Programming type safe transformations in Beluga

Olivier Savary-Belanger
McGill University
osavary@cs.mcgill.ca

Mathieu Boespflug
McGill University
mboes@cs.mcgill.ca

Stefan Monnier
Université de Montréal
stefan@iro.umontreal.ca

Brigitte Pientka
McGill University
bpientka@cs.mcgill.ca

1. Problem and Motivation

Compilers are elaborate programs, often spanning thousands of lines of code, making it likely for an error to creep its way into the implementation. Moreover, bugs in compilers can have catastrophic consequences, as they can lead to the compiler generating erroneous programs from perfectly fine source code.

Code transformations are essential to optimizing compilers and code analysis tools. They translate away features of the source language, making subsequent analysis of the code easier.

One such feature, which closure conversion [Appel 1992] eliminates, is the ability to write functions that take higher order structures as arguments. Another, which translation to continuation passing style [Plotkin 1975] removes, is the ability to write a non-tail call to a function.

The usefulness of these transformations crucially hinges upon their correctness, which is that they preserve the semantics of the source program. Although it is possible to prove correctness separately, such developments are lengthy and complex. One of the best example of this is the *CompCert* project [Leroy 2009], a certified compiler for a growing subset of the C language. Another approach to safe compilers is to preserve the type of programs throughout the compilation pipeline, taken as an approximation to correctness. Verifying that the program transformation are type-preserving is simply accomplished by type checking them. This approach is lightweight, and scales gracefully to modifications and extensions to the transformations. Type-preserving code transformations have been described numerous times in the literature [Fischer 1972; Minamide et al. 1996; Steele 1978], but have only recently been fully implemented [Chen and Xi 2003; Chlipala 2008; Guillemette and Monnier 2008; Linger and Sheard 2004], thanks to improved programming environments with richer type systems.

Our development consists of a full compiler front-end over the simply typed lambda calculus, implemented in BELUGA that is statically guaranteed to preserve the types of its input, including transformation to continuation-passing style, closure conversion and hoisting. We believe our compilation pipeline to be the first to be implemented over higher-order abstract syntax (HOAS) without having to devolve, even locally, to a first-order representation.

2. Background and Related Work

2.1 Beluga

Beluga [Cave and Pientka 2012; Pientka 2008; Pientka and Dunfield 2010] is a dependently-typed proof and programming environment. Beluga uses a two level approach, where data is implemented in the logical framework LF [Harper et al. 1993], thereby providing support for higher-order abstract syntax (HOAS).

HOAS is a representation technique where binding constructs from object languages are represented as functions of the meta-language, LF. For example, the term λ`x.x y` would be represented as `lam` λ`x.app x y`. Substitution is obtained for free, reducing to function application. This contrasts with first-order encodings such as de Bruijn indices or nominal [Gabbay and Pitts 1999], where the programmer has to provide his own implementation of capture-avoiding substitution.

```
datatype  source: tp → type  =
| app : source (arr A B) → source A → source B
| lam : (source A → source B) → source (arr A B)
| ...
;
```

At the computational level, we program using pattern-matching on contextual objects in dependently typed recursive functions. A contextual object M of type $A[\Psi]$ [Nanevski et al. 2008], which is open LF term M which has type A within a context Ψ. Its type precisely characterizes what variables are allowed to occur free in M. Pattern-matching on a contextual object refines our knowledge of both the form of its context and its type, allowing us to statically reason about well-scoped objects.

2.2 Code Transformations

2.2.1 CPS

Continuation passing style [Plotkin 1975] is a code transformation after which every function call appears in tail position. To do so, we modify every function so that it takes in a continuation, and it will call a continuation its result, which proceed with the rest of the computation. The type of the resulting function enforces that every call will appear in tail position.

Continuation passing style is a simple transformation, but applying it by itself creates a number of redexes which bloats the resulting term. Danvy and Filinski [1992] present a single pass CPS algorithm performing the reduction of those redexes at the moment of the transformation.

2.2.2 Closure Conversion

Closure conversion is a code transformation that results in a program whose functions are closed. Variables bound outside of a function's body are replaced by projections of an environment. The resulting function λ`(env,x).M` is paired with an environment tuple `L` filled with variables in a closure $\langle$ λ`(env,x).M , L` $\rangle$. Applications break apart these closures and apply the tuple, in addition to the original argument, to the function.

A closure conversion algorithm as we just described would however be space inefficient, as the closure would contain all the variables in the scope, instead of solely those occurring in the term. A more realistic algorithm, such as the one Guillemette and Monnier [2008] implement and ours, includes computing the variables occurring free in each functions, and closing the functions over these.

Under our algorithm, the program:

```
((λx.λy.x+y) 5) 2
```

is translated as:

```
open ⟨f1, c1⟩ =
  open ⟨f2, c2⟩ =
    ⟨ λ(e2, x).
        ⟨ λ(e1, y). π1e1+y,(x,·) ⟩
      ,(·) ⟩
  in f2 (c2, 5)
in f1 (c1, 2)
```

Which would, when evaluated, reduce to:

```
open ⟨f1, c1⟩ =
  ⟨ λ(e1, y). π1e1+y,(5,·) ⟩
in f1 (c1, 2)
```

and then further to:

```
(λ(e1, y). π1e1+y) ((5,·), 2)
```

Closure conversion is notoriously difficult using higher-order abstract syntax, due to its reliance on variable identity, which is harder to establish when reusing the binding construct of the host language. Previous implementations [Chlipala 2008; Guillemette and Monnier 2008] reverted to a first order encoding for this transformation even after using a higher order one for previous phases.

2.2.3 Hoisting

Hoisting is a code transformation that lifts the lambda-abstractions, closed by closure conversion, to the top level of the program. The function declaration is replaced in the body of the program by a reference to a global function environment.

Under our algorithm, the closure-converted program presented in Section 2.2.2 would be translated as:

```
let l =
    (λl1.λ(c1, x).⟨(π1 l1) (π2 l1),(x,·)⟩,
     λl2.λ(c2, y).π1 c2 + y, ·)
in open ⟨f1, c1⟩ =
   in open ⟨f2, c2⟩ =
        ⟨(π1 l) (π2 l), (·)⟩
   in f2 (c2, 5)
in f1 (c1, 2)
```

3. Approach and Uniqueness

The source language of our compilation pipeline is the simply typed lambda calculus extended with **let** construct, pairs and natural numbers. We use an intrinsically well-typed representation for terms, indexing them by their type. Type-preservation is enforced by having the type index of the source term and of the result of the transformation be the same.

3.1 Continuation-passing style

Our continuation-passing style transformation is adapted from the algorithm presented in Danvy and Filinski [1992].

It consists of a single downward pass on the input program, reducing administrative redexes on the fly.

```
schema ctx = some [a:tp]
             block x:source a, _t:target a ;

rec cpse :(g:ctx) [g. source A]
          → [g, c: target A → contra. contra]
```

Our context consists of pairs of source and target variables of the same type. We replace one with the other when reaching a variable in the `cps` conversion. Using this context invariant, both the source term and the target can be represented in the same context. As this transformation maps variables from the source language directly to a variable of the target language, we prefer this joined context to a relational approach, that will be used for closure-conversion and hoisting. The alternative would be to include as argument a datatype relating a context containing `values` to a context containing `targets` of the same types.

Our continuation-passing style transformation in Beluga is, at around 100 lines of code, similar to previous developments in other programming environments, such as in Twelf [Pfenning and Schürmann 1999]. It has also been verified to be covering, which means that all of our sets of patterns are exhaustive.

3.2 Closure Conversion

Previous implementations of closure conversion in similar settings, including Guillemette and Monnier [2007], resort to computing the free-variables of a term as a list, and to using this list to create a map from the variable to its projection when present in the list, and to $\perp$ otherwise. Using the support for pattern-matching on context in Beluga, we can find the minimal context for our function by strengthening the term by every variable not present in it. The map between the strengthened context and the desired minimal environment tuple can then be represented as `Map [h] [g]`, a substitution-like datatype.

```
schema ctx = source T;
schema tctx = target T;

rec cc : [. source T] → [. target T]

rec cc': Map [h] [g] → [g. source T]
          → [h. target T]
```

Our closure conversion algorithm consists of a downward pass on the input program, maximally strengthening the context of its lambda-abstractions, reifying their strengthened contexts, creating substitutions from those to projections of an environment variable, and applying them *under the binders* while modifying both their definitions and call sites in a type-preserving manner.

```
datatype Map:{h:tctx}{g:ctx} ctype =
| Id:{h:tctx} Map [h] []
| Dot: Sub [h] [g] → [h. target S]
       → Map [h] [g,x:source S]
;
```

`Map` is a mapping which, for every variable of type `S` in the context `g`, provides a target term of the same type. If our source and target language were to coincide, this would be the exact definition of a substitution, bringing a term from context `g` to context `h`. As the function space of the `source` language corresponds to a closure in the `target` language by our type translation, our `Map` can only be applied to a general `source` term while closure-converting it, in `cc'`.

Our implementation is much more succinct than comparable implementations [Chlipala 2008; Guillemette and Monnier 2007], at a third of their size in terms of lines of code. It has also been verified to be covering.

3.3 Hoisting

When hoisting all functions from a program, each function will depend on functions nested in them. One way of performing hoisting

(see Guillemette and Monnier [2008]) consists of binding the functions at the top level individually, in reverse order. In our setting, due to limitations of the notion of context provided, this would require a lot of machinery to separate variables which represent functions bound at the top-level from those which do not. We instead merge all the functions in a single tuple, abstracting the function environment from collected functions.

An important operation in our hoisting algorithm is the merging of the function environments together. When hoisting out terms with more than one subterm, each recursive call on those subterms results in a different function environment, which needs to be merged together before combining the subterms back.

```
datatype   concatTp: prodtp→prodtp→prodtp→type =
| cNil : concatTp unit S S
| cCons  : concatTp S1 S2 S3
         → concatTp (cross S S1) S2 (cross S S3)
;

rec  appendList: [. target (prod S1)]
                 → [. target (prod S2)]
                 → [. concatTp S1 S2 S3]
                 → [. target (prod S3)]
```

To do this, we compute the type resulting of appending the tuples of collected functions, using the relation `concatTp`. We then append the tuples together with `appendList`. The type of `appendList` enforces that the resulting tuple is of the right type. We enforce that terms given to `appendList` are tuples and not general terms by requiring their types to be product types, `prodtp`, which are separated from general types and are inhabited solely by tuples of terms.

```
rec  weakenLbyS1:(h:tctx) [. concatListTp S1 S2 S3]
               → [h, l:target (prod S2). target T]
               → [h, l:target (prod S3). target T]
```

Finally, we weaken, for each subterm, the type of their function environment variables to the merged one. We do this by replacing `l` by `tsnd l`, and this for each occurrence of `l` in the input term. A similar function is tailored for appending to the back of the current function environment, intuitively rebasing the type of each projection. By reusing the same `concatTp` relation on `weaken` and `appendList` calls, we know the type of the tuple consisting of the functions hoisted out coincides with the tuple abstracted in the program.

```
rec  hoist: [. source T] → [. target T]

datatype   HoistReturn:{h:tctx} [. tp] → ctype =
| HRet : [h,l:target (prod S). target T]
          → [. target (prod S)]
          → HoistReturn [h] [. T]
;

rec  hoist': Map [h] [g] → [g. source T]
             → HoistReturn [h] [.S]
```

Our implementation of hoisting is performed in the same pass as closure conversion as we can not express in *LF* that a term is closed, where the contextual object returned by the closure-conversion and hoisting on the lambda-abstraction case does assert closedness, which is required for the well-scopedness of hoisting.

3.4 Previous Implementations

Guillemette and Monnier [2007, 2008] implement a CPS and closure conversion for System F using both a higher-order and first order encoding of binders, taking an extension of Haskell supported by GHC as the host language. This extension relies on *generalized algebraic data types (GADTs)* [Xi et al. 2003] and *type families* [Schrijvers et al. 2008] instead of full-blown dependent types. Another limitation of their programming environment is its weak support for higher-order abstract encoding of binders, which they are not able to sustain throughout the compiler, reverting to a first-order encoding after CPS. For closure-conversion, the author creates a partial map of variables, which means that in the event where free-variables of a term are computed erroneously, the result of closure-conversion would contain $\perp$ in place of the missing variable. Their datatype syntactically enforces that functions are closed after closure-conversion, making it possible for them to perform hoisting safely in a separate phase.

Chlipala [2008] presents an implementation of the same pipeline over a simply typed source language in Coq. The author also presents CPS over a polymorphic language, and mechanically proves the semantically correctness of the transformations. As with Guillemette and Monnier [2008], higher-order encodings prove to be challenging for closure conversion, and the author reverts to a first order encoding for this phase. Hoisting is performed during closure-conversion, for the same reasons as ours.

4. Results and Contributions

We present a full compiler front-end implemented in Beluga that is statically guaranteed to preserve the types of its input, including transformation to continuation-passing style, closure conversion and hoisting. We believe our closure conversion to be the first to be implemented over true higher-order abstract syntax (HOAS) without having to devolve, even locally, to a first-order representation. The scope safety and type safety of this transformation is moreover statically guaranteed, and we achieve these safety guarantees with nearly no proof witnesses — making ours a *programmed* safe implementation with little to no proof work. Compared to previous attempts, our compilation pipeline is concise and arguably more readable, with explicit contexts giving more insight into the work of the transformations.

Security of compilers is of extreme importance. Type-preservation is a method bringing verified programs closer to the general programmer; while they may not be familiar with the tools needed to prove semantics correctness on the side, or do not have time to do so, they should readily adopt a method internal to their programming environment, whose added cost is minimal. We believe that Beluga in its current state is close to providing enough support for dependently-typed development to have the additional conceptual cost inherent to certified programming countervailed by type-annotation driven development and shorter testing phase.

References

A. W. Appel. *Compiling with continuations*. Cambridge University Press, New York, NY, USA, 1992. ISBN 0-521-41695-7.

A. Cave and B. Pientka. Programming with binders and indexed data-types. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 413–424. ACM Press, 2012.

C. Chen and H. Xi. Implementing typeful program transformations. In *Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, PEPM '03, pages 20–28, New York, NY, USA, 2003. ACM. ISBN 1-58113-667-6.

A. J. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In J. Hook and P. Thiemann, editors, *13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 143–156. ACM, 2008.

O. Danvy and A. Filinski. Representing control: a study of the cps transformation, 1992.

M. J. Fischer. Lambda calculus schemata. In *Proceedings of ACM conference on Proving assertions about programs*, pages 104–109, New York, NY, USA, 1972. ACM.

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. IEEE Computer Society Press, 1999.

L.-J. Guillemette and S. Monnier. A type-preserving closure conversion in Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, Haskell '07, pages 83–92, 2007.

L.-J. Guillemette and S. Monnier. A type-preserving compiler in haskell. In J. Hook and P. Thiemann, editors, *13th ACM SIGPLAN international conference on Functional programming (ICFP'08)*, pages 75–86. ACM, 2008.

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

X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52 (7):107–115, July 2009. ISSN 0001-0782.

N. Linger and T. Sheard. Programming with static invariants in omega. Unpublished, 2004.

Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *In 23rd ACM Symposium on Principles of Programming Languages*, pages 271–283. ACM Press, 1996.

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

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

B. 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.

B. Pientka and J. Dunfield. Beluga: a framework for programming and reasoning with deductive systems (System Description). In J. Giesl and R. Haehnle, editors, *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI 6173), pages 15–21, 2010.

G. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 51–62, New York, NY, USA, 2008. ACM.

G. L. Steele, Jr. Rabbit: A compiler for scheme. Master's thesis, Massachusetts Institute of Technology, 1978.

H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*, pages 224–235. ACM Press, 2003.