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

Adam Poswolsky\textsuperscript{1} and Carsten Schürmann\textsuperscript{2}

\textsuperscript{1} Yale University poswolsky@cs.yale.edu
\textsuperscript{2} IT University of Copenhagen carsten@itu.dk

Abstract. Delphin is a functional programming language [PS08] utilizing dependent higher-order datatypes. Delphin is a two level system, which cleanly separates data from computation. The data level is LF [HHP93], which allows for the specification of deductive systems following the judgments-as-types methodology utilizing higher-order abstract syntax (HOAS). The computation level facilitates the manipulation of such encodings by providing a newness constructor to create parameters (fresh constants) and the ability to write functions over parameters, which we also call parameter functions. A wealth of documentation and examples are available online at http://www.cs.yale.edu/~delphin.

1 The Delphin System

Delphin is a functional programming language built to facilitate the encoding, manipulation, and reasoning over dependent higher-order datatypes. Delphin is a two level system distinguishing between data and computation. The data level is the logical framework LF [HHP93], 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 data.

Delphin is first and foremost a general purpose programming language supporting complex data structures. However, it is also well-suited to be used in the setting of the Logosphere project [SPS], 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 via partial theory morphisms. Delphin has been successfully used in expressing translations from HOL, Nuprl, and various other logics.

Specification of Data. As the data level is LF, one may encode languages using HOAS as well as dependent types. The language being encoded is referred to as the object language. For example, an encoding of the untyped $\lambda$-calculus can represent its functions as LF functions. This means that variables of the object language are represented as LF variables and the application of object-level functions is just LF application. This deceptively simple idea allows for the encoding of complex data structures (object languages) without having to worry about
the representation of variables, renamings, or substitutions that are prevalent in logic derivations, typing derivations, operational semantics, and more.

The representation of deductive systems utilizes the judgments-as-types technique. This means that we represent a judgment as a type, and derivation trees correspond to objects of that type. Checking the correctness of a derivation is reduced to simply type-checking the LF object representing said derivation. Dependent types are necessary to encode non-trivial judgments. Besides utilizing HOAS to represent languages, it is also used in this paradigm to represent hypothetical judgments.

Computation. The operational semantics of Delphin is fairly straightforward. The key feature is that we distinguish between two methods of variable binding. The function type constructor \(\forall\) binds variables that are intended for instantiation, which means that computation is delayed until application. This gives us the typical function space that is expected in any programming language. Additionally, the newness type constructor \(\nu\) binds variables that will always remain uninstantiated and hence computation will not be delayed. The introduction form of \(\nu\) is the \(\nu\) (pronounced new) construct, \(\nu x. e\), where \(x\) can occur free in \(e\). Evaluation of \(e\) occurs while the binding \(x\) remains uninstantiated. Therefore, for the scope of \(e\), the variable \(x\) behaves as a fresh constructor, which we will henceforth call a parameter. One may view \(\nu\) as a method of dynamically extending the signature, where the signature refers to the collection of datatype constructors. A detailed explanation can be found in [PS08].

Delphin pervasively distinguishes between LF-types \(A\), parameter types \(A^\#\), and computation-level types \(\tau\). Note that the only terms of type \(A^\#\) are variables \(x\), which also have type \(A\). Thus, one may view the type \(A^\#\) as a subtype of \(A\).

The dynamic creation of parameters is used to compute under data level (LF) \(\lambda\)-binders. The introduction of parameters, via \(\nu x. e\), introduces the \(\nu\)-type, making it a local effect. This is in contrast to nominal approaches [GP02], such as FreshML, where the creation of parameters is a global effect and then additional reasoning is necessary to ensure parameters do not escape their scope [Pot07], which is instead an inherent property of our system. Additionally, other recent work [?] ensures parameters do not escape their scope by using an enriched data level with explicit substitutions and contexts.

The manipulation of LF objects is facilitated by case-analysis and recursion. Dynamically introduced parameters may be eliminated using higher-order matching. Finally, reasoning explicitly about parameters is also supported via parameter functions, i.e. computational functions whose domains range over parameters. For example, passing around a function \(f\) of type \(\forall x \in A^\#. B\) allows us to save a map between parameters of type \(A\) and objects of type \(B\). From a first-order perspective, \(f\) can be thought of as a substitution, but parameter functions can do much more. For example, the codomain of the function may be a pair, which comes up heavily in our example suite. This provides a natural way to reason about parameters without resorting to a first-order encoding, making substitutions explicit, or adding ad-hoc specifications. Unlike Delphin,
the Twelf [PS98] meta-theory cannot support such functions as it is not higher-order (i.e. inputs and outputs cannot be meta-level functions). Instead, Twelf has the user define a block specification to save this relationship which is naturally expressible using Delphin’s parameter functions.

Reasoning. As computation occurs with respect to a dynamic collection of parameters, determining if a list of cases is exhaustive is a non-trivial question. 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. progress holds).

The problem of determining if a list of cases is exhaustive is also referred to as coverage checking. Earlier work [SP03] determines coverage of LF objects in an empty context. This was extended for use in Twelf by supporting an open context of blocks [Sch00]. In Delphin, we support LF objects in any context and also handle computation-level higher-order functions, e.g. parameter functions.

Finally, 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 termination order is simply the order of all explicit arguments.

2 Case Studies

Delphin has been used for a number of experiments, which are all available on our website. Here we outline just a few.

Hindley-Milner Type Inference 3. We have an extensive case study of a subset of ML, called Mini-ML. We can implement the operational semantics, prove value soundness and prove type preservation. The most novel feature is a Hindley-Milner type-inference algorithm. This illustrates the power of reasoning about parameters. Parameters are used in place of references, where a new parameter can be thought of as a fresh memory location. We then pass around environments, which are implemented as parameter functions.

Logic. Other examples include the proof of the Church-Rosser theorem for the untyped λ-calculus under β-reduction using the method of parallel reduction 4. We have implemented cut-elimination of the intuitionistic sequent calculus 5. In the framework of expressing morphisms between logics, we have expressed

3 http://www.cs.yale.edu/~delphin/delphin-examples/mini-ml
4 http://www.cs.yale.edu/~delphin/delphin-examples/church-rosser
5 http://www.cs.yale.edu/~delphin/delphin-examples/cut-elim
translations from HOL to Nuprl\(^6\) as well as translations between sequent and natural deduction calculi.

All of these examples were based on similar proofs in Twelf, but the Twelf blocks are replaced with parameter functions. In all these examples, the parameter functions save mappings between parameters, which are extended with additional mappings when we introduce new parameters. As these extensions are prevalent in converting Twelf examples into Delphin, we provide syntactic sugar to allow the programmer to concisely express the extension. During type reconstruction this is converted into an appropriate function of the form found in our discussion of the underlying type-theory [PS08].

**Isomorphism between HOAS and deBruijn\(^7\).** A more advanced use of parameter functions occurs in our example converting between terms of the untyped \(\lambda\)-calculus encoded in HOAS to one utilizing deBruijn indices. In this example we use a parameter function to maintain a mapping of parameters to deBruijn indices. However, we do more than just simply extend this mapping. When handling the \(\lambda\) case, we create a new parameter mapped to deBruijn index 1 and update all other mappings to be offset by 1.

Furthermore, we use Delphin to 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 deBruijn indices.

### 3 Implementation

Delphin is implemented in Standard ML and has been compiled under SML of New Jersey and PolyML. In this section we will discuss the key implementation concerns involving the data level, type reconstruction, implicit arguments, operational semantics, and coverage checking.

**Data Level.** Recall that the programmer can take advantage of HOAS by representing object-language functions as LF functions. Here we will briefly discuss the implementation of LF which happens under-the-hood.

Delphin borrows some Twelf code to deal with LF-level unification and LF-level type reconstruction, but with extensions. The implementation of LF in Twelf exploits the fact that one cannot have nested functions, allowing one to assume the context is *empty*, an invariant that does not hold in Delphin.

The main property of LF is that all terms possess a unique canonical form modulo \(\alpha\beta\eta\). The implementation utilizes deBruijn notation to avoid dealing with \(\alpha\)-equivalence. Instead of constantly computing canonical forms, we use a

---

\(^6\) [http://www.cs.yale.edu/~delphin/delphin-examples/hol-nuprl](http://www.cs.yale.edu/~delphin/delphin-examples/hol-nuprl)

\(^7\) [http://www.cs.yale.edu/~delphin/delphin-examples/debruijn](http://www.cs.yale.edu/~delphin/delphin-examples/debruijn)
spine notation and weak-head normal forms [DHKP98]. This is a very useful optimization as it is wasteful to compute the entire canonical form only to discover that two terms differ in their top-level constructors.

Coverage analysis and type reconstruction make use of logic variables and unification. In the presence of logic variables, we must add some form of closures. Therefore, we utilize explicit substitutions and follow the unification algorithm in [DHKP98] which describes higher-order unification in this setting. Equations that fall within the fragment of higher-order patterns are solved immediately, while all other equations are postponed as constraints. Such constraints are reawakened when a variable in the postponed equation is instantiated. It is also important to note that the Delphin unification algorithm extends the LF version with support for variables of parameter type (i.e., variables that can only be instantiated with parameters) and computation-level types.

**Type Reconstruction.** The type reconstruction algorithm converts terms into the Delphin internal syntax [PS08]. Our algorithm will always report either a principal type, a type error, or in the case of leftover constraints, an indication that the type is ambiguous and needs further annotations. The result of this algorithm neither has free variables nor has logic variables, and thus is a valid Delphin expression. We have found that, in practice, pattern-variables never need to be explicitly quantified, as we also find in mainstream programming languages such as ML and Haskell. Reconstruction occurs in three stages:

1. The first stage is an approximate type reconstruction, which effectively throws out all dependent types and determines the simply-typed form of the type.
2. With this approximate information we create appropriate logic variables and utilize unification to infer the exact type (with dependencies).
3. The formal system (internal syntax) requires an explicit declaration of pattern-variables, but we allow the user to express patterns by just using free variables. Therefore, the final step finds all free variables and explicitly quantifies them to the respective pattern where the variable first occurs.

**Implicitness.** It is redundant to explicitly supply an input argument which is indexed in another input argument. Therefore, we extend the reconstruction algorithm above to allow free variables in types. These free variables are implicitly $\forall$-quantified. Similarly, we also implicitly quantify free variables in LF constants, just as Twelf does. When applying a function, the reconstruction algorithm will automatically fill in the implicit arguments. This support is just a frontend convenience and does not affect the underlying theory. Evaluation, coverage, and termination are unaware of what is implicit/explicit in the frontend.

Additionally, Delphin supports the syntax “_” for terms that are explicit but can be inferred. This syntax instructs the reconstructor to fill in the term automatically, which is possible when the type forces it to be a particular term.

**Operational Semantics.** The evaluator for Delphin expressions is fairly straightforward. If multiple branches match, it will pick the first branch that matches
and continue execution. Pattern variables are filled in with logic variables and we employ unification during runtime to instantiate such variables.

Coverage Checking. The problem of coverage checking refers to determining if a function’s list of cases is exhaustive. If a function has an exhaustive list of cases, then it will never get stuck, i.e. return Match Non-Exhaustive Failure during execution. Delphin allows different functions to make sense with respect to different collections of parameters, also called its world. A function’s world is declared as a set of types. The function can be executed in any extension of the context with parameters incorporating said types.

The flexibility of allowing functions in different worlds generates concerns when one function calls another. For example, assume function $f$ is covered for closed terms (no parameters allowed) and assume function $g$ is covered for terms allowing some parameters. In this scenario, function $f$ can use function $g$, but the reverse is not true. Therefore, after type reconstruction, Delphin employs the coverage checker which is also a world checker which makes sure that functions defined with respect to one set of parameters can only call functions which also handle at least the same set of parameters. More specifically, the programmer uses a params keyword to specify the intended world of a function, and determining if a function is accessible corresponds to determining if all fresh parameters (with respect to the context in which the function was defined) are compatible with its world. The extension to Delphin for specifying worlds has been proved correct and is also available on our webpage.

In contrast, Twelf specifies a context-schema stating explicitly the form of the entire context. However, this is only reasonable since Twelf only allows top-level functions and hence all functions are declared in the empty context. As Delphin is a higher-order language and hence supports the definition of nested functions, functions may be declared in different contexts. Therefore, the important characteristic is not the form of the entire context, but the extensions of parameters allowed.

4 Environment

Although Delphin is implemented in ML, it can be executed as a stand-alone program. The frontend provides a top-level similar to that of ML itself. Error and Warning messages are reported in the same form as in SML of New Jersey, allowing one to use SML Emacs scripts to jump to error-locations easily. The parser and lexer are implemented using ML-Yacc and ML-Lex, respectively.

The frontend supports loading LF signatures as well as loading Delphin programs. 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.

Just as in any other programming language, one may write, execute, and experiment with programs directly using the frontend. We provide a large set
of illustrative examples, to hopefully allow one to get quickly acquainted with Delphin. Delphin and its related publications are all available for download on our website – http://www.cs.yale.edu/~delphin.

References

[DHKP98] Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. Rapport de Recherche 3591, INRIA, December 1998. Preliminary version appeared at JICSLP’96.

[GP02] Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. Formal Aspects Computing, 13(3-5):341–363, 2002.

[HHP93] 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.

[Pot07] François Pottier. Static name control for FreshML. In Twenty-Second Annual IEEE Symposium on Logic In Computer Science (LICS’07), Wroclaw, Poland, July 2007.

[PS98] 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.

[PS08] Adam Poswolsky and Carsten Schürmann. Practical programming with higher-order encodings and dependent types. In European Symposium on Programming (ESOP), 2008.

[Sch00] Carsten Schürmann. Automating the Meta-Theory of Deductive Systems. PhD thesis, Carnegie Mellon University, 2000. CMU-CS-00-146.

[SP03] Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In David Basin and Burkhard Wolff, editors, Proceedings of Theorem Proving in Higher Order Logics (TPHOLs’03), volume LNCS-2758, Rome, Italy, 2003. Springer Verlag.

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