omega \vdash \nu x{\in}A^{\#}.\ e \to \nu x{\in}A^{\#}.\ f} \qquad \dfrac{\Omega \vdash e \to e'}{\Omega \vdash (e,\ f) \to (e',\ f)} \qquad \dfrac{\Omega \vdash f \to f'}{\Omega \vdash (e,\ f) \to (e,\ f')} \qquad \dfrac{\Omega \vdash e \to e'}{\Omega \vdash e\ f \to e'\ f}$$

$$\dfrac{\Omega \vdash f \to f'}{\Omega \vdash e\ f \to e\ f'} \qquad \dfrac{\Omega \vdash e \to f}{\Omega, x\overset{\triangledown}{\in}A^{\#}, \Omega_2 \vdash e\backslash x \to f\backslash x} \qquad \dfrac{}{\Omega, x\overset{\triangledown}{\in}A^{\#}, \Omega_2 \vdash (\nu x'{\in}A^{\#}.\ e)\backslash x \to e[x/x']}$$

$$\dfrac{}{\Omega \vdash (\text{fn } (c_1 \mid \ldots \mid c_n))\backslash x \to \text{fn } ((c_1\backslash x) \mid \ldots \mid (c_n\backslash x))} \qquad \dfrac{\Omega \vdash c_i \to c_i'}{\Omega \vdash (\text{fn } (\ldots \mid c_i \mid \ldots))\ v \to (\text{fn } (\ldots \mid c_i' \mid \ldots))\ v}$$

$$\dfrac{}{\Omega \vdash (\text{fn } (\ldots \mid v \mapsto e \mid \ldots))\ v \to e}\,* \qquad \dfrac{}{\Omega \vdash \mu u{\in}\tau.\ e \to e[\mu u{\in}\tau.\ e/u]}$$

$$\dfrac{\Omega \vdash v \in \delta}{\Omega \vdash \epsilon\alpha{\in}\delta.\ c \to c[v/\alpha]} \qquad \dfrac{\Omega, x\overset{\triangledown}{\in}A^{\#} \vdash c \to c'}{\Omega \vdash \nu x{\in}A^{\#}.\ c \to \nu x{\in}A^{\#}.\ c'} \qquad \dfrac{\Omega \vdash c \to c'}{\Omega, x\overset{\triangledown}{\in}A^{\#}, \Omega_2 \vdash c\backslash x \to c'\backslash x}$$

$$\dfrac{}{\Omega, x\overset{\triangledown}{\in}A^{\#}, \Omega_2 \vdash (\nu x'{\in}A^{\#}.\ c)\backslash x \to c[x/x']} \qquad \dfrac{\Omega \vdash e \to e'}{\Omega \vdash (e \mapsto f) \to (e' \mapsto f)}$$

Fig. 5. Small-Step Operational Semantics

stuck, which corresponds to a *match non-exhaustive* error. However, we say that a program "coverage checks" if the list of patterns is exhaustive.

Recall that all LF terms possess a unique canonical form. Given any Delphin term, we implicitly reduce all LF terms to canonical form allowing us to express matching via syntactic equality in the rule marked with *.

7 Meta-Theoretic Results

We show here that Delphin is type-safe when all cases are exhaustive.

Lemma 2 (Substitution).
If $\Omega \vdash e \in \delta$ and $\Omega, \alpha{\in}\delta \vdash f \in \tau$, then $\Omega \vdash f[e/\alpha] \in \tau[e/\alpha]$.

Proof. We actually prove this for a more general notion of simultaneous substitutions. See Technical Report [6] for details.

Theorem 2 (Type Preservation).
If $\Omega \vdash e \in \tau$ and $\Omega \vdash e \to f$ then $\Omega \vdash f \in \tau$.

Proof. By induction on the structure of $\mathcal{E} :: \Omega \vdash e \to f$ and $\mathcal{F} :: \Omega \vdash c \to c'$. See Technical Report [6] for details.

Corollary 1 (Soundness). *Parameters cannot escape their scope. If $\Omega \vdash e \in \tau$ and $\Omega \vdash e \to e'$ then all parameters in e and e' are declared in Ω.*

Theorem 3 (Progress).
Under the condition that all cases in e are exhaustive, if $\Omega \vdash e \in \tau$ and Ω only contains declarations of the form $x\overset{\triangledown}{\in}A^{\#}$, then $\Omega \vdash e \to f$ or e is a value.

Proof. *By induction over $\mathcal{E} :: \Omega \vdash e \in \tau$. In matching (rule *) we assume that cases are exhaustive and defer to an orthogonal "coverage check." The Delphin implementation contains a prototype coverage algorithm extending ideas from [8], but a formal description is left for future work. Although the problem of checking an arbitrary list of cases is undecidable, it is always possible to generate an exhaustive list of cases for any type δ.*

8 Combinator Example

Recall the definition of the natural deduction calculus from Example 3. We will give an algorithmic procedure that converts natural deduction derivations into the Hilbert calculus, i.e. simply typed λ-terms into combinators. We omit the declaration of inferable pattern variables (as is also allowed in the implementation).

$$\frac{}{\vdash A \supset B \supset A}\,\mathsf{K} \qquad \frac{\vdash A \supset B \quad \vdash A}{\vdash B}\,\mathsf{MP}$$

$$\frac{}{\vdash (A \supset B \supset C) \supset (A \supset B) \supset (A \supset C)}\,\mathsf{S}$$

comb : o $\rightarrow$ *type*,

K	: comb $(A \Rightarrow B \Rightarrow A)$
MP	: comb$(A \Rightarrow B) \rightarrow$ comb A $\rightarrow$ comb B
S	: comb $((A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C)$

Our translation follows a two-step algorithm. The first step is bracket abstraction, or ba, which internalizes abstraction. If M has type (comb $A \rightarrow$ comb B) and N has type (comb A) then we can use ba to get a combinator, d, of type (comb $A \Rightarrow B$). Subsequently, we can do (MP d N) to get a term that is equivalent to (M N) in combinator logic. Formally, ba is written as.

```
$\mu\mathsf{ba} \in \forall A{\in}\mathrm{o}.\ \forall B{\in}\mathrm{o}.\ \langle \mathrm{comb}\ A \rightarrow \mathrm{comb}\ B\rangle \supset \langle \mathrm{comb}\ (A \Rightarrow B)\rangle.$
 $\mathrm{fn}\ A \mapsto$
    $(\mathrm{fn}\ A \mapsto \mathrm{fn}\ F \mapsto \langle \mathrm{MP}\ (\mathrm{MP\ S\ K})\ \mathrm{K}\rangle$
     $\mid\ B \mapsto \mathrm{fn}\ \ \langle \lambda x.\ \mathrm{MP}\ (D_1\ x)\ ((D_2{:}\mathrm{comb\ A} \rightarrow \mathrm{comb}\ C)\ x)\rangle$
                              $\mapsto\ \ \mathrm{let}\ \langle D_1'\rangle = (\mathsf{ba}\ A\ (C \Rightarrow B)\ \langle D_1\rangle)\ \mathrm{in}$
                                  $\mathrm{let}\ \langle D_2'\rangle = (\mathsf{ba}\ A\ \ C\ \langle D_2\rangle)\ \mathrm{in}\ \ \langle \mathrm{MP}\ (\mathrm{MP\ S}\ D_1')\ D_2'\rangle$
                $\mid\ \langle \lambda x.\ U\rangle \mapsto \langle \mathrm{MP\ K}\ U\rangle)$
```

Next we write the function convert which traverses a natural deduction derivation and uses ba to convert them into Hilbert style combinators. In this function, we will need to introduce new parameters of (nd A) and (comb A) together. In order to hold onto the relationship between these parameters, we pass around a function of type $\forall A{\in}\mathrm{o}.\ \forall D{\in}(\mathrm{nd}\ A)^{\#}.\ \langle \mathrm{comb}\ A\rangle$. We will employ type aliasing and abbreviate this type as convParamFun.

```
$\mu\mathsf{convert} \in \mathsf{convParamFun} \supset \forall A{\in}\mathrm{o}.\ \forall D{\in}\langle \mathrm{nd}\ A\rangle.\ \langle \mathrm{comb}\ A\rangle.$
    $\lambda f{\in}\ \mathsf{convParamFun}.$
       $\mathrm{fn}\ (B \Rightarrow C) \mapsto \mathrm{fn}\ \langle \mathrm{impi}\ D'\rangle \mapsto$
             $(\mathrm{case}\ (\nu d{\in}(\mathrm{nd}\ B)^{\#}.\ \nu d_u{\in}(\mathrm{comb}\ B)^{\#}.$
```

$$\begin{array}{l}
\qquad\qquad\text{let } f' = \text{fn } B \mapsto \text{fn } d \mapsto \langle d_u \rangle \\
\qquad\qquad\qquad\quad \mid (\epsilon B'.\ \epsilon d'.\ \nu d.\ \nu d_u.\ (B' \mapsto \text{fn } d' \mapsto \\
\qquad\qquad\qquad\qquad\quad (\text{let } R = f\ B'\ d' \text{ in } \nu d.\ \nu d_u.\ R)\backslash d\backslash d_u))\backslash d\backslash d_u \\
\qquad\qquad\text{in } \mathsf{convert}\ \ f'\ \ C\ \langle D'\ d \rangle) \\
\qquad\quad \text{of } \nu d{\in}(\text{nd } B)^{\#}.\ \nu d_u{\in}(\text{comb } B)^{\#}.\ \langle D''\ d_u \rangle \mapsto \mathsf{ba}\ B\ \ C\ \langle D'' \rangle) \\
\mid\ A \mapsto \text{fn } \langle \text{impe } D_1\ (D_2{:}\text{nd } B) \rangle \mapsto \\
\qquad\qquad\qquad \text{let } \langle U_1 \rangle = (\mathsf{convert}\ f\ (B \Rightarrow A)\ \langle D_1 \rangle) \text{ in} \\
\qquad\qquad\qquad \text{let } \langle U_2 \rangle = (\mathsf{convert}\ f\ B\ \langle D_2 \rangle) \text{ in } \langle \text{MP } U_1\ U_2 \rangle \\
\mid\ A \mapsto \text{fn } \epsilon x{\in}(\text{nd } A)^{\#}.\ \langle x \rangle \mapsto f\ A\ x
\end{array}$$

The first argument to convert is a computation-level function f of type convParamFun that handles the parameters.

The first case, $\langle \text{impi } D \rangle$, requires recursion under a representation-level λ. We create two new parameters (or equivalently extend the signature with) d and d_u in order to continue our computation by recursing on $\langle D'\ d \rangle$. As we are in an extended signature, if f was a total function on input, it is no longer total. We therefore extend the function f into f' mapping d to d_u before recursing. We then use the same techniques from Examples 6 and 7 to abstract the result into an LF function D'' exploiting that d cannot occur free in the result. Finally, we employ ba to yield our desired combinator.

The second case does not create any parameters and hence all recursive calls are called with f. Finally, the last case handles the parameters by simply calling the input function f which has been built up to handle all parameters.

The above definition of f' illustrates how one can build up parameter functions. The second branch of f' utilizes $e\backslash x$ and $c\backslash x$ (Section 5.1) to ensure that the input function f is not executed in scope of d and d_u. The Delphin implementation offers a shorthand to extend a function f by writing "f `with` $d \ \mapsto\ d_u$".

Example 8 (Sample Execution).

$$\begin{array}{l}
\nu A.\ \mathsf{convert}\ (\text{fn } \cdot)\ (A \Rightarrow A)\ \langle \text{impi } \lambda x.\ x \rangle \\
\ldots \rightarrow \nu A.\ \ \text{case } (\nu d.\ \nu d_u.\ \mathsf{convert}\ (\text{fn } A \mapsto \text{fn } d \mapsto \langle d_u \rangle \mid \ldots)\ A\ \langle d \rangle) \\
\qquad\qquad\quad \text{of } \nu d.\ \nu d_u.\ \langle D''\ d_u \rangle \mapsto \mathsf{ba}\ A\ A\ \langle D'' \rangle \\
\ldots \rightarrow \nu A.\ \ \text{case } (\nu d.\ \nu d_u.\ (\text{fn } A \mapsto \text{fn } d \mapsto \langle d_u \rangle \mid \ldots)\ A\ d) \\
\qquad\qquad\quad \text{of } \nu d.\ \nu d_u.\ \langle D''\ d_u \rangle \mapsto \mathsf{ba}\ A\ A\ \langle D'' \rangle \\
\ldots \rightarrow \nu A.\ \ \text{case } (\nu d.\ \nu d_u.\ \langle d_u \rangle) \text{ of } \nu d.\ \nu d_u.\ \langle D''\ d_u \rangle \mapsto \mathsf{ba}\ A\ A\ \langle D'' \rangle \\
\ldots \rightarrow \nu A.\ \mathsf{ba}\ A\ A\ \langle \lambda x.\ x \rangle \\
\ldots \rightarrow \nu A.\ \langle \text{MP (MP S K) K} \rangle
\end{array}$$

9 Implementation

An implementation is available at `http://www.cs.yale.edu/~delphin`. Delphin is implemented in approximately 12K lines of code in SML/NJ offering a powerful type reconstruction algorithm, typechecker, and evaluator.

The non-deterministic instantiation of pattern variables from Section 6 is implemented by using logic variables to delay the choice until matching. Additionally, when writing a curried function with multiple arguments we look at all the arguments together before committing to a branch. We implement this feature by partially evaluating functions. For example, convert $(A \Rightarrow A)$ will result in a function with three cases rather than committing to the first branch. This is an enhancement to allow the programmer to write more concise code.

We employ a unification/matching algorithm based on the one designed by Dowek et al. [1], but extended to handle parameters. Therefore, we only allow LF patterns that fall into the decidable pattern fragment of higher-order unification. Formally, this means that we only allow LF patterns of the form $E\ x_1\ \ldots\ x_n$ where x_i is a fresh parameter (with respect to E) and all x_i's are distinct. It is important to note that this restriction is only an implementation limitation as it is also possible to use different unification algorithms.

The Delphin code for all examples in this paper and many more can be found on our website. We have implemented a function translating HOL proofs into Nuprl proofs (approximately 400 lines of code) and a Hindley-Milner style type-inference algorithm for Mini-ML (approximately 300 lines of code).

10 Related Work

Twelf: LF is well suited for *representation* but does not directly afford the ability to reason over representations. Twelf utilizes a logic programming methodology to conduct such reasoning by providing meta-level constructs to interpret a type family as a function. Delphin affords the user the ability to write the functions themselves, and we envision this will replace the underlying meta-logic of Twelf.

Higher-order encodings: The predecessor of our work was on the ∇-calculus [9], which provided a stack based system only supporting a simply-typed logical framework. The ∇ also referred to something different than what it does here.

Our work is related to Miller and Tiu's [4]. In their setting, they use ∇ as a logic quantifier designed to reason about scoped constants. However, their reasoning occurs over formulas *with* an explicit local context. In our setting there is only a global context, which renders it more useful for functional programming.

Pientka[5] also proposes a system for programming with HOAS, however only for a simply-typed logical framework. Programming over HOAS resorts to the explicit handling of substitutions and contexts. In contrast, we believe the purpose of HOAS is to provide an implicit notion of substitution. Therefore, we provide a computation-level in the same spirit, keeping these constructs hidden.

Dependent types: DML provides indexed datatypes whose domains were recently generalized to LF objects to form the ATS/LF system. In contrast, the Cayenne language supports full dependent types and even computation with types, rendering it more expressive but at the expense of an undecidable type checker. Agda and Epigram are two more languages inspired by dependent type theories. All but the ATS/LF system lack support for higher-order encodings. Although ATS/LF supports HOAS they resort to encoding the context explicitly, or as they say representing terms as *terms-in-contexts*. By making this information explicit they can reason about parameters in the context, but they must also define substitutions. We suspect that they can also add a ∇-type similar to ours.

Freshness: Also related to our work are programming languages with freshness [2], such as FreshML, which utilizes Fraenkel-Mostowski (FM) set theory to provide a built-in α-equivalence relation for first-order encodings. This allows for limited support of HOAS as substitution lemmas must still be explicit, albeit easier to write. Lately, Pottier has developed a logic for reasoning about values and the names they contain in FreshML [7]. As the creation of names is a *global* effect in FreshML, his work is used to prove that names cannot escape their scope, which is an inherent property of Delphin's type system.

11 Conclusion

In this paper we have presented the underlying calculus and semantics of the Delphin programming language. This is the only functional system tackling programming over a logical framework with both higher-order encodings and dependent types. The novelty of this work is in providing a way to reason under LF λ-binders, such that the notions of context and substitutions remain implicit in computations as well as representations.

Acknowledgments. We would like to thank Jeffrey Sarnat and Lucas Dixon for many helpful discussions on this and many earlier designs of the system.

References

1. G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Unification via explicit substitutions: The case of higher-order patterns. Rapport de Recherche 3591, INRIA, Dec. 1998. Preliminary version appeared at JICSLP'96.
2. M. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects Computing*, 13(3-5):341–363, 2002.
3. 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.
4. D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, Oct. 2005.
5. B. Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Principles of Programming Languages, POPL*, 2008.
6. A. Poswolsky and C. Schürmann. Extended report on delphin: A functional programming language with higher-order encodings and dependent types. Technical Report YALEU/DCS/TR-1375, Yale University, 2007.
7. F. Pottier. Static name control for FreshML. In *Twenty-Second Annual IEEE Symposium on Logic In Computer Science (LICS'07)*, Wroclaw, Poland, July 2007.
8. C. Schürmann and F. Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proccedings of Theorem Proving in Higher Order Logics (TPHOLs'03)*, volume LNCS-2758, Rome, Italy, 2003. Springer Verlag.
9. C. Schürmann, A. Poswolsky, and J. Sarnat. The ∇-calculus. Functional programming with higher-order encodings. In *Typed Lambda Calculus and Applications, TLCA*, 2005.