Bringing Higher-Order Abstract Syntax to Programmers

Research Contribution Outline

Francisco Ferreira

April 16, 2013

1 Problem and Motivation

A key aspect when implementing compilers, code generators, interpreters, type inference engines, or theorem provers is the representation of data structures with names and binders present in most if not all programming languages. Historically, binders in data-structures such as abstract syntax trees have been represented in a variety of ways, mostly governed by software engineering concerns: whether the representation is efficient, reasonably simple to understand, and sufficiently easy to manipulate without introducing bugs by accident. Depending on the particular domain, different representations were used: e.g., de Bruijn indices when fast α-equivalence checking is important or unique small integers when the flexibility of moving code freely is a key concern.

Besides supporting clear and easy to maintain code, the growing interest in formal methods has added a new concern: how can we establish and track properties of binders and the code that manipulates them without obfuscating the development.

To this end, several systems have been developed with specialized support for binders, either via libraries or as first-class notions, to try and encapsulate as much as possible the common functionality of binders and prove their properties ideally once and for all, or at least more easily. Currently, there are two main theoretical approaches: supporting *nominals*, that is *names* as first-class notions, in the language ([Pitts, 2003], [Pouillard and Pottier, 2010]) and Higher-Order Abstract Syntax (HOAS) [Pfenning and Elliott, 1988], which represents binders in the object level by re-using binders of the meta-language, implemented by systems like Twelf [Pfenning and Schürmann, 1999], Delphin[Poswolsky and Schürmann, 2009], and Beluga [Pientka and Dunfield, 2010].

Over the past decade, we have made substantial progress in using sophisticated binder support and manipulating HOAS representations to effectively model proofs. However, in programming languages such support for binders is at best spotty. In the case of languages and libraries, techniques such as FreshML [Shinwell et al., 2003], folds or catamorphisms [Washburn and Weirich, 2008], and Hobbits [Westbrook et al., 2011] all

address the problem, but they either lack the formal guarantees, are complicated and inconvenient to use, or do not scale to dependent types.

Beluga [Pientka, 2008, Pientka and Dunfield, 2010] is a novel language specifically designed to manipulate data-structures with binders, providing all the usual operations we know and love, such as seamless α-renaming and capture-avoiding substitution, cleanly integrating them with dependent types. It starts with the logical framework LF [Harper et al., 1993] which allows for HOAS representations and from where it gets its high-level and clean semantics. LF objects together with their surrounding context in which they are meaningful are then embedded into computations thereby lifting the usual limitations of HOAS which typically prevent us from accessing and comparing variables. We call LF objects with their surrounding context *contextual objects* [Nanevski et al., 2008]. On the level of computations, Beluga provides recursion and a powerful pattern matching construct to analyze contextual objects. This makes it feasible to elegantly write code transformations such as for example closure conversion and hoisting.

While the theory behind Beluga is well developed, it remains unclear how to implement it efficiently and make its technology available in a realistic programming language. The current Beluga implementation uses an interpreter that largely follows the theoretical presentation. This allows us to explore the power of the language in small scale examples. In order to make it into a practical programming language however, it is important to be able to compile it into efficient code. Efficiency of the compiled code depends mostly on two different aspects:

- How to represent binders and contexts: Experience in compilers indicates that the best representation to use can depend on the particular binders. For example, SML/NJ's [Shao, 1997] internal representation uses names for variables bound in the computation code, but de Bruijn indices for variables bound within types.

- How to implement pattern matching: Beluga's patterns are powerful, and we use a specialization of higher-order pattern unification [Dowek et al., 1996, Abel and Pientka, 2011], a decidable fragment of higher-order unification, for pattern matching. But of course, we do not want to compile Beluga's to a sequence of calls to a generic higher-order pattern unification function.

2 Background and Related Work

Pouillard and Pottier [2010] present a high-level, abstract representation for names and contexts and then show how this abstract interface can simply be instantiate with a choice of various concrete representations. We reuse a lot of their work but start from an even higher-level representation.

Urban [2008] presents an Isabelle package that lets the user write proofs in terms of an abstract notion of names, which are under the hood represented by plain first order data. Contrary to our work he does not worry about efficiency and instead focuses on proving that the lower-level representation actually provides the expected semantic properties.

Along the same lines, Felty and Momigliano [2012] provide a package that lets users use HOAS in their specifications and proofs, and where the properties of those binders are proved by relating them to some lower-level first-order representation. But here as well, the focus is on writing proofs rather than programs, so efficiency of the code manipulated is a secondary concern.

Westbrook et al. [2011], Washburn and Weirich [2008], Shinwell et al. [2003] present various ways to provide support for programming with data-structures that contain binders within existing languages making them more practical than our work, but at some significant costs: they do not provide capture-avoiding substitution and do not support dependent types, for example. As the interface and the implementation are tightly integrated, it is more difficult to change the implementation strategy.

Chlipala [2008] shows a similar, though more light-weight, effort but within the context of a proof assistant. It can support dependent types, but the concrete representation of binders cannot be changed.

3 Approach and Uniqueness

In this research, we study how to compile contextual objects and contexts. The compilation is split into three parts:

1. A translation of code into an internal first-order representation that we call *Fresh-Style* , inspired by [Pouillard and Pottier, 2010]. The Fresh-Style representation abstracts over the actual final representation and provides an abstract interface from which we can derive a concrete representation using names or de Bruijn indices. This constitutes our intermediate representation. Our translation between contextual objects and their corresponding Fresh-Style representation connects for the first time formally the gap between programming using the nominal approach and programming using contextual objects.

2. We describe pattern-matching compilation that turns the statements in Beluga programs into a decision tree as is usually done with ML patterns. In particular, we extend ML-style pattern matching compilation to the more general case of Beluga's patterns, that can match on contexts, open terms, and λ-abstractions.

3. We show how to instantiate the abstract Fresh-Style representation to obtain a concrete representation of the binders using names and de Bruijn indices.

Our compiler framework is implemented in OCaml. The internal representation of the program in the prototype makes it easy to target a generic simply typed functional language. At this point, the prototype generates JavaScript code. For more details, consider the the joint publication with my advisors at the Programming Languages meet Program Verification workshop [Ferreira et al., 2013].

4 Results and Contributions

As already mentioned, this resulted, so far, in a joint publication with my advisors Stefan Monnier and Brigitte Pientka, presented earlier this year at the Programming Languages meet Program Verification workshop [Ferreira et al., 2013].

We have developed a generic framework for compiling contextual objects which can be instantiated to yield compiled code using names and code using de Bruijn indices. This opens up the possibility to choose the best representation depending on how binders are used. A key aspect of our framework is the pattern matching compilation for contextual objects which allows matching under the λ-binder. Our framework also provides for the first time a connection between programming with HOAS as done in Beluga and programming with nominal techniques. We believe our work is the first step in adding HOAS and contextual objects to existing programming languages and making this sophisticated technology available to the ordinary programmer.

In the future, we plan to prove that our compilation scheme is faithful to the theory, so that the meta-theoretical results developed for the theoretical foundation of Beluga carry over to the code generated by our compiler. Furthermore, we plan to explore the relation between the splitting trees used in pattern matching and those generated during coverage checking [Dunfield and Pientka, 2009, Schürmann and Pfenning, 2003].

We also plan to explore alternative compilation schemes such as targeting a functional language that supports pattern matching (e.g. OCaml). This can be done by translating contextual objects to linear higher-order patterns [Pientka and Pfenning, 2003] and reusing the pattern matching algorithm of the target language together with guards enforcing higher-order constraints. This would decouple the aspects of compiling patterns with binders and efficiently compiling simple patterns.

Finally, we aim to improve the concrete low-level representations with the optimizations used in typical hand-implementations, such as explicit substitutions [Nadathur and Qi, 2003] or hash-consing [Shao et al., 1998]. Many optimizations can also be done in the Fresh-Style representation, which makes them available to all the back-ends. Last but not least, we plan to investigate how to combine and choose various concrete representations, such that some binders can use one representations while others can use another depending on which operations are most often used on them.

5 Bibliography

References

Andreas Abel and Brigitte Pientka. Higher-order dynamic pattern unification for dependent types and records. In Luke Ong, editor, *10th International Conference on Typed Lambda Calculi and Applications (TLCA'11)*, Lecture Notes in Computer Science (LNCS 6690), pages 10–26. Springer, 2011.

Adam J. Chlipala. Parametric higher-order abstract syntax for mechanized semantics.

In James Hook and Peter Thiemann, editors, *13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 143–156. ACM, 2008.

Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Joint International Conference and Symposium on Logic Programming*, pages 259–273. MIT Press, September 1996.

Joshua Dunfield and Brigitte Pientka. Case analysis of higher-order data. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08)*, volume 228 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 69–84. Elsevier, June 2009.

Amy P. Felty and Alberto Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *Journal of Automated Reasoning*, 48(1): 43–105, 2012.

Francisco Ferreira, Stefan Monnier, and Brigitte Pientka. Compiling contextual objects: Bringing higher-order abstract syntax to programmers. In *7th Workshop on Programming Languages meets Program Verification (PLPV'13)*, pages 13–24. ACM, 2013. ISBN 978-1-4503-1860-0.

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

Gopalan Nadathur and Xiaochu Qi. Explicit substitutions in the reduction of lambda terms. In *5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 195–206. ACM, 2003.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):1–49, 2008.

Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Symposium on Language Design and Implementation (PLDI'88)*, pages 199–208, June 1988.

Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *16th International Conference on Automated Deduction (CADE-16)*, Lecture Notes in Artificial Intelligence (LNAI 1632), pages 202–206. Springer, 1999.

Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 371–382. ACM Press, 2008.

Brigitte Pientka and Joshua Dunfield. Beluga: a framework for programming and reasoning with deductive systems (System Description). In Jürgen Giesl and Reiner Haehnle,

editors, *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI 6173), pages 15–21. Springer-Verlag, 2010.

Brigitte Pientka and Frank Pfenning. Optimizing higher-order pattern unification. In F. Baader, editor, *19th International Conference on Automated Deduction (CADE-19)*, Lecture Notes in Artificial Intelligence (LNAI) 2741, pages 473–487. Springer-Verlag, 2003.

Andrew Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165–193, November 2003. ISSN 0890-5401.

Adam Poswolsky and Carsten Schürmann. System description: Delphin—a functional programming language for deductive systems. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08)*, volume 228 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 135–141. Elsevier, 2009.

Nicolas Pouillard and François Pottier. A fresh look at programming with names and binders. In *15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010)*, pages 217–228, 2010.

Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'03)*, pages 120–135. Springer, 2003.

Z. Shao. An overview of the FLINT/ML compiler. In *1997 Workshop on Types in Compilation*, 1997.

Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *International Conference on Functional Programming*, pages 313–323. ACM Press, September 1998.

Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. FreshML: programming with binders made simple. In *8th International Conference on Functional Programming (ICFP'03)*, pages 263–274. ACM Press, 2003.

Christian Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.

Geoff Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *Journal of Functional Programming*, 18(01):87–140, 2008.

Edwin Westbrook, Nicolas Frisby, and Paul Brauner. Hobbits for Haskell: a library for higher-order encodings in functional programming languages. In *4th ACM Symposium on Haskell (Haskell'11)*, pages 35–46. ACM, 2011.