Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 228 (2009) 113–120
www.elsevier.com/locate/entcs

System Description: Delphin – A Functional Programming Language for Deductive Systems

Adam Poswolsky[1,2]

Yale University
New Haven, CT, USA

Carsten Schürmann[1,3]

IT University of Copenhagen
Copenhagen, Denmark

Abstract

Delphin is a functional programming language [6] utilizing dependent higher-order datatypes. Delphin's two-level type-system cleanly separates data from computation, allowing for decidable type checking. The data level is LF [2], which allows for the specification of deductive systems following the judgments-as-types methodology. The computation level facilitates the manipulation of such encodings by providing facilities for pattern matching, recursion, and the dynamic creation of new parameters (which can be thought of as scoped constants). Delphin's documentation and examples are available online at `http://delphin.logosphere.org`.

Keywords: LF, dependent types, higher-order abstract syntax, HOAS, Twelf, Delphin

1 Introduction

Delphin is a functional programming language built to facilitate the encoding of, manipulation of, and reasoning over dependent higher-order datatypes. Delphin is a two level system that separates data-level functions from computation-level functions. The data level is the logical framework LF [2], which supports dependent types and *higher-order abstract syntax* (HOAS). The computation level provides mechanisms such as case analysis and recursion to allow for the manipulation of

[1] This research has been funded by NSF grants CCR-0325808 and CCR-0133502.
[2] Email: `adam.poswolsky@yale.edu`
[3] Email: `carsten@itu.dk`

doi:10.1016/j.entcs.2008.12.120

data. Its most novel feature is in its support of the dynamic creation of parameters (i.e. scoped constants).

Delphin is first and foremost a general purpose programming language supporting complex data structures. However, it also contains strong tools for meta-reasoning allowing one to determine if functions are total and hence Delphin can also be used to formalize proofs. Thus, it is well-suited to be used in the setting of the Logosphere project [9], a digital library of formal proofs that brings together different proof assistants and theorem provers with the goal to facilitate the exchange of mathematical knowledge by converting proofs from one logical formalism into another. Delphin has been successfully used in expressing translations between HOL, Nuprl, and various other logics.

As a running example, we will describe a translator from the natural deduction calculus (implicational fragment) to the Hilbert calculus, or equivalently a translator from the simply-typed λ-calculus into combinators.

2 Specification of Data

We refer to the language being encoded as the *object language*. We write $\ulcorner \_ \urcorner$ for the representation function mapping elements of the object language to their representations in LF.

We adopt the same syntax for LF as in Twelf [5] where a λ-abstraction is written as `[x:A]e` and application as juxtaposition. A function's type is written as `{x:A}B`, but we often write `A -> B` when `x` does not occur in `B`. LF is well-suited to represent complex object languages because variable binders can be represented using LF functions, eliminating the need to worry about the representation of variables, renamings, or substitutions.

We distinguish LF terms from computation-level expressions by enclosing the former in `<...>`. For example, the expression `e f` refers to computation-level application whereas the application in `<M N>` occurs at the LF level. Similarly, terms of type `<A1 -> A2>` are `<M>` where `M` is an LF function.

Figure 1 illustrates how to define datatypes. The Delphin keyword `sig` constructs a new type given the name of the type and its constructors, analogous to the `datatype` keyword in ML. We refer to the collection of datatypes as the *signature*. Formulas $A, B ::= A \supset B \mid b$ are represented as terms of type `tp`. We use `ar` as a right-associative infix constructor and represent the base type using `b`. Terms of type A in the simply-typed λ-calculus are represented as terms of type `exp` $\ulcorner A \urcorner$. This illustrates an example with both dependent types as well as HOAS. For example, $\ulcorner \lambda x{:}(b \to b).\ \lambda y{:}b.\ x\ y \urcorner$ is `lam [x : exp (b ar b)] lam [y : exp b] app x y`. Finally, we represent derivations in the Hilbert calculus, also known as typed combinators, as terms of type `comb` $\ulcorner A \urcorner$.

```
(* Types *)
sig <tp : type>
    <ar : tp -> tp -> tp> %infix right 10
    <b  : tp> ;
(* Simply-Typed λ-calculus *)
sig <exp : tp -> type>
    <lam : (exp A -> exp B) -> exp (A ar B)>
    <app : exp (A ar B) -> exp A -> exp B> ;
(* Combinators *)
sig <comb : tp -> type>
    <k  : comb (A ar B ar A)>
    <s  : comb ((A ar B ar C) ar (A ar B) ar A ar C) >
    <mp : comb (A ar B) -> comb A -> comb B> ;
```

Fig. 1. Translator (Part 1/4): Datatype Declarations

```
params = <exp A>, <comb A>;
```

Fig. 2. Translator (Part 2/4): Params Declarations

3 Computation

Delphin's most novel feature is the *new* construct, which is written as {<x>}e and has type {<x>}T [4]. Evaluation of e occurs while the binding x remains uninstantiated. Therefore, for the scope of e, the variable x can be thought of as a fresh constant, which we call a parameter. One may view this as providing a way to dynamically extend the signature. Expressions {<x>}e evaluate to {<x>}v, where v is a value [6].

Delphin pervasively distinguishes between LF types A, parameter types A#, and computation-level types T. In the expression {<x>}e, the variable x has type A#. Additionally, x also has type A, and as such A# can be seen as a subtype of A.

Since one may dynamically create arbitrary parameters, a Delphin function expects arguments that are constructed from constants in the signature as well as dynamically introduced parameters. For example, given a function foo, the evaluation of {<x : (exp A)#>}foo <x> would get stuck if foo did not provide a case for parameters of type exp A. In Delphin, we use the params keyword to indicate which parameters our functions are intended to handle. Note that a function may call a function with a different params specification as long as the callee can handle all the parameters of the caller. The ability to call functions that make sense with respect to different parameters is known as *world subsumption*.

We proceed to motivate the Delphin programming language with our example of writing a translator from simply typed λ-terms into combinators. The entire construction is depicted in Figures 1, 2, 3, and 4.

Figure 2 declares that the functions that follow are intended to handle an arbitrary collection of dynamically created simply-typed λ-terms and combinators.

[4] In the formal system [6], the term is $\nu x.\, e$ and the type is $\nabla x.\, \tau$.

```
fun ba : <comb A -> comb B> -> <comb (A ar B)>
  = fn <[x] x> => <mp (mp s k) (k : comb (A ar A ar A))>
     | <[x] mp (C1 x) (C2 x)> => (case ((ba <C1>), (ba <C2>))
                                  of (<C1'>, <C2'>) => <mp (mp s C1') C2'>)
     | <[x] C> => <mp k C> ;
```

Fig. 3. Translator (Part 3/4): Bracket Abstraction

The first step in our translator is *bracket abstraction*, or `ba`, which internalizes abstraction in the combinator calculus. If `M` has type `comb A -> comb B`, then we can use `ba` to get a combinator `M'` of type `comb (A ar B)`. Subsequently, if we have a `C` of type `comb A`, then the term `mp M' C` is equivalent to `M C` in the combinator calculus.

The code for `ba` is shown in Figure 3. Functions are defined by cases, and in this example we are performing case analysis over LF functions. The first case handles the creation of *identity* combinators, i.e. of type `A ar A`. The second case handles `mp` by performing bracket abstraction on both parts and using the `s` combinator on the results. Notice that we perform case analysis on the results of the recursive calls since we need to extract the LF-level `C1'` from the computation-level `<C1'>`. The third case handles combinators `C` of type `comb B` that do not use the hypothetical combinator `x`; in this situation, we use the `k` combinator. This function indeed covers all cases.

We write `T1 -> T2` for the type of non-dependent functions, and we write `<x : A> -> T` when `x` can occur in `T`. Observe that the variables `A` and `B` occur free in the type of `ba`. The full type of `ba` is `<A : tp> -> <B : tp> -> <comb A -> comb B> -> <comb (A ar B)>`. The omission of the first two argument types tells the frontend to treat the first two arguments *implicitly*. This greatly simplifies our code as it is redundant to explicitly supply an input argument which is indexed in another input argument. When applying the function, the reconstruction engine will automatically fill in the implicit arguments. This support is just a frontend convenience and does not affect the underlying theory.

Next we write `convert`, which traverses simply-typed λ-terms and uses `ba` to convert them into combinators. We will see that it is necessary to introduce new parameters of `exp A` and `comb A` together. In order to maintain the relationship between these parameters, we pass around a *parameter function*, i.e. a computational function whose domains ranges over parameters. In this example, we will pass around a function `W` of type `<(exp B)#> -> <comb B>`. From a first-order perspective, `W` can be thought of as a substitution since it maps parameters to terms.

Figure 4 shows us the entire `convert` function. For readability we employ type aliasing and abbreviate the type of the parameter function as `paramFun`. The first argument is the parameter function `W`, and we will need to update this function when we create new parameters. One may think of `W` as a continuation to be applied to parameters. In the `lam` case, we recurse on `E` by first creating a parameter `x` to which `E` can be applied. However, `W` is not defined with respect to the new

```
type paramFun = <(exp B)#> -> <comb B>;
fun convert : paramFun -> <exp A> -> <comb A>
  = fn W <lam [x] E x> =>
         (case ({<x>}{<u>} convert (W with <x> => <u>) <E x>)
            of ({<x>}{<u>} <C u>) => ba <C>)
     | W <app E1 E2> => (case ((convert W <E1>), (convert W <E2>))
                           of (<C1>, <C2>) => <mp C1 C2>)
     | W <x#> => W <x>;
```

Fig. 4. Translator (Part 4/4): Main Conversion

`x`. Therefore, before recursing, we create a fresh combinator `u` and extend `W`, via the `with` keyword, to map `x` to `u`. The `with` construct is syntactic sugar used to extend parameter functions; its desugared version is described in [6]. The expression "`{<x>}{<u>} convert (W with <x> => <u>) <E x>`" contains a recursive call in the presence of the new `x` and `u`. During the recursion, the `x` will be converted to `u` by the last case of `convert`. Because the `u` may occur in the result, we utilize higher-order matching by matching the result against "`{<x>}{<u>}<C u>`". The variable `C` is a LF function resulting from abstracting away all occurrences of `u` in the result. The call to bracket abstraction `ba` internalizes this abstraction in the combinator calculus. The `app` case recurses on both components and glues the results together with `mp`. Finally, the last case handles parameters by applying the parameter function `W`. Note that the pattern `<x#>` matches only parameters and the pattern variable `x` has type `(exp A)#`.

4 Reasoning

Since computation occurs with respect to a dynamic collection of parameters, determining if a list of cases is exhaustive is a non-trivial problem. If a list of cases is incomplete, Delphin will return a *Match Non-Exhaustive Warning* providing a list of cases which are missing. Additionally, if one tries to call a function which makes sense with respect to an incompatible collection of parameters, Delphin will return a *World Subsumption Error*. If no warning message is generated, then programs are guaranteed not to get stuck (i.e. type safety holds).

The problem of determining if a list of cases is exhaustive is also referred to as *coverage checking*. Earlier work [8] determines coverage of closed LF objects. This was extended for use in Twelf by supporting LF objects which are open with respect to a collection of blocks [7]. In Delphin, we support LF objects that are open with respect to a collection of parameters, and handle coverage on computation-level expressions in a similar way.

If a function passes the coverage checker and is terminating, then it is total and may also be interpreted as a proof. The termination checker for Delphin is currently only a prototype and supports lexicographic extensions of the subterm ordering over the inputs. The default termination order is lexicographic on the first input, followed by the second, and so on. However, we omit (1) implicit arguments

and (2) computation-level functions from the termination order. For example, the termination checker for our `convert` function just checks that the second explicit argument gets smaller.

Our converter (Figure 4) is proven total by Delphin, and hence we can view it as a proof that every simply-typed λ-term can be converted into a combinator of the appropriate type, or as an embedding of the natural deduction calculus into the Hilbert calculus. The reverse direction can also be proven in Delphin by coding a similar translation function.

5 Case Studies

Delphin has been used for numerous other examples. Here we just outline a few.

Hindley-Milner Type Inference. We have an extensive case study of Mini-ML. We have implemented the operational semantics, proved value soundness, and proved type preservation. The most interesting feature is a Hindley-Milner style type-inference algorithm. Parameters are used in place of references, where a new parameter can be thought of as a fresh memory location.

Logic. We have implemented and proved cut-elimination of the intuitionistic sequent calculus. In the framework of expressing morphisms between logics, we have expressed translations from HOL to Nuprl as well as translations between sequent and natural deduction calculi.

Church-Rosser. We have proved the Church-Rosser theorem for the untyped λ-calculus using the method of parallel reduction.

Isomorphism between HOAS and de Bruijn notation. A more advanced use of parameter functions occurs when we convert terms of the untyped λ-calculus encoded in HOAS to an encoding utilizing de Bruijn indices. In this example we use a parameter function to maintain a mapping of parameters to de Bruijn indices. However, we do more than just simply extend this mapping. When handling the λ case, we create a new parameter mapped to de Bruijn index 1 and *update* all other mappings to be offset by 1. Furthermore, we show that the translation between these two languages is an isomorphism. This example illustrates the expressivity of parameter functions, as we can use it to express complex statements about the underlying parameters (or context). For example, to prove the translation is an isomorphism, we used parameter functions to express the invariant that all parameters are mapped to different de Bruijn indices.

6 Related Work

Unlike Delphin, Twelf [5] cannot support parameter functions as its meta-logic is not higher-order (i.e. inputs and outputs cannot be meta-level functions). Instead, Twelf uses *blocks* to specify that certain parameters are introduced together. One can convert Twelf programs to Delphin, where blocks are replaced by parameter functions, whose extensions can be naturally and succinctly expressed using the `with` keyword.

```
Delphin, Release Version 1.5.0, April 20, 2008
D-- use "lfmtp08.d";
...
D-- val test1 = convert (fn .) <lam [u:exp b] lam [v:exp b] u> ;
val test1 : <comb (b ar b ar b)>
            = <mp (mp s (mp k k)) (mp (mp s k) k)>
D-- val test2 = convert (fn .)  <lam [x:exp b] x> ;
val test2 : <comb (b ar b)>
            = <mp (mp s k) k>
```

Fig. 5. Delphin Sample Execution

Delphin's type system guarantees that parameters cannot escape their scope. This is in contrast to nominal systems [1], such as FreshML, where the creation of parameters is a global effect and additional reasoning is necessary to ensure that parameters do not escape their scope [4]. Like our system, the Beluga [3] system supports HOAS and also guarantees that parameters do not escape their scope; but rather than providing a means to extend the signature, they pass around the entire context explicitly.

7 Environment

Delphin provides a top-level interactive loop where one may write, execute, and experiment with programs. Error and warning messages are reported in the same format as in SML of New Jersey, allowing one to use the SML Emacs mode to jump to error locations easily. One may load LF Signatures (Twelf files) by typing `sig use "filename"` and may run Delphin files by typing `use "filename"`. For illustrative and debugging purposes, Delphin provides the ability to pretty-print arbitrary Delphin expressions with options to make pattern variables and implicit arguments explicit. Additionally, Delphin allows the disabling/enabling of the coverage checker and the termination checker.

Figure 5 illustrates a use of the top-level in converting a couple of simply-typed λ-terms into combinators. The file `lfmtp08.d` is the code from our four previous figures. Recall that the first argument to `convert` is a parameter function. We supply it with an empty function "`fn .`", which is appropriate as no parameters exist at the top-level.

Delphin and its related publications are all available for download on our website – http://delphin.logosphere.org.

References

[1] Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects Computing*, 13(3-5):341–363, 2002.

[2] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[3] Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Principles of Programming Languages, POPL*, 2008.

[4] François Pottier. Static name control for FreshML. In *Twenty-Second Annual IEEE Symposium on Logic In Computer Science (LICS'07)*, Wroclaw, Poland, July 2007.

[5] Frank Pfenning and Carsten Schürmann. *Twelf User's Guide*, 1.2 edition, September 1998. Available as Technical Report CMU-CS-98-173, Carnegie Mellon University.

[6] Adam Poswolsky and Carsten Schürmann. Practical programming with higher-order encodings and dependent types. In *European Symposium on Programming (ESOP)*, 2008.

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

[8] Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In David Basin and Burkhart Wolff, editors, *Proccedings of Theorem Proving in Higher Order Logics (TPHOLs'03)*, volume LNCS-2758, Rome, Italy, 2003. Springer Verlag.

[9] Carsten Schürmann, Frank Pfenning, and Natarajan Shankar. Logosphere. A Formal Digital Library. Logosphere homepage: http://www.logosphere.org.