

# Towards Practical Functional Programming with Logical Frameworks Extended version

Carsten Schürmann  
Yale University

July 26, 2003

## Abstract

In this paper we show that the logical framework LF [6] extended by  $\Sigma$ -types serves as an excellent candidate for the representation of special purpose domains that are best encoded using higher-order abstract syntax and hypothetical judgments. It has an elegant meta-theory and  $\Sigma$ -types prove enormously useful in the design of a functional programming language Delphin that permits programmers to work directly with those encodings. Applications of this work include predominantly deductive software development, special purpose theorem provers especially for proof carrying safety architectures such as proof carrying authentication PCA [2], foundational proof carrying code PCC [8, 1], and interpreters and compilers for experimental programming languages. This paper presents the underlying logical framework  $\text{LF}^\Sigma$  and the functional programming language Delphin.

## 1 Introduction

The logical framework LF [6] is a meta-language designed to encode deductions that are prevalent when working with proofs, derivations, certificates, and type systems. With  $\beta\eta$  as definitional equality it is strongly normalizing and every term is identified by its canonical form. From a theoretical perspective, this means that the widely studied concept of higher-order encodings and hypothetical judgments find nice and elegant representations in the logical framework, but from a practical perspective neither LF nor the higher-order encodings are often used in programming practice. The absence of a programming language that supports those encodings as first-class objects leaves programmers retreating to traditional languages, s.t. ML or Haskell that are arranged around the notion of a standard datatype, which is largely incompatible with the higher-orderedness of LF encodings. In this paper, we present the language Delphin, that is designed to fill this void.

In deductive software, which includes all software that has to do with theorem provers, model checkers, security applications, . . . , programmers typically have to handle complex and composite objects, such as proof trees, substitutions, theorems, or translators. Special purpose theorem provers designed for proof carrying safety architectures such as proof carrying authentication PCA [2], proof carrying code PCC [8, 1] are important pieces of software that must be programmed, maintained, and refined. The more direct complex data objects are encoded, the clearer and more intuitive the software tends to be.

In this paper we argue that a prime candidate for representing complex encodings is the logical framework  $\text{LF}^\Sigma$  [14], a variant of LF extended with  $\Sigma$ -types.  $\text{LF}^\Sigma$  enjoys almost all the pleasant properties of LF except for unique types. Datatypes in  $\text{LF}^\Sigma$  support full-higher order encodings without any positivity restriction. To pinpoint the property that makes  $\text{LF}^\Sigma$  and Delphin interact so nicely is that  $\text{LF}^\Sigma$  offers a *parametric* function space orthogonal to the *non-parametric* space of Delphin. In Delphin functions can be defined by cases, but not in  $\text{LF}^\Sigma$ . In addition, different from the regular formulation of LF,  $\text{LF}^\Sigma$  does not permit object level constant declarations.

Because of  $\text{LF}^\Sigma$ , the benefits of Delphin include full capabilities of higher-order abstract syntax and hypothetical judgments, an expressive type system that captures invariant at a finer level of detail than non dependent typed systems, and elegant code. The drawbacks include a necessary and inadvertent distinction of functions that are used for the encoding of data vs. functions designated for programming, and the absence of polymorphic functions, which we have not studied yet but plan to address in future work. In this sense, Delphin is more limited than other functional languages, we would however argue that for many applications, especially deductive software, the benefits outweigh the drawbacks by far.

The basic idea underlying this work is to eradicate the idea of datatypes being given as a set of constructors. Instead we propose to think of a datatype as a type together with  $\Sigma$ -types, representing the types of all constructors, and each constructor as a projection. For example, trees are defined as follows:

$$\begin{aligned} \text{tree} & : \text{ type.} \\ \text{c.tree} & = \Sigma c_1 : \text{ tree. tree} \rightarrow \text{ tree} \rightarrow \text{ tree.} \\ \text{leaf} & = \lambda m : \text{ c.tree. } \pi_1 m. \\ \text{node} & = \lambda m : \text{ c.tree. } \pi_2 m. \end{aligned}$$

To create a tree, we require an instance of type `c.tree` which “is” the datatype. Another example are lists whose types encode size information as well.

$$\begin{aligned} \text{list} & : \text{ nat} \rightarrow \text{ type.} \\ \text{c.list} & = \Sigma c_1 : \text{ list } 0. \\ & \quad \Pi n : \text{ nat. nat} \rightarrow \text{ list } n \rightarrow \text{ list (succ } n\text{)}. \\ \text{nil} & = \lambda m : \text{ c.list. } \pi_1 m. \\ \text{cons} & = \lambda m : \text{ c.list. } \pi_2 m. \end{aligned}$$

Viewing constructors as projections changes the traditional character of datatype declarations. Datatypes may be declared in local contexts, and appropriate

binding constructs of a programming language may introduce, select, and discharge datatypes. With respect to higher-order encodings, recursion under  $\lambda$ -binders is now possible. All those features are available in the implemented experimental version of Delphin. In its current state, Delphin is a domain specific language for encodings in  $\text{LF}^\Sigma$ . Support for constraint domains, such as rationals, integers, and strings are planned. For more information on Delphin, and an extended version of this paper, visit [www.cs.yale.edu/~carsten/delphin](http://www.cs.yale.edu/~carsten/delphin).

The paper is organized as follows. In Section 2 we present the logical framework  $\text{LF}^\Sigma$  illustrate the representation technology that we plan to use, and discuss its meta theory. Next, we leave the representational layer and present the functional programming language Delphin in Section 3. We start with a description of the language features, followed by a presentation of its type system, operational semantics, and some meta-theoretical considerations. Delphin’s implementation is described in Section 4. We mention related work in Section 5 before we conclude with Section 6 where we assess results and outline future work.

## 2 The Logical Framework $\text{LF}^\Sigma$

In proof carrying authentication, Church’s higher-order logic is used to formulate access and safety properties, and proofs play the role of certificates, that a user has to present to a server to gain access to a desired resource. In turn, the server checks the certificate for validity and grants access if it is correct. What is important in this setting is that certificates (derivations in higher-order logic) are encoded in LF and a user must have the ability to created, compress, and exchange certificates, using software that has to be written in some kind of a language. The choices are limited. Traditional functional programming languages cannot work with LF encodings because first, they are not dependently typed, and second, the negative occurrence of types in constructor types requires the user to fall back into a deBruijn style implementation. None of the dependently typed programming languages surveyed in Section 5 can do it either. Hence, programming with PCA certificates requires the user either to retreat to traditional programming language converting the LF format into an internal format and then casting the result back into LF, or to use a programming language that is especially developed for these kind of applications, such as Delphin. The former is quite challenging, because certain properties that come for free with LF, such as substitutivity has to be encoded for every datatype.

As an illustrative example consider the natural deduction rules for propositional logic

$$\begin{array}{ll} \text{Formulas} & A ::= P \mid A_1 \wedge A_2 \mid A_1 \supset A_2 \mid \neg A \\ \text{Assumptions} & \Delta ::= \cdot \mid \Delta, A \mid \Delta, [p] \end{array}$$

depicted in Figure 1 that can be found in the core logics of PCA or PCC. Here  $[p]$  marks a new propositional constant that may appear free in assumptions  $A$  to the right of it in  $\Delta$ .

$$\begin{array}{c}
\frac{}{\Delta, A \vdash A} \text{ax} \quad \frac{\Delta \vdash A_1 \quad \Delta \vdash A_2}{\Delta \vdash A_1 \wedge A_2} \text{andI} \\
\frac{\Delta \vdash A_1 \wedge A_2}{\Delta \vdash A_1} \text{andE}_1 \quad \frac{\Delta \vdash A_1 \wedge A_2}{\Delta \vdash A_2} \text{andE}_2 \\
\frac{\Delta, A_1 \vdash A_2}{\Delta \vdash A_1 \supset A_2} \text{impl} \quad \frac{\Delta \vdash A_1 \supset A_2 \quad \Delta \vdash A_1}{\Delta \vdash A_2} \text{impE} \\
\frac{\Delta, [p], A \vdash p}{\Delta \vdash \neg A} \text{notI}^p \quad \frac{\Delta \vdash \neg A \quad \Delta \vdash A}{\Delta \vdash B} \text{notE}
\end{array}$$

Figure 1: Propositional logic

We begin now with the formal definition of the logical framework  $\text{LF}^\Sigma$  that extends the Edinburgh Logical framework LF [6] by dependent products.

$$\begin{array}{lcl}
\text{Kinds:} & K & ::= \text{type} \mid \Pi x : A. K \\
\text{Types:} & A & ::= a \mid A M \mid \Pi x : A_1. A_2 \\
& & \quad \mid \Sigma x : A_1. A_2 \mid \lambda x : A_1. A_2 \\
\text{Objects:} & M & ::= x \mid M_1 M_2 \mid \lambda x : A. M \\
& & \quad \mid \langle M_1; M_2 \rangle \mid \pi_1 M \mid \pi_2 M \\
\text{Contexts:} & \Gamma & ::= \cdot \mid \Gamma, x : A \\
\text{Signature:} & \Sigma & ::= \cdot \mid \Sigma, a : K \mid \Sigma, a : K = A \\
& & \quad \mid \Sigma, c : A = M
\end{array}$$

Contexts assign type information to object level variables. There are no type level variables. Substitutions are as usual capture avoiding which is achieved by tacitly renaming variables during substitution application. As a variation to LF, definitional equality [6, 4, 14] is defined as the congruence closure of

$$\begin{array}{ll}
(\lambda x : A. M_1) M_2 \equiv M_1[M_2/x] & (\beta) \\
\lambda x : A. M x \equiv M & (\eta) \\
\pi_1 \langle M_1; M_2 \rangle \equiv M_1 & (\pi_1) \\
\pi_2 \langle M_1; M_2 \rangle \equiv M_2. & (\pi_2)
\end{array}$$

For this work, the critical feature of  $\text{LF}^\Sigma$  is that every object has a canonical ( $\beta$ -normal,  $\eta$ -long,  $\pi_i$ -normal) form, for which we write  $\Gamma \vdash_\Sigma M \uparrow A$ . Different from [6], constant declarations are disallowed. Instead, datatypes are declared in the context, and projections play the role of constructors. Besides the common type family declarations, signatures in  $\text{LF}^\Sigma$  can only contain type level definitions  $a : K = A$  and object level definitions  $c : A = M$ , for which we write as  $a = A$  and  $c = M$ , respectively, when the kind/type information is inferable from the context. Often, we write  $A_1 \rightarrow A_2$  for  $\Pi x : A_1. A_2$  and  $A_1 \times A_2$  for

$$\begin{array}{c}
\frac{\Gamma(x) = A}{\Gamma \vdash_{\Sigma} x \downarrow A} \text{obj\_var} \quad \frac{\Gamma \vdash_{\Sigma} M \downarrow a}{\Gamma \vdash_{\Sigma} M \uparrow A} \text{obj\_atmcan} \\
\\
\frac{\Gamma, x : A_1 \vdash_{\Sigma} M \uparrow A_2}{\Gamma \vdash_{\Sigma} \lambda x : A_1. M \uparrow \Pi x : A_1. A_2} \text{obj\_lam} \\
\\
\frac{\Gamma \vdash_{\Sigma} M_1 \downarrow \Pi x : A_2. A_1 \quad \Gamma \vdash_{\Sigma} M_2 \uparrow A_2}{\Gamma \vdash_{\Sigma} M_1 M_2 \downarrow A_1[M_2/x]} \text{obj\_app} \\
\\
\frac{\Gamma \vdash_{\Sigma} M_1 \uparrow A_1 \quad \Gamma \vdash_{\Sigma} M_2 \uparrow A_2[M_1/x]}{\Gamma \vdash_{\Sigma} \langle M_1; M_2 \rangle \uparrow \Sigma x : A_1. A_2} \text{obj\_pair} \\
\\
\frac{\Gamma \vdash_{\Sigma} M \downarrow \Sigma x : A_1. A_2}{\Gamma \vdash_{\Sigma} \pi_1 M \downarrow A_1} \text{obj\_fst} \quad \frac{\Gamma \vdash_{\Sigma} M : \Sigma x : A_1. A_2}{\Gamma \vdash_{\Sigma} \pi_2 M \downarrow A_2[\pi_1 M/x]} \text{obj\_snd}
\end{array}$$

Figure 2: LF canonical forms.

$\Sigma x : A_1. A_2$  if  $x$  does not occur free in  $A_2$ . We write  $\pi_i^k$  for the  $k$  fold application of  $\pi_i$ . Furthermore an inference system for canonical forms in combination with typing is given in Figure 2. From [14] we repeat the main results about  $\text{LF}^{\Sigma}$ .

**Theorem 2.1 ( $\text{LF}^{\Sigma}$ )**

1. *Type checking of  $\text{LF}^{\Sigma}$  is decidable.*
2. *If  $\Gamma \vdash_{\Sigma} M : A$  then there exists an  $M' \equiv M$  s.t.  $\Gamma \vdash_{\Sigma} M' \uparrow A$ . Types have canonical forms as well.*
3.  *$\text{LF}^{\Sigma}$  is a conservative extension of  $\text{LF}$ .*

In  $\text{LF}^{\Sigma}$  judgments are represented as types and derivations as objects. For example, the encoding of a derivation of  $\mathcal{D}$  of judgment  $\Delta \vdash A$  is captured by the definition of the type family  $\text{nd}$ :

$$\ulcorner \mathcal{D} :: \Delta \vdash A \urcorner = \Gamma, \ulcorner \Delta \urcorner \vdash_{\Sigma} \ulcorner \mathcal{D} \urcorner \uparrow \text{nd} \ulcorner A \urcorner \quad (1)$$

where we write  $\ulcorner \cdot \urcorner$  for the representation function, explained below. In a slight deviation from the usual way of representing deductive systems in  $\text{LF}$  we use a prefix of the context ( $\Gamma$ ) to declare connectives and inference rules, and  $\Sigma$  to introduce the necessary type declarations, type definitions, and constant definitions exploiting the newly added expressiveness of  $\Sigma$ -types. The signature

contains the following.

$$\begin{aligned}
\mathbf{o} & : \text{ type.} \\
\mathbf{c\_o} & = (\mathbf{o} \rightarrow \mathbf{o} \rightarrow \mathbf{o}) \times (\mathbf{o} \rightarrow \mathbf{o} \rightarrow \mathbf{o}) \times (\mathbf{o} \rightarrow \mathbf{o}). \\
\mathbf{and} & = \lambda m : \mathbf{c\_o}. \pi_1 m. \\
\mathbf{imp} & = \lambda m : \mathbf{c\_o}. \pi_1(\pi_2 m). \\
\mathbf{not} & = \lambda m : \mathbf{c\_o}. \pi_2(\pi_2 m).
\end{aligned}$$

$\mathbf{c\_o}$  accounts for the three connectives conjunction, implication, and negation. Let  $m : \mathbf{c\_o}$ .

$$\begin{aligned}
\ulcorner A_1 \wedge A_2 \urcorner & = \mathbf{and} m \ulcorner A_1 \urcorner \ulcorner A_2 \urcorner \\
\ulcorner A_1 \supset A_2 \urcorner & = \mathbf{imp} m \ulcorner A_1 \urcorner \ulcorner A_2 \urcorner \\
\ulcorner \neg A \urcorner & = \mathbf{not} m \ulcorner A \urcorner
\end{aligned}$$

A dependent datatype for natural deduction derivations can be formulated in a similar way. The following is also contained in the signature.

$$\begin{aligned}
\mathbf{nd} & : \mathbf{o} \rightarrow \text{ type.} \\
\mathbf{c\_nd} & = \lambda m : \mathbf{c\_o}. \\
& \quad \Pi a : \mathbf{o}. \Pi b : \mathbf{o}. \mathbf{nd} a \rightarrow \mathbf{nd} b \rightarrow \mathbf{nd} (\mathbf{and} m a b) \\
& \quad \times \Pi a : \mathbf{o}. \Pi b : \mathbf{o}. \mathbf{nd} (\mathbf{and} m a b) \rightarrow \mathbf{nd} a \\
& \quad \times \Pi a : \mathbf{o}. \Pi b : \mathbf{o}. \mathbf{nd} (\mathbf{and} m a b) \rightarrow \mathbf{nd} b \\
& \quad \times \Pi a : \mathbf{o}. \Pi b : \mathbf{o}. (\mathbf{nd} a \rightarrow \mathbf{nd} b) \rightarrow \mathbf{nd} (\mathbf{imp} m a b) \\
& \quad \times \Pi a : \mathbf{o}. \Pi b : \mathbf{o}. \mathbf{nd} (\mathbf{imp} m a b) \rightarrow \mathbf{nd} a \rightarrow \mathbf{nd} b \\
& \quad \times \Pi a : \mathbf{o}. (\Pi p : \mathbf{o}. \mathbf{nd} a \rightarrow \mathbf{nd} p) \rightarrow \mathbf{nd} (\mathbf{not} m a) \\
& \quad \times \Pi a : \mathbf{o}. \Pi b : \mathbf{o}. \mathbf{nd} a \rightarrow \mathbf{nd} (\mathbf{not} m a) \rightarrow \mathbf{nd} b \\
\mathbf{andI} & = \lambda m_1 : \mathbf{c\_o}. \lambda m_2 : \mathbf{c\_nd} m_1. \pi_1 m_2. \\
\mathbf{andE}_1 & = \lambda m_1 : \mathbf{c\_o}. \lambda m_2 : \mathbf{c\_nd} m_1. \pi_1(\pi_2 m_2). \\
\mathbf{andE}_2 & = \lambda m_1 : \mathbf{c\_o}. \lambda m_2 : \mathbf{c\_nd} m_1. \pi_1(\pi_2^2 m_2). \\
\mathbf{impl} & = \lambda m_1 : \mathbf{c\_o}. \lambda m_2 : \mathbf{c\_nd} m_1. \pi_1(\pi_2^3 m_2). \\
\mathbf{impE} & = \lambda m_1 : \mathbf{c\_o}. \lambda m_2 : \mathbf{c\_nd} m_1. \pi_1(\pi_2^4 m_2). \\
\mathbf{notI} & = \lambda m_1 : \mathbf{c\_o}. \lambda m_2 : \mathbf{c\_nd} m_1. \pi_1(\pi_2^5 m_2). \\
\mathbf{notE} & = \lambda m_1 : \mathbf{c\_o}. \lambda m_2 : \mathbf{c\_nd} m_1. \pi_2^6 m_2.
\end{aligned}$$

$\mathbf{c\_nd}$ , parametrized by the datatype for formulas  $m : \mathbf{c\_o}$ , encodes all inference rules depicted in Figure 1. We only show the case of  $\mathbf{notI}$ , all the others are defined analogously. Let  $m_1 : \mathbf{c\_o}$  and  $m_2 : \mathbf{c\_nd} m_1$ . The encoding of a derivation ending in  $\mathbf{notI}$  with premiss  $\mathcal{D} :: \Delta, A \vdash p$  is given as

$$\mathbf{notI} m_1 m_2 \ulcorner A \urcorner (\lambda p : \mathbf{o}. \lambda u : \mathbf{nd} (\mathbf{not} m_1 \ulcorner A \urcorner). \ulcorner \mathcal{D} \urcorner)$$

Thus  $\Gamma$  in (1) has the form  $m_1 : \mathbf{c\_o}, m_2 : \mathbf{c\_nd} m_1$ . As for the encoding of  $\Delta$ , each assumption is to be named, encoded, and added to  $\Gamma$ .

$$\ulcorner \cdot \urcorner = \cdot \tag{2}$$

$$\ulcorner \Delta, A \urcorner = \ulcorner \Delta \urcorner, u : \mathbf{nd} \ulcorner A \urcorner \tag{3}$$

$$\ulcorner \Delta, [p] \urcorner = \ulcorner \Delta \urcorner, p : \mathbf{o} \tag{4}$$

The idea of representing constructors defining a datatype in terms of products is certainly not new. What is new however, is to consider datatypes and the defining constructors consistently as objects of type level definitions, that can be statically or dynamically introduced, discharged, accessed, and compared by a programming language. Delphin, the language described in Section 3, implements this interpretation of datatypes, which implicitly removes the restriction to negative occurrences as described below.

$\Sigma$ 's are often used dependently. For example, it is easy to extend the natural deduction calculus by the two Boolean constants  $\top$  and  $\perp$ , and the corresponding rules

$$\frac{}{\Delta \vdash \top} \top I \quad \frac{\Delta \vdash \perp \quad \Delta, A \vdash B}{\Delta \vdash B} \perp E$$

The corresponding signature is

$$\begin{aligned} \text{c\_ext} &= \Sigma t : \text{o}. \Sigma f : \text{o}. \text{nd } t \\ &\quad \times \Pi a : \text{o}. \Pi b : \text{o}. \text{nd } f \rightarrow (\text{nd } a \rightarrow \text{nd } b) \rightarrow \text{nd } b \\ \text{true} &= \lambda m : \text{c\_ext}. \pi_1 m. \\ \text{false} &= \lambda m : \text{c\_ext}. \pi_1(\pi_2 m). \\ \text{trueI} &= \lambda m : \text{c\_ext}. \pi_1(\pi_2^2 m). \\ \text{falseE} &= \lambda m : \text{c\_ext}. \pi_2^3 m. \end{aligned}$$

With  $\text{LF}^\Sigma$ , datatypes are encoded via  $\Sigma$ -types, declared in the context, and often fragmented in several individual pieces. The technique presented here is so general, that each piece may contain constructors of different types.

## 2.1 Regular Worlds

When working with complex encodings, such as the encodings of logic derivations, type systems, or operational semantics, we need a guarantee that the encoding is *adequate* which means that there exists a one-to-one correspondence between mathematical concepts and their representations in the type theory. Objects of type  $\text{nd } A$  and derivations of  $A$  in the natural deduction calculus must correspond bijectively. Adequacy is proved by induction over the canonical forms of type  $\text{nd } A$ . Due to Theorem 2.1 (2) every well-typed object in the  $\text{LF}^\Sigma$  calculus reduces to a canonical form. Without adequacy, one can never be sure that the result of manipulating an object still corresponds to a valid object.

It should not surprise that the concepts we introduce and the techniques we develop in this section also apply to the following sections on functional programming with Delphin. Induction and function definitions by cases are tightly intertwined, and therefore regular worlds lay the ground work for both.

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 judgment for natural deductions 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  $A$  in  $\Delta$ . 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 natural deduction derivations provides further illustration of the open nature of LF objects. Any Delphin program that recurses on subderivations of  $\text{nd } A$  ending in  $\text{impl}$  or  $\text{notE}$ , for example, must traverse several  $\lambda$ -binders, which introduce new parameters into the context. Programs of this kind occur when converting derivation of one logic to another or when programming theorem provers as we describe in Section 3.

**Example 2.2 (Encoding)** Consider the context

$$\Gamma = c_1 : \text{c.o}, c_2 : \text{c.nd } c_1$$

which declares constructors a set of constructors for formulas  $c_1$ , and another for the corresponding inference rules  $c_2$ . The derivation

$$\frac{\frac{\frac{\frac{}{A, \neg A \vdash A} \text{ax}}{A, \neg A \vdash \neg A} \text{ax}}{A, \neg A \vdash p} \text{negE}}{A \vdash \neg \neg A} \text{negI}^p}{\vdash A \supset \neg \neg A} \text{impl}}$$

is encoded in  $\text{LF}^\Sigma$  as

$$\begin{aligned} \Gamma \vdash_\Sigma & \text{impl } c_2 \ulcorner A \urcorner (\text{not } c_1 (\text{not } c_1 \ulcorner A \urcorner)) (\lambda u : \text{nd } \ulcorner A \urcorner. \\ & \text{negI } c_2 \ulcorner A \urcorner (\lambda p : \text{o}. \lambda v : \text{nd } (\text{not } c_1 \ulcorner A \urcorner). \\ & \text{negE } c_2 \ulcorner A \urcorner p u v)) \\ & : \text{nd } (\text{imp } c_1 \ulcorner A \urcorner (\text{not } c_1 (\text{not } c_1 \ulcorner A \urcorner))) \end{aligned}$$

and illustrates how each constructors refers to the declaration of the signature that contains it. We write  $\ulcorner \cdot \urcorner$  for the representation function that maps formulas and natural deduction derivations into LF, respectively. Recall that all object level constants given **sans serif** font, are in fact projections.

Consider as another example a function that counts the number of occurrences of the  $\text{ax}$  rule in the LF representation of a natural deduction derivation. Furthermore consider it to traverse the structure of the LF object given above while currently deconstructing the subderivation ending in  $\text{negE}$ . Without question, this LF object is open, because it is valid only in context  $c_1 : \text{c.o}, c_2 : \text{c.nd } c_1, u : \text{nd } \ulcorner A \urcorner, p : \text{o}, v : \text{nd } (\text{not } c_1 \ulcorner A \urcorner)$ .  $u$ ,  $p$ , and  $v$  are parameters that cannot be subjected to further case analysis without rendering the function unsound.

This observation bares already the crucial insight into our solution to this problem. Viewed from the meta level, the context  $\Gamma$  declares parametric and non parametric variables.  $c_1$  and  $c_2$  are parametric, because it is unsound to consider

cases over constants. Thus, a mechanism to instantiate  $\lambda$  bound variables by parameters is in order. This mechanism permits dynamic declarations of partial datatypes and their discharge thereafter. Consequently, the datatypes declared by  $c_1$  and  $c_2$  are only partial, open-ended, and may be extended dynamically by new declarations of the form  $x \sqsupset A$  during runtime. For non-parametric declarations, we continue to write  $x : A$ .

$$\text{Contexts } \Psi ::= \cdot \mid \Psi, x : A \mid \Psi, x \sqsupset A$$

Below we refer to contexts  $\Psi$  that contain exclusively parametric assumptions  $x \sqsupset A$  as  $\Phi$ . How to replace the parametric looking  $u$  and  $v$  in our running example is discussed in Section 3. The distinction of parametric from non-parametric declarations on the meta level is of no interest to  $\text{LF}^\Sigma$  prompting the definition of an embedding function from meta-level to  $\text{LF}^\Sigma$ -level  $[\Psi] = \Gamma$ , which simply replaces  $\sqsupset$  by “:”.

**Example 2.3 (Dynamic context extensions)** Recall Example 2.2. Let

$$\begin{aligned} \Phi = & c_1 \sqsupset \text{c.o}, c_2 \sqsupset \text{c.nd } c_1, \\ & c_3 \sqsupset \text{nd } \ulcorner A \urcorner, c_4 \sqsupset \text{o}, c_5 \sqsupset \text{nd } (\text{not } c_1 \ulcorner A \urcorner). \end{aligned}$$

During the traversal of that derivation, the LF term starting with `negE` is encountered as

$$[\Phi] \vdash_\Sigma \text{negE } c_2 \ulcorner A \urcorner c_4 c_3 c_5 : \text{nd } c_4.$$

In the interest of adequate representations, meta-level contexts must adhere to certain specifications. If contexts were to allow arbitrary declarations, it is impossible to reason about or program with them. Specifications of this kind are called *worlds* [16] and play the role of types for contexts  $\Phi$ . In traditional functional programming languages, such as ML or Haskell, datatypes are static and cannot be extended during runtime which leads to a trivial world system, in Delphin, however, dynamic extensions are allowed as long as they are predictable. In this case, the parametric constituents of a context may be described by a grammar, and although it is plausible to base the world system on context free grammars, we have found that regular expressions suffice.

$$\text{Worlds } W ::= A \mid 1 \mid 0 \mid W_1 + W_2 \mid W_1 \cdot W_2 \mid W^*$$

Following standard regular expression terminology, the type assignment of a world to a context is written as  $\vdash \Phi \in W$  and defined in Figure 3.  $A$  describes the set of constructors, usually defined in the  $\text{LF}^\Sigma$  signature  $\Sigma$ , such as `c.o`, `c.nd`, or `c.ext`. The language generated by 1 contains only the empty context, by 0 is empty, + stands for alternative,  $\cdot$  for concatenation, and \* for repetition.

**Example 2.4 (Valid worlds)** The context  $\Phi$  defined in Example 2.3 is valid in world

$$W = \text{c.o} \cdot \text{c.nd} \cdot (\text{nd} + \text{o})^*.$$

$$\begin{array}{c}
\frac{}{\Phi_0 \vdash (x \sqcap A) \in A} \text{ axiom} \\
\frac{[\Phi_0] \vdash M \uparrow A_1 \quad \Phi_0 \vdash A \in [M/x]A_2}{\Phi_0 \vdash (x \sqcap A) \in \lambda x : A_1. A_2} \text{ inst} \\
\frac{}{\Phi_0 \vdash \cdot \in 1} \text{ one} \quad \text{no rule for } 0 \\
\frac{}{\Phi_0 \vdash \cdot \in W^*} \text{ empty} \\
\frac{\Phi_0 \vdash \Phi_1 \in W^* \quad \Phi_0, \Phi_1 \vdash \Phi_2 \in W}{\Phi_0 \vdash \Phi_1, \Phi_2 \in W^*} \text{ unfold} \\
\frac{\Phi_0 \vdash \Phi \in W_1}{\Phi_0 \vdash \Phi \in W_1 + W_2} \text{ left} \quad \frac{\Phi_0 \vdash \Phi \in W_2}{\Phi_0 \vdash \Phi \in W_1 + W_2} \text{ right} \\
\frac{\Phi_0 \vdash \Phi_1 \in W_1 \quad \Phi_0, \Phi_1 \vdash \Phi_2 \in W_2}{\Phi_0 \vdash \Phi_1, \Phi_2 \in W_1 \cdot W_2} \text{ concat}
\end{array}$$

Figure 3: Valid (well-worlded) context.

## 2.2 Adequacy

Adequacy refers to the existence of a bijection between informal deductions and objects in the type theory, It is a property that must be established individually for every type 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 deductions in one and on the canonical forms in the other direction.

We show that the encoding of the natural deduction calculus is adequate with respect to world  $W$  from Example 2.4. Informally, the adequacy theorem states that the representation function  $\ulcorner \cdot \urcorner$  defined in Equation (1) is a bijection. Let  $\Phi_0 = c_1 \sqcap c_{\circ}, c_2 \sqcap c_{\text{nd}} c_1$  and  $W$  defined as in Example 2.4. Assumptions  $\Delta$  can be identified with  $\Phi$  such that  $\Phi_0 \vdash \Phi \in (\text{nd} + \circ)^*$ , formulas with canonical forms of type  $\circ$ , and derivations of  $A$  with canonical forms of type  $\text{nd} \ulcorner A \urcorner$ .

**Theorem 2.5 (Adequacy)** 1. For all context  $\Phi$  s.t.  $\Phi_0 \vdash \Phi \in (\text{nd} + \circ)^*$ , there exists a  $\Delta$ , s.t.  $\ulcorner \Delta \urcorner = \Phi$ , and vice versa.

2. For all objects  $[\Phi_0, \Phi] \vdash M \uparrow \circ$  there exists assumptions  $\Delta$  and a formula  $A$  with free propositional variables contained in  $\Delta$ , s.t.  $\ulcorner \Delta \urcorner = \Phi$  and  $\ulcorner A \urcorner = M$ , and vice versa.

3. For all objects  $[\Phi_0, \Phi] \vdash M \uparrow \text{nd} \ulcorner A \urcorner$  there exists assumptions  $\Delta$ , and a natural deduction derivation  $\mathcal{D} :: \Delta \vdash A$ , s.t.  $\ulcorner \Delta \urcorner = \Phi$  and  $\ulcorner \mathcal{D} \urcorner = M$ , and

*vice versa.*

**Proof:** The first, by structural induction on the derivation of  $\Phi_0 \vdash \Phi \in W$  in one direction and  $\Delta$  in the other, the other two by structural induction on the canonicity derivations in one direction,  $A$  and  $\mathcal{D} :: \text{nd } \ulcorner A \urcorner$  in the other.

1. “ $\Rightarrow$ ”:

- (a)  $\Phi_0 \vdash \cdot \in (\text{nd} + \text{o})^*$  by empty  
Choose  $\Delta = \cdot$ , because  $\ulcorner \cdot \urcorner = \cdot$  by (2)
- (b)  $\Phi_0 \vdash \Phi, u \sqcap \text{nd } M \in (\text{nd} + \text{o})^*$  by unfold  
 $\Phi_0 \vdash \Phi \in (\text{nd} + \text{o})^*$   
 $\Phi_0, \Phi \vdash u \sqcap \text{nd } M \in \text{nd} + \text{o}$   
there exists a  $\Delta$ , s.t.  $\ulcorner \Delta \urcorner = \Phi$  by i.h. (1)  
 $\Phi_0, \Phi \vdash u \sqcap \text{nd } M \in \text{nd}$  by inversion  
 $\Phi_0, \Phi \vdash M \uparrow \text{o}$  by inversion  
there exists an  $A$ , s.t.  $\ulcorner A \urcorner = M$  by i.h. (2)  
 $\ulcorner \Delta, A \urcorner = \ulcorner \Delta \urcorner, u \sqcap \text{nd } \ulcorner A \urcorner = \Phi, u \sqcap \text{nd } M$
- (c)  $\Phi_0 \vdash \Phi, p \sqcap \text{o} \in (\text{nd} + \text{o})^*$  by unfold  
 $\Phi_0 \vdash \Phi \in (\text{nd} + \text{o})^*$   
 $\Phi_0, \Phi \vdash p \sqcap \text{o} \in \text{nd} + \text{o}$   
there exists a  $\Delta$ , s.t.  $\ulcorner \Delta \urcorner = \Phi$  by i.h. (1)  
 $\Phi_0, \Phi \vdash p \sqcap \text{o} \in \text{o}$  by inversion  
 $\ulcorner \Delta, [p] \urcorner = \ulcorner \Delta \urcorner, p \sqcap \text{o} = \Phi, p \sqcap \text{o}$

“ $\Leftarrow$ ”:

- (a)  $\Delta = \cdot$ ,  $\ulcorner \cdot \urcorner = \cdot$  by (2)  
 $\Phi_0 \vdash \cdot \in (\text{nd} + \text{o})^*$  by empty
- (b)  $\Delta = \Delta', A$ .  
 $\Phi_0 \vdash \ulcorner \Delta' \urcorner \in (\text{nd} + \text{o})^*$  by i.h. (1)  
 $\Phi_0 \vdash \ulcorner A \urcorner \uparrow \text{o}$  by i.h. (2)  
 $\Phi_0, \ulcorner \Delta' \urcorner \vdash u \sqcap \text{nd } \ulcorner A \urcorner \in \text{nd } \ulcorner A \urcorner$  by axiom  
 $\Phi_0, \ulcorner \Delta' \urcorner \vdash u \sqcap \text{nd } \ulcorner A \urcorner \in \text{nd}$  by inst  
 $\Phi_0, \ulcorner \Delta' \urcorner \vdash u \sqcap \text{nd } \ulcorner A \urcorner \in \text{nd} + \text{o}$  by left  
 $\Phi_0 \vdash, \ulcorner \Delta', A \urcorner \in (\text{nd} + \text{o})^*$  by unfold
- (c)  $\Delta = \Delta', [p]$ .  
 $\Phi_0 \vdash \ulcorner \Delta' \urcorner \in (\text{nd} + \text{o})^*$  by i.h. (1)  
 $\Phi_0, \ulcorner \Delta' \urcorner \vdash p \sqcap \text{o} \in \text{o}$  by axiom  
 $\Phi_0, \ulcorner \Delta' \urcorner \vdash p \sqcap \text{o} \in \text{nd} + \text{o}$  by right  
 $\Phi_0 \vdash, \ulcorner \Delta', [p] \urcorner \in (\text{nd} + \text{o})^*$  by unfold

2. “ $\Rightarrow$ ”:

- (a)  $[\Phi_0, \Phi] \vdash_{\Sigma} p \uparrow \circ$  by assumption  
 $[\Phi_0, \Phi] \vdash_{\Sigma} p \downarrow \circ$  by inversion  
 $p : \circ \in [\Phi_0, \Phi]$  by inversion  
 $p : \circ \in [\Phi]$  by inversion  
 $p \Box \circ \in \Phi$  by inversion  
there exists a  $\Delta$ , s.t.  $\ulcorner \Delta \urcorner = \Phi$  by i.h. (1)  
 $[p] \in \Delta$ , and thus  $\ulcorner p \urcorner = p$  by (4)
- (b)  $\Phi_0 = c_1 \Box_{c.o.} c_2 \Box_{c.nd} c_1$  by assumption  
 $[\Phi_0, \Phi] \vdash_{\Sigma} (\text{not } c_1) M \uparrow \circ$  by assumption  
 $[\Phi_0, \Phi] \vdash_{\Sigma} (\text{not } c_1) M \uparrow \circ$  by inversion  
 $[\Phi_0, \Phi] \vdash_{\Sigma} (\text{not } c_1) \downarrow \circ \rightarrow \circ$  by inversion  
 $[\Phi_0, \Phi] \vdash_{\Sigma} M \uparrow \circ$  by inversion  
there exists a  $\Delta$ , s.t.  $\ulcorner \Delta \urcorner = \Phi$  by i.h. (2)  
there exists a  $A$ , s.t.  $\ulcorner A \urcorner = M$  by i.h. (2)  
 $(\text{not } c_1) M = (\text{not } c_1) \ulcorner A \urcorner = \ulcorner \neg A \urcorner$
- (c)  $\Phi_0 = c_1 \Box_{c.o.} c_2 \Box_{c.nd} c_1$  by assumption  
 $[\Phi_0, \Phi] \vdash_{\Sigma} (\text{and } c_1) M_1 M_2 \uparrow \circ$  by assumption  
 $[\Phi_0, \Phi] \vdash_{\Sigma} (\text{and } c_1) M_1 M_2 \uparrow \circ$  by inversion  
 $[\Phi_0, \Phi] \vdash_{\Sigma} (\text{and } c_1) \downarrow \circ \rightarrow \circ \rightarrow \circ$  by inversion  
 $[\Phi_0, \Phi] \vdash_{\Sigma} M_1 \uparrow \circ$  by inversion  
 $[\Phi_0, \Phi] \vdash_{\Sigma} M_2 \uparrow \circ$  by inversion  
there exists a  $\Delta$ , s.t.  $\ulcorner \Delta \urcorner = \Phi$  by i.h. (2)  
there exists a  $A_1$ , s.t.  $\ulcorner A_1 \urcorner = M_1$  by i.h. (2)  
there exists a  $A_2$ , s.t.  $\ulcorner A_2 \urcorner = M_2$  by i.h. (2)  
 $(\text{and } c_1) M_1 M_2 = (\text{and } c_1) \ulcorner A_1 \urcorner \ulcorner A_2 \urcorner$   
 $= \ulcorner A_1 \wedge A_2 \urcorner$
- (d)  $\Phi_0 = c_1 \Box_{c.o.} c_2 \Box_{c.nd} c_1$  by assumption  
 $[\Phi_0, \Phi] \vdash_{\Sigma} (\text{imp } c_1) M_1 M_2 \uparrow \circ$  by assumption  
 $[\Phi_0, \Phi] \vdash_{\Sigma} (\text{imp } c_1) M_1 M_2 \uparrow \circ$  by inversion  
 $[\Phi_0, \Phi] \vdash_{\Sigma} (\text{imp } c_1) \downarrow \circ \rightarrow \circ \rightarrow \circ$  by inversion  
 $[\Phi_0, \Phi] \vdash_{\Sigma} M_1 \uparrow \circ$  by inversion  
 $[\Phi_0, \Phi] \vdash_{\Sigma} M_2 \uparrow \circ$  by inversion  
there exists a  $\Delta$ , s.t.  $\ulcorner \Delta \urcorner = \Phi$  by i.h. (2)  
there exists a  $A_1$ , s.t.  $\ulcorner A_1 \urcorner = M_1$  by i.h. (2)  
there exists a  $A_2$ , s.t.  $\ulcorner A_2 \urcorner = M_2$  by i.h. (2)  
 $(\text{imp } c_1) M_1 M_2 = (\text{imp } c_1) \ulcorner A_1 \urcorner \ulcorner A_2 \urcorner$   
 $= \ulcorner A_1 \supset A_2 \urcorner$

“ $\Leftarrow$ ”:

- (a)  $A = p$  by assumption  
 $[p] \in \Delta$  by definition  
 $[\Phi_0, \ulcorner \Delta \urcorner] \vdash_{\Sigma} \ulcorner p \urcorner \uparrow \circ$  by obj\_var
- (b)  $A = \neg A'$  by assumption  
 $\Phi_0 = c_1 \Box_{c.o.} c_2 \Box_{c.nd} c_1$  by assumption

$$\begin{array}{ll}
[\Phi_0, \ulcorner \Delta \urcorner] \vdash_{\Sigma} \ulcorner A' \urcorner \uparrow \circ & \text{by i.h. (2)} \\
[\Phi_0, \ulcorner \Delta \urcorner] \vdash_{\Sigma} \text{not } c_1 \downarrow \circ \rightarrow \circ & \text{by obj\_var} \\
[\Phi_0, \ulcorner \Delta \urcorner] \vdash_{\Sigma} \text{not } c_1 \ulcorner A' \urcorner \downarrow \circ & \text{by obj\_app} \\
[\Phi_0, \ulcorner \Delta \urcorner] \vdash_{\Sigma} \text{not } c_1 \ulcorner A' \urcorner \uparrow \circ & \text{by obj\_atmcan} \\
[\Phi_0, \ulcorner \Delta \urcorner] \vdash_{\Sigma} \ulcorner \neg A' \urcorner \uparrow \circ & \text{by obj\_atmcan}
\end{array}$$

(c)  $A = A_1 \wedge A_2$ , analogously.

(d)  $A = A_1 \supset A_2$ , analogously.

3. Analogously to 2.

□

### 3 Delphin

Delphin is a special purpose functional programming language that is designed to support programmers to work directly with  $\text{LF}^{\Sigma}$  encodings. When programming with proofs, theorems, and derivations in PCC or PCA, a language like this is the natural candidate to program in. In Delphin, programming is lifted to a new level of abstraction, where datatype constructors need not be given a priori, but may be introduced during runtime. In the form presented here, Delphin may be seen as a domain specific language for  $\text{LF}^{\Sigma}$  encodings that are by nature higher-order and dependently typed. Examples include proofs, but also typing derivation and computation traces. Nevertheless this work sheds also some light on possible further design directions for traditional languages such as ML and Haskell. We currently consider extending Delphin to a general purpose programming language as well.

#### 3.1 Language Features

Delphin’s design is based on a variation of the type theory  $\mathcal{T}_{\omega}^+$  [16] rendering it suitable for practical programming. It features dependent functions and dependent products, which are described in Section 3.1.1, and function definition by cases as described in Section 3.1.2. In addition, for higher-order encodings of data, Delphin provides a mechanism to add, retract, and lookup resources in the implicit and not user accessible  $\text{LF}^{\Sigma}$  context  $\Phi$  that stores fragments of the data constructors. Delphin is inherently non-deterministic. A “choice” operator selects parametric declarations from  $\Phi$  (no other declarations are present during runtime), an “alternative” operator allows programs to be executed non-deterministically (see Section 3.1.3), and a “new” operator extends the set of constructors by adding declarations to  $\Phi$  (which are subsequently internalized) as described in Section 3.1.4.

As running example for the remainder of this section, we use the Delphin code fragment in Figure 4 that shows an implementation of a theorem prover for propositional logic (Figure 1) with implication, conjunction, and negation. We have taken the freedom to use pseudo code to improve the presentation.

```

world W= c.o · c.nd · (nd + o)*
with
  prove :: ∀A : o. ∃D : nd A. ⊤
  infer :: ∀A : o. ∀P : o. ∀D : nd A. ∃D : nd P. ⊤

fun prove (and M1 A1 A2) =
  choose (M2 □ c.nd M1)
  in let val <D1, <>> = prove A1
      val <D2, <>> = prove A2
      in <andI M2 A1 A2 D, <>>
      end
  end
| prove (imp M1 A1 A2) =
  choose (M2 □ c.nd M1)
  in let val <D, <>> = new {b □ nd A1}
      in prove A2
      end
      in <impl M2 A1 A2 D, <>>
      end
  end
| prove (not M1 A') =
  choose (M2 □ c.nd M1)
  in let val <D, <>> =
      new {b □ o} {c □ nd A'}
      in prove b
      end
      in <notI M2 A' D, <>>
      end
  end
| prove (M1 : o) =
  choose (M2 □ nd A')
  in infer A' M1 M2
  end
end

fun infer (imp M1 A1 A2) P D =
  choose (M2 □ c.nd M1)
  in let val <D1, <>> = prove A1
      in infer A2 P (impE M2 D1 D)
      end
  end
| infer (and M1 A1 A2) P D =
  choose (M2 □ c.nd M1)
  in (let val <D1, <>> = prove A1
      in infer A2 P (andE1 M2 D1 D)
      end)
  || (let val <D2, <>> = prove A2
      in infer A2 P (andE2 M2 D2 D)
      end)
  end
| infer (not M1 A2) P D =
  choose (M2 □ c.nd M1)
  in let val <D2, <>> = prove A2
      in infer A2 P (notE M2 D2 D)
      end
  end
| infer (M1 □ o) P D = <D, <>>
end

```

Figure 4: A Theorem Prover For Propositional Logic written in Delphin.

Although it may look academic, `prove` may be used in applications related to PCC where theorems and proofs are used to guarantee safety of mobile code, and PCA that requires users to construct certificates (proofs) to gain access to a resources on servers or in databases. Detailed case studies in these particular application domains, however, are left to future work.

### 3.1.1 Dependent Functions and Products

The program in Figure 4 defines the world  $W$  that is given in Example 2.4 and inhabits it with a theorem prover for natural deduction derivations that consists of two mutually recursive parts called `prove` and `infer`. We fix the signature  $\Sigma$  to contain the type constants `o`, `c.o`, `nd`, `c.nd`, and all respective constructor definitions as described in Section 2.

The theorem prover is invoked via `prove` and expects a formula as argument, which it then transforms into a natural deduction proof should it exist. To this end it applies introduction rules bottom-up, until the formula to be proven is a proposition. Then, function `infer` is invoked as a helper function,

applying elimination rules in top-down fashion on some non-deterministically chosen hypothesis.

The body of the world declaration contains two type ascriptions for `prove` and `infer` and their respective implementations. Delphin level types  $F$  should not be confused with  $\text{LF}^\Sigma$  types  $A$ . When we say type, it will always be clear from the context which one is meant. Universal quantifiers  $\forall$  range over  $\text{LF}^\Sigma$  derivations and existential quantifiers  $\exists$  denote the result type of the function. The fact that both functions are declared before they are defined tells Delphin that they are mutually recursive. Internally they are stored as one single recursive function of type

$$\begin{aligned} F_0 = \forall A : \circ. \exists D : \text{nd } A. \top \wedge \\ \forall A : \circ. \forall P : \circ. \forall D : \text{nd } A. \exists D : \text{nd } P. \top \end{aligned} \quad (5)$$

where projections are used to select the appropriate part. Formally the argument to `prove` is a  $\text{LF}^\Sigma$  derivation  $\Gamma \vdash_\Sigma A \uparrow \circ$  and it produces a result that contains the  $\text{LF}^\Sigma$  derivation of  $\Gamma \vdash_\Sigma D \uparrow \text{nd } A$ . When programming in Delphin the user may omit all references to  $\Gamma$ . This is a fundamental design decision underlying Delphin which is made possible by requiring that all input and output arguments of a Delphin function are valid in one and the same context  $\Gamma$ . In addition  $\Gamma$  is required to be the image of  $[\Phi]$  for a meta-level context  $\Phi$  valid in world  $W$ . Thus,  $\Gamma$  can be hidden from the user, who can refer to it only through the choice and the new operator described below.

Not illustrated in Figure 4 are the function space  $\forall x \square A. F$  and its corresponding product space  $\exists x \square A. F$  that are necessary when the type of a Delphin function refers to constructors.

**Example 3.1 (Double negation introduction)** That a derivation of  $\Delta \vdash A$  can be transformed into a derivation  $\Delta \vdash \neg\neg A$  can be expressed in Delphin as a function of type

$$\begin{aligned} \text{notnotI} :: \forall c_1 \square c. \circ. \forall A : \circ. \forall D_1 : \text{nd } A. \\ \exists D_1 : \text{nd } (\text{not } c_1 (\text{not } c_1 A)). \top \end{aligned}$$

whose straightforward implementation we omit.

Also missing from the example are Delphin-level functions  $F_1 \supset F_2$  that map meta-level objects from  $F_1$  to  $F_2$ . This function space is best explained in analogy to the function space used in traditional functional programming languages. Delphin does currently not permit user-defined types. The only type constant is  $\top$ , which corresponds to the `unit` type of ML. For the precise characterization of types supported by Delphin, consult Figure 5.

Figure 4 uses quite a bit of syntactic sugar to make the source code more readable. Internally recursive functions are introduced by a leading  $\mu x \in F. P$ , followed by several occurrences of  $\Lambda x : A. P$ ,  $\Lambda x \square A. P$ , or  $\Lambda x \in F. P$ , depending on the type of the function. We have chosen the capital  $\Lambda$  to distinguish it clearly from the  $\lambda$  defined for  $\text{LF}^\Sigma$  and it should not be confused with polymorphic

Declarations	$D ::= x : A \mid x \square A$
Types	$F ::= \forall D. F \mid \exists D. F$ $\mid F_1 \supset F_2 \mid F_1 \wedge F_2 \mid \top$
Values	$V ::= \{\eta; \Lambda D. P\} \mid \langle M; V \rangle$ $\mid \{\eta; \Lambda x \in F. P\} \mid \langle V_1; V_2 \rangle \mid \langle \rangle$
Programs	$P ::= \{\eta; P\} \mid \Lambda D. P \mid \langle M; P \rangle$ $\mid \Lambda x \in F. P \mid \langle P_1; P_2 \rangle \mid \langle \rangle$ $\mid x \mid P M \mid P_1 P_2$ $\mid \text{case } \Omega \mid \mu x \in F. P$ $\mid \text{let } (x \in F) = P_1 \text{ in } P_2$ $\mid \text{let } D = M \text{ in } P \mid P_1 \parallel P_2$ $\mid \kappa(\Psi \triangleright x \square A). P \mid \nu x \square A. P$
Patterns	$\sigma ::= id_\Psi \mid \sigma, M/x \mid \sigma, V/x$
Cases	$\Omega ::= . \mid \Omega, (\Psi \triangleright \sigma \mapsto P)$
Contexts	$\Psi ::= . \mid \Psi, D \mid \Psi, x \in F$
Environments	$\eta ::= id_\Psi \mid \eta, M/x \mid \eta, P/x$

Figure 5: Delphin type and program syntax

abstraction, which is not treated here at all. In Delphin, functions are often defined by cases which is described in detail in Section 3.1.2.

We write  $P M$  for application to a  $\text{LF}^\Sigma$  object (for both universal quantifiers), and  $P_1 P_2$  for the application of a Delphin functions to Delphin programs. We refrain from introducing a different symbol for this kind of application, since from looking at the argument, it is always clear which application is meant. Similarly, the objects of product types are of the form  $\langle M; P \rangle$ , or  $\langle P_1; P_2 \rangle$ , and the elimination form is covered as well by case analysis which is discussed in Section 3.1.2. Unit is written as  $\langle \rangle$ . Delphin provides three let statements binding  $x : A$ ,  $x \square A$ , and  $x \in F$ . Applications, pairs, and let statement correspond almost directly to the ones used in Figure 4. We write  $\{\eta; P\}$  for closures.

### 3.1.2 Function Definition by Cases

Patterns in Delphin are non-linear, higher-order, dependently typed, and non-local. Objects may occur as index objects to type families, and case analysis over one object may inevitably lead to instantiations of index objects as well, automatically annihilating many impossible cases. Thus, traditional pattern matching techniques are only partially applicable. Consider, for example, the Delphin context

$$m_1 \square_{\text{c.o}}, m_2 \square_{\text{c.nd}} m_1, A : \text{o}, D : \text{nd } A.$$

$D$  may be split into eight cases, one for each inference rule depicted in Figure 1, but in some cases, e.g. **andl**, **impl**, or **negl**,  $A$ 's form is instantiated to  $A_1 \wedge A_2$ ,  $A_1 \supset A_2$  or  $\neg A'$ , respectively. Recall that no assumption of the form  $x \sqsupset A$  can ever be split.

An elegant solution to this problem was proposed in [15]. Patterns as defined in Figure 5 are valid substitutions with strict co-domains. As usual we exclude program of functional types from patterns  $\sigma$ . In our example, each of the four cases of **prove** is guarded by one pattern (given here) and each of the four cases of **infer** is guarded by similar pattern as well (omitted). Recall  $F_0$ , the type of the recursive function in Figure 4, defined in (5). Each pattern has domain  $\Psi, a : \circ$ , where  $\Psi = f \in F_0$ .

$$\begin{array}{ll} \Psi, m_1 \sqsupset \mathbf{c.o}, a_1 : \circ, a_2 : \circ & \vdash_{\Sigma;W} \text{id}_{\Psi}, (\mathbf{and} \ m_1 \ a_1 \ a_2)/a \\ \Psi, m_1 \sqsupset \mathbf{c.o}, a_1 : \circ, a_2 : \circ & \vdash_{\Sigma;W} \text{id}_{\Psi}, (\mathbf{imp} \ m_1 \ a_1 \ a_2)/a \\ \Psi, m_1 \sqsupset \mathbf{c.o}, a' : \circ & \vdash_{\Sigma;W} \text{id}_{\Psi}, (\mathbf{not} \ m_1 \ a')/a \\ \Psi, p \sqsupset \circ & \vdash_{\Sigma;W} \text{id}_{\Psi}, p/a \end{array}$$

Cases  $\Omega$  are defined as a list of individual triples  $(\Psi \triangleright \sigma \mapsto P)$  consisting of a co-domain, the pattern, and the program that is to be executed if the case applies. The strictness requirement [11] guarantees that matching against  $a$  will indeed instantiate all variables in the co-domain which may occur free in the program.

The elegance of this formulation of patterns comes to light when defining pattern matching on environments that are substitutions as well (see Figure 5).

**Definition 3.2 (Pattern-matching)** *Let  $\eta$  be an environment such that  $\Phi \vdash_{\Sigma;W} \eta : \Psi$  and  $\sigma$  be a pattern such that  $\Psi' \vdash_{\sigma;W} \sigma : \Psi$ . We say  $\sigma$  matches  $\eta$  if and only if there is an environment  $\eta'$  (satisfying  $\Phi \vdash_{\sigma;W} \eta' : \Psi'$ ) such that  $\eta = \sigma \circ \eta'$ , where  $\circ$  stands for substitution composition.*

### 3.1.3 Non-Deterministic Choice

Without a global supply of constructor names, Delphin charts new territory on how users can nevertheless make use of them for the destruction and construction of  $\text{LF}^{\Sigma}$  objects. In Delphin, the only way to destruct  $\text{LF}^{\Sigma}$  objects is through pattern matching. Since each occurrence of a constructor in a pattern is parametrized by an  $m \sqsupset A$ , their origins are quite easy to discover. The previous example illustrates this observation in that each of the first three co-domains contains a declaration  $m_1 \sqsupset \mathbf{c.o}$  and the fourth a declaration  $p \sqsupset \circ$ .  $m_1$  and  $p$  denote the source for the respective constructors.

On the one hand, the basic requirement for construction is that each resource  $m \sqsupset A$  providing constructors is in fact declared in the context. Otherwise type checking would be impossible. Of course, by inspection of the world  $W$  in Example 2.4, it is clear that there can be at most one declaration  $m_1$ , and for that matter only one  $m_2 \sqsupset \mathbf{c.nd} \ m_1$ . For all other constituents of the world that may occur repeatedly, such as  $n_i \sqsupset \mathbf{nd}$  and  $o_i \sqsupset \circ$ , things are a little bit more difficult. Without explicit reference to the local context  $\Gamma$ , the user can neither

iterate nor access any declarations during run-time besides the ones singled out by pattern-matching. Unless of course, Delphin provides him or her with the appropriate functionality called *non-deterministic choice*.

In Figure 4, some cases start with **choose** ( $M_2 \square_{\text{c.nd}} M_1$ ). The intended meaning is that Delphin chooses a declaration  $m_2 \square_{\text{c.nd}} m_1$  from  $\Phi$  non-deterministically, match it against  $M_2 \square_{\text{c.nd}} M_1$ , and instantiates  $M_2$  with  $m_2$  and  $M_1$  with  $m_1$  such that the overall evaluation leads to a value. Each variable that occurs free in **choose** is a binding occurrence. Often, variables are constrained due to dependencies justifying the already bound  $M_1$  as an argument to **c.nd**. **choose** ( $M_2 \square_{\text{nd}} A'$ ) in the parameter case of **prove** illustrates non-deterministic choice, since there may be several assumptions of this form in  $\Phi$ . Formally, we write  $\kappa(\Psi \triangleright x \square A).P$ , where  $\Psi$  denote the new variables to be bound, and  $x \square A$ , the pattern, and  $P$  the program to be executed afterwards. Non-determinism has a second face in Delphin. We write  $P_1 \parallel P_2$  for alternative execution of  $P_1$  and  $P_2$ .

### 3.1.4 Dynamic Constructors

Programming with  $\text{LF}^\Sigma$  encodings requires Delphin to recurse under  $\lambda$ -binders for many applications. The **imp** and **not** cases of **prove** in Figure 4 pose only simple examples. Others include, type derivation preserving compilers, type checkers, and interpreters for experimental languages. And again, since the overall context  $\Gamma$  is hidden, users cannot add to it directly, but must instead use Delphin's *dynamic constructors* that allow new constructors  $m \square A$  to be introduced during runtime, and discharged upon return from a subcomputation.

As an example, we consider the **not** case in the definition of **prove** in more detail. Both  $b \square_{\text{o}}$  and  $c \square_{\text{nd}} A'$  are new declarations that are added to  $\Phi$  when the **new** evaluates. It is easy to verify that  $\vdash \Phi \in W$  implies that

$$\vdash \Phi, b \square_{\text{o}}, c \square_{\text{nd}} A' \in W$$

as well. In this extended context **prove**  $b$  is invoked. If it terminates it terminates with result  $D' b c$ , s.t.

$$[\Phi, b \square_{\text{o}}, c \square_{\text{nd}} A'] \vdash_{\Sigma} D' b c \uparrow_{\text{nd}} b$$

holds. All objects must be valid in the same context, which can be easily established by two invocations of **obj\_lam** resulting in

$$[\Phi] \vdash_{\Sigma, W} \lambda b : \text{o}. \lambda c : \text{nd } A'. D' b c : \Pi b : \text{o}. \text{nd } A' \rightarrow \text{nd } b.$$

Thus, execution continues, and

$$\text{notl } M_2 A' (\lambda b : \text{o}. \lambda c : \text{nd } A'. D' b c)$$

is returned. In the general case, of course, extensions  $m \square A$  may declare more than just one constructor, all of which may occur free in the returned value of a subcomputation. Examples include Delphin programs that can translate proofs from one proof theory into another [17]. In Delphin, we write  $\nu m \square A.P$  for introducing constructors dynamically.

## 3.2 Type System

Delphin's type system is specified in Figure 6, which defines the following three typing judgments:

$$\begin{array}{ll} \text{Valid programs} & \Psi \vdash_{\Sigma;W} P \in F \\ \text{Valid cases} & \Psi \vdash_{\Sigma;W} \Omega \in F \\ \text{Valid substitutions} & \Psi \vdash_{\Sigma;W} \eta \in \Psi'. \end{array}$$

In the interest of space, we omit the definition of validity for Delphin types and Delphin contexts. The validity for values follows from the validity of programs because values are subsumed by programs. Similarly, patterns and environments are subsumed by substitutions.

As a short hand notation, we write  $x?A$  for  $x : A$  or  $x \square A$ . The universal introduction and elimination rules  $\forall I$ ,  $\forall E$ ,  $\supset I$ , and  $\supset E$  are standard. Note that  $\Psi$  is converted into a  $\text{LF}^\Sigma$  level context using the  $[\ ]$  operation defined in Section 2.1 bridging the gap between object and meta level. Regarding the existentials, programs are always pairs. It is sufficient to give only the introduction rules  $\exists I$  and  $\wedge E$ , since the respective elimination rules can be safely omitted because they are subsumed by the `case` rule. `var` and `true` are standard. `rec` and `case` are two rules that introduce recursion and pattern-matching. Clearly, well-typed programs neither need not terminate nor make progress. There is a `let` rule for each possible assumption  $x : A$ ,  $x \square A$ , and of course  $x \in F$ , and a typing rule `clo` for closures.

The new rule is inspired by [17] and is one of the novel features made available to functional programming. It permits new dynamic constructors to be introduced into the context during runtime and discharged after termination as illustrated by Example in Section 3.1.4. The type  $A \nearrow F$  in the conclusion of rule `new` does not denote a new type, but it stands for an operation on the Delphin type level. The idea is simple. This operation traverses  $A$ , and parametrizes every  $\text{LF}^\Sigma$  type by the respective constructors. We define it first for the  $\text{LF}^\Sigma$  level and then for Delphin. Let  $m \square A = \Sigma x_1 : A_1. \dots \Sigma x_n : A_n. A_{n+1}$  be the most generic form of a  $\Sigma$ -type, and  $M, A'$  valid in a context that contains  $m \square A$ .

### Definition 3.3 ( $\text{LF}^\Sigma$ abstraction)

$$\begin{aligned} (m : A) \nearrow A' &= \Pi x_1 : A_1. \dots \Pi x_n : A_n. \Pi x_{n+1} : A_{n+1}. \\ &\quad A'[\langle x_1; \dots \langle x_n; x_{n+1} \rangle \rangle / m] \\ (m : A) \nearrow M &= \lambda x_1 : A_1. \dots \lambda x_n : A_n. \lambda x_{n+1} : A_{n+1}. \\ &\quad M[\langle x_1; \dots \langle x_n; x_{n+1} \rangle \rangle / m] \end{aligned}$$

Applied to the example from Section 3.1.4 we obtain:  $(b \square \text{o}) \nearrow (c \square \text{nd } A') \nearrow \text{nd } b = \Pi b : \text{o}. \text{nd } A' \rightarrow \text{nd } b$ . Let  $M_2 : A$  be an argument to an  $M_1 : (m \square A) \nearrow A'$ .

### Definition 3.4 ( $\text{LF}^\Sigma$ application)

$$M_1 \searrow M_2 = M_1 (\pi_1 M_2) \dots (\pi_2^{n+1} M_2)$$

The arguments enumerate the individual constructors  $\pi_1, \pi_1 \circ \pi_2, \dots, \pi_1 \circ \pi_2^n, \pi_2^{n+1}$ . Finally, we exploit  $\text{LF}^\Sigma$  level abstraction on application, and generalize it to Delphin level types  $F$ , also valid in a context that contains  $m \sqsupset A$ . Note, that abstraction can only work on  $\text{LF}^\Sigma$  objects hidden inside a Delphin values. Thus, abstraction must leave functional Delphin types and Delphin objects alone.

**Definition 3.5 (Delphin type abstraction)**

$$\begin{aligned}
(m \sqsupset A) \nearrow \top &= \top \\
(m \sqsupset A) \nearrow (\forall x? A'. F') &= \forall m \sqsupset A. \forall x? A'. F' \\
(m \sqsupset A) \nearrow (\exists x? A'. F') &= \exists x? ((m : A) \nearrow A'). \\
&\quad ((m \sqsupset A) \nearrow F'[x \searrow m/x]) \\
(m \sqsupset A) \nearrow (F_1 \supset F_2) &= \forall m \sqsupset A. F_1 \supset F_2 \\
(m \sqsupset A) \nearrow (F_1 \wedge F_2) &= ((m \sqsupset A) \nearrow F_1) \wedge (m \sqsupset A) \nearrow F_2
\end{aligned}$$

**Definition 3.6 (Delphin value abstraction)** *Let  $\Phi, m \sqsupset A \vdash_{\Sigma; w} V \in F$ .  $A \nearrow V$  is defined as follows.*

$$\begin{aligned}
(m \sqsupset A) \nearrow \{\eta; \Lambda D. P\} &= \{\text{id}_\Phi; \Lambda m \sqsupset A. \{\eta; \Lambda D. P\}\} \\
(m \sqsupset A) \nearrow \langle M; V \rangle &= \langle (m : A) \nearrow M; (m \sqsupset A) \nearrow V \rangle \\
(m \sqsupset A) \nearrow \{\eta; \Lambda x \in F. P\} &= \{\text{id}_\Phi; \Lambda m \sqsupset A. \{\eta; \Lambda x \in F. P\}\} \\
(m \sqsupset A) \nearrow \langle V_1; V_2 \rangle &= \langle (m \sqsupset A) \nearrow V_1; (m \sqsupset A) \nearrow V_2 \rangle \\
(m \sqsupset A) \nearrow \langle \rangle &= \langle \rangle
\end{aligned}$$

In this form, abstraction takes into account every single constructor declared in  $m \sqsupset A$ , including those that may in reality never occur within an object of the type being abstracted. As described in [17], those can be filtered out by strengthening captured by the so called dependency relation [18]. This theory, however, has not been extended to the realm of  $\text{LF}^\Sigma$  as of yet, that we omit the discussion here. However, we do not foresee and conceptual problems in generalizing dependency relations to  $\text{LF}^\Sigma$ .

**Lemma 3.7 (Abstraction)**

1. If  $\Gamma, m : A' \vdash_\Sigma M \uparrow A$  then  $\Gamma \vdash_\Sigma (m : A') \nearrow M \uparrow (m : A') \nearrow A$ .
2. If  $\Phi, m \sqsupset A \vdash_{\Sigma; w} V \in F$  then  $\Phi \vdash_{\Sigma; w} (m \sqsupset A) \nearrow V \in (m \sqsupset A) \nearrow F$ .
3.  $M \equiv ((m : A) \nearrow M) \searrow m$ .

**Proof:** 1. By induction on the type  $A$ . Proof omitted.

2. By induction on the formula  $F$ .

**Case:**  $F = \top$

$$\begin{array}{l} \Phi, m \Box A \vdash_{\Sigma; w} \langle \rangle \in \top \quad \text{by assumption} \\ \Phi, m \Box A \vdash_{\Sigma; w} (m \Box A) \nearrow \langle \rangle \in (m \Box A) \nearrow \top \quad \text{by Definition 3.6} \end{array}$$

**Case:**  $F = \forall x?A'. F'$

$$\begin{array}{l} \Phi, m \Box A \vdash_{\Sigma; w} \{\eta; \Lambda x?A'. P\} \in \forall x?A'. F' \quad \text{by assumption} \\ \Phi \vdash_{\Sigma; w} \Lambda m \Box A. \{\eta; \Lambda x?A'. P\} \in \forall m \Box A. \forall x?A'. F' \quad \text{by } \forall I \\ \Phi \vdash_{\Sigma; w} id_{\Phi} \in \Phi \quad \text{by id} \\ \Phi \vdash_{\Sigma; w} \{id_{\Phi}; \Lambda m \Box A. \{\eta; \Lambda x?A'. P\}\} \in \forall m \Box A. \forall x?A'. F' \quad \text{by clo} \\ \Phi \vdash_{\Sigma; w} (m \Box A) \nearrow \{\eta; \Lambda x?A'. P\} \in (m \Box A) \nearrow \forall x?A'. F' \quad \text{by Definitions 3.5, 3.6} \end{array}$$

**Case:**  $F = \exists x?A'. F'$

$$\begin{array}{l} \Phi, m \Box A \vdash_{\Sigma; w} \langle M; V \rangle \in \exists x?A'. F' \quad \text{by assumption} \\ [\Phi, m \Box A] \vdash_{\Sigma} M \uparrow A' \quad \text{by inversion} \\ \Phi, m \Box A \vdash_{\Sigma; w} V \in F'[M/x] \quad \text{by inversion} \\ [\Phi], m : A \vdash_{\Sigma} M \uparrow A' \quad \text{by Definition } [\cdot] \\ [\Phi] \vdash_{\Sigma} (m : A) \nearrow M \uparrow (m : A) \nearrow A' \quad \text{by i.h. (1)} \\ \Phi \vdash_{\Sigma; w} (m \Box A) \nearrow V \in (m \Box A) \nearrow F'[M/x] \quad \text{by i.h. (2)} \\ M \equiv ((m : A) \nearrow M) \searrow m \quad \text{by i.h. (3)} \\ \Phi \vdash_{\Sigma; w} (m \Box A) \nearrow V \in (m \Box A) \nearrow F'[((m : A) \nearrow M) \searrow m/x] \\ \Phi \vdash_{\Sigma; w} \langle (m : A) \nearrow M; (m \Box A) \nearrow V \rangle \in \exists x?(m : A) \nearrow A'. (m \Box A) \nearrow F'[x \searrow m/x] \\ \quad \text{by } \exists I \\ \Phi \vdash_{\Sigma; w} (m \Box A) \nearrow \langle M; V \rangle \in (m \Box A) \nearrow \exists x?A'. F' \quad \text{by Definitions 3.5, 3.6} \end{array}$$

**Case:**  $F = F_1 \supset F_2$

$$\begin{array}{l} \Phi, m \Box A \vdash_{\Sigma; w} \{\eta; \Lambda x \in F'_1. P\} \in F_1 \supset F_2 \quad \text{by assumption} \\ \Phi \vdash_{\Sigma; w} \Lambda m \Box A. \{\eta; \Lambda x \in F'_1. P\} \in \forall m \Box A. F_1 \supset F_2 \quad \text{by } \forall I \\ \Phi \vdash_{\Sigma; w} id_{\Phi} \in \Phi \quad \text{by id} \\ \Phi \vdash_{\Sigma; w} \{id_{\Phi}; \Lambda m \Box A. \{\eta; \Lambda x \in F'_1. P\}\} \in (\forall m \Box A. F_1 \supset F_2) \quad \text{by clo} \\ \Phi \vdash_{\Sigma; w} (m \Box A) \nearrow \{\eta; \Lambda x \in F'_1. P\} \in (m \Box A) \nearrow (F_1 \supset F_2) \quad \text{by Definitions 3.5, 3.6} \end{array}$$

**Case:**  $F = F_1 \wedge F_2$

$$\begin{array}{l} \Phi, m \Box A \vdash_{\Sigma; w} \langle V_1; V_2 \rangle \in F_1 \wedge F_2 \quad \text{by assumption} \\ \Phi, m \Box A \vdash_{\Sigma; w} V_1 \in F_1 \quad \text{by inversion} \\ \Phi, m \Box A \vdash_{\Sigma; w} V_2 \in F_2 \quad \text{by inversion} \\ \Phi \vdash_{\Sigma; w} (m \Box A) \nearrow V_1 \in (m \Box A) \nearrow F_1 \quad \text{by i.h. (2)} \\ \Phi \vdash_{\Sigma; w} (m \Box A) \nearrow V_2 \in (m \Box A) \nearrow F_2 \quad \text{by i.h. (2)} \\ \Phi \vdash_{\Sigma; w} \langle (m \Box A) \nearrow V_1; (m \Box A) \nearrow V_2 \rangle \in ((m \Box A) \nearrow F_1) \wedge ((m \Box A) \nearrow F_2) \\ \quad \text{by } \wedge I \\ \Phi \vdash_{\Sigma; w} (m \Box A) \nearrow \langle V_1; V_2 \rangle \in (m \Box A) \nearrow (F_1 \wedge F_2) \quad \text{by Definitions 3.5, 3.6} \end{array}$$

3. By induction on the type  $A$ . Proof omitted.  $\square$

The `choose` rule assigns a type to the non-deterministic choose operator described in Section 3.1.3.  $\Psi'$  accounts for all variables that may occur free in  $A$ . Recall that the strictness premiss guarantees that each variable declared in  $\Psi'$  occurs in a rigid position [7], and is therefore guaranteed to be instantiated during evaluation. Because strictness is not necessary for the type preservation proof below, we omit its definition altogether. The interested reader is referred to [15] for a detailed account. The `par` typing rule for nondeterministic execution is self explanatory.

The programs  $\nu$  and  $\kappa$  used in accordance with the correct world are a novel concepts in functional programming languages. In this respect, Delphin differs significantly from traditional functional programming languages such as ML and Haskell.

The second block of rules in Figure 6 establish the meaning of the validity of cases where `empty` assigns a type to the empty list, and `cons` to the non empty list. In analogy to `choose`,  $\Psi_2$  contains a set set of variables that occur free in the pattern  $\sigma$ , and as above, it is expected to satisfy the strictness requirement.

And finally the bottom block of rules in Figure 6 establishes a typing discipline for patterns, substitutions, and environments. The rule `id` is trivial, `env1` and `env1` cover the  $\text{LF}^\Sigma$ , and the Delphin case alike. The notation  $A[\eta]$  stands for LF substitution level application. Because of Theorem 2.1 (2),  $A[\eta]$  possesses a canonical form and so does  $M$ .

### 3.3 Operational Semantics

As usual in programming language design, Delphin's operational semantics may be call-by-value or call-by-name. One possible but by no means the only operational semantics is given in Figure 7. It is a big-step semantics, because a small-step version would be more convoluted and more difficult to understand. Although, more properties of Delphin would follow. The evaluation judgment  $\Phi; \eta \vdash_W P \hookrightarrow V$  relates the program  $P$  to be evaluated with the outcome of the evaluation  $V$ . Evaluation takes place in context  $\Phi$  that can only consist of declarations of the form  $m \square A$  which is ensured in the rules by premiss  $\vdash \Phi \in W$ .  $\eta$  is an environment.

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_new` introduces, retracts, and abstracts the dynamic constructors following Definition 3.6, `ev_choose` selects non-deterministically one set of constructors from  $\Phi$  that matches  $A$ , and `ev_par1` and `ev_par2` select non-deterministically which of the two alternative programs to execute. `ev_case`, `ev_yes`, and `ev_no` govern case analysis. Following Definition 3.2, pattern-matching is successful only if the second premiss of `ev_yes` manages to construct a new and refined environment  $\eta'$ , which can only be determined during runtime using higher-order matching.

### 3.4 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 constructors  $m \square A$  may be dynamically introduced, but they are guaranteed to be discharged by the time the computation terminates with a value. Delphin's operational semantics is type preserving. We begin the formal presentation with a few infrastructure lemmas.

**Lemma 3.8 (Properties of substitutions)** *Let  $\Phi \vdash_{\Sigma;W} \sigma \in \Psi$  and  $[\Phi] \vdash_{\Sigma} M \uparrow A[\sigma]$ . Then the following properties hold:*

1.  $(\sigma, M/x) = (\sigma, x/x) \circ M/x$
2.  $(\sigma, M[\sigma]/x) = M/x \circ \sigma$

**Proof:** The proof follows by unfolding the definition of substitution. □

**Lemma 3.9 (Substitution property for LF)** *Let  $\Phi \vdash_{\Sigma;W} \sigma \in \Psi$ . If  $[\Psi] \vdash_{\Sigma} M \uparrow A$  then  $[\Phi] \vdash_{\Sigma} M[\sigma] \uparrow A[\sigma]$ .*

**Proof:** By induction on the canonicity derivation. □

**Lemma 3.10 (Composition)** 1. *If  $\Psi_2 \vdash_{\Sigma;W} \sigma_1 \in \Psi_1$  and  $\Psi_1 \vdash_{\Sigma;W} \sigma_0 \in \Psi_0$  then  $\Psi_2 \vdash_{\Sigma;W} \sigma_0 \circ \sigma_1 \in \Psi_0$ .*

2. *If  $\Psi_2 \vdash_{\Sigma;W} \sigma_0 \circ \sigma_1 \in \Psi_0$  and  $\Psi_1 \vdash_{\Sigma;W} \sigma_0 \in \Psi_0$  then  $\Psi_2 \vdash_{\Sigma;W} \sigma_1 \in \Psi_1$ .*

3. *If  $\Psi_2 \vdash_{\Sigma;W} \sigma_0 \in \Psi_0$  and  $\Psi_2 \vdash_{\Sigma;W} \sigma_1 \in \Psi_1$  then  $\Psi_2 \vdash_{\Sigma;W} \sigma_0, \sigma_1 \in \Psi_0, \Psi_1$ .*

**Proof:** By induction on one of the given typing derivation for substitutions. □

In our system, weakening holds as expected. These particular weakening properties are needed for the proof of type preservation.

**Lemma 3.11 (Weakening)** *If  $\Gamma \vdash_{\Sigma} A : \text{type}$  and  $\Gamma' \vdash_{\Sigma} \sigma : \Gamma$  then for all  $\Gamma' \vdash_{\Sigma} \sigma, \sigma' : \Gamma, \Gamma'$  it holds that  $\Gamma \vdash_{\Sigma} A[\sigma] \equiv A[\sigma, \sigma'] : \text{type}$*

**Proof:** By induction on the validity derivation of  $A$ . Since we have omitted the rules, we also omit the proof. □

**Lemma 3.12 (Weakening of Substitution)** *If  $\Psi_1 \vdash_{\Sigma;W} \eta \in \Psi$  then  $\Psi_1, D \vdash_{\Sigma;W} \eta \in \Psi$ .*

**Proof:** By induction on the typing derivation for substitutions. □

It is trivial, that declarations can be looked up in the context.

**Lemma 3.13 (Lookup)** *If  $(x \in F) \in \Psi$  and  $\Phi \vdash_{\Sigma;W} \eta \in \Psi$  then  $\Phi \vdash_{\Sigma;W} \eta(x) \in F[\eta]$ .*

**Proof:** By induction on  $\Psi$ . □

**Lemma 3.14 (Commutativity)** *It holds that*

1.  $((m : A) \nearrow A')[\eta] \equiv (m : A[\eta]) \nearrow A'[\eta]$ .
2.  $((m \sqcap A) \nearrow F)[\eta] \equiv (m \sqcap A[\eta]) \nearrow F[\eta]$ .

**Proof:** In the first case by induction on the type  $A$ , and in the second case, by induction on the formula  $F$ . □

All is in place for the proof of the main theorem, type preservation.

**Theorem 3.15 (Type-preservation)**

1. *If  $\Psi \vdash_{\Sigma;W} P \in F$  and  $\Phi \vdash_{\Sigma;W} \eta \in \Psi$  and  $\Phi \vdash_W P \hookrightarrow V$  then  $\Phi \vdash_W V \in F[\eta]$ .*
2. *If  $\Psi \vdash_{\Sigma;W} \Omega \in F$  and  $\Phi \vdash_{\Sigma;W} \eta \in \Psi$  and  $\Phi \vdash_W \eta \sim \Omega \hookrightarrow V$  then  $\Phi \vdash_W V \in F[\eta]$ .*

**Proof:** By structural induction on the evaluation derivation  $\Phi; \eta \vdash_W P \hookrightarrow V$ .

**Case:**  $\text{ev-}\Lambda_{1,2}$

$\Phi; \eta \vdash_W \Lambda D. P' \hookrightarrow \{\eta; \Lambda D. P\}$	by assumption
$\vdash \Phi \in W$	by assumption
$\Phi \vdash_{\Sigma;W} \eta \in \Psi$	by assumption
$\Psi \vdash_{\Sigma;W} \Lambda D. P' \in \forall D. F'$	by assumption
$\Psi \vdash_{\Sigma;W} \{\eta; \Lambda D. P'\} \in (\forall D. F')[\eta]$	by clo

**Case:**  $\text{ev-app}_1$

$\Phi; \eta \vdash_W P M \hookrightarrow V$	by assumption
$\Phi; \eta \vdash_W P \hookrightarrow \{\eta'; \Lambda x?A. P'\}$	by assumption
$\Phi; \eta', M[\eta]/x \vdash_W P' \hookrightarrow V$	by assumption
$\Phi \vdash_{\Sigma;W} \eta \in \Psi$	by assumption
$\Psi \vdash_{\Sigma;W} P M \in F[M/x]$	by assumption
$\Psi \vdash_{\Sigma;W} P \in \forall x?B. F$	by inversion
$[\Psi] \vdash_{\Sigma} M \uparrow B$	by inversion
$\Phi \vdash_{\Sigma;W} \{\eta'; \Lambda x?A. P'\} \in \forall x?B[\eta]. F'[\eta, x/x]$	by inversion
$\Phi \vdash_{\Sigma;W} \eta' \in \Psi'$	by assumption
$\Psi' \vdash_{\Sigma;W} \Lambda x?A. P' \in \forall x?A. F'$	by inversion
$(\forall x?B[\eta]. F')[\eta, x/x] = (\forall x?A. F')[\eta'] = \forall x?A[\eta']. F'[\eta', x/x]$	
$B[\eta] = A[\eta']$	by inversion
$F'[\eta', x/x] = F[\eta, x/x]$	by inversion
$[\Phi] \vdash_{\Sigma} M[\eta] \uparrow B[\eta]$	by Lemma 3.9
$\Psi', x?A \vdash_{\Sigma;W} P' \in F'$	by inversion

$$\begin{aligned}
\Phi \vdash_{\Sigma;w} \eta', M[\eta]/x \in \Psi', x?A[\eta] & && \text{by inversion} \\
\Phi \vdash_{\Sigma;w} V \in F'[\eta', M[\eta]/x] & && \text{by i.h. (1)} \\
F'[\eta', M[\eta]/x] = (F'[\eta', x/x])[M[\eta]/x] = (F[\eta, x/x])[M[\eta]/x] & && \\
= (F[\eta, M[\eta]/x]) = (F[M/x])[\eta] & && \text{by Lemma 3.8} \\
\Phi \vdash_{\Sigma;w} V \in (F[M/x])[\eta] & && 
\end{aligned}$$

**Case: ev\_app<sub>2</sub>**

$$\begin{aligned}
\Phi; \eta \vdash_w P_1 P_2 \hookrightarrow V & && \text{by assumption} \\
\Phi; \eta \vdash_w P_1 \hookrightarrow \{\eta'; \Lambda x \in F'. P'_1\} & && \text{by assumption} \\
\Phi; \eta \vdash_w P_2 \hookrightarrow V_2 & && \text{by assumption} \\
\Phi; \eta', V_2/x \vdash_w P'_1 \hookrightarrow V & && \text{by assumption} \\
\Phi \vdash_{\Sigma;w} \eta \in \Psi & && \text{by assumption} \\
\Psi \vdash_{\Sigma;w} P_1 P_2 \in G & && \text{by assumption} \\
\Psi \vdash_{\Sigma;w} P_1 \in F \supset G & && \text{by assumption} \\
\Psi \vdash_{\Sigma;w} P_2 \in F & && \text{by assumption} \\
\Phi \vdash_{\Sigma;w} \{\eta'; \Lambda x \in F'. P'_1\} \in (F \supset G)[\eta] & && \text{by i.h. (1)} \\
\Phi \vdash_{\Sigma;w} \eta' \in \Psi' & && \text{by inversion} \\
\Psi' \vdash_{\Sigma;w} \Lambda x \in F'. P'_1 \in F' \supset G' & && \text{by inversion} \\
F[\eta] = F'[\eta'] & && \text{by inversion} \\
G[\eta] = G'[\eta'] & && \text{by inversion} \\
\Psi', x \in F' \vdash_{\Sigma;w} P'_1 \in G' & && \text{by assumption} \\
\Phi \vdash_{\Sigma;w} V_2 \in F[\eta] & && \text{by i.h. (1)} \\
\Phi \vdash_{\Sigma;w} \eta', V_2/x \in \Psi', x \in F' & && \text{by env}_2 \\
\Phi \vdash_{\Sigma;w} V \in G'[\eta'] & && \text{by assumption} \\
\Phi \vdash_{\Sigma;w} V \in G[\eta] & && 
\end{aligned}$$

**Case: ev\_inx**

$$\begin{aligned}
\Phi; \eta \vdash_w \langle M; P \rangle \hookrightarrow \langle M[\eta]; V \rangle & && \text{by assumption} \\
\Phi; \eta \vdash_w P \hookrightarrow V & && \text{by assumption} \\
\Phi \vdash_{\Sigma;w} \eta \in \Psi & && \text{by assumption} \\
\Psi \vdash_{\Sigma;w} \langle M; P \rangle \in \exists x : A. F & && \text{by assumption} \\
[\Psi] \vdash_{\Sigma} M \uparrow A & && \text{by inversion} \\
\Psi \vdash_{\Sigma;w} P \in F[M/x] & && \text{by inversion} \\
\Psi \vdash_{\Sigma;w} V \in F[M/x][\eta] & && \text{by i.h. (1)} \\
\Psi \vdash_{\Sigma;w} V \in F[\eta, M[\eta]/x] & && \text{by Lemma 3.8} \\
\Psi \vdash_{\Sigma;w} V \in F[\eta, x/x][M[\eta]/x] & && \text{by Lemma 3.8} \\
[\Phi] \vdash_{\Sigma} M[\eta] \uparrow A[\eta] & && \text{by Lemma 3.9} \\
\Psi \vdash_{\Sigma;w} \langle M[\eta]; V \rangle \in \exists x : A[\eta]. F[\eta, x/x] & && \text{by } \exists \\
\Psi \vdash_{\Sigma;w} \langle M[\eta]; V \rangle \in (\exists x : A. F)[\eta] & && 
\end{aligned}$$

**Case: ev\_pair**

$\Phi; \eta \vdash_w \langle P_1; P_2 \rangle \leftrightarrow \langle V_1; V_2 \rangle$	by assumption
$\Phi; \eta \vdash_w P_1 \leftrightarrow V_1$	by inversion
$\Phi; \eta \vdash_w P_2 \leftrightarrow V_2$	by inversion
$\Phi \vdash_{\Sigma; W} \eta \in \Psi$	by assumption
$\Psi \vdash_{\Sigma; W} \langle P_1; P_2 \rangle \in F_1 \wedge F_2$	by assumption
$\Psi \vdash_{\Sigma; W} P_1 \in F_1$	by inversion
$\Psi \vdash_{\Sigma; W} P_2 \in F_2$	by inversion
$\Psi \vdash_{\Sigma; W} V_1 \in F_1[\eta]$	by i.h. (1)
$\Psi \vdash_{\Sigma; W} V_2 \in F_2[\eta]$	by i.h. (1)
$\Psi \vdash_{\Sigma; W} \langle V_1; V_2 \rangle \in (F_1 \wedge F_2)[\eta]$	by $\wedge$

**Case: ev\_var**

$\Phi; \eta \vdash_W x \leftrightarrow \eta(x)$	by assumption
$\Phi \vdash_{\Sigma; W} \eta \in \Psi$	by assumption
$\Psi \vdash_{\Sigma; W} x \in F$	by assumption
$(x \in F) \in \Psi$	by inversion
$\Phi \vdash_{\Sigma; W} \eta(x) \in F[\eta]$	by Lemma 3.13

**Case: ev\_unit**

$\Phi; \eta \vdash_w \langle \rangle \leftrightarrow \langle \rangle$	by assumption
$\Phi \vdash_{\Sigma; W} \eta \in \Psi$	by assumption
$\Psi \vdash_{\Sigma; W} \langle \rangle \in \top$	by assumption
$\Phi \vdash_{\Sigma; W} \langle \rangle \in \top[\eta]$	by $\top$

**Case: ev\_clo**

$\Phi; \eta \vdash_w \{\eta'; P\} \leftrightarrow V$	by assumption
$\Phi; \eta' \circ \eta \vdash_w P \leftrightarrow V$	by inversion
$\Phi \vdash_{\Sigma; W} \eta \in \Psi$	by assumption
$\Psi \vdash_{\Sigma; W} \{\eta'; P\} \in F[\eta']$	by assumption
$\Psi \vdash_{\Sigma; W} \eta' \in \Psi'$	by inversion
$\Psi' \vdash_{\Sigma; W} P \in F$	by inversion
$\Phi \vdash_{\Sigma; W} \eta' \circ \eta \in \Psi'$	by Lemma 3.10 (1)
$\Psi' \vdash_{\Sigma; W} V \in F[\eta' \circ \eta]$	by i.h. (1)
$\Psi' \vdash_{\Sigma; W} V \in F[\eta'][\eta]$	

**Case: ev\_rec**

$\Phi; \eta \vdash_w \mu x \in F. P \leftrightarrow V$	by assumption
$\Phi; \eta, \mu x \in F. P/x \vdash_w P \leftrightarrow V$	by inversion
$\Phi \vdash_{\Sigma; W} \eta \in \Psi$	by assumption
$\Psi \vdash_{\Sigma; W} \mu x \in F. P \in F$	by assumption
$\Psi, x \in F \vdash_{\Sigma; W} P \in F$	by inversion

$$\begin{array}{l} \Phi \vdash_{\Sigma;W} \{\eta; \mu x \in F. P\} \in F[\eta] \\ \Phi \vdash_{\Sigma;W} \eta, \{\eta; \mu x \in F. P\}/x \in (\Psi, x \in F) \\ \Phi \vdash_{\Sigma;W} V \in F[\eta] \end{array} \quad \begin{array}{l} \text{by clo} \\ \text{by assumption} \\ \text{by i.h. (1)} \end{array}$$

**Case: ev\_case**

$$\begin{array}{l} \Phi; \eta \vdash_W \text{case } \Omega \leftrightarrow V \\ \Phi \vdash_W \eta \sim \Omega \leftrightarrow V \\ \Phi \vdash_{\Sigma;W} \Omega \in F \\ \Phi \vdash_{\Sigma;W} \eta \in \Psi \\ \Phi \vdash_{\Sigma;W} V \in F[\eta] \end{array} \quad \begin{array}{l} \text{by assumption} \\ \text{by inversion} \\ \text{by assumption} \\ \text{by assumption} \\ \text{by i.h. (1)} \end{array}$$

**Case: ev\_let**

$$\begin{array}{l} \Phi; \eta \vdash_W \text{let } (x?A) = M \text{ in } P \leftrightarrow V \\ \Phi; \eta, M[\eta]/x \vdash_W P \leftrightarrow V \\ \Phi \vdash_{\Sigma;W} \eta \in \Psi \\ \Psi \vdash_{\Sigma;W} \text{let } (x?A) = M \text{ in } P \in F \\ [\Psi] \vdash_{\Sigma} M \uparrow A \\ \Psi, x?A \vdash_{\Sigma;W} P \in F \\ [\Phi] \vdash_{\Sigma} M[\eta] \uparrow A[\eta] \\ \Phi \vdash_{\Sigma;W} \eta, M[\eta]/x \in \Psi, x?A \\ \Phi \vdash_{\Sigma;W} V \in F[\eta, M[\eta]/x] \\ \Phi \vdash_{\Sigma;W} V \in F[M/x][\eta] \end{array} \quad \begin{array}{l} \text{by assumption} \\ \text{by inversion} \\ \text{by assumption} \\ \text{by assumption} \\ \text{by inversion} \\ \text{by Lemma 3.9} \\ \text{by env}_1 \\ \text{by i.h. (1)} \\ \text{by Lemma 3.8 (2)} \end{array}$$

**Case: ev\_let<sub>2</sub>**

$$\begin{array}{l} \Phi; \eta \vdash_W \text{let } (x \in F_1) = P_1 \text{ in } P_2 \leftrightarrow V \\ \Phi; \eta \vdash_W P_1 \leftrightarrow V_1 \\ \Phi; \eta, V_1/x \vdash_W P_2 \leftrightarrow V \\ \Phi \vdash_{\Sigma;W} \eta \in \Psi \\ \Psi \vdash_{\Sigma;W} \text{let } (x \in F_1) = P_1 \text{ in } P_2 \in F_2 \\ \Psi \vdash_{\Sigma;W} P_1 \in F_1 \\ \Psi, x \in F_1 \vdash_{\Sigma;W} P_2 \in F_2 \\ \Phi \vdash_{\Sigma;W} V_1 \in F_1[\eta] \\ \Phi \vdash_{\Sigma;W} \eta, V_1/x \in \Psi, x \in F_1 \\ \Phi \vdash_{\Sigma;W} V \in F_2[\eta] \end{array} \quad \begin{array}{l} \text{by assumption} \\ \text{by inversion} \\ \text{by inversion} \\ \text{by assumption} \\ \text{by assumption} \\ \text{by inversion} \\ \text{by inversion} \\ \text{by i.h. (1)} \\ \text{by env}_2 \\ \text{by i.h. (1)} \end{array}$$

**Case: ev\_new**

$$\begin{array}{l} \Phi; \eta \vdash_W \nu(x \square A). P \leftrightarrow (A[\eta] \nearrow V) \\ \Phi, x \square A[\eta]; \eta, x/x \vdash_W P \leftrightarrow V \\ \Phi \vdash_{\Sigma;W} \eta \in \Psi \end{array} \quad \begin{array}{l} \text{by assumption} \\ \text{by inversion} \\ \text{by assumption} \end{array}$$

$\Psi \vdash_{\Sigma;W} \nu(x \Box A). P \in A \nearrow F$	by assumption
$\Psi, x \Box A \vdash_{\Sigma;W} P \in F$	by inversion
$\Phi, x \Box A[\eta] \vdash_{\Sigma;W} \eta \in \Psi$	by Lemma 3.12
$\Phi, x \Box A[\eta] \vdash_{\Sigma;W} \eta, x/x \in \Psi, x \Box A$	by env <sub>1</sub>
$\Phi, x \Box A[\eta] \vdash_{\Sigma;W} V \in F[\eta, x/x]$	by inversion
$\Phi \vdash_{\Sigma;W} A[\eta] \nearrow V \in A[\eta] \nearrow F[\eta]$	by Lemma 3.7
$\Phi \vdash_{\Sigma;W} A[\eta] \nearrow V \in (A \nearrow F)[\eta]$	by Lemma 3.14

**Case: ev\_choose**

$\Phi; \eta \vdash_w \kappa(\Psi' \triangleright x \Box A). P \leftrightarrow V$	by assumption
$\Phi \vdash_w \eta' : \Psi'$	by inversion
$(y \Box A[\eta']) \in \Phi$	by inversion
$\Phi; \eta, \eta', y/x \vdash_w P \leftrightarrow V$	by inversion
$\Phi \vdash_{\Sigma;W} \eta \in \Psi$	by assumption
$\Psi \vdash_{\Sigma;W} \kappa(\Psi' \triangleright x \Box A). P \in F'$	by assumption
$\Psi, \Psi', x \Box A \vdash_{\Sigma;W} P \in F$	by inversion
$\Phi \vdash_{\Sigma;W} \eta, \eta' \in \Psi, \Psi'$	by Lemma 3.10 (3)
$A[\eta] \equiv A[\eta, \eta']$	by Lemma 3.11
$\Phi \vdash_{\Sigma;W} \eta, \eta', y/x \in \Psi, \Psi', x \Box A$	by env <sub>1</sub>
$\Phi \vdash_{\Sigma;W} V \in F[\eta, \eta', y/x]$	by i.h. (1)
$\Phi \vdash_{\Sigma;W} V \in F[\eta]$	by strengthening

**Case: ev\_par<sub>1</sub>**

$\Phi; \eta \vdash_w P_1 \parallel P_2 \leftrightarrow V$	by assumption
$\Phi; \eta \vdash_w P_1 \leftrightarrow V$	by inversion
$\Phi \vdash_{\Sigma;W} \eta \in \Psi$	by assumption
$\Psi \vdash_{\Sigma;W} P_1 \parallel P_2 \in F$	by assumption
$\Psi \vdash_{\Sigma;W} P_1 \in F$	by inversion
$\Phi \vdash_{\Sigma;W} V \in F[\eta]$	by i.h. (1)

**Case: ev\_par<sub>2</sub>**

$\Phi; \eta \vdash_w P_1 \parallel P_2 \leftrightarrow V$	by assumption
$\Phi; \eta \vdash_w P_2 \leftrightarrow V$	by inversion
$\Phi \vdash_{\Sigma;W} \eta \in \Psi$	by assumption
$\Psi \vdash_{\Sigma;W} P_1 \parallel P_2 \in F$	by assumption
$\Psi \vdash_{\Sigma;W} P_2 \in F$	by inversion
$\Phi \vdash_{\Sigma;W} V \in F[\eta]$	by i.h. (1)

**Case: ev\_yes**

$\Phi \vdash_w \eta \sim (\Omega, (\Psi_2 \triangleright \psi \mapsto P)) \leftrightarrow V$	by assumption
--	---------------

$\Phi; \eta' \vdash_w P \hookrightarrow V$	by inversion
$\psi \circ \eta' = \eta$	by inversion
$\Phi \vdash_{\Sigma; W} \eta \in \Psi_1$	by assumption
$\Psi_1 \vdash_{\Sigma; W} \Omega, (\Psi_2 \triangleright \sigma \mapsto P) \in F$	by assumption
$\Psi_2 \vdash_{\Sigma; W} \sigma \in \Psi_1$	by inversion
$\Psi_2 \vdash_{\Sigma; W} P \in F[\sigma]$	by inversion
$\Phi \vdash_{\Sigma; W} \eta' \in \Psi_2$	by Lemma 3.10 (2)
$\Psi_2 \vdash_{\Sigma; W} P \in F[\sigma][\eta']$	by i.h.(1)
$\Psi_2 \vdash_{\Sigma; W} P \in F[\eta]$	

**Case: ev\_no**

$\Phi \vdash_w \eta \sim (\Omega, (\Psi \triangleright \psi \mapsto P)) \hookrightarrow V$	by assumption
$\Phi \vdash_w \eta \sim \Omega \hookrightarrow V$	by inversion
$\Phi \vdash_{\Sigma; W} \eta \in \Psi_1$	by assumption
$\Psi_1 \vdash_{\Sigma; W} \Omega, (\Psi_2 \triangleright \sigma \mapsto P) \in F$	by assumption
$\Psi_1 \vdash_{\Sigma; W} \Omega \in F$	by inversion
$\Psi_1 \vdash_{\Sigma; W} V \in F[\eta]$	by i.h. (2)

□

The current version of the typing rules given in Figure 6 do not enforce that  $\Phi$  remains “well-worlded” during execution. The world plays an important role, in that they govern which patterns to consider. Without worlds, a program may introduce constructors in arbitrary forms, for which a programmer might not have provided a case. The only rule that extends the world  $\Phi$  during execution is `ev_new`. Hence, augmenting `new` with the side condition

$$\begin{aligned} &\text{If } \vdash \Phi \in W \text{ and } \Phi \vdash_{\Sigma; W} \eta \in \Psi \\ &\text{then } \vdash \Phi, x \sqcap A[\eta] \in W. \end{aligned} \tag{6}$$

guarantees that the current world  $W$  is not left during execution.

## 4 Implementation

An experimental version of Delphin that is described in Section 3 is implemented in Standard ML of New Jersey. It features  $\text{LF}^\Sigma$ -like datatypes, a parser, a type reconstruction algorithm, a type checker, and an implementation of the operational semantics.

**Datatypes:** Delphin employs Twelf [12]’s internal representation to model  $\text{LF}^\Sigma$ .  $\Sigma$ -types are currently not part of the Twelf system, but in view of this work will be added soon. On the other hand, Twelf already offers a useful implementation of the concept of blocks [15] that were used in our implementation of Delphin.

**Type reconstruction:** When programming in Delphin, users are allowed to omit a significant portion of reconstructible information. Each universal and existential Delphin type constructors can be flagged as “inferable”, which tells Delphin to determine the omitted information via higher-order unification, if possible. For example, the second argument to `notnotI` in Example 3.1 can be safely omitted because the third argument contains information about  $A$ . But there is more redundant information, Delphin programmers are currently not allowed to omit, because of the use of blocks.  $\text{LF}^\Sigma$  objects contain often much redundant information inferable from the type information of other arguments [10]. For example, the arguments  $A_1$  and  $A_2$  to `andI` in the first case of `prove` in Figure 4 may be omitted as well, because it can be reconstructed from the type of  $D$ .

**Type checking:** Type checking with dependent  $\Sigma$ -types can be quite challenging, because principal types do not always exist. Which type to assign is a complicated type inference problem that can be solved using a bi-directional type inference technique. Type annotations are often in order. 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  $\text{LF}^\Sigma$  and Delphin are decidable.

**Interpreter:** The interpreter implements the operational semantics from Figure 7. Due to the regular nature of this context in which execution takes place, Delphin functions bare in general a high degree of parallelism, useful not only for programming but also for compiling Delphin programs to parallel architectures, e.g. computing grids as described in Section 3.1.3. We plan to investigate these observations further in future work.

## 5 Related Work

The languages DML [21] and Cayenne [3] 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  $\text{LF}^\Sigma$  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  $\text{LF}$ ’s  $\lambda^{\text{II}}$  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.

Dependent higher-order datatypes have also attracted a lot of attention in staged computation since they safely support tag-elimination [9]. Guarded recursive datatype constructors [20] solves this problem by passing higher-order encodings for run-time type analysis.

Washburn and Weirich [19] have given an implementation of the modal  $\lambda$ -calculus in Haskell. Using abstract type classes and polymorphism they have shown that programming with higher-order encodings in a simply typed programming language is possible. Their work, however does not support dependent types.

FreshML [13, 5] 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. Delphin’s two-level design allows programming with recursion and pattern matching to coexist with elegant higher-order encodings in LF.

## 6 Conclusion

We have presented Delphin, a functional programming language based on  $\text{LF}^\Sigma$  an extension of LF [6] by  $\Sigma$ -types. The principal novel feature of Delphin is a strict separation between the representation of data and the programs that manipulate such data. Constructors are encoded as projections from  $\Sigma$ -types. In  $\text{LF}^\Sigma$  every object has a canonical form, making it a prime candidate for higher-order encodings, such as proofs, type systems, program transformations, and operational semantics, etc. Auxiliary functionality such as access to resources in a context or application of substitutions are provided implicitly and “for free” by  $\text{LF}^\Sigma$ . Thus, Delphin enables programmers to compute with these complex data structures as easily as programmers can manipulate conventional data structures in mainstream functional programming languages.

For these domains, Delphin can be seen as domain specific programming language, without exceptions, state, or module system, but a world system instead. In future work we plan to make Delphin a more general purpose programming language by adding support for the standard constraint domains, including integers, reals, doubles, strings, and Booleans.

## 7 Acknowledgments

I would like to thank Frank Pfenning, Jeffrey Sarnat, for their input on the theoretical end of this work, and Richard Fontana, Yu Liao, Adam Poswolsky, and Henrik Nilsson for many helpful discussions that proved to be directly relevant to the implementation of Delphin.

## References

- [1] A. W. Appel. Foundational proof-carrying code. In *Logic in Computer Science*, 2001.
- [2] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *ACM Conference on Computer and Communications Security*, pages 52–62, 1999.
- [3] L. Augustsson. Cayenne - a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] G. Huet. A unification algorithm for typed  $\lambda$ -calculus. *Theoretical Computer Science*, 1:27–57, 1975.
- [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] E. Pašalić, W. Taha, and T. Sheard. Tagless staged interpreters for typed languages. In *Proceedings of International Conference on Functional Programming*, Pittsburgh, PA, 2002. ACM Press.
- [10] 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.
- [11] F. Pfenning and C. Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*. Springer-Verlag LNCS 1657, 1998. To appear.

- [12] 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.
- [13] 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.
- [14] J. Sarnat. Lf with  $\Sigma$ -types. Technical report, Yale University, 2003. 690 report.
- [15] C. Schürmann. *Automating the Meta-Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, 2000. CMU-CS-00-146.
- [16] 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.
- [17] 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.
- [18] R. Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 1999. Forthcoming.
- [19] G. Washburn and S. Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *Proceedings of International Conference on Functional Programming*. ACM Press, 2003. to appear.
- [20] H. Xi. Guarded recursive datatype constructors. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, January 2003. ACM press.
- [21] 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.

$\frac{[\Psi] \vdash_{\Sigma} A \uparrow \text{type} \quad \Psi, x?A \vdash_{\Sigma;W} P \in F}{\Psi \vdash_{\Sigma;W} \Lambda x?A. P \in \forall x?A. F} \forall I$	$\frac{\Psi \vdash_{\Sigma;W} P \in \forall x?A. F \quad [\Psi] \vdash_{\Sigma} M \uparrow A}{\Psi \vdash_{\Sigma;W} P M \in F[M/x]} \forall E$
$\frac{\Psi, x \in F_1 \vdash_{\Sigma;W} P \in F_2}{\Psi \vdash_{\Sigma;W} \Lambda x \in F_1. P \in F_1 \supset F_2} \supset I$	$\frac{\Psi \vdash_{\Sigma;W} P_1 \in F_2 \supset F_1 \quad \Psi \vdash_{\Sigma;W} P_2 \in F_2}{\Psi \vdash_{\Sigma;W} P_1 P_2 \in F_1} \supset E$
$\frac{[\Psi] \vdash_{\Sigma} M \uparrow A \quad \Psi \vdash_{\Sigma;W} P \in F[M/x]}{\Psi \vdash_{\Sigma;W} \langle M; P \rangle \in \exists x?A. F} \exists I$	$\frac{\Psi \vdash_{\Sigma;W} P_1 \in F_1 \quad \Psi \vdash_{\Sigma;W} P_2 \in F_2}{\Psi \vdash_{\Sigma;W} \langle P_1; P_2 \rangle \in F_1 \wedge F_2} \wedge I$
$\frac{(x \in F) \in \Psi}{\Psi \vdash_{\Sigma;W} x \in F} \text{var}$	$\frac{}{\Psi \vdash_{\Sigma;W} \langle \rangle \in \top} \text{true}$
$\frac{\Psi, x \in F \vdash_{\Sigma;W} P \in F}{\Psi \vdash_{\Sigma;W} \mu x \in F. P \in F} \text{rec}$	$\frac{\Psi \vdash_{\Sigma;W} \Omega \in F}{\Psi \vdash_{\Sigma;W} \text{case } \Omega \in F} \text{case}$
$\frac{\Psi \vdash_{\Sigma;W} M \in A \quad \Psi, x?A \vdash_{\Sigma;W} P \in F}{\Psi \vdash_{\Sigma;W} \text{let } (x?A) = M \text{ in } P \in F[M/x]} \text{let}_1$	$\frac{\Psi \vdash_{\Sigma;W} P_1 \in F_1 \quad \Psi, x \in F_1 \vdash_{\Sigma;W} P_2 \in F_2}{\Psi \vdash_{\Sigma;W} \text{let } (x \in F_1) = P_1 \text{ in } P_2 \in F_2} \text{let}_2$
$\frac{\Psi, x \square A \vdash_{\Sigma;W} P \in F}{\Psi \vdash_{\Sigma;W} \nu(x \square A). P \in A \nearrow F} \text{new}$	$\frac{\vdash \Psi' \text{ strict } A \quad \Psi, \Psi', x \square A \vdash_{\Sigma;W} P \in F}{\Psi \vdash_{\Sigma;W} \kappa(\Psi' \triangleright x \square A). P \in F'} \text{choose}$
$\frac{\Psi \vdash_{\Sigma;W} P_1 \in F \quad \Psi \vdash_{\Sigma;W} P_2 \in F}{\Psi \vdash_{\Sigma;W} P_1 \parallel P_2 \in F} \text{par}$	$\frac{\Psi \vdash_{\Sigma;W} \eta \in \Psi' \quad \Psi' \vdash_{\Sigma;W} P \in F}{\Psi \vdash_{\Sigma;W} \{\eta; P\} \in F[\eta]} \text{clo}$
$\frac{}{\Psi \vdash_{\Sigma;W} \cdot \in F} \text{empty}$	
$\frac{\Psi_1 \vdash_{\Sigma;W} \Omega \in F \quad \vdash \Psi_2 \text{ strict } \sigma \quad \Psi_2 \vdash_{\Sigma;W} \sigma \in \Psi_1 \quad \Psi_2 \vdash_{\Sigma;W} P \in F[\sigma]}{\Psi_1 \vdash_{\Sigma;W} \Omega, (\Psi_2 \triangleright \sigma \mapsto P) \in F} \text{cons}$	
$\frac{}{\Psi \vdash_{\Sigma;W} \text{id}_{\Psi} \in \Psi} \text{id}$	
$\frac{\Psi \vdash_{\Sigma;W} \eta \in \Psi' \quad [\Psi] \vdash_{\Sigma} M \uparrow A[\eta]}{\Psi \vdash_{\Sigma;W} \eta, M/x \in \Psi', x?A} \text{env}_1$	$\frac{\Psi \vdash_{\Sigma;W} \eta \in \Psi' \quad \Psi \vdash_{\Sigma;W} P \in F[\eta]}{\Psi \vdash_{\Sigma;W} \eta, P/x \in \Psi', x \in F} \text{env}_2$

Figure 6: Delphin Typing Rules.

$$\begin{array}{c}
\frac{\vdash \Phi \in W}{\Phi; \eta \vdash_W \Lambda D. P \hookrightarrow \{\eta; \Lambda D. P\}} \text{ev-}\Lambda_{1,2} \\
\frac{\Phi; \eta \vdash_W P \hookrightarrow \{\eta'; \Lambda x?A. P'\} \quad \Phi; \eta', M[\eta]/x \vdash_W P' \hookrightarrow V}{\Phi; \eta \vdash_W P M \hookrightarrow V} \text{ev\_app}_1 \\
\frac{\Phi; \eta \vdash_W P_1 \hookrightarrow \{\eta'; \Lambda x \in F. P'_1\} \quad \Phi; \eta \vdash_W P_2 \hookrightarrow V_2 \quad \Phi; \eta', V_2/x \vdash_W P'_1 \hookrightarrow V}{\Phi; \eta \vdash_W P_1 P_2 \hookrightarrow V} \text{ev\_app}_2 \\
\frac{\Phi; \eta \vdash_W P \hookrightarrow V}{\Phi; \eta \vdash_W \langle M; P \rangle \hookrightarrow \langle M[\eta]; V \rangle} \text{ev\_inx} \quad \frac{\Phi; \eta \vdash_W P_1 \hookrightarrow V_1 \quad \Phi; \eta \vdash_W P_2 \hookrightarrow V_2}{\Phi; \eta \vdash_W \langle P_1; P_2 \rangle \hookrightarrow \langle V_1; V_2 \rangle} \text{ev\_pair} \\
\frac{\vdash \Phi \in W}{\Phi; \eta \vdash_W x \hookrightarrow \eta(x)} \text{ev\_var} \quad \frac{\vdash \Phi \in W}{\Phi; \eta \vdash_W \langle \rangle \hookrightarrow \langle \rangle} \text{ev\_unit} \quad \frac{\Phi; \eta' \circ \eta \vdash_W P \hookrightarrow V}{\Phi; \eta \vdash_W \{\eta'; P\} \hookrightarrow V} \text{ev\_clo} \\
\frac{\Phi; \eta, \mu x \in F. P/x \vdash_W P \hookrightarrow V}{\Phi; \eta \vdash_W \mu x \in F. P \hookrightarrow V} \text{ev\_rec} \quad \frac{\Phi \vdash_W \eta \sim \Omega \hookrightarrow V}{\Phi; \eta \vdash_W \text{case } \Omega \hookrightarrow V} \text{ev\_case} \\
\frac{\Phi; \eta, M[\eta]/x \vdash_W P \hookrightarrow V}{\Phi; \eta \vdash_W \text{let } (x?A) = M \text{ in } P \hookrightarrow V} \text{ev\_let}_1 \\
\frac{\Phi; \eta \vdash_W P_1 \hookrightarrow V_1 \quad \Phi; \eta, V_1/x \vdash_W P_2 \hookrightarrow V}{\Phi; \eta \vdash_W \text{let } (x \in F) = P_1 \text{ in } P_2 \hookrightarrow V} \text{ev\_let}_2 \\
\frac{\Phi, x \sqcap A[\eta]; \eta, x/x \vdash_W P \hookrightarrow V}{\Phi; \eta \vdash_W \nu(x \sqcap A)P \hookrightarrow (A[\eta] \nearrow V)} \text{ev\_new} \\
\frac{\Phi \vdash_W \eta' : \Psi \quad (y \sqcap A[\eta']) \in \Phi \quad \Phi; \eta, \eta', y/x \vdash_W P \hookrightarrow V}{\Phi; \eta \vdash_W \kappa(\Psi \triangleright x \sqcap A). P \hookrightarrow V} \text{ev\_choose} \\
\frac{\Phi; \eta \vdash_W P_1 \hookrightarrow V}{\Phi; \eta \vdash_W P_1 \| P_2 \hookrightarrow V} \text{ev\_par}_1 \quad \frac{\Phi; \eta \vdash_W P_2 \hookrightarrow V}{\Phi; \eta \vdash_W P_1 \| P_2 \hookrightarrow V} \text{ev\_par}_2 \\
\frac{\Phi; \eta' \vdash_W P \hookrightarrow V \quad \psi \circ \eta' = \eta}{\Phi \vdash_W \eta \sim (\Omega, (\Psi \triangleright \psi \mapsto P)) \hookrightarrow V} \text{ev\_yes} \quad \frac{\Phi \vdash_W \eta \sim \Omega \hookrightarrow V}{\Phi \vdash_W \eta \sim (\Omega, (\Psi \triangleright \psi \mapsto P)) \hookrightarrow V} \text{ev\_no}
\end{array}$$

Figure 7: Delphin Operational Semantics.