Programming type-safe transformations using higher-order abstract syntax

Olivier Savary-Belanger[1], Stefan Monnier[2], and Brigitte Pientka[1]

[1] McGill University
[2] Université de Montréal

Abstract. Compiling syntax to native code requires complex code transformations which rearrange the abstract syntax tree. This can be particularly challenging for languages containing binding constructs, and often leads to subtle, hard to find errors. In this paper, we exploit higher-order abstract syntax (HOAS) to implement a type-preserving compiler for the simply-typed lambda-calculus, including transformations such as closure conversion and hoisting, in the dependently-typed language Beluga. Unlike previous implementations, which have to abandon HOAS locally in favor of a first-order binder representation, we are able to take advantage of HOAS throughout the compiler pipeline, so that we do not have to include any lemmas about binder manipulation. Scope and type safety of the code transformations are statically guaranteed, and our implementation directly mirrors the proofs of type preservation. Our work demonstrates that HOAS encodings offer substantial benefits to certified programming.

1 Introduction

Type-based verification methods support building correct-by-construction software, and hold the promise of dramatically reducing the costs of quality assurance. Instead of verifying properties post-hoc about software, we rely on rich type abstractions which can be checked statically during the development.

Compiler implementers have long recognized the power of types to establish key properties about complex code transformations. However, the standard approach is to type-check the intermediate representations produced by compilation. This amounts to *testing* the result of compilation via type-checking. In this paper, we explore the use of sophisticated type systems to implement a correct-by-construction compiler for the simply typed lambda-calculus, including translation to continuation-passing style (CPS), closure conversion and hoisting. We concentrate here on the last two phases which are particularly challenging since they rearrange the structure of the abstract syntax tree.

A central question when implementing code transformations is the representation of the source and target languages. Shall we represent binders via first-order abstract syntax using de Bruijn indices or names or higher-order abstract syntax (HOAS) where we map binders in our source and target language to

binders in our meta-language? - Arguably HOAS is the more sophisticated representation technique, eliminating the need to deal with common and notoriously tricky aspects such as renaming, fresh name generation and capture-avoiding substitution. However, while the power and elegance of HOAS encodings have been demonstrated in representing proofs, for example in the Twelf system [Pfenning and Schürmann, 1999], it has been challenging to exploit its power in program transformations which rearrange abstract syntax trees and move possibly open code fragments. Previous implementations (for example Chlipala [2008]; Guillemette and Monnier [2008]) have been unable to take advantage of HOAS throughout the full compiler pipeline and have to abandon HOAS in closure conversion and hoisting. In this work, we rely on the rich type system and abstraction mechanisms of the dependently-typed language Beluga [Pientka and Dunfield, 2010; Cave and Pientka, 2012] to implement a type and scope preserving compiler for the simply-typed lambda-calculus using HOAS for all the stages. There are two key ingredients crucial to the success: First, we encode our source and target languages using HOAS within the logical framework LF [Harper et al., 1993] reusing the LF function space to model object-level binders. As a consequence, we inherit support for α-renaming, capture-avoiding substitution, and fresh name generation from LF. Second, we represent and embed open code fragments using the notions of contextual objects and first-class contexts. A contextual object, written as $[\Psi.M]$, characterizes an open LF object M which may refer to the bound variables listed in the context Ψ [Nanevski et al., 2008]. We internalize this notion on the level of types using the contextual type $[\Psi.A]$ which classifies the contextual objects $[\Psi.M]$ where M has type A in the context Ψ. By embedding contextual objects into computations, users can not only characterize abstract syntax trees with free variables, but also manipulate and rearrange open code fragments using pattern matching.

Our implementation of a type-preserving compiler is very compact avoiding tedious infrastructure for manipulating binders. Our code directly manipulates intrinsically typed source terms and is an *executable* version of the proof that the compiler is type-preserving.

We believe our work shows that programming with contextual objects offers significant benefits to certified programming. For the full development see: `http://complogic.cs.mcgill.ca/beluga/cc-code`.

2 Source language: Simply typed lambda-calculus

We describe first the source language of our compiler, the simply typed lambda-calculus (STLC) extended with n-ary tuples, selectors, let-expressions and unit.

$$
\begin{array}{lll}
\text{(Type)} & T, S & ::= S \to T \mid \mathsf{code}\ S\ T \mid S \times T \mid \mathsf{unit} \\
\text{(Source)} & M, N & ::= x \mid \lambda x.\, M \mid M\ N \mid \mathsf{fst}\ \ M \mid \mathsf{rst}\ \ M \mid (M_1, M_2) \\
 & & \quad \mid \mathsf{let}\ x = N\ \mathsf{in}\ M \mid () \\
\text{(Context)} & \Gamma & ::= \cdot \mid \Gamma, x : T
\end{array}
$$

Each of our type-preserving algorithms transforms the source language to a separate target language, but uses the same language for types. N-ary products are constructed using the binary product $S \times T$ and unit. In closure conversion, we will use n-ary tuples to describe the environment. Foreshadowing the subsequent explanation of closure conversion, we also add a special type code $S\ T$; this type only arises as a result in closure conversion where it describes closed functions.

We omit the typing rules for our source language, since they are standard.

The encoding of the source language into the logical framework LF is straightforward. In this paper, we are using the dependently typed language Beluga, which supports writing LF specifications and programs about them. By indexing `source` terms by their types, we only represent well-typed terms.

```
datatype source : tp → type =
| lam   : (source S → source T) → source (arr S T)
| app   : source (arr S T) → source S → source T
| fst   : source (cross S T) → source S
| rst   : source (cross S T) → source T
| cons  : source S → source T → source (cross S T)
| nil   : source unit
| letv  : source S → (source S → source T) → source T;
```

In Beluga's concrete syntax, the *kind* **type** declares an LF type family, as opposed to a computational data type. Binders in our object languages are represented via the LF function space. For example, the `lam` constructor takes as argument a function `source S → source T` and constructs an object of type `source (arr S T)`. As a consequence, we inherit α-renaming from LF and substitution is modelled via function application. N-ary tuples are represented using the constructor `cons` and () is represented as `nil`, emphasizing that n-ary tuples are encoded as lists.

3 Closure conversion

Closure conversion is a code transformation that makes the manipulation of closure objects explicit and results in a program whose functions are closed so that they can be hoisted to the top-level.

3.1 Target language

Our target language for closure conversion contains, in addition to functions ($\lambda y.\, P$), function application $P\ Q$, tuples (P, Q), selectors (fst and rst), and let-expressions ($\mathsf{let}\ y = P\ \mathsf{in}\ Q$), two new constructs: 1) we can form a closure $\langle P, Q \rangle$ of an expression P with its environment Q, represented as an n-ary tuple. 2) we can break apart a closure P using $\mathsf{let}\ \langle y_f, y_{env} \rangle = P\ \mathsf{in}\ Q$.

$$
\begin{array}{lll}
(\mathsf{Target}) & P, Q ::= & x \mid \lambda x.\, P \mid P\ Q \mid \mathsf{fst}\ P \mid \mathsf{rst}\ P \mid \mathsf{let}\ x = P\ \mathsf{in}\ Q \\
& & \mid (P, Q) \mid () \mid \langle P, Q \rangle \mid \mathsf{let}\ \langle y_f, y_{env} \rangle = P\ \mathsf{in}\ Q \\
(\mathsf{Context})\ \Delta & ::= & \cdot \mid \Delta, x : T
\end{array}
$$

The essential idea of closure conversion is to make the evaluation context of functions explicit; variables bound outside of a function are replaced by projections from an environment variable. Given a source-level function of type $T \to S$, we return a closure $\langle \lambda y_c.P, Q \rangle$ consisting of a closed function $\lambda y_c.P$, where y_c pairs the input argument y and the environment variable y_{env}, and its environment Q, containing all its free variables. Such packages are traditionally given an existential type such as $\exists l.(\mathsf{code}\ (T \times l)\ S) \times l$ where l is the type of the environment. We instead reuse the source type $T \to S$ which also hides l and saves us from having to handle existential types in their full generality. The rules for t_pack and t_letpack are modelling implicitly the introduction and elimination rules for existential types. Moreover, we enforce that $\lambda x.P$ is closed. The remaining typing rules for the target language are mostly straightforward and summarized next.

$\boxed{\Delta \vdash P : T}$ Target P has type T

$$\frac{\Delta, x : T \vdash P : S}{\Delta \vdash \lambda x.\, P : \mathsf{code}\ T\ S}\ \mathsf{t\_lam} \qquad \frac{\Delta \vdash P : \mathsf{code}\ T\ S \quad \Delta \vdash Q : T}{\Delta \vdash P\ Q : S}\ \mathsf{t\_app}$$

$$\frac{x : T \in \Delta}{\Delta \vdash x : T}\ \mathsf{t\_var} \qquad \frac{\cdot \vdash P : \mathsf{code}\ (T \times T_{env})\ S \quad \Delta \vdash Q : T_{env}}{\Delta \vdash \langle P, Q \rangle : T \to S}\ \mathsf{t\_pack}$$

$$\frac{\Delta \vdash P : T \to S \quad \Delta, y_f : \mathsf{code}\ (T \times l)\ S, y_{env} : l \vdash Q : S}{\Delta \vdash \mathsf{let}\ \langle y_f, y_{env} \rangle = P\ \mathsf{in}\ Q : S}\ \mathsf{t\_letpack}^l$$

$$\frac{\Delta \vdash P : T \quad \Delta \vdash Q : S}{\Delta \vdash (P, Q) : T \times S}\ \mathsf{t\_cons} \qquad \frac{}{\Delta \vdash () : \mathsf{unit}}\ \mathsf{t\_unit}$$

3.2 Closure conversion algorithm

Before describing the algorithm in detail, let us illustrate briefly the idea of closure conversion using an example. Our algorithm translates the program $(\lambda x.\lambda y.x + y)\ 5\ 2$ to

$$\begin{array}{l}
\mathsf{let}\ \langle f_1, c_1 \rangle = \\
\quad \mathsf{let}\ \langle f_2, c_2 \rangle = \\
\quad \langle\ \lambda e_2.\ \mathsf{let}\ x = \mathsf{fst}\ e_2\ \mathsf{in}\ \mathsf{let}\ x_{env} = \mathsf{rst}\ e_2\ \mathsf{in} \\
\quad \langle \lambda e_1.\ \mathsf{let}\ y = \mathsf{fst}\ e_1\ \mathsf{in}\ \mathsf{let}\ y_{env} = \mathsf{rst}\ e_2\ \mathsf{in}\ \mathsf{fst}\ y_{env} + y, (x, ()) \rangle \\
\quad ,\ ()\ \rangle \\
\quad \mathsf{in}\ f_2\ (5\ ,\ c_2) \\
\mathsf{in}\ f_1\ (2, c_1)
\end{array}$$

Closure conversion introduces an explicit representation of the environment, closing over the free variables of the body of an abstraction. We represent the environment as a tuple of terms, corresponding to the free variables in the body of the abstraction.

We define the algorithm for closure conversion using $[\![M]\!]_\rho$, where M is a source term which is well-typed in the context Γ and ρ a mapping of source

variables in Γ to target terms in the context Δ. Intuitively, ρ maps source variables to the corresponding projection of the environment. It is defined as follows:

$\boxed{\Delta \vdash \rho : \Gamma}$ ρ maps variables from source context Γ to target context Δ

$$\frac{}{\Delta \vdash id : \cdot}\ \mathsf{m\_id} \qquad \frac{\Delta \vdash \rho : \Gamma \quad \Delta \vdash P : T}{\Delta \vdash \rho, x \mapsto P : \Gamma, x : T}\ \mathsf{m\_dot}$$

For convenience, we write π_i for the i-th projection instead of using the selectors fst and rst. We give here only the cases for variables, functions and function applications.

$$\begin{aligned}
[\![x]\!]_\rho &= \rho(x) \\
[\![\lambda x.M]\!]_\rho &= \langle\ \lambda x_c.\, \mathsf{let}\ x = \mathsf{fst}\ x_c\ \mathsf{in}\ \mathsf{let}\ x_{env} = \mathsf{rst}\ x_c\ \mathsf{in}\ P\ ,\ P_{env}\ \rangle \\
&\quad \text{where } \{x_1, \ldots, x_n\} = \mathsf{FV}(\lambda x.M) \\
&\quad \text{and } \rho' = x_1 \mapsto \pi_1\ x_{env}, ..., x_n \mapsto \pi_n\ x_{env}, x \mapsto y \\
&\quad \text{and } P_{env} = (\rho(x_1), \ldots, \rho(x_n)) \text{ and } P = [\![M]\!]_{\rho'} \\
[\![M\ N]\!]_\rho &= \mathsf{let}\ \langle x_f, x_{env}\rangle = P\ \mathsf{in}\ x_f\ (Q\ ,\ x_{env}) \quad \text{where } P = [\![M]\!]_\rho \text{ and } Q = [\![N]\!]_\rho
\end{aligned}$$

To translate a source variable, we look up its binding in the map ρ. When translating a lambda-abstraction $\lambda x.M$, we first compute the set $\{x_1, \ldots, x_n\}$ of free variables occurring in $\lambda x.M$. We then form a closure consisting of two parts: 1) a term P which is obtained by converting M with the new map ρ' which maps variables $x_1, \ldots, x_n$ to their corresponding projection of the environment variable and x to itself, thereby eliminating all free variables in M. 2) an environment tuple P_{env}, obtained by applying ρ to each variable in $(x_1, \ldots, x_n)$.

When translating an application $M\ N$, we first translate M and N to target terms P and Q. Since the source term M denotes a function, the target term P will denote a closure. We unpack the closure obtaining x_f, the part denoting the function, and x_{env}, the part denoting the environment. We then apply x_f to the extended environment (Q, x_{env}).

Our goal is to implement the described algorithm as a recursive program which manipulates intrinsically well-typed source terms. This is non-trivial. To understand the general idea behind our program, we discuss how to prove that given a well-typed source term M we can produce a well-typed target term which is the result of converting M. The proof relies on several straightforward lemmas which correspond exactly to auxiliary functions needed in our implementation.

Auxiliary lemmas:

- **Strengthening**: If $\Gamma \vdash M : T$ and $\Gamma' = FV(M)$, then $\Gamma' \vdash M : T$ and $\Gamma' \subseteq \Gamma$
- **Weakening**: If $\Gamma' \vdash M : T$ and $\Gamma' \subseteq \Gamma$ then $\Gamma \vdash M : T$.
- **Context reification**: Given a context $\Gamma = x_1 : T_1, \ldots, x_n : T_n$, there exists a type $T_\Gamma = (T_1 \times \ldots \times T_n)$ and there is a $\rho = x_1 \mapsto \pi_1\ x_{env}, \ldots, x_n \mapsto \pi_n\ x_{env}$ s.t. $x_{env} : T_\Gamma \vdash \rho : \Gamma$ and $\Gamma \vdash (x_1, \ldots x_n) : T_\Gamma$.
- **Map extension**: If $\Delta \vdash \rho : \Gamma$, then $\Delta, x : T \vdash \rho, x \mapsto x : \Gamma, x : T$.

– **Map lookup**: If $x : T \in \Gamma$ and $\Delta \vdash \rho : \Gamma$, then $\Delta \vdash \rho(x) : T$.
– **Map lookup (tuple)**:
If $\Gamma \vdash (x_1, \ldots, x_n) : T$ and $\Delta \vdash \rho : \Gamma$ then $\Delta \vdash (\rho(x_1), \ldots, \rho(x_n)) : T$.

We show here the key cases of the proof concentrating on lambda abstractions and variables.

Theorem 1. *If* $\Gamma \vdash M : T$ *and* $\Delta \vdash \rho : \Gamma$ *then* $\Delta \vdash [\![M]\!]_\rho : T$

Proof. By induction on the structure of the term M.

Case : $M = x$.

$\Gamma \vdash x : T$ and $\Delta \vdash \rho : \Gamma$	by assumption
$\Delta \vdash \rho(x) : T$	by Map lookup
$\Delta \vdash [\![x]\!]_\rho$: T	by definition

Case $M = \lambda x.M$

$\Gamma \vdash \lambda x.M : T \to S$ and $\Delta \vdash \rho : \Gamma$	by assumption
$\Gamma' \vdash \lambda x.M : T \to S$ and $\Gamma' \subseteq \Gamma$	
where $\Gamma' = FV(\lambda x.M)$	by Term strengthening
$\Gamma', x : T \vdash M : S$	by inversion on t_lam
$\Gamma' \vdash (x_1, \ldots, x_n) : T_{\Gamma'}$ and $x_{env} : T_{\Gamma'} \vdash \rho' : \Gamma'$	by Context reification
$\Gamma \vdash (x_1, \ldots, x_n) : T_{\Gamma'}$	by Term Weakening
$\Delta \vdash \rho : \Gamma$	by assumption
$(\rho(x_1), \ldots, \rho(x_n)) = P_{env}$	by assumption
$\Delta \vdash P_{env} : T_{\Gamma'}$	by Map lookup (tuple)
$x_{env} : T_{\Gamma'}, x : T \vdash \rho', x \mapsto x : \Gamma', x : T$	By Map extension
$x_{env} : T_{\Gamma'}, x : T \vdash P : S$	
where $P = [\![M]\!]_{\rho', x \mapsto x}$	by i.h. on M
$c : T \times T_{\Gamma'}, x : T, x_{env} : T_{\Gamma'} \vdash P : S$	by Term weakening
$c : T \times T_{\Gamma'}, x : T \vdash \mathsf{let}\ x_{env} = \mathsf{rst}\ c\ \mathsf{in}\ P : S$	by rule t_let
$c : T \times T_{\Gamma'} \vdash \mathsf{let}\ x = \mathsf{fst}\ x\ \mathsf{in}\ \mathsf{let}\ x_{env} = \mathsf{rst}\ c\ \mathsf{in}\ P : S$	by rule t_let
$\cdot \vdash \lambda c.\, \mathsf{let}\ x = \mathsf{fst}\ c\ \mathsf{in}\ \mathsf{let}\ x_{env} = \mathsf{rst}\ c\ \mathsf{in}\ P : \mathsf{code}\ (T \times T_{\Gamma'})\ S$	by rule t_lam
$\Delta \vdash \langle \lambda c.\, \mathsf{let}\ x = \mathsf{fst}\ c\ \mathsf{in}\ \mathsf{let}\ x_{env} = \mathsf{rst}\ c\ \mathsf{in}\ P\ ,\ P_{env} \rangle : T \to S$	by rule t_pack
$\Delta \vdash [\![\lambda x.M]\!]_\rho : T \to S$	by definition

□

3.3 Representation of target language in LF

We now describe the implementation of the closure conversion algorithm in Beluga. We begin by defining the target language, showing the constructs for lambda-abstraction, application, creating a closure and taking a closure apart.

```
datatype target: tp → type =
| clam    : (target T → target S) → target (code T S)
| capp    : target (code T S) → target T → target S
| cpack   : target  (code (cross T L) S) → target L → target (arr T S)
| cletpack: target (arr T S)
          → ({l:tp} target (code (cross T l)) S)→target l → target S)
          → target S;
```

The data-type definition directly reflects the typing rules with one exception: our typing rule t_pack enforced that P was closed. This cannot be achieved in the LF encoding, since the context of assumptions is ambient. As a consequence, hoisting, which relies on the fact that the closure converted functions are closed, cannot be implemented as a separate phase after closure conversion. We will come back to this issue in Section 4.

3.4 Type-preserving closure conversion in Beluga: an overview

The top-level closure conversion function `cc` translates a closed source term of type `T` to a closed target term of type `T`, which is encoded in Beluga using the computation-level type `[.source T] → [.target T]`. We embed closed contextual LF object of type `source T` and `target T` into computation-level types via the modality `[ ]`. The `.` separates the context of assumptions from the conclusion. Since we are describing closed objects, the context is left empty.

However, when closure converting and traversing source terms, our source terms do not remain closed. We generalize the closure conversion function to translate well-typed source terms in a source context Γ to well-typed target terms in the target context Δ given a map of the source context Γ to the target context Δ. Δ will consists of an environment variable $\mathtt{x}_{env}$ and the variable `x` bound by the last abstraction, along with variables introduced by let bindings.

```
cc': Map [Δ] [Γ] → [Γ. source T] → [Δ. target T]
```

Just as types classify terms, schemas classify contexts in Beluga, similarly to world declarations in Twelf [Pfenning and Schürmann, 1999]. The schema `tctx` defines a context where the type of each declaration is an instance of `target T`; similarly the schema `sctx` defines a context where the type of each declaration is an instance of `source T`. While type variables appear in the typing rule `t_letpack`, they only occur locally and are always bound before the term is returned by our functions, such that they do not appear in the context variables indexing them.

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

We use the indexed recursive type `Map` to relate the target context Δ and source context Γ [Cave and Pientka, 2012]. In Beluga's concrete syntax, the *kind* `ctype` indicates that we are not defining an LF datatype, but a recursive type on the level of computations. → is overloaded to mean computation-level strong functions rather than the LF function space. `Map` is defined recursively on the source context Γ directly encoding our definition $\Delta \vdash \rho : \Gamma$ given earlier.

```
datatype Map:{Δ:tctx}{Γ:sctx} ctype =
| Id :{Δ:tctx} Map [Δ] []
| Dot: Map [Δ] [Γ] → [Δ. target S] → Map [Δ] [Γ,x:source S];
```

Beluga reconstructs the type of free variables Δ, Γ, and `S` and implicitly abstracts over them. In the constructor `Id`, we choose to make Δ an explicit argument to `Id`, since we often need to refer to Δ explicitly in the recursive programs we are writing about `Map`. The next section presents the implementation of the necessary auxiliary functions, followed by `cc'`.

3.5 Implementation of auxiliary lemmas

Term strengthening and weakening Both operations rely on an inclusion relation $\Gamma' \subseteq \Gamma$ where we preserve the order, which is defined using the indexed recursive computation-level data-type `SubCtx`.

```
datatype SubCtx: {Γ':sctx} {Γ:sctx} ctype =
| WInit: SubCtx [] []
| WDrop: SubCtx [Γ'] [Γ] → SubCtx [Γ'] [Γ,x:source T]
| WKeep: SubCtx [Γ'] [Γ] → SubCtx [Γ',x:source T] [Γ,x:source T];
```

Given a source term `M` in Γ the function `strengthen` computes the strengthened version of `M` which is well-typed in Γ' characterizing the free variables in `M` together with the proof `SubCtx [Γ'] [Γ]`. We represent the result using the indexed recursive type `StrTerm'` encoding the existential in the specification as a universal quantifier using the constructor `STm'`. The fact that Γ' describes exactly the free variables of `M` is not captured by the type definition.

```
datatype StrTerm': {Γ:sctx} [.tp] → ctype =
| STm': [Γ'. source T] → SubCtx [Γ'] [Γ] → StrTerm' [Γ] [.T];

rec strengthen: [Γ.source T] → StrTerm' [Γ] [.T]
```

Just as in the proof of the term strengthening lemma, we cannot implement this function directly. This is because, while we would like to perform induction on the size of Γ, we cannot appeal to the induction hypothesis while maintaining a well-scoped source term in the case of an occurring variable in front of Γ. Instead, we implement `str`, which, intuitively, implements the lemma

$$\textit{If } \Gamma_1, \Gamma_2 \vdash M : T \textit{ and } \Gamma_1', \Gamma_2 = \mathsf{FV}(M), \textit{ then } \Gamma_1', \Gamma_2 \vdash M : T \textit{ and } \Gamma_1' \subseteq \Gamma_1.$$

In Beluga, contextual objects can only refer to one context variable - we cannot simply write `[Γ1, Γ2. source T]`. To express this, we use a data-type `wrap` which abstracts over all the variables in Γ_2. `wrap` is indexed by the type `T` of the source term and the size of Γ_2. `str` then recursively analyses Γ_1, adding variables occurring in the input term to Γ_2. The type of `str` asserts, through its index `N`, the size of Γ_2.

```
datatype wrap: tp → nat → type =
| ainit: (source T) → wrap T z
| add: (source S → wrap T N) → wrap (arr S T) (suc N);

datatype StrTerm: {Γ:sctx} [.tp] → [.nat]  → ctype =
| STm: [Γ'. wrap T N] → SubCtx [Γ'] [Γ] → StrTerm [Γ] [.T] [.N];

rec str: [Γ. wrap T N] →  StrTerm [Γ] [.T] [.N]
```

The function `str` is implemented recursively on the structure of Γ exploits higher-order pattern matching to test whether a given variable `x` occurs in a term `M`. As a consequence, we can avoid the implementation of a function which recursively analyzes `M` and test whether `x` occurs in it. While one can implement term weakening following similar ideas, we incorporate it into the variable lookup function defined next.

Map extension and lookup The lookup function takes a source variable of type `T` in the source context Γ and `Map [`Δ`] [`Γ`]` and returns the corresponding target expression of the same type.

```
rec lookup: {#p:[Γ.source T]} Map [Δ] [Γ] → [Δ. target T] =
λ□#p ⇒ fn ρ ⇒ let (ρ: Map [Δ] [Γ]) = ρ in case [Γ. #p… ] of
| [Γ',x:source T. x]    ⇒ let  Dot ρ' [Δ.M… ] = ρ in [Δ.M… ]
| [Γ',x:source S. #q… ] ⇒ let  Dot ρ' [Δ.M… ] = ρ in lookup [Γ'.#q… ] ρ';
```

We quantify over all variables in a given context by `{#p:[`Γ`.source T]}` where `#p` denotes a variable of type `source T` in the context Γ. In the function body, $\lambda^{\square}$-abstraction introduces an explicitly quantified contextual object and **fn**-abstraction introduces a computation-level function. The function `lookup` is implemented by pattern matching on the context Γ and the parameter variable `#p`.

To guarantee coverage and termination, it is pertinent that we know that an n-ary tuple is composed solely of source variables from the context Γ, in the same order. We therefore define `VarTup` as a computational datatype for such variable tuples. `Nex v` of type `VarTup [`Γ`] [.L`$_\Gamma$`]`, where $\Gamma = \mathtt{x}_1\mathtt{:T}_1,\ldots,\mathtt{x}_n\mathtt{:T}_n$, is taken to represent the source language tuple `(`$\mathtt{x}_1,\ldots,\mathtt{x}_n$`)` of type $\mathtt{T}_1 \times \ldots \times \mathtt{T}_n$ in the context Γ.

```
datatype VarTup: {Γ:sctx} [.tp] → ctype =
| Emp: VarTup [] [.unit]
| Nex: VarTup [Γ] [.L] → VarTup [Γ,x:source T] [.cross T L];
```

The function `lookupVars` applies a map ρ to every variable in a variable tuple.

```
rec lookupVars: VarTup [Γ'] [.L_Γ'] → SubCtx [Γ'] [Γ] → Map [Δ] [Γ]
              → [Δ. target L_Γ']
```

`lookupVars` allows the application of a `Map` defined on a more general context Γ provided that $\Gamma' \subseteq \Gamma$. This corresponds, in the theoretical presentation, to weakening a variable tuple before applying a mapping on it.

`extendMap`, which implements the Map extension lemma, weakens a mapping with the identity on a new variable `x`. It is used to extend the `Map` with local variables, for example when we encounter a **let** binding construct.

```
rec extendMap: Map [Δ] [Γ] → Map [Δ,x:target S] [Γ,x:source S]
```

A Reification of the Context as a Term Tuple The context reification lemma is proven by induction on Γ; to enable pattern matching on the context Γ, we wrap it in the indexed data-type `Ctx`.

```
datatype Ctx: {Γ:sctx} ctype =
| Ctx: {Γ:sctx} Ctx [Γ];

datatype CtxAsTup: {Γ:sctx} ctype =
| CTup: VarTup [Γ] [.L_Γ] → Map [x:target L_Γ] [Γ] → CtxAsTup [Γ];

rec reify: Ctx [Γ] → CtxAsTup [Γ]
```

The function `reify` translates the context Γ to a source term. It produces a tuple containing variables of Γ in order, along with `Map [x:target T`$_\Gamma$`] [`Γ`]` describing the mapping between those variables and their corresponding projections. The type of `reify` enforces that the returned `Map` contains, for each of the variables in Γ, a target term of the same type referring solely to a variable `x`

```
rec cc': Map [Δ] [Γ] → [Γ. source T] → [Δ. target T] =
fn ρ ⇒ fn m ⇒ case m of
| [Γ. #p… ] ⇒ lookup ρ [Γ. #p… ]
| [Γ. lam λx.M… x] ⇒
  let  STm [Γ'. add (λx. ainit (M' … x))] rel = str [Γ. add λx.ainit (M … x)] in
  let  CTup [Γ'. E… ] (ρ'':Map [x_env:target T_Γ'] [Γ']) = reify (Ctx [Γ']) in
  let  ρ' = extendMap ρ'' in
  let  [x_env:target T_Γ',x:target T. (P x_env x)] = cc' ρ' [Γ',x:source _. M… x] in
  let  [Δ. P_env… ] = lookupVars [Γ'. E… ] rel ρ in
     [ Δ. cpack (clam (λc. (clet (cfst c)
                                 (λx.(clet (crst c)
                                           (λx_env. P x_env x))))))
                (P_env… ) ]
```

Fig. 1. Implementation of closure conversion in BELUGA

of type $\mathtt{T}_\Gamma$. This means the tuple of variables of type $\mathtt{T}_\Gamma$ also returned by `reify` contain enough *information* to replace occurrences of variables in any term in context Γ perserving types - it contains either the variables themselves or terms of the same type.

3.6 Closure conversion: Top-level function

The function `cc'` (see Fig. 1) implements our closure conversion algorithm recursively by pattern matching on objects of type `[Γ. source T]` . It follows closely the earlier proof (Thm. 1). We describe here on the cases for variables and lambda-abstractions omitting the case for applications. When we encounter a variable, we simply lookup its corresponding binding in ρ.

Given a lambda abstraction in context Γ and ρ which represents the map from Γ to Δ, we begin by strengthening the term to some context Γ'. We then reify the context Γ' to obtain a tuple `E` together with the new map ρ'' of type `Map [x_env:target T_Γ'] [Γ']`. Next, we extend ρ'' with the identity on the lambda-abstraction's local variable to obtain ρ', and recursively translate `M` using ρ', obtaining a `target` term in context `x_env,x`. Abstracting over `x_env` and `x` gives us the desired closure-converted lambda-abstraction. To obtain the environment `P_env`, we apply ρ on each variables in `E` using `lookupVars`. Finally, we pack the converted lambda-abstraction and the environment `P_env` as a closure, using the constructor `cpack`.

Our implementation of closure conversion, including all definitions and auxiliary functions, consists of approximately 250 lines of code.

4 Hoisting

Hoisting is a code transformation that lifts the lambda-abstractions, closed by closure conversion, to the top level of the program. Function declarations in the program's body are replaced by references to a global function environment.

As we alluded to earlier, our encoding of the target language of closure conversion does not guarantee that functions in a closure converted term are indeed closed. While this information is available during closure conversion, it cannot easily be captured in our meta-language. We therefore extend our closure conversion algorithm to perform hoisting at the same time. Hoisting can however be understood by itself; we highlight here its main ideas.

Performing hoisting on the closure-converted program presented in Sec. 3

$$
\begin{array}{l}
\mathsf{let}\ \langle f_1, c_1\rangle = \\
\quad \mathsf{let}\ \langle f_2, c_2\rangle = \\
\quad \langle\ \lambda e_2.\ \mathsf{let}\ x = \mathsf{fst}\ e_2\ \mathsf{in}\ \mathsf{let}\ x_{env} = \mathsf{rst}\ e_2\ \mathsf{in} \\
\quad \langle \lambda e_1.\ \mathsf{let}\ y = \mathsf{fst}\ e_1\ \mathsf{in}\ \mathsf{let}\ y_{env} = \mathsf{rst}\ e_2\ \mathsf{in}\ \mathsf{fst}\ y_{env} + y, (x, ())\rangle \\
\quad ,\ ()\ \rangle \\
\quad \mathsf{in}\ f_2\ (5\ ,\ c_2) \\
\mathsf{in}\ f_1\ (2, c_1)
\end{array}
$$

will return

$$
\begin{array}{l}
\mathsf{let}\ l = (\lambda l_2.\lambda e_2.\mathsf{let}\ x = \mathsf{fst}\ e_2\ \mathsf{in}\ \mathsf{let}\ x_{env} = \mathsf{rst}\ e_2\ \mathsf{in}\ \langle\ (\mathsf{fst}\ l_2)\ (\mathsf{rst}\ l_2)\ ,\ (x,\ ())\ \rangle, \\
\qquad \lambda l_1.\lambda e_1.\mathsf{let}\ y = \mathsf{fst}\ e_1\ \mathsf{in}\ \mathsf{let}\ y_{env} = \mathsf{rst}\ e_2\ \mathsf{in}\ \mathsf{fst}\ y_{env} + y, ()) \\
\mathsf{in}\ \mathsf{let}\ \langle f_1, c_1\rangle = \\
\quad \mathsf{let}\ \langle f_2, c_2\rangle = \langle(\mathsf{fst}\ l)\ (\mathsf{rst}\ l), (\cdot)\rangle \\
\quad \mathsf{in}\ f_2\ (5, c_2) \\
\mathsf{in}\ f_1\ (2, c_1)
\end{array}
$$

4.1 Source and target languages - revisited

We define hoisting on the target language of closure conversion and keep the same typing rules (see Fig. 3.1) with one exception: the typing rule for t_pack is replaced by the one below.

$$
\frac{l : T_f \vdash P : \mathsf{code}\ (T \times T_x)\ S \quad \Delta, l : T_f \vdash Q : T_x}{\Delta, l : T_f \vdash \langle P, Q\rangle : T \to S}\ \mathsf{t\_pack'}
$$

When hoisting is performed at the same time as closure conversion, P is not completely closed anymore, as it refers to the function environment `l`. Only at top-level, where we bind the collected tuple as `l`, will we recover a closed term. The distinction between t_pack and t_pack' is irrelevant in our implementation, as in our representation of the typing rules in LF the context is ambient.

We now define the hoisting algorithm as $[\![P]\!]_l = Q \bowtie E$. Hoisting takes as input a target term P and returns a hoisted target term Q together with its function environment E, represented as a n-ary of product type L. We write $E_1 \circ E_2$ for appending tuple E_2 to E_1 and $L_1 \circ L_2$ for appending the product type L_2 to L_1. We concentrate here on the cases for variables and closures.

$$
\begin{array}{lll}
[\![x]\!]_l & = x \bowtie () & \\
[\![\langle P_1, P_2\rangle]\!]_l & = \langle(\mathsf{fst}\ l)\ (\mathsf{rst}\ l), Q_2\rangle \bowtie E & \text{where } Q_1 \bowtie E_1 = [\![P_1]\!]_l \\
 & & \text{and } Q_2 \bowtie E_2 = [\![P_2]\!]_l \\
 & & \text{and } E = (\lambda l.Q_1,\ E_1 \circ E_2)
\end{array}
$$

While the presented hoisting algorithm is simple to implement in an untyped setting, its extension to a typed language demands more care with respect to the form and type of the functions environment. As $\circ$ is only defined on n-ary tuples and product types and not on general terms and types, we enforce that the returned E and its type L are of the right form. We take $\Delta \vdash_l E : L$ to mean $\Delta \vdash E : L$ for a n-ary tuple E of product type L.

Auxiliary lemmas:

- **Append function environments**
  If $\Delta \vdash_l E_1 : L_1$ and $\Delta \vdash_l E_2 : L_2$, then $\Delta \vdash_l E_1 \circ E_2 : L_1 \circ L_2$.
- **Function environment weakening (1)**
  If $\Delta, l : L_{f_1} \vdash P : T$ and $L_{f_1} \circ L_{f_2} = L_f$, then $\Delta, l : L_f \vdash P : T$.
- **Function environment weakening (2)**
  If $\Delta, l : L_{f_2} \vdash P : T$ and $L_{f_1} \circ L_{f_2} = L_f$, then $\Delta, l : L_f \vdash P : T$.

Theorem 2. *If $\Delta \vdash P : T$ then $\cdot \vdash_l E : L_f$ and $\Delta, l : L_f \vdash Q : T$ for some L_f where $[\![P]\!]_l = Q \bowtie E$.*

Proof. By induction on the term P.

4.2 Auxiliary functions

Defining environments Our hoisting algorithm uses operations such as $\circ$, which are only defined on n-ary tuples and on product types. To guarantee coverage, we define an indexed datatype encoding the judgement $\Delta \vdash_l E : L_f$, which asserts that environment E and its type L_f are of the right form.

```
datatype Env: {Lf:[.tp]} [.target Lf] → ctype =
| EnvNil: Env [.unit] [.cnil]
| EnvCons: {P:[.target T]}
           Env [.L] [.E]  → Env [.cross T L] [.ccons P E];
```

Appending function environments When hoisting terms with more than one subterm, each recursive call on those subterms results in a different function environment; they need to be merged before combining the subterms again. This is accomplished by the function `append` which takes in `Env [.L1] [.E1]` and `Env [.L2] [.E2]`, and constructs the function environment `Env [.L1 ∘ L2] [.E1 ∘ E2]`. As Beluga does not support functions in types, we return some function environment `E` of type `L`, and a proof that `E` and `L` are the results of concatenating respectively `E1` and `E2`, and `L1` and `L2`.

```
datatype App: {T:[.tp]}{S:[.tp]}{TS:[.tp]} [.target T] → [.target S]
           → [.target TS] → ctype =
| AStart: Env [.S] [.Q] → App [.unit] [.S] [.S] [.cnil] [.Q] [.Q]
| ACons: App [.T] [.S] [.TS] [.P] [.Q] [.PQ]
      → App [.(cross T' T)] [.S] [.(cross T' TS)]
            [.(ccons P' P)] [.Q] [.(ccons P' PQ)];

datatype ExApp: {T:[.tp]}{S:[.tp]} [.target T] → [.target S] → ctype =
| AP: App [.L1] [.L2] [.L] [.E1] [.E2] [.E] → Env [.L] [.E]
   → ExApp [.L1] [.L2] [.E1] [.E2];

rec append: Env [.L1] [.E1]→ Env [.L2] [.E2]→ ExApp [.L1] [.L2] [.E1] [.E2]
```

`App [.L1] [.L2] [.L] [.E1] [.E2] [.E]` can be read as $\texttt{E}_1$ and $\texttt{E}_2$ being tuples of type $\texttt{L}_1$ and $\texttt{L}_2$, and concatening them yields the tuple `E` of type `L`.

Next, we show the type of the two lemmas about function environment weakening. They are a direct encoding of their specifications.

```
rec weakenEnv1: (Δ:tctx) App [.L1] [.L2] [.L] [.E1] [.E2] [.E]
             → [Δ, l:target L1. target T] → [Δ, l:target L. target T]
rec weakenEnv2: (Δ:tctx) App [.L1] [.L2] [.L] [.E1] [.E2] [.E]
             → [Δ, l:target L2. target T] → [Δ, l:target L. target T]
```

4.3 The main function

The top-level function `hcc` generalizes `cc` such that it performs hoisting at the same time as closure conversion. Again we only concentrate on the case for variables and lambda-abstraction to illustrate that only small changes are required. We generalize `hcc` and implement `hcc'` to closure convert and hoist open terms when given a map between the source and target context.

```
datatype HCCRet:{Δ:tctx} [.tp] → ctype =
| HRet: [Δ,l:target Lf. target T] → Env [.Lf] [.E] → HCCRet [Δ] [.T];
rec hcc': Map [Δ] [Γ] → [Γ. source T] → HCCRet [Δ] [.T] =
fn ρ ⇒ fn m ⇒ case m of
| [Γ. #p… ] ⇒
  let [Δ. Q… ] = lookup [Γ] [Γ. #p… ] ρ in
    HRet [Δ,l:target (prod unit). Q… ] EnvNil
| [Γ. lam λx.M… x] ⇒
  let STm [Γ'.add λx.ainit (M'… x)] rel = str [Γ.add λx. ainit (M … x)] in
  let CTup vt (ρ'':Map [xenv:target TΓ'] [Γ']) = reify (Ctx [Γ']) in
  let [Δ. Penv… ] = lookupTup vt rel ρ in
  let HRet r e = hcc' (extendMap ρ'') [Γ',x:source _. M'… x] in
  let [xenv:target TΓ', x:target T, l:target Tf. (Q xenv x l)] = r in
  let e' = EnvCons [.clam λl. clam λc.
                      clet (cfst c) (λx.clet (crst c) (λxenv. Q xenv x l))]
                   e in
  let [.T'] = [.cross (code Tf (code (cross T TΓ') S)) Tf]
  in HRet  [Δ,l:target T'. cpack (capp (cfst l) (crst l))  (Penv… )] e'
;
rec hcc: [.source T] → [.target T] =
fn m ⇒ let HRet r (e: Env [._] [.E]) = hcc' (IdMap []) m  in
        let [l:target S. Q l] = r in
          [.clet E (λl. Q l)];
```

`hcc` calls `hcc'` with the initial map and the source term `m` of type `T`. It then binds, with `clet`, the function environment as `l` in the hoisted term, resulting in a closed target term of the same type.

`hcc'` converts a source term in the context Γ given a map between the source context Γ and the target context Δ following the algorithm described in Sec. 4. It returns a target term of type `T` which depends on a function environment `l` of some product type $\texttt{L}_f$ together with a concrete function environment of type $\texttt{L}_f$. The result of `hcc'` is described by the datatype `HCCRet` which is indexed by the target context Δ and the type `T` of the target term.

`hcc'` follows closely the structure of `cc'`. When we encounter a variable, we look it up in ρ and return the corresponding target term with an empty well-formed function environment `EnvNil`. When reaching a lambda-abstraction of

type `arr S T`, we again strengthen the body `lam λx.M … x` to some context Γ'. We then reify Γ' to obtain a variable tuple $(x_1, \ldots, x_n)$ and convert the strengthened `M` recursively using the map ρ extended with the identity. As a result, we obtain a closed target term `Q` together with a well-formed function environment `e` containing the functions collected so far. We then build the variable environment $(\rho(x_1), \ldots, \rho(x_n))$, extend the function environment with the converted result of `M` which is known to be closed, and return `capp (cfst l) (crst l)` where `l` abstracts over the current function environment.

Our implementation of hoisting adds in the order of 100 lines to the development of closure conversion and retains its main structure.

An alternative to the presented algorithm would be to thread through the function environment as an additional argument to `hcc`. This avoids the need to append function environments and obviates the need for `weakenEvn1`. Other properties around `concat` would however still have to be proven, some of which require multiple nested inductions; therefore, the complexity and length of the resulting implementation is similar or even larger.

5 Related Work

While HOAS holds the promise of dramatically reducing the overhead related to manipulating abstract syntax trees with binders, the implementation of a certified compiler, in particular the phases of closure conversion and hoisting, using HOAS has been elusive.

One of the earliest studies of using HOAS in implementing compilers was presented in Hannan [1995], where the author describes the implementation of a type-directed closure conversion in *Elf* [Pfenning, 1989], leaving open several implementation details, such as how to reason about variables equality.

Abella [Gacek, 2008] is an interactive theorem prover which supports HOAS, but not dependent types at the specification level. The standard approach would be to specify source terms, typing judgments, and the closure conversion algorithm, and then prove that it is type-preserving. However, one cannot obtain an executable program from the proof. Moreover, it is not obvious how to specify closure conversion algorithm since one of its arguments is the mapping ρ which itself inductively defined

Closely related to our work is Guillemette and Monnier [2007]'s implementation of a type-preserving closure conversion algorithm over STLC in Haskell. While HOAS is used in the CPS translation, the languages from closure conversion onwards use de Bruijn indices. Since the language targeted by their closure conversion syntactically enforces that functions are closed, it is possible for them to perform hoisting in a separate phase. In Guillemette and Monnier [2008], the authors extend the closure conversion implementation to System F.

Chlipala [2008] presents a certified compiler for STLC in Coq using parametric higher-order abstract syntax (PHOAS), a variant of weak HOAS. He however annotates his binders with de Bruijn level before the closure conversion pass, thus

degenerating to a first-order representation. His closure conversion is hence similar to the one of Guillemette and Monnier [2007]. As in our work, hoisting is done at the same time as closure conversion, because his target language does not capture that functions are closed.

In both implementations, infrastructural lemmas dealing with binders constitute a large part of the development. Moreover, additional information in types is necessary to ensure the program type-checks, but is irrelevant at a computational level. In contrast, we rely on the rich type system and abstraction mechanisms of Beluga to avoid all infrastructural lemmas.

The closure conversion algorithm has also served as a key benchmark for systems supporting first-class nominal abstraction such as FreshML [Pottier, 2007] and αProlog [Cheney and Urban, 2004]. Both languages provide facilities for generating names and reasoning about their freshness, which proves to be useful when computing the free variables in a term. However, capture-avoiding substitution still needs to be implemented separately. Since these languages lack dependent types, implementing a certified compiler is out of their reach.

6 Conclusion

In addition to closure conversion and hoisting, we also have implemented the translation to continuation-passing style. Our compiler not only type checks, but also coverage checks. Termination can be verified straightforwardly by the programmer, as every recursive call is made on a structurally smaller argument, such that all our functions are total. The fact that we are not only preserving types but also the scope of terms guarantees that our implementation is *essentially* correct by construction.

Although HOAS is one of the most sophisticated encoding techniques for structures with binders and offers significant benefits, problems such as closure conversion, where reasoning about the identity of free variables is needed, have been difficult to implement using an HOAS encoding. In Beluga, contexts are first-class; we can manipulate them, and indeed recover the identity of free variables by observing the context of the term. This is unlike other system supporting HOAS such as Twelf [Pfenning and Schürmann, 1999] or Delphin [Poswolsky and Schürmann, 2008]; in Abella [Gacek, 2008], we can test variables for identity, but users need to represent and reason about contexts explicitly. More importantly, we cannot obtain an executable program from the proof.

In addition, Beluga's computation-level recursive datatypes provide us with an elegant tool to encode properties about contexts and contextual object. Our case study clearly demonstrates the elegance of developing certified programs in Beluga. We rely on built-in substitutions to replace bound variables with their corresponding projections in the environment; we rely on the first-class context and recursive datatypes to define a mapping of source and target variables as well as computing a strengthened context only containing the relevant free variables in a given term.

In the future, we plan to extend our compiler to System F. While the algorithms seldom change from STLC to System F, open types pose a significant challenge. This will provide further insights into what tools and abstractions are needed to make certified programming accessible to the every day programmer.

Acknowledgements. We thank Mathieu Boespflug for his feedback and work on the implementation of Beluga, and anonymous referees for helpful suggestions and comments on an earlier version of this paper.

7 Bibliography

A. Cave and B. Pientka. Programming with binders and indexed data-types. In *Symposium on Principles of Programming Languages*, pages 413–424. ACM, 2012.

J. Cheney and C. Urban. Alpha-Prolog: A logic programming language with names, binding and alpha-equivalence. In *International Conference on Logic Programming*, pages 269–283, 2004.

A. J. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *International Conference on Functional Programming*, pages 143–156. ACM, 2008.

A. Gacek. The Abella interactive theorem prover (system description). In *International Joint Conference on Automated Reasoning*, pages 154–161. Springer, 2008.

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

L.-J. Guillemette and S. Monnier. A type-preserving compiler in Haskell. In *International Conference on Functional Programming*, pages 75–86. ACM, 2008.

J. Hannan. Type systems for closure conversions. In *Workshop on Types for Program Analysis*, pages 48–62, 1995.

R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993. doi: 10.1145/138027.138060.

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

F. Pfenning. Elf: A language for logic definition and verified meta-programming. In *Symposium on Logic in Computer Science*, pages 313–322. IEEE, 1989.

F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *Conference on Automated Deduction*, pages 202–206. Springer, 1999.

B. Pientka and J. Dunfield. Beluga: a framework for programming and reasoning with deductive systems (System Description). In *International Joint Conference on Automated Reasoning*, pages 15–21. Springer, 2010.

A. B. Poswolsky and C. Schürmann. Practical programming with higher-order encodings and dependent types. In *European Symposium on Programming*, pages 93–107. Springer, 2008.

F. Pottier. Static name control for FreshML. In *Symposium on Logic in Computer Science*, pages 356–365. IEEE, July 2007.