

## REALIZING THE DEPENDENTLY TYPED $\lambda$ -CALCULUS

ZACHARY SNOW<sup>1</sup>

<sup>1</sup> Department of Computer Science and Engineering  
University of Minnesota  
4-192 EE/CS Building  
200 Union Street SE  
Minneapolis, MN 55455  
*E-mail address:* `snow@cs.umn.edu`

---

**ABSTRACT.** Dependently typed  $\lambda$ -calculi such as the Edinburgh Logical Framework (LF) can encode relationships between terms in types and can naturally capture correspondences between formulas and their proofs. Such calculi can also be given a logic programming interpretation: the system is based on such an interpretation of LF. We have considered whether a conventional logic programming language can also provide the benefits of a Twelf-like system for encoding type and term dependencies through dependent typing, and whether it can do so in an efficient manner. In particular, we have developed a simple mapping from LF specifications to a set of formulas in the higher-order hereditary Harrop (*hohh*) language, that relates derivations and proof-search between the two frameworks. We have shown that this encoding can be improved by exploiting knowledge of the well-formedness of the original LF specifications to elide much redundant type-checking information. The resulting logic program has a structure that closely follows the original specification, thereby allowing LF specifications to be viewed as meta-programs that generate *hohh* programs. We have proven that this mapping is correct, and, using the Teyjus implementation of  $\lambda$ Prolog, we have shown that our translation provides an efficient means for executing LF specifications, complementing the ability the Twelf system provides for reasoning about them. In addition, the translation offers new avenues for reasoning about such specifications, via reasoning over the generated *hohh* programs.

### 1. Introduction and problem description

There is significant and growing interest in tools for specifying and reasoning about formal systems. These systems, such as programming languages and logics are typically defined in terms of a rules-based operational semantics. This leads to one obvious technique for specification: through the use of predicate logics, and languages like Prolog. In this setting we can encode expressions in the formal system as terms in the language, and use predicates to define the operational semantics. The systems that we might wish to specify can have a rich structure, for instance they may include a notion of binding or abstraction, and require operations like capture-avoiding substitutions and properties like  $\alpha$ -equivalence. Implementing these features anew, for each logic or language that one wishes to specify, is time consuming and error prone, and so might benefit from language integration. Therefore

---

*1998 ACM Subject Classification:* Languages, Theory.

*Key words and phrases:* logical frameworks, logic programming.

logics or languages that embody these notions are often preferred, and have been widely used in the specification, and particularly the implementation, of such systems [Wha05].

Moving in a different direction, we might think of encoding properties of terms, and relationships between terms, not through *explicit* predicate definitions, but instead *implicitly* through types. Dependent types, like those provided by the dependently typed  $\lambda$ -calculus, provide powerful and natural methods for expressing these kinds of constraints. Furthermore, analyzing such specification for properties of correctness can often be reduced to type checking. This approach, distinct from that of predicate encodings, has also found widespread adoption in specifying and implementing formal systems, as well [Nec97, Ler06].

But specification is only the first goal. Given a specification of a formal system, we can think of doing several things: we can reason over the specifications, and thereby prove various properties about the logic or language being specified. We can also *animate* the specifications: for instance, having specified a language and its operational semantics, we might execute the specification in order to evaluate programs written in that language. The latter possibility actually benefits from the former; whereas traditionally we might specify and reason in one language, and then implement in another, here we execute exactly the same program about which we have reasoned. This handily removes the question of whether the implementation matches the specification.

Therefore our focus has been on problems associated with specifying formal systems using dependently typed languages, and then efficiently animating these specifications. In particular, we have sought to leverage existing work in the realm of efficient implementations of predicate logics when doing so, by designing translations from dependently typed languages to predicate logics.

## 2. Background and overview of the existing literature

As we have described, we can think of using various languages for specifying systems. On the one hand, we have higher order predicate logics like *hohh*, a logic based on Horn clauses, in which terms are those of the  $\lambda$ -calculus, but with support for handling the binding structure inherent in such terms.  $\lambda$ Prolog [Nad88] is a higher order logic programming language based on *hohh*, and extended in various ways (for instance, with a module system that supports programming in the large, with *ad hoc* polymorphism, and with facilities for interacting with the outside world). Furthermore,  $\lambda$ Prolog admits an efficient compiled implementation, as realized by the Teyjus system [Gac08]. Finally, there has already been work in analyzing and reasoning over programs written in  $\lambda$ Prolog [Gac09b, Bae10a], and there exist tools [Gac09a, Bae10b] for reasoning over it, both interactively and automatically, as well.

On the other hand, we have logics and languages founded on the dependently typed  $\lambda$ -calculus, for instance the The Edinburgh Logical Framework (LF) [Har93]. Twelf [Pfe99] is an implementation of LF that allows for reasoning over such specifications, and animating them. In and of itself, LF is strictly a specification language; it has no operational semantics of its own. However, one can apply the Curry-Howard Isomorphism [How80] to realize a *logic programming interpretation* of LF. In this context one defines types that correspond to judgments; then searching for an inhabitant of such a type corresponds to searching for a proof of the given judgment. Constructors for the type play the role of inference rules for constructing derivations of the judgment. And the discovered inhabitant, called a *proof term*, is itself a proof of the relevant judgment.

Twelf animates specifications in an interpreted fashion. There has already been research into improving this implementation by way of optimization (*e.g.*, [Pie06, Pie03]), which have proved quite fruitful. In the end, however, the existing implementation of Twelf suffers due to its interpreted nature, and we find that it cannot be used on many realistically sized programs.

### 3. Goal of the research

The specific goal of my research as described herein has been to develop an efficient implementation of logic programming search for LF specifications, in particular through translation to  $\lambda$ Prolog, so that they may be executed using the Teyjus system. In addition, an important aspect of this work has been to ensure that this translation is *transparent*, so that the structure of the LF specification is clear from the structure of the generated logic program. This facilitates an understanding of the translation that allows the programmer to view LF specifications as a meta-programs, and enables reasoning over LF specifications using existing tools for reasoning over *hohh* programs.

### 4. Current status of the research

We have developed several translations from LF specifications into *hohh*. The problem of translating an LF specification into equivalent *hohh* has been investigated by Felty [Fel89, Fel90] — in this context, “equivalence” should be understood to mean the following: if an LF judgment has a derivation under a particular LF specification, then the translated judgment has a derivation under the translated specification in *hohh*. However this translation is not suitable for the purposes of logic programming, as it assumes that the proof term is already known, whereas when animating specifications this is exactly what is *not* known. Thus, taking inspiration from this translation we have developed our “simplified” translation that is suitable for logic programming.

Next we have improved this translation in two ways. First, the simplified translation is inefficient in that there are redundancies in proof search, that can be avoided through various observations about the nature of valid LF specifications. Indeed, this aspect of LF specifications, (that is, that they contain significant amounts of redundant typing information) has been investigated by, *e.g.*, Reed [Ree08], for the purpose of limiting the size of proof terms, which can become quite large. Addressing these redundancies is critical to the usefulness of the translation as an implementation mechanism for a separate reason: these redundancies can lead to inefficiencies, and even asymptotic changes in the complexity of algorithms implemented in specifications. Second, the simplified translation generates *hohh* logic programs that are relatively opaque, in the sense that it is not obvious that the logic program corresponds to the original specification. This is largely due to the fact that the simplified translation does not make much use of the rich type system afforded us by *hohh*.

We address these issues in a second, “optimized” translation. We first develop a technique for identifying and eliminating redundancies in proof search. And we improve the transparency of the translation by making a deeper use of the type system of *hohh*, to, for instance, reflect the non-dependent aspects of LF types as *hohh* types. Our final translation includes these optimizations, along with a few others, and results in a translation that is efficient and transparent. Because the generated  $\lambda$ Prolog programs share the same structure as the original LF specification we can view LF specifications as meta-programs. What’s

more, as we've proved that our various translations are equivalent to LF, it is possible to reason over the resulting logic programs in order to reach conclusions about the properties of the original specification. And finally, we've developed a system that implements the translation.

Our implementation, named Parinati [Sno10] and written in Objective Caml, is released under the GNU General Public License version 3. Given a valid LF specification written in the concrete syntax of Twelf, along with various types for which inhabitants should be sought, it generates a  $\lambda$ Prolog program that can be compiled and run using the Teyjus system.

## 5. Preliminary results accomplished

Preliminary experimental results comparing the efficiency of our implementation with that of Twelf are quite good: we have obtained an increase in efficiency of anywhere from 2 times to over 100 times in many cases. What's more, for sufficiently large problem sizes our implementation is almost always more efficient in terms of running time, apparently due to the extreme memory consumption that Twelf can exhibit — this is characteristic of certain kinds of interpreted implementations [Bri94] of logic programming search, including Twelf.

Beyond various performance metrics we have also demonstrated the transparency of the translation. In fact, our translation generates  $\lambda$ Prolog programs that almost exactly matches code that might be written “by hand”, and the underlying structure of the original LF specification is completely clear. As already described, this allows the programmer to view the LF specification as a kind of meta-program for generating  $\lambda$ Prolog, and furthermore allows for reasoning over the resulting program as a method for reasoning over the original LF specification. What's more, this transparency is not only enabling, it is also elucidating: the generated *hohh* program is easier to reason about because it highlights those types that could have logical importance, and elides those that do not.

## 6. Open issues and expected achievements

There are a number of possible directions to take this work. First, there are still some examples in which our implementation is only as efficient, or even less efficient, than that of Twelf. We have begun a series of experiments to determine what factors are causing this slowdown, which we believe to be due to differences in the treatment of occurs checking between the two systems. Next, the efficiency of the implementation depends on our ability to accurately identify and eliminate redundancies. Any improvements we might make to this identification process should lead to performance increases.

Much of our work has been on optimizing our translation to  $\lambda$ Prolog; however, a different approach is to compile directly to, for instance, the Teyjus virtual machine's instruction set. By employing such an approach we might avoid some of the thorny questions associated with redundancy elimination. More generally, direct compilation could allow us to regain opportunities for those improvements that might be lost by translating first to  $\lambda$ Prolog and then relying on its implementation that is not specially optimized to treat LF-specific programs. However, this would clearly eliminate the possibility of treating LF as a meta-programming language for writing complex  $\lambda$ Prolog programs, as the requirement of transparency could not be fulfilled.

Twelf has several extensions aimed at the practicalities of programming. One particularly useful extension is the ability to use metavariables in the type for which an inhabitant is to be sought; these are instantiated during search. While the translation we have described includes this extension, we have not yet fully understood the theoretical aspects of it in terms of correctness of the translated programs.

Finally, we have only begun to understand how our translation fares when the purpose is to reason over an LF specification by analyzing the resulting *hohh* program. In the future we could apply existing tools to both LF specifications and their *hohh* counterparts generated by the translation, to judge the relative merits of reasoning in either system.

## Acknowledgements

This work has been supported by the NSF grants CCR-0429572 and CCF-0917140. Opinions, findings, and conclusions or recommendations expressed in this papers are those of the authors and do not necessarily reflect the views of the National Science Foundation.

## References

- [Bae10a] David Baelde, Dale Miller, and Zach Snow. Focused inductive theorem proving, 2010. Accepted for publication at IJCAR'10.
- [Bae10b] David Baelde, Zach Snow, and Alexandre Viel. The Tac system, 2010.
- [Bri94] Pascal Brisset and Olivier Ridoux. The architecture of an implementation of lambda-prolog: Prolog/mali. In *ILPS Workshop: Implementation Techniques for Logic Programming Languages*. 1994.
- [Fel89] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language*. Ph.D. thesis, University of Pennsylvania, 1989.
- [Fel90] Amy Felty and Dale Miller. Encoding a dependent-type  $\lambda$ -calculus in a logic programming language. In Mark Stickel (ed.), *Proceedings of the 1990 Conference on Automated Deduction, LNAI*, vol. 449, pp. 221–235. Springer, 1990.
- [Gac08] Andrew Gacek, Steven Holte, Gopalan Nadathur, Xiaochu Qi, and Zach Snow. The Teyjus system – version 2, 2008. Available from <http://teyjus.cs.umn.edu/>.
- [Gac09a] Andrew Gacek. Abella, 2009. Available from <http://abella.cs.umn.edu/>.
- [Gac09b] Andrew Gacek. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems*. Ph.D. thesis, University of Minnesota, 2009.
- [Har93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.
- [How80] William A. Howard. The formulae-as-type notion of construction, 1969. In J. P. Seldin and R. Hindley (eds.), *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pp. 479–490. Academic Press, New York, 1980.
- [Ler06] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones (eds.), *POPL*, pp. 42–54. ACM, 2006.
- [Nad88] Gopalan Nadathur and Dale Miller. An Overview of  $\lambda$ Prolog. In *Fifth International Logic Programming Conference*, pp. 810–827. MIT Press, Seattle, 1988.  
URL <http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/iclp88.pdf>
- [Nec97] George C. Necula. Proof-carrying code. In *Conference Record of the 24th Symposium on Principles of Programming Languages 97*, pp. 106–119. ACM Press, Paris, France, 1997.
- [Pfe99] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger (ed.), *16th Conference on Automated Deduction (CADE)*, no. 1632 in LNAI, pp. 202–206. Springer, Trento, 1999.
- [Pie03] Brigitte Pientka and Frank Pfenning. Optimizing higher-order pattern unification. In *19th International Conference on Automated Deduction*, pp. 473–487. Springer-Verlag, 2003.

- [Pie06] Brigitte Pientka. Eliminating redundancy in higher-order unification: A lightweight approach. In Ulrich Furbach and Natarajan Shankar (eds.), *IJCAR, Lecture Notes in Computer Science*, vol. 4130, pp. 362–376. Springer, 2006.
- [Ree08] Jason Reed. Redundancy elimination for LF. *Electron. Notes Theor. Comput. Sci.*, 199:89–106, 2008. doi:<http://dx.doi.org/10.1016/j.entcs.2007.11.014>.
- [Sno10] Zach Snow. Parinati, 2010. Available from <http://www.cs.umn.edu/~snow/parinati>.
- [Wha05] Michael William Whalen. *Trustworthy translation for the requirements state machine language without events*. Ph.D. thesis, Minneapolis, MN, USA, 2005. Adviser-Heimdahl, Mats Per.