Programming type-safe transformations using higher-order abstract syntax

OLIVIER SAVARY BELANGER
McGill University
STEFAN MONNIER
Universite de Montreal
and
BRIGITTE PIENTKA
McGill University

When verifying that compiler phases preserve some property of the compiled program, a major difficulty lies in how to represent and manipulate variable bindings. This often imposes extra complexity both on the compiler writer and the verification effort.

In this paper, we show how Beluga's dependent contextual types let us use 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. 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 the compiler code stays clean and we do not need extra lemmas about binder manipulation. Our work demonstrates that HOAS encodings offer substantial benefits to certified programming.

Scope and type safety of the code transformations are statically guaranteed, and our implementation nicely mirrors the paper proof of type preservation. It can hence be seen as an encoding of the proof which happens to be executable as well.

1. INTRODUCTION

Formal methods are particularly important for compilers, since any compiler bug could potentially completely defeat all the efforts put into proving properties of your program.

For this reason, while mainstream compilers still rely mostly on extensive test suites, there have been many efforts at developing other ways to ensure the proper behaviour of compilers. Proving actual correctness of a compiler is a very large undertaking. Even just formalizing what it means for a compiler to be correct is itself a lot of work, which involves formalizing the lexer, parser, and semantics of both the source and the target languages.

While some applications do require this amount of assurance, our work focuses on a simpler problem, which is to prove that the compiler preserves the types. This is an interesting compromise in the design space, because on the one hand we believe type-preservation can be proved at a reasonable cost, and on the other hand this property is sufficient to cover the vast majority of cases, namely those programs where the only property proved and thus preservable is that it is properly typed.

Earlier work in this area started with so-called *typed intermediate representations*, which amounted to *testing* type-preservation throughout the execution of the compiler [Tarditi et al., 1996; Shao and Appel, 1995; Benton et al., 1998]. This came at a significant runtime cost, associated with engineering costs to try and keep this runtime cost under control [Shao et al., 1998]. More recent work instead tries to write the compiler in such a way that type-preservation can be *verified* statically [Chlipala, 2008; Guillemette and Monnier, 2008]. One of the main difficulties when implementing code transformations relates to the representation of binders in the source and target languages: in order for the verification tool to understand how bindings are manipulated, the compiler typically needs to use techniques such as de Bruijn indices, which tend to be very inconvenient and make the code harder to understand and debug.

A solution to this last problem is to use higher-order abstract syntax (HOAS), an elegant and high-level representation of bindings in which binders are represented in the object languages as binders in the meta-language. This eliminates the risk of scoping errors, and saves the programmer the trouble to write the usual α-renaming and the notoriously delicate 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. For example the closure conversion phase of a compiler has generally been considered challenging to implement with HOAS as we must calculate and reason about the free variables of a term.

In this work, we show how the rich type system and abstraction mechanisms of the dependently-typed language Beluga [Pientka and Dunfield, 2010; Cave and Pientka, 2012; Pientka and Cave, 2015] enable us to implement a type and scope preserving compiler for the simply-typed lambda-calculus using HOAS for its stages, including translation to continuation-passing style (CPS), closure conversion, and hoisting. This hence also demonstrates that HOAS does support elegant implementations of program transformations such as closure conversion.

Beluga is a dependently-typed proof and programming environment which provides a sophisticated infrastructure for implementing languages and formal systems based on the logical framework LF [Harper et al., 1993]. This allows programmers to uniformly specify the syntax, inference rules, and derivation trees using higher-order abstract syntax (HOAS) and relieves users from having to build up a common infrastructure to manage variable binding, renaming, and (single) substitution. Beluga provides in addition support for first-class contexts [Pientka, 2008] and simultaneous substitutions [Cave and Pientka, 2013], two common key concepts that frequently arise in practice. Compared to existing approaches, its infrastructure is one of the most advanced for prototyping formal systems [Felty et al., 2015].

While Beluga's infrastructure for specifying languages and formal systems is very expressive, the proof and programming language used to analyze and manipulate HOAS trees that depend on assumptions remains simple. In terms of proof-theoretical strength, Beluga provides a first-order proof language with inductive definitions [Cave and Pientka, 2012] and domain-specific induction principles [Pientka and Abel, 2015]. However instead of inductively reasoning about simple domains such as natural numbers or lists, Beluga supports representing, analyzing and manipulating HOAS trees that may depend on assumptions.

There are two key ingredients we crucially rely on in our work of encoding type-preserving program transformations: First, we encode our source and target languages using HOAS within the logical framework LF, 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 exploit BELUGA's ability to analyze and manipulate HOAS trees that depend on a context of assumptions using pattern matching. In fact this is crucial to characterize abstract syntax trees with free variables, and to manipulate and rearrange open code fragments. Both these ingredients provide an elegant conceptual framework to tackle code transformations which re-arrange (higher-order) abstract syntax trees.

Taking advantage of BELUGA's support for dependent types, our code uses the technique of intrinsic typing. This means that rather than writing the compilation phases and separately writing their type preservation proof, we write a single piece of code which is both the compiler and the proof of its type preservation. If we look at it from the point of view of a compiler writer, the code is made of fairly normal compilation phases manipulating normal abstract syntax trees, except annotated with extra type annotations. But if we look at it from the point of view of formal methods, what seemed like abstract syntax trees are actually encoding typing derivations, and the compilation phases are really encoding the proofs of type preservation. In a sense, we get the proof of type preservation "for free", although in reality it does come at the cost of extra type annotations. This kind of technique is particularly beneficial when the structure of the proof mirrors the structure of the program, as is the case here.

In the rest of this article, we first present our source code and how to encode it in BELUGA, and then for each of the three compilation phases we present, we first show a *manual* proof of its type preservation and then show how that proof is translated into an *executable* compilation phase in BELUGA. The full development is available online at `http://complogic.cs.mcgill.ca/beluga/cc-code`.

2. SOURCE LANGUAGE:

We present, in this section, the source language for our compiler as well as its encoding in Beluga. All our program transformations share the same source language, a simply typed lambda calculus extended with tuples, selectors fst and rst, let-expressions and unit written as () (see Fig. 1).

We represent n-ary tuples as nested binary tuples and unit, i.e. a triple for example is represented as $(M_1, (M_2,\ (M_3, ())\,)\,)$. In particular, environments arising during the closure conversion phase will be represented as n-ary tuples.

Dually, n-ary product types are constructed using the binary product $T \times S$ and unit; the type of a triple can then be described as $T_1 \times (T_2 \times (T_3 \times \mathsf{unit}))$. Foreshadowing closure conversion and inspired by the type language of Guillemette and Monnier [2007], we add a special type code $S\ T$. This type only arises as a

(Type)	T, S	$::=$	$S \to T \mid \mathsf{code}\ S\ T \mid T \times S \mid \mathsf{unit}$
(Source)	M, N	$::=$	$x \mid \mathsf{lam}\, x.\, M \mid M\ N \mid \mathsf{fst}\ \ M \mid \mathsf{rst}\ \ M \mid (M_1, M_2) \mid \mathsf{let}\ x = N\ \mathsf{in}\ M \mid ()$
(Context)	Γ	$::=$	$\cdot \mid \Gamma, x : T$

Fig. 1. Syntax of the source language

$\boxed{\Gamma \vdash M : T}$ Source term M has type T in context Γ

$$\frac{\Gamma, x : T \vdash M : S}{\Gamma \vdash \mathsf{lam}\, x.\, M : T \to S}\ \mathsf{t\_lam} \qquad \frac{\Gamma \vdash M : T \to S \quad \Gamma \vdash N : T}{\Gamma \vdash M\ N : S}\ \mathsf{t\_app}$$

$$\frac{\Gamma \vdash M : T \quad \Gamma, x : T \vdash N : S}{\Gamma \vdash \mathsf{let}\ x\ =\ M\ \mathsf{in}\ N : S}\ \mathsf{t\_let} \qquad \frac{\Gamma \vdash M : T \times S}{\Gamma \vdash \mathsf{fst}\ M : T}\ \mathsf{t\_first} \qquad \frac{\Gamma \vdash M : T \times S}{\Gamma \vdash \mathsf{rst}\ M : S}\ \mathsf{t\_rest}$$

$$\frac{\Gamma \vdash M : T \quad \Gamma \vdash N : S}{\Gamma \vdash (M, N) : T \times S}\ \mathsf{t\_cons} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \mathsf{t\_var} \qquad \frac{}{\Gamma \vdash () : \mathsf{unit}}\ \mathsf{t\_unit}$$

Fig. 2. Typing rules for the source language

result of closure conversion. However since the type language will remain unchanged during all our transformations, we include it from the beginning, although there are no source terms of type $\mathsf{code}\ S\ T$.

The typing rules for the source language are given in Fig. 2 and are standard. The lambda-abstraction $\mathsf{lam}\, x.\, M$ is well-typed, if M is well-typed in a typing context extended with a typing assumption for x. The application $M\ N$ has type S, if the source term M has the function type $T \to S$ and N has type T. The let-expression $\mathsf{let}\ x = M\ \mathsf{in}\ N$ is well-typed at type S, if M has some type T and N has type S in a context extended with the typing assumption $x : T$. Variables are well-typed, if a typing assumption for them is present in the context. Selectors $\mathsf{fst}\ M$ and $\mathsf{rst}\ M$ are well-typed, if M has a product type $T \times S$.

2.1 Representing the Source Language in LF

The first obvious question when trying to encode the source language in a programming environment is how to represent the binders present in the lambda-abstraction $\mathsf{lam}\, x.\, M$ and the let-expression $\mathsf{let}\ x = M\ \mathsf{in}\ N$. We encode the source language in the logical framework LF [Harper et al., 1993] which allows us to take advantage of higher-order abstract syntax (HOAS), a technique where we model binders in our object language (for example the variable x in $\mathsf{lam}\, x.\, M$) using the binders in the logical framework LF. In other words, variables bound by lambda-abstraction and let-expressions in the object-language will be bound by λ-abstraction in the meta-language. As a consequence, we inherit α-renaming and term-level substitution. HOAS encodings relieve users from building up a common infrastructure for dealing with variables, assumptions, and substitutions.

In Beluga's concrete syntax, the *kind* `type` declares an LF type family (see Fig. 3). In particular, we declare the type family `tp` together with constructors `nat`, `arr`, `code`, `cross`, and `unit` to model the types in our object language. This is unsurprising. For representing source terms we use an intrinsically typed representation: by indexing `source` terms with their types, we represent typing derivations of terms rather than terms themselves, such that we only manipulate well-typed source terms. An LF term of type `source` T in a context Γ, where T is a `tp`, corresponds to a typing derivation $\Gamma \vdash M : T$. We note that in LF itself, the context Γ is ambient and implicit. However, as we will see shortly, we can pack the LF object `source T` together with the context Γ in which it is meaningful forming a contextual object [Nanevski et al., 2008] which can then be manipulated and analyzed via

```
LF tp: type =
| nat   : tp
| arr   : tp → tp → tp
| code  : tp → tp → tp
| cross : tp → tp → tp
| unit  : tp
;

LF 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;
```

Fig. 3. Encoding of the source language in LF

pattern matching in Beluga.

We model lambda-abstractions in the source language by the constructor `lam` which takes as argument an LF-abstraction of type `source S → source T`. Hence, the source term $\mathsf{lam}\, x.\, \mathsf{lam}\, y.\, x\ y$ is represented as `lam (λx.lam (λy.app x y))` where `λ` is an LF binder. Similarly, we represent the let-expression $\mathsf{let}\ x = M\ \mathsf{in}\ N$ in the source language by the constructor `letv` which takes in two arguments, the representation of the term M and the body of the let-expression N. The fact that the body of the let-expression depends on the variable x is enforced by giving it the LF-function type `source S → source T`. Hence the source term $\mathsf{lam}\, x.\, \mathsf{lam}\, y.\, \mathsf{let}\ w = x\ y\ \mathsf{in}\ x\ w$ is represented as `lam (λx.lam (λy.letv (app x y) (λw.app x w)))`. We encode n-ary tuples as lists, using constructors `cons` and `nil`, the latter doubling as representation of $()$. We represent selectors fst and rst using the constructor `fst` and `rst` respectively. We prefer this representation to one that would group an arbitrary number of terms in a tuple for simplicity: as we index our terms with their types, and without computation in types, such tuples would have to carry additional proofs of the well-typedness of tuples and of their projections, which, while possible, would be cumbersome.

When manipulating such objects, we often have to manipulate open subterms, which requires some way to clarify the context in which they are meant to be used. Beluga represents and embeds such open fragments using the notions of contextual objects and first-class contexts. A contextual object, written as `[Γ ⊢ M]`, characterizes an open LF object M which may refer to the bound variables listed in the context Γ. Correspondingly, the contextual type `[Γ ⊢ A]` classifies the contextual objects `[Γ ⊢ M]` where M has type A in the context Γ. 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. Both these ingredients provide an elegant conceptual framework to tackle code transformations which re-arrange (higher-order) abstract syntax trees.

Since contexts are first-class, we can also pattern match on them and for example determine which variables indeed occur in M. This is critical in our implementation of closure conversion.

3. CONTINUATION PASSING STYLE

As a first program transformation, we consider translating a direct-style source language into a continuation-passing style (CPS) target language. This is typically the first step in a compiler for functional languages. Continuation-passing style means that the control flow of the program is passed explicitly to functions, as an extra argument. This argument, called a continuation, will consume the result of the function before proceeding with the execution of the program. In essence, translating a program into continuation-passing style moves all function calls to a tail position and enables further analysis and optimizations. Our continuation-passing style transformation algorithm is adapted from Danvy and Filinski [1992].

3.1 Target Language

Let us first review the language targeted by our CPS transformation.

$$\begin{array}{llcl}
\text{(Value)} & V & ::= & x \mid \mathsf{lam}\,(x,k).\,P \mid (V_1, V_2) \mid () \\
\text{(Expression)} & P, Q & ::= & V_1\ V_2\ K \mid \mathsf{let}\ x = V\ \mathsf{in}\ P \\
 & & \mid & \mathsf{let}\ x = \mathsf{fst}\ V\ \mathsf{in}\ P \mid \mathsf{let}\ x = \mathsf{rst}\ V\ \mathsf{in}\ P \mid K\ V \\
\text{(Continuation)} & K & ::= & k \mid \lambda x.P \\
\text{(Context)} & \Delta & ::= & \cdot \mid \Delta, x : T \mid \Delta, k \perp_T
\end{array}$$

$\boxed{\Delta \vdash V : T}$ Value V has type T in context Δ

$$\frac{x : T \in \Delta}{\Delta \vdash x : T}\ \mathsf{t\_kvar} \qquad \frac{\Delta, x : T, k \perp_S \vdash P \perp}{\Delta \vdash \mathsf{lam}\,(x,k).\,P : T \to S}\ \mathsf{t\_klam}$$

$$\frac{\Delta \vdash V_1 : S \quad \Delta \vdash V_2 : T}{\Delta \vdash (V_1, V_2) : S \times T}\ \mathsf{t\_kcons} \qquad \frac{}{\Delta \vdash () : \mathsf{unit}}\ \mathsf{t\_knil}$$

$\boxed{\Delta \vdash P \perp}$ Expression P is well-formed in context Δ

$$\frac{\Delta \vdash V_1 : S \to T \quad \Delta \vdash V_2 : S \quad \Delta \vdash K \perp_T}{\Delta \vdash V_1\ V_2\ K \perp}\ \mathsf{t\_kapp} \qquad \frac{\Delta \vdash V : T \quad \Delta, x : T \vdash P \perp}{\Delta \vdash \mathsf{let}\ x = V\ \mathsf{in}\ P \perp}\ \mathsf{t\_klet}$$

$$\frac{\Delta \vdash V : S \times T \quad \Delta, x : S \vdash P \perp}{\Delta \vdash \mathsf{let}\ x = \mathsf{fst}\ V\ \mathsf{in}\ P \perp}\ \mathsf{t\_kfst} \qquad \frac{\Delta \vdash V : S \times T \quad \Delta, x : T \vdash P \perp}{\Delta \vdash \mathsf{let}\ x = \mathsf{rst}\ V\ \mathsf{in}\ P \perp}\ \mathsf{t\_ksnd}$$

$\boxed{\Delta \vdash K \perp_T}$ Continuation K expects as input values of type T in context Δ

$$\frac{k \perp_T \in \Delta}{\Delta \vdash k \perp_T}\ \mathsf{t\_kvark} \qquad \frac{\Delta, x : T \vdash P \perp}{\Delta \vdash \lambda x.P \perp_T}\ \mathsf{t\_klamk}$$

Fig. 4. Typing Rules for the Target Language of CPS

The target language is divided into values, expressions, and continuations and we define typing judgments for each. Values consist of variables, lambda-abstractions and tuples of values. They are typed using judgement $\Delta \vdash V : T$, describing that a value V has type T in the typing context Δ. The typing context Δ contains typing assumptions for values $x : T$, where the variable x stands for a value of type T, and for continuations $k \perp_T$, where the variable k stands for a well-typed

$$
\begin{array}{llllll}
\llbracket x \rrbracket_k & = & k\ x & & & \\
\llbracket \mathsf{lam}\, x.\, M \rrbracket_k & = & k\ (\mathsf{lam}\,(x, k_1).\, P) & \text{where} & \llbracket M \rrbracket_{k_1} & = P \\
\llbracket M_1\ M_2 \rrbracket_k & = & [(\lambda p.[(\lambda q.p\ q\ k)/k_2]Q)/k_1]P & \text{where} & \llbracket M_1 \rrbracket_{k_1} & = P \\
 & & & \text{and} & \llbracket M_2 \rrbracket_{k_2} & = Q \\
\llbracket (M, N) \rrbracket_k & = & [(\lambda p.[(\lambda q.k\ (p, q))/k_2]Q)/k_1]P & \text{where} & \llbracket M \rrbracket_{k_1} & = P \\
 & & & \text{and} & \llbracket N \rrbracket_{k_2} & = Q \\
\llbracket \mathsf{let}\ x = N\ \mathsf{in}\ M \rrbracket_k & = & [(\lambda q.\mathsf{let}\ x = q\ \mathsf{in}\ P)/k_2]Q & \text{where} & \llbracket M \rrbracket_{k} & = P \\
 & & & \text{and} & \llbracket N \rrbracket_{k_2} & = Q \\
\llbracket \mathsf{fst}\ \ M \rrbracket_k & = & [(\lambda p.\mathsf{let}\ x = \mathsf{fst}\ p\ \mathsf{in}\ k\ x)/k_1]P & \text{where} & \llbracket M \rrbracket_{k_1} & = P \\
\llbracket \mathsf{rst}\ \ M \rrbracket_k & = & [(\lambda p.\mathsf{let}\ x = \mathsf{rst}\ p\ \mathsf{in}\ k\ x)/k_1]P & \text{where} & \llbracket M \rrbracket_{k_1} & = P \\
\llbracket () \rrbracket_k & = & k\ () & & &
\end{array}
$$

Fig. 5. CPS Algorithm

continuation of the type $\bot_T$. Continuations are formed either by variables k or lambda-abstractions $\lambda x.P$. We think of continuations here as meta-level functions and we eagerly reduce any β-redexes arising from applying a continuation.

The typing rules for values are straightforward and resemble the ones for source terms, with the exception of t_lam: lambda-abstractions of the target language take a continuation as an additional argument. We say that the target term $\mathsf{lam}\,(x, k).\, P$ has type $T \to S$, if P is a well-formed expression which may refer to a variable x of type T and a continuation variable k denoting a well-formed continuation expecting values of type S as input.

Expressions P are typed using the judgment $\Delta \vdash P\ \bot$; it simply states that an expression P is well-formed in the context Δ. By writing $P\ \bot$ in the judgment we represent the fact that the expression P does not return anything by itself but instead relies on the continuation to carry on the execution of the program. Expressions include application of a continuation $K\ V$, application of a function $V_1\ V_2\ K$, a general let-construct, $\mathsf{let}\ x = V\ \mathsf{in}\ P$ and two let-constructs to observe lists, $\mathsf{let}\ x = \mathsf{fst}\ V\ \mathsf{in}\ P$ and $\mathsf{let}\ x = \mathsf{rst}\ V\ \mathsf{in}\ P$.

Finally, we define when a continuation K is well-typed using the following judgment $\Delta \vdash K \bot_T$: it must be either a continuation variable or a well-typed lambda-abstraction expecting an input value of type T in the context Δ. By choosing to represent continuations here using lambda-abstractions we are already foreshadowing our encoding of continuations in the logical framework LF. As we will see, continuations will be modelled using the LF function space.

3.2 CPS Algorithm

We describe the translation to continuation-passing style following Danvy and Filinski [1992] in Fig. 5 using the function $\llbracket M \rrbracket_k = P$ which takes as input a source term M and produces a target term P depending on k, where k is a (fresh) variable standing for the top-level continuation in the translated expression.

In the variable and unit case, we simply call the continuation. To translate the lambda-abstraction $\mathsf{lam}\, x.\, M$ with the continuation k, we translate M using a new continuation k_1 to obtain a term P and then call the continuation k with the term $\mathsf{lam}\,(x, k_1).\, P$.

The interesting cases are the ones for applications, pairs, projections, and let-expressions. Consider translating the application $M_1\ M_2$ given a continuation k:

we first recursively translate M_i using a new continuation k_i and obtain resulting terms P and Q. We now need to create a term which only depends on the original continuation k. This is accomplished by replacing any occurrence of k_2 in Q with the meta-level function $\lambda q.p\ q\ k$ resulting in a term Q' and replacing any occurrence of k_1 in P with $\lambda p.Q'$. We eagerly reduce meta-level β-redexes arising from the application of these substitutions in the algorithm. This amounts to eliminating administrative redexes *on the fly*. The remaining cases follow a similar principle.

We now give the proof that the given transformation preserves types. While this proof is standard, we give it in detail, since it will serve as a guide for explaining our implementation of CPS translation as a function in BELUGA. In particular, we draw attention to structural lemmas, such as weakening and exchange, which are often used silently in paper proofs, but can pose a challenge when mechanizing and implementing the CPS-translation together with the guarantee that types are preserved.

As the type preservation proof relies crucially on the fact that we can replace variables denoting continuations with a concrete continuation K, we first state and prove the following substitution lemma.

LEMMA 3.1. ***Substitution (Continuation in Expression)***
If $\Gamma, k\ \bot_T \vdash P\ \bot$ *and* $\Gamma \vdash K\ \bot_T$ *then* $\Gamma \vdash [K/k]P\ \bot$

PROOF. Proof by induction on the derivation $\Gamma, k\ \bot_T \vdash P\ \bot$. $\square$

Our main Theorem 3.1 states that a source expression of type T will be transformed by the algorithm given in Fig. 5 to a well-formed target expression expecting a continuation k of type $\bot_T$. The typing context of the target language *subsumes* the typing context of source term, such that we can use Γ in the conclusion, reading typing assumptions for source terms as assumptions for target values. The proof of Theorem 3.1 follows by structural induction on the typing derivation of the source term. In the proof, terms with a substitution appearing in them should be read as the terms resulting from the substitution, corresponding to the algorithm reducing continuation applications eagerly, rather than as terms with a delayed substitution.

THEOREM 3.1. ***Type Preservation*** *If* $\Gamma \vdash M : T$ *then* $\Gamma, k\ \bot_T \vdash [\![M]\!]_k\ \bot$.

PROOF. By induction on the typing derivation $\mathcal{D}_0 :: \Gamma \vdash M : T$. We consider here some representative cases.

Case $\mathcal{D}_0 = \dfrac{x : T \in \Gamma}{\Gamma \vdash x : T}\,\mathsf{t\_var}$

where $M = x$ and $[\![M]\!]_k = k\ x$.

$\Gamma \vdash x : T$	by assumption
$\Gamma, k\ \bot_T \vdash x : T$	by weakening
$\Gamma, k\ \bot_T \vdash k\ \bot_T$	by t_kvark
$\Gamma, k\ \bot_T \vdash k\ x\ \bot$	by t_kappk
$\Gamma, k\ \bot_T \vdash [\![M]\!]_k\ \bot$	by definition

Case $\mathcal{D}_0 = \dfrac{\begin{array}{c}\mathcal{D}\\ \Gamma, x : T \vdash M' : S\end{array}}{\Gamma \vdash \mathsf{lam}\, x.\, M' : T \to S}\,\mathsf{t\_lam}$

where $M = \mathsf{lam}\, x.\, M'$ and $[\![M]\!]_k = k\ (\mathsf{lam}\,(.\, x, k)\, [\![M']\!]_k)$.

$\Gamma, x : T, k_1\ \bot_S \vdash [\![M']\!]_{k_1}\ \ \bot$	by i.h. on $\mathcal{D}$
$\Gamma \vdash \mathsf{lam}\,(x, k_1).\ [\![M']\!]_{k_1} : T \to S$	by t_klam
$\Gamma, k\ \bot_{T\to S} \vdash \mathsf{lam}\,(x, k_1).\ [\![M']\!]_{k_1} : T \to S$	by weakening
$\Gamma, k\ \bot_{T\to S} \vdash k\ \bot_{T\to S}$	by t_kvark
$\Gamma, k\ \bot_{T\to S} \vdash k\ (\mathsf{lam}\,(x, k_1).\ [\![M']\!]_{k_1})\ \bot$	by t_kappk
$\Gamma, k\ \bot_{T\to S} \vdash [\![\mathsf{lam}\, x.\, M']\!]_k\ \ \bot$	by definition

$$\textit{Case } \mathcal{D}_0 = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Gamma \vdash M' : S \to T & \Gamma \vdash N : S\end{array}}{\Gamma \vdash M'\ N : T}\ \mathsf{t\_app}$$

where $M = M'\ N$ and $[\![M]\!]_k = [\lambda p.[\lambda q.p\ q\ k/k_2]\, [\![N]\!]_{k_2} /k_1]\, [\![M']\!]_{k_1}$.

$\Gamma, k_2\ \bot_S \vdash [\![N]\!]_{k_2}\ \ \bot$	by i.h. on $\mathcal{D}_2$
$\Gamma, k\ \bot_T, p : S \to T, q : S \vdash p : S \to T$	by t_kvar
$\Gamma, k\ \bot_T, p : S \to T, q : S \vdash q : S$	by t_kvar
$\Gamma, k\ \bot_T, p : S \to T, q : S \vdash k\ \bot_T$	by t_kvark
$\Gamma, k\ \bot_T, p : S \to T, q : S \vdash p\ q\ k\ \bot$	by t_kapp
$\Gamma, k\ \bot_T, p : S \to T \vdash \lambda q.p\ q\ k\ \bot_S$	by t_klamk
$\Gamma, k\ \bot_T, p : S \to T, k_2\ \bot_S \vdash [\![N]\!]_{k_2}\ \ \bot$	by weakening
$\Gamma, k\ \bot_T, p : S \to T \vdash [\lambda q.p\ q\ k/k_2]\, [\![N]\!]_{k_2}\ \ \bot$	by substitution (Lemma 3.1)
$\Gamma, k\ \bot_T \vdash \lambda p.[\lambda q.p\ q\ k/k_2]\, [\![N]\!]_{k_2}\ \ \bot_{S\to T}$	by t_klamk
$\Gamma, k_1\ \bot_{S\to T} \vdash [\![M']\!]_{k_1}\ \ \bot$	by i.h. on $\mathcal{D}_1$
$\Gamma, k\ \bot_T, k_1\ \bot_{S\to T} \vdash [\![M']\!]_{k_1}\ \ \bot$	by weakening
$\Gamma, k\ \bot_T \vdash [\lambda p.[\lambda q.p\ q\ k/k_2]\, [\![N]\!]_{k_2} /k_1]\, [\![M']\!]_{k_1}\ \ \bot$	by substitution (Lemma 3.1)
$\Gamma, k\ \bot_T \vdash [\![M'\ N]\!]_k\ \ \bot$	by definition □

3.3 Representing the Target Language in LF

As with the source language (see Section 2), we encode the target language in the LF logical framework as intrinsically well-typed terms (see Fig. 6). We reuse `tp`, the type index of the `source` language, to index elements of type `value`, which correspond to values. An LF term of type `value` T in a context Γ, where T is a `tp`, corresponds to a typing derivation $\Gamma \vdash V : T$ for a (unique) value V in the target language. Similarly, we use the type `exp` here to represent well-typed CPS-expressions: an LF term of type `exp` corresponds directly to a typing derivation $\Gamma \vdash E\ \bot$ for a (unique) expression E. Continuations are not present at this level; they are directly represented as LF functions from values to expressions written as `value T → exp`.

3.4 Implementation of the Main Theorem

We now implement the translation $[\![-]\!]_k$ on intrinsically well-typed expressions as a recursive function in Beluga. Its development mirrors the type preservation proof and may be viewed as an executable version of it. A central challenge lies in the fact that our CPS translation (see Fig. 5) is defined on open terms M, meaning that they may contain free variables. Since we aim to implement the translation on intrinsically well-typed terms, M must be well-typed in the context

```
LF exp : type =
 | kapp    : value (arr S T) → value S → (value T → exp) → exp
 | klet    : value T → (value T → exp) → exp
 | klet-fst: value (cross S T) → (value S → exp) → exp
 | klet-rst: value (cross S T) → (value T → exp) → exp

and value : tp → type =
 | klam    : (value S → (value T → exp) → exp) → value (arr S T)
 | kcons   : value S → value T → value (cross S T)
 | knil    : value unit ;
```

Fig. 6. Encoding of the Target Language of CPS in LF

Γ, i.e. all the free variables in M must be declared in Γ. Similarly, P, the result of our CPS-translation, is open and is well-typed in a context Γ' which stands in a one-to-one relation to Γ, i.e. for every source variable declared in Γ there is a corresponding target variable in Γ'. Note that in the type preservation proof (Theorem 3.1) we were cheating a little by conflating the source and target context. In a mechanization however we are forced to be more precise.

Because every source variable corresponds to exactly one target variable, we can give a more uniform view of the context in which both the source and target term are well-typed, namely as a context containing pairs of variables of type `source T` and `value T`. This could be for example captured informally as follows where x stands for a source variable and y for a target variable of the same type.

$$\text{(Context)}\ \ \Delta \ ::= \ \cdot \mid \Delta, (x : T, y : T)$$

As an alternative we could have stated the relationship between the source and target contexts explicitly, but we then would need to separately establish the fact that the i-th declaration in the source context corresponds exactly to the i-th declaration in the target context. Using a joint context where we pair assumptions about source and target variables, we obtain this correspondance for free.

In Beluga, we define the type of this joint context using a schema, more precisely we define the type each declaration in the context must satisfy. The keyword **block** defines a variable declaration in the context that stands for a pair. So we use it to associate our source and target variables and we explicitly quantify over the shared type using the keyword **some**:

```
schema ctx = some [t:tp] block x:source t, y:value t;
```

Contexts of schema `ctx` are ordered sequences of declarations where each variable declaration in the context is an instances of the given schema. Here are a few examples.

`b1:block (x1:source nat, y1:value nat),` `b2:block (x2:source (arr nat nat), y2:value (arr nat nat))`	✓
`b1:block (x:source nat, y:value nat),` `b2:block (x:source unit, y:value unit)`	✓
`b1:block (x1:source nat, y1:value unit),` `b2:block (x2:source unit, y2:value unit)`	↯
`b1:block (x1:source nat, y1:value nat),` `x2:source (arr nat nat), y2:value (arr nat nat)`	↯

The first two contexts are well-formed according to the given schema. Each element is an instance of the schema declaration. The last two contexts are not well-formed according to the given schema: in the first counter example the elements in the block do not share the same type and in the second counter example the context declarations are not grouped into pairs (i.e. blocks).

Beluga's schema declarations are similar to world declarations in Twelf [Pfenning and Schürmann, 1999]. However, there is a subtle distinction: in Beluga a schema declaration simply declares the type of a context; we do not verify that a given LF type satisfies this context. Instead we use the type of a context to express invariants about contextual object and their types when we pack an LF term together with the context in which it is meaningful.

Our CPS translation function `cpse`, presented in Fig. 7, takes as input a source term M of type `source S` in a joined context Γ and returns a well-typed expression which depends on Γ and the continuation `k` of type $\bot_S$. This is stated in Beluga as follows:

```
rec cpse : (Γ:ctx)[Γ ⊢ source S[]] → [Γ, k: value S[] → exp ⊢ exp]
```

where `[Γ ⊢ source S[]]` is the *contextual type* describing the type and scoping of the source argument. Similarly, `[Γ, k: value S[] → exp ⊢ exp]` describes how the output can depend on Γ and on the continuation `k` of type $\bot_S$.

By writing `(Γ:ctx)` we quantify over Γ in the given type and state the schema Γ must satisfy. The type checker will then guarantee that the function `cpse` only manipulates contexts of schema `ctx` and we are only considering source terms in such a context. By wrapping the context declaration in round parenthesis we express at the same time that Γ remains implicit in the use of this function, where curly brackets would denote an explicit dependent argument. In other words, `(Γ:ctx)` is simply a type annotation stating the schema the context Γ must have, but we do not need to pass an instantiation for Γ explicitly to the function `cpse` when making a function call.

We refer to variables occurring inside a contextual object (such as the `S` in `[Γ ⊢ source S[]]` above) as meta-variables to distinguish them from variables bound by function abstraction in the actual program. All meta-variables are associated with a postponed substitution which can be omitted, if the substitution is the identity. In stating the type of the function `cpse` we however want to make sure that the type `S` of the expression is closed. We therefore associate it with a weakening substitution, written as `[]`, which weakens the closed type `S` (i.e. `⊢ S:tp`). In other words `[]` is a substitution from the empty context to the context Γ.

We further note that `S` was free in the type annotation for `cpse`. Beluga's type reconstruction [Pientka, 2013; Ferreira and Pientka, 2014] will infer the type of `S` as `[⊢ tp]` which can be read as "`S` has type `tp` in the empty context".

Our implementation of the CPS translation (see Fig. 7) consists of a single downward pass on the input `source` program. It follows closely the proof of type preservation (Theorem 3.1) for the algorithm. The case analysis on the typing derivation in the proof corresponds to the case analysis via pattern matching on the intrinsically well-typed source expression. The appeals to the induction hypothesis in the proof correspond to the recursive calls in our program.

Let us look at the program more closely. The first pattern, `#p.1` matches the first

```
rec cpse : (Γ:ctx)[Γ ⊢ source S[]] → [Γ, k: value S[] → exp ⊢ exp]
fn e ⇒ case e of
| [Γ ⊢ #p.1] ⇒ [Γ, k:value _ → exp ⊢ k (#p.2[…])]
| [Γ ⊢ app M N] ⇒
  let [Γ, k1:value (arr T[] S[]) → exp ⊢ P] = cpse [Γ ⊢ M] in
  let [Γ, k2:value T[] → exp           ⊢ Q] = cpse [Γ ⊢ N] in
    [Γ, k:value S[] → exp ⊢ P […, (λf. Q[…, (λx. kapp f x k)])]]
| [Γ ⊢ lam λx. M] ⇒
   let [Γ, x:source S ⊢ M] = [Γ, x:source _ ⊢ M] in
   let [Γ, b:block (x:source S[], y:value S[]), k1:value T[]→ exp
        ⊢ P[…,b.2, k1] =
          cpse [Γ, b:block (x:source S[], y:value S[]) ⊢ M […, b.1] in
      [Γ, k:value (arr S[] T[]) → exp ⊢ k (klam (λx.λk1. P[…, x, k1]))]
...;
```

Fig. 7. Implementation of CPS in Beluga

field of elements of Γ, corresponding to source variables. Formally, `#p` describes a variable of type `[Γ ⊢ block (x:source T[], y:value T[]])` for some closed type `T` and `#p.1` extracts the first element. As mentioned above, meta-variables appearing in contextual objects are associated with a substitution describing which part of the context they may depend on. For convenience, Beluga allows users to omit writing the identity substitution stating that the meta-variable depends on the whole context Γ. Therefore, we simply write `#p.1` in the previous pattern. Alternatively, we could have written `#p.1[…]`, where … represents the identity substitution over Γ, making explicit the fact that a given parameter variable `#p` may depend on the declarations from the context Γ.

When we encounter a source variable `#p.1`, we apply the continuation `k` to the corresponding target variable. Since we have paired up source and target variables in Γ and `#p` describes such a pair, we simply call `k` with the second field of `#p`. Type reconstruction fills in the `_` in the type of `k` with the type of the matched variable. Note that we apply the weakening substitution `[..]` to `#p.2`, as we return a target term in the context `Γ, k:value _ → exp`.

In the application case, we match on the pattern `app M N` and recursively transform `M` and `N` to target terms `P` and `Q` respectively. We then substitute for the continuation variable `k2` in `Q` a continuation consuming the local argument of an application. A continuation is then built from this, expecting the function to which the local argument is applied and substituted for `k1` in `P` producing a well-typed expression, if a continuation for the resulting type `S` is provided.

We take advantage of Beluga's built-in substitution here to model the substitution operation present in our algorithm in Fig. 5. The term `(λx. kapp f x k)` that we substitute for references to `k2` in `Q` will be β-reduced wherever that `k2` appears in a function call position, such as the function calls introduced by the `#p.1` case. We hence reduce administrative redexes using the built-in LF application.

Note that in the type preservation proof we rely on weakening to justify applying the substitution lemma. In our implementation we also rely on weakening although it is subtle to see, since Beluga's type system silently takes care of it. Weakening is used to justify using the target term `Q` which is well-typed in `Γ, k2:value T[] → exp` in the target term `(λf. Q[…, (λx. kapp f x k)])` which is well-typed in the context `Γ, k:value S[] → exp`.

In the λ-abstraction case, we want to recursively translate `M` which is well-typed in the context `Γ, x:source S[]`. Note that from the pattern `Γ ⊢ lam λx.M` we only know `[Γ, x: source _ ⊢ M]` where the underscore makes explicit the fact that we do not yet have the name `S` for the type of `x`. We explicitly introduce a name for the type of `x` by pattern matching on `[Γ, x: source _ ⊢ M]` and then recursively translate `M` in the context `Γ, b:block (x:source S[],y:value S[])`. Note that introducing a name `S` is necessary in Beluga, since Beluga's type reconstruction presently does not take into account dependencies among blocks of declarations and hence fails to infer the type for `y`. Moreover, we note that while `M` was well-typed in the context `Γ, x: source S[]`, when making the recursive call, we want to use it in the context `Γ, b:block (x:source S[], y:value S[])`. We hence associate `M` with a weakening substitution which maps `x` to the first part of the block `b`. The result of recursively translating `M` is described as a target expression `P` which depends on the continuation `k1` and the target variable given by the second part of the block. In other words, `P` is well-typed in `Γ, y:value S[], k1:value T[]→ exp`. In general, `P` could depend on both variables declared in the block `b`, but because a target expression can never depend on source variables restricting `P` is justified. This is verified in Beluga by the coverage checker which takes into account strengthening based on subordination which constructs a dependency graph for type families (see for a similar analysis [Virga, 1999; Harper and Licata, 2007]). Using this dependency graph, we can justify that an object M of type A cannot depend on some objects of type B, i.e. there is no way of constructing M using an assumption of type B. Hence, if a term M was well-typed in a context that contained also declarations of type B, we can always strengthen M to be well-typed in a context where we drop all the assumptions of type B.

We return `[Γ, k:value (arr S[] T[]) → exp ⊢ k (klam λx.λk1.P[…,x, k1])]` as the final result. To guarantee that the term `k (klam λx.λk1. P[…, x, k1])` is well-typed, we observe that `P[…, x, k1]` is well-typed in `Γ, x:value S,k1:value T[] → exp`, the extended context, and Beluga's typing rules will silently employ weakening (by `k:value (arr S[] T[]) → exp`) to type-check `(klam λx.λk1. P[…, x, k1])` in the context `Γ, k:value (arr S[] T[]) → exp`.

The remaining cases are similar and in direct correspondence with the proof of Theorem 3.1. In the let-expressions, we rely on inferring the type of variables in a similar fashion as in the lambda-abstraction case.

3.5 Discussion

The implementation of the type-preserving transformation into continuation-passing style in Beluga, including the definition of the type, source and target languages, amounts to less than 65 lines of code. We rely crucially on the built-in support for weakening, strengthening, and substitution. Beluga's features such as pattern variables, built-in substitution and first-class contexts make for a straightforward representation of the transformation as a single function, while dependent types allow us to build the type preservation proof into the representation of the transformation with little overhead.

The fact that this algorithm can be implemented elegantly in languages supporting HOAS is not a new result. A similar implementation was given as an example in the Twelf tutorial presented at POPL 2009 [Car, 2009]. Guillemette and Monnier [2006] and Chlipala [2008], both using a term representation based on HOAS, achieve similar results, respectively in Haskell and in Coq.

We have also implemented a CPS transformation in Beluga over System F, an extension of the simply typed lambda calculus where types may depend on type variables. Few changes are required for our simply typed implementation to support System F: removing the weakening substitution (written as `[]`) associated with types so that they may depend on type variables in the context suffices as sole modification to the cases included in the implementation over a simply-typed language.

While we present here the implementation using a joint context, an alternative would be to have distinct contexts for source and target variables. We would have to carry an explicit relation stating that for every variable in the source contexts, a variable of the same type is present in the target context. This would be closer to the technique used for the other transformations presented in this paper (see Sections 4 and 5); however we would need to also explicitly infer that the i-th declaration in the source context corresponds to the i-th declaration in the target context. Carrying an explicit relation between contexts would also complicate the extension to System F, as types would have to be transported from the source to the target contexts in order to appear in different contextual objects.

4. CLOSURE CONVERSION

The second program transformation we consider is closure conversion. It 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.

In a typical compiler pipeline, the CPS transformation precedes closure conversion. However, closure conversion does not depend in any essential way on the fact that its input is continuation-passing. For clarity, we present closure conversion as a transformation on programs in direct style, i.e. not in continuation passing style.

4.1 Target Language

In addition to functions $\mathsf{lam}\, y.\, P$, function application $P\ Q$, tuples (P, Q), selectors fst and rst, and let-expressions $\mathsf{let}\ y = P\ \mathsf{in}\ Q$, our target language for closure conversion now contains two new constructs (see Fig. 8): 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 $\mathsf{let}\ \langle y_f, y_{env}\rangle = P\ \mathsf{in}\ Q$.

The 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 $\mathsf{lam}\, x.\, M$ of type $T \to S$, we return a closure $\langle \mathsf{lam}\, y_c.\, P, Q\rangle$ consisting of a closed function $\mathsf{lam}\, y_c.\, P$ and its environment Q. The variable y_c stands for a pair of the local argument y, standing in for x, and an environment variable y_{env} whose projections replace free

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

Fig. 8. Syntax of the Target Language for Closure Conversion

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

$$\frac{\Delta, x : T \vdash P : S}{\Delta \vdash \mathsf{lam}\, x.\, P : \mathsf{code}\ T\ S}\ \mathsf{t\_clam} \qquad \frac{\Delta \vdash P : \mathsf{code}\ T\ S \quad \Delta \vdash Q : T}{\Delta \vdash P\ Q : S}\ \mathsf{t\_capp}$$

$$\frac{\Delta \vdash P : T \quad \Delta, x : T \vdash Q : S}{\Delta \vdash \mathsf{let}\ x\ =\ P\ \mathsf{in}\ Q : S}\ \mathsf{t\_clet} \qquad \frac{x : T \in \Delta}{\Delta \vdash x : T}\ \mathsf{t\_cvar}$$

$$\frac{\Delta \vdash P : T \times S}{\Delta \vdash \mathsf{fst}\ P : T}\ \mathsf{t\_cfst} \qquad \frac{\Delta \vdash P : T \times S}{\Delta \vdash \mathsf{rst}\ P : S}\ \mathsf{t\_csnd} \qquad \frac{}{\Delta \vdash () : \mathsf{unit}}\ \mathsf{t\_cnil}$$

$$\frac{\cdot \vdash P : \mathsf{code}\ (T \times T_{\mathrm{env}})\ S \quad \Delta \vdash Q : T_{\mathrm{env}}}{\Delta \vdash \langle P, Q \rangle : T \to S}\ \mathsf{t\_cpack} \qquad \frac{\Delta \vdash P : T \quad \Delta \vdash Q : S}{\Delta \vdash (P, Q) : T \times S}\ \mathsf{t\_ccons}$$

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

Fig. 9. Typing Rules for the Target Language of Closure Conversion

variables of M. 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 use $T \to S$ to type the closure packages, hiding l and saving 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, with the rule t_pack, we enforce that P is closed. The remaining typing rules are similar to the typing rules for the source language (see Fig. 9).

4.2 Closure Conversion Algorithm

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

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

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 in Fig. 10 using $[\![M]\!]_\rho$, where M is a source term which is well-typed in some context Γ and ρ is a mapping of source variables in Γ to target terms in some context Δ. Intuitively, ρ maps source variables to the corresponding projection of the environment.

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

Fig. 10. Closure Conversion Algorithm

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

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

The identity, written as id, maps the empty source context to any target context (see rule m_id). We may extend the domain of a map ρ from Γ to $\Gamma, x : T$ by appending the mapping $x \mapsto P$ to ρ, where P has type T in the target context Δ, using rule m_dot.

Were the `source` and `target` languages the same, a map $\Delta \vdash \rho : \Gamma$ would be the exact encoding of substitution from context Γ to context Δ. For this reason, we refer to maps as substitutions interchangeably in the remainder of this paper. For convenience, we write π_i for the i-th projection instead of using the selectors fst and rst. For example, $\pi_2\ M$ corresponds to the term fst (rst M).

The closure conversion algorithm given in Fig. 10 translates a well-typed source term M using the map ρ. To translate a source variable, we look up its binding in the map ρ. To translate tuples and projections, we translate the subterms before reassembling the result using target language constructs. () is directly translated to its target equivalent. Translating the let-expression let $x = M$ in N involves translating M using the mapping ρ and translating N with the extended map $\rho, x \mapsto y$, therefore guaranteeing that the map provides instantiations for all the free variables in N, before reassembling the converted terms using the target let-construct. The interesting cases of closure conversion arise for λ-abstraction and application.

When translating a λ-abstraction lam $x. M$, we first compute the set $\{x_1, ..., x_n\}$ of free variables occurring in lam $x. M$. We then form a closure consisting of two parts:

(1) A term P, obtained by converting M with the new map ρ' which maps variables $x_1, ..., x_n$ to their corresponding projection of the environment y_{env} 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 $(x_1, ..., x_n)$.

When translating an application $M\ N$, we first translate M and N to target terms P and Q respectively. Since the source term M denotes a function, the target term P will denote a closure and we extract the two parts of the closure using a let-pack construct where the variable y_f describes the function and y_{env} stands for the environment. We then apply the extended environment (Q, y_{env}) to the function described by the variable y_f.

Following our earlier ideas, we implement the described algorithm in Beluga as a recursive program which manipulates intrinsically well-typed source terms. Recall that intrinsically typed terms represent typing derivations and hence, our program can be viewed as an executable transformation over typing derivations. To understand better the general idea behind our implementation and appreciate its correctness, we again discuss first 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 we discuss below. We then use this proof as a guide to explain our type-preserving implementation of closure conversion. As we will see the lemmas necessary in the type preservation proof correspond exactly to auxiliary functions needed in our implementation.

Most of the lemmas arise from proving that translating λ-abstractions preserves typing. For example when translating a λ-abstraction, we compute the set of free variables that in fact occur in the body of the abstraction. This set of free variables and their associated types form a subset of the overall typing context. To argue that types are preserved we need a strengthening lemma for source terms (see Lemma 4.1) which justifies that the body remains well-typed in the smaller context which only tracks the free variables occurring in it.

Lemma 4.1. ***Term Strengthening***
If $\Gamma \vdash M : T$ *and* $\Gamma' = FV(M)$ *then* $\Gamma' \subseteq \Gamma$ *and* $\Gamma' \vdash M : T$.

Proof. Proof using an auxiliary lemma: if $\Gamma_1, \Gamma_2 \vdash M : T$ and $\Gamma_1' = FV(M)\backslash\Gamma_2$ then $\Gamma_1' \subseteq \Gamma_1$ and $\Gamma_1', \Gamma_2 \vdash M : T$ which is proven by induction on Γ_1. □

Term Strengthening (Lemma 4.1) says that the set of typing assumption Γ' as computed by FV is sufficient to type any term M provided that this term M is well-typed in the context Γ.

The result of translating the body of the abstraction is a target term P; for it to be meaningful again in the original typing context, we will need to use weakening (see Lemma 4.2). As we will see in the implementation of our algorithm in Beluga, this form of weakening is obtained for free while the strengthening properties must be established separately. In the type preservation proof itself, we also rely on weakening source terms which allows us to weaken the tuple describing the free source variables in the body of the λ-abstraction to its original context. Hence we state two weakening lemmas.

Lemma 4.2. ***Term Weakening***

(1) *If* $\Gamma' \vdash M : T$ *and* $\Gamma' \subseteq \Gamma$ *then* $\Gamma \vdash M : T$.

(2) *If* $\Delta, \Delta' \vdash P : T$ *then* $\Delta, x : S, \Delta' \vdash P : T$.

Proof. Proof (1) using an auxiliary lemma: if $\Gamma_1', \Gamma_2 \vdash P : T$ and $\Gamma_1' \subseteq \Gamma_1$ then $\Gamma_1, \Gamma_2 \vdash P : T$ which is proven by induction on Γ_1'.
Proof of (2) is the standard lemma for weakening target terms. □

Term Weakening (Lemma 4.2) says that a source term M stays well-typed at type T if we weaken its typing context Γ to a context Γ' containing all of the assumptions in Γ. It is stated dual to the previous strengthening lemma such that first strengthening a term and then weakening it again, results in the same original source term. To prove term weakening for source terms, we need to generalize. The weakening lemma for target terms is stated in the standard way.

Since our algorithm abstracts over the free variables $x_1, \ldots, x_n$ and creates an environment as an n-ary tuple, we also need to argue that the type of the environment can always be inferred and exists. Intuitively, given the set of free variables and their types, i.e. $x_1{:}T_1, \ldots, x_n{:}T_n$, we can form the type $T_1 \times \ldots \times T_n$ together with a well-typed map which associates each x_i with the i-th projection. This is justified by the following context reification lemma.

Lemma 4.3. ***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 $\rho = x_1 \mapsto \pi_1\ y_{\text{env}}, \ldots, x_n \mapsto \pi_n\ y_{\text{env}}$ such that $y_{\text{env}} : T_1 \times \ldots \times T_n \vdash \rho : \Gamma$ and $\Gamma \vdash (x_1, \ldots, x_n) : T_1 \times \ldots \times T_n$.

Proof. By induction on Γ. □

Context Reification (Lemma 4.3) says that it is possible to represent a context Γ of typing assumptions as a single typing assumption y_{env} by creating a product type which consists of all the typing assumptions in Γ. The substitution ρ acts as a witness and transports any term meaningful in Γ to one solely referring to y_{env}.

Finally, we establish some basic lemmas about looking up an element in our mapping ρ between source variables and their corresponding target terms and about extending the mapping (see lemma 4.4 and 4.5).

Lemma 4.4. ***Map Extension***
If $\Delta \vdash \rho : \Gamma$ then $\Delta, y : T \vdash \rho, x \mapsto y : \Gamma, x : T$.

Proof. Induction on the definition of $\Delta \vdash \rho : \Gamma$. □

The Map Extension Lemma 4.4 says that we can extend any substitution ρ by the identity, mapping a source variable `x` to a new target variable of the same type. This does not follow directly from the definition of the map. Instead, it is necessary to weaken all judgments of the form $\Delta \vdash P : S$ contained in ρ by the formation rule m_dot to judgments of the form $\Delta, x : T \vdash P : S$.

Lemma 4.5. ***Map Lookup***
If $x : T \in \Gamma$ and $\Delta \vdash \rho : \Gamma$, then $\Delta \vdash \rho(x) : T$.

Proof. Induction on the definition of $\Delta \vdash \rho : \Gamma$. □

Map Lookup (Lemma 4.5) states, intuitively, that substitutions as encoded by our mapping judgment work as intended: ρ associates any variable in Γ to a term of the same type in the target context Δ.

LEMMA 4.6. ***Map Lookup (Tuple)***
If $\Gamma \vdash (x_1, \ldots, x_n) : T$ *and* $\Delta \vdash \rho : \Gamma$ *then* $\Delta \vdash (\rho(x_1), \ldots, \rho(x_n)) : T$.

PROOF. By Lemma 4.5 and inversion on the typing rules. □

Map Lookup (Tuple) (Lemma 4.6) says that applying a substitution ρ to each component of a variable tuple will transport the tuple from the domain of the substitution Γ to its codomain Δ while preserving the type of the tuple.

We now are ready to show that our closure conversion algorithm preserves types. The proof is mostly straightforward except for the λ-abstraction case which is the most difficult.

THEOREM 4.1. ***Type Preservation***
If $\Gamma \vdash M : T$ *and* $\Delta \vdash \rho : \Gamma$ *then* $\Delta \vdash [\![M]\!]_\rho : T$.

PROOF. By induction on the typing derivation $\mathcal{D}_0 :: \Gamma \vdash M : T$. We only show a few key cases. The other cases are similar.

$$\textit{Case } \mathcal{D}_0 = \cfrac{\begin{array}{c}\mathcal{D}\\ x : T \in \Gamma\end{array}}{\Gamma \vdash x : T}\ \mathsf{t\_var}$$

where $M = x$ and $\Delta \vdash \rho : \Gamma$.

$\Delta \vdash \rho(x) : T$ — by Map Lookup
$\Delta \vdash [\![x]\!]_\rho$: T — by definition

$$\textit{Case } \mathcal{D}_0 = \cfrac{\begin{array}{c}\mathcal{D}\\ \Gamma, x : T \vdash M' : S\end{array}}{\Gamma \vdash \mathsf{lam}\, x.\, M' : T \to S}\ \mathsf{t\_lam}$$

where $M = \mathsf{lam}\, x.\, M'$ and $\Delta \vdash \rho : \Gamma$.

$\Gamma' \vdash \mathsf{lam}\, x.\, M' : T \to S$ and $\Gamma' \subseteq \Gamma$ where $\Gamma' = FV(\mathsf{lam}\, x.\, M')$ — by Term Str

$\Gamma' \vdash (x_1, \ldots, x_n) : T_{\Gamma'}$ and $y_{env} : T_{\Gamma'} \vdash \rho' : \Gamma'$ — by Context Reification
$\Gamma \vdash (x_1, \ldots, x_n) : T_{\Gamma'}$ — by Term Weakening (1)
$\Delta \vdash \rho : \Gamma$ — by assumption
let $(\rho(x_1), \ldots, \rho(x_n)) = P_{env}$
$\Delta \vdash P_{env} : T_{\Gamma'}$ — by Map Lookup (tuple)
$y_{env} : T_{\Gamma'}, y : T \vdash \rho', x \mapsto y : \Gamma', x : T$ — By Map Extension
$y_{env} : T_{\Gamma'}, y : T \vdash P : S$ where $P = [\![M']\!]_{\rho', x \mapsto y}$ — by i.h. on $\mathcal{D}$
$y_c : T \times T_{\Gamma'}, y_{env} : T_{\Gamma'}, y : T \vdash P : S$ — by Term Weakening (2)
$y_c : T \times T_{\Gamma'}, y : T, y_{env} : T_{\Gamma'} \vdash P : S$ — by Exchange
$y_c : T \times T_{\Gamma'}, y : T \vdash \mathsf{let}\ y_{env} = \mathsf{rst}\ y_c\ \mathsf{in}\ P : S$ — by rule $\mathsf{t\_clet}$
$y_c : T \times T_{\Gamma'} \vdash \mathsf{let}\ y = \mathsf{fst}\ y_c\ \mathsf{in}\ \mathsf{let}\ y_{env} = \mathsf{rst}\ y_c\ \mathsf{in}\ P : S$ — by rule $\mathsf{t\_clet}$
$\cdot \vdash \mathsf{lam}\, y_c.\, \mathsf{let}\ y = \mathsf{fst}\ y_c\ \mathsf{in}\ \mathsf{let}\ y_{env} = \mathsf{rst}\ y_c\ \mathsf{in}\ P : \mathsf{code}\ (T \times T_{\Gamma'})\ S$ — by rule $\mathsf{t\_clam}$
$\Delta \vdash \langle \mathsf{lam}\, y_c.\, \mathsf{let}\ y = \mathsf{fst}\ y_c\ \mathsf{in}\ \mathsf{let}\ y_{env} = \mathsf{rst}\ y_c\ \mathsf{in}\ P\ ,\ P_{env} \rangle : T \to S$ — by rule $\mathsf{t\_cpack}$
$\Delta \vdash [\![\mathsf{lam}\, x.\, M']\!]_\rho : T \to S$ — by definition

```
LF 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
| cfst : target (cross T S) → target T
| crst : target (cross T S) → target S
| ccons: target T → target S → target (cross T S)
| cnil : target unit
| clet : target T → (target T → target S) → target S;
```

Fig. 11. Encoding of the Target Language of Closure Conversion in LF

$$\textit{Case } \mathcal{D}_0 = \dfrac{\begin{matrix}\mathcal{D}_1 \\ \Gamma \vdash M' : S \to T\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ \Gamma \vdash N : S\end{matrix}}{\Gamma \vdash M'\ N : T}\ \mathsf{t\_app}$$

where $M = M'\ N$ and $\Delta \vdash \rho : \Gamma$.

$\Delta \vdash [\![M']\!]_\rho : S \to T$	by i.h. on $\mathcal{D}_1$
$\Delta \vdash [\![N]\!]_\rho : S$	by i.h. on $\mathcal{D}_2$
$\Delta, y_{\text{env}} : l \vdash [\![N]\!]_\rho : S$	by Term Weakening (2)
$\Delta, y_{\text{env}} : l \vdash y_{\text{env}} : l$	by rule t_cvar
$\Delta, y_{\text{env}} : l \vdash ([\![N]\!]_\rho\ ,\ y_{\text{env}}) : (S \times l)$	by rule t_ccons
$\Delta, y_f : \mathsf{code}\ (S \times l)\ T, y_{\text{env}} : l \vdash ([\![N]\!]_\rho\ ,\ y_{\text{env}}) : (S \times l)$	by Term Weakening (2)
$\Delta, y_f : \mathsf{code}\ (S \times l)\ T \vdash y_f : \mathsf{code}\ (S \times l)\ T$	by rule t_cvar
$\Delta, y_f : \mathsf{code}\ (S \times l)\ T, y_{\text{env}} : l \vdash y_f : \mathsf{code}\ (S \times l)\ T$	by Term Weakening (2)
$\Delta, y_f : \mathsf{code}\ (S \times l)\ T, y_{\text{env}} : l \vdash y_f\ ([\![N]\!]_\rho\ ,\ y_{\text{env}}) : T$	by rule t_capp
$\Delta \vdash \mathsf{let}\ \langle y_f, y_{\text{env}}\rangle = [\![M']\!]_\rho\ \mathsf{in}\ y_f\ ([\![N]\!]_\rho\ ,\ y_{\text{env}}) : T$	by rule t_cletpack
$\Delta \vdash [\![M'\ N]\!]_\rho : T$	by definition □

4.3 Representing the Target Language in LF

Our implementation of type-preserving closure conversion in Beluga translates an intrinsically well-typed source term to an intrinsically well-typed target term. For source terms we will reuse the data type definition given earlier (see Fig. 3).

Target terms are defined in LF (see Fig. 11) following the typing rules from Fig. 9 using the type family `target` which is indexed by types. This allows us to only consider well-typed target terms. Note that our target terms for closure conversion differ from the target terms used in CPS. We are taking advantage of HOAS and model the binding structure in abstractions and let-expressions by piggybacking on LF's function space. Our 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 5.

4.4 Type Preserving Closure Conversion in Beluga: an Overview

We now implement the closure conversion algorithm as a transformation of intrinsically typed source terms to intrinsically typed target terms. Our main function `cc` corresponds closely to the type preservation proof. Its type is a direct encoding of the type-preservation theorem and can be read as follows: given well-typed source terms in a source context Γ and a map of the source context Γ to the target context Δ, it returns a well-typed target term in the target context Δ. We again ensure that the type `T` is closed by associating it with a weakening substitution.

```
cc: Map [Δ] [Γ] → [Γ ⊢ source T[]] → [Δ ⊢ target T[]]
```

Here `Map [Δ] [Γ]` refers to an indexed recursive data-type which encodes the context relation between the target context Δ and the source context Γ (see page 15 for the formal description given earlier). Before we give its definition, we define the type of a context using a schema declaration. The schema for source and target contexts can be defined as follows:

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

The schema `tctx` describes contexts where each declaration is an instance of type `target T` and encodes the structure of target contexts as defined by the grammar; similarly the schema `sctx` describes contexts where each declaration is an instance of type `source T`. In the remainder of this section and in Section 5, we will use the convention of writing Γ to name contexts characterized by the schema `sctx`, and Δ for contexts of schema `tctx`. We note that our typing rule for t_letpack also introduces type variables. This might suggest that we must include them in our schema definition for the target context. However, type variables only occur locally and are always bound before the term is returned by our closure conversion function. Therefore well-typed terms never depend on these type variables. As we remarked earlier, a schema declaration simply declares an invariant that the context satisfies when we translate well-typed source terms to well-typed target terms. The schema declaration does not enforce that in general we can only build terms that have this form. In fact, our data type for target terms allows it – however the schema declaration defines a context invariant that is more restrictive and is satisfied in the type preservation proof and also in our implementation of `cc`.

We now define `Map [Δ] [Γ]` as an indexed recursive type [Cave and Pientka, 2012] in Beluga to relate the target context Δ and source context Γ following the definition given earlier on page 15. Each of the constructors given directly corresponds to one of the inference rules defining the relationship between the target and source context.

```
inductive Map:{Δ:tctx}{Γ:sctx} ctype =
| M_id : {Δ:tctx} Map [Δ] []
| M_dot: Map [Δ] [Γ] → [Δ ⊢ target S[]] → Map [Δ] [Γ, x:source S[]];
```

In Beluga's concrete syntax, the *kind* `ctype` and the keyword `inductive` indicate that we are not defining an LF datatype, but an inductive type on the level of computations. $\rightarrow$ is overloaded to mean computation-level 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 (see page 15). Note that we can only embed contextual LF into computation-level (inductive) types not the other way. Once we encounter a box-type or box-object enclosed with `[` and `]` we leave the computation-level and descend to contextual LF.

Beluga reconstructs the type of free variables Δ, Γ, and `S` and implicitly abstracts over them. In the constructor `M_id`, we choose to make Δ an explicit argument to `M_id`, since we often need to refer to Δ explicitly in the recursive programs we are writing about `Map`.

4.5 Implementation of Auxiliary Lemmas

Before giving the implementation of the closure conversion function `cc`, we present several auxiliary functions which closely correspond to the auxiliary lemmas needed in the type preservation proof.

Term strengthening. Our previous strengthening lemma on page 17 stated: if the source term M is well-typed in the context Γ, then M is also well-typed in $\Gamma' = FV(M)$ which is a sub-context of Γ. Here we give an algorithmic interpretation mirroring the proof. We again rely on intrinsic typing here, and hence use a single function to implement both the FV computation and the strengthening proof.

Our strengthening function takes as input a well-typed source term `M` in the context Γ and returns a strengthened term `M'` which is well-typed in some context Γ' where Γ' is a sub-context of Γ, written as $\Gamma' \subseteq \Gamma$. By construction, Γ' will contain all the free variables occurring in `M`.

The subset relation between the context Γ and the context Γ' is defined using the following indexed recursive computation-level data-type.

```
inductive 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[]];
```

In the type preservation proof, we strengthen a term $\mathsf{lam}\, x.\, M'$ and compute $\Gamma' = FV(\mathsf{lam}\, x.\, M')$. Then we recursively translate the term M' which is well-typed in $\Gamma', x : S$. Here, we implicitly rely on the fact that strengthening $\mathsf{lam}\, x.\, M'$ does not change the shape of the overall term. We might even use the same variable names. However, because variables can be understood as referring to a specific position in the contexts (for example as a de Bruijn index), strengthening changes what position a variable refers to. This is subtle and in fact difficult to obtain for free in an implementation. Our implementation strengthens a source term `M` in Γ, `x:source S[]` and returns a strengthened version of `M`, which is well-typed in the source context Γ', `x:source S[]` together with the proof `SubCtx [Γ'] [Γ]`. By construction, Γ' characterizes the free variables in the expression `lam λx.M`. Since Beluga does not directly support existential types, we encode this result using the indexed recursive type `StrTerm`.

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

rec strengthen:[Γ, x:source S[]⊢source T[]] → StrTerm [Γ,x:source S[]] [ ⊢ T]
```

Just as in the proof of the term strengthening lemma, we cannot implement the function `strengthen` directly. When performing induction on Γ, we cannot appeal to the induction hypothesis while maintaining a well-scoped source term. Instead, we implement `str`, which intuitively implements the lemma

```
LF wrap: tp → nat → type =
| init: (source T) → wrap T z
| abs : (source S → wrap T N) → wrap (arr S T) (suc N);

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


rec str: {Γ:ctx} [Γ ⊢  wrap T[] K[]] →  StrTerm' [Γ] [ ⊢ T] [⊢ K]=
λ□ Γ ⇒ fn e ⇒  case [Γ] of
| [] ⇒ let [⊢ M] = e  in STm' [⊢ M] WInit
| [Γ, x:source _ ] ⇒
  case e of
  | [Γ',x:source _ ⊢ M […]] ⇒
    let STm' [h ⊢  M'] rel = str [Γ'] [Γ' ⊢  M] in
      STm' [h ⊢ M'] (WDrop rel)
  | [Γ',x:source _ ⊢ M] ⇒
    let  STm' [h ⊢ abs λx.M'] rel = str [Γ'] [Γ' ⊢ abs λx. M] in
      STm' [h,x:source _ ⊢ M'] (WKeep rel)
;


rec strengthen:[Γ,x:source S[]⊢source T[]]→ StrTerm [Γ, x:source S[]] [⊢T] =
fn m ⇒
let [Γ,x:source _ ⊢ M] = m in
let STm' [Γ'⊢abs λx.init M'] wk = str [Γ] [Γ⊢abs λx.init M] in
  STm [Γ',x:source _ ⊢M'] wk;
```

Fig. 12. Implementation of the Function `str`

If $\Gamma_1, \Gamma_2 \vdash M : T$ *and* $\Gamma'_1 = FV(M) \setminus \Gamma_2$
then $\Gamma'_1, \Gamma_2 \vdash M : T$ *and* $\Gamma'_1 \subseteq \Gamma_1$.

In Beluga, contextual objects can only refer to one context variable, such that we cannot simply write `[Γ1,Γ2 ⊢ source T[]]`. Instead we define a type family `wrap` which abstracts over all the variables in Γ_2. The type family `wrap` is indexed by the type `T` of the source term and the size of Γ_2 described by `N`. The function `str` then recursively analyses Γ_1, adding variables occurring in the input term to Γ_2. The type of `str` carries with it the index `N` the size of Γ_2 which is preserved. This is only needed to verify coverage; in the case where we call `str`, this ensures that the returned wrapped term will be of the same form.

The function `str`, given in Fig. 12, is implemented recursively on the structure of Γ and 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` to test whether `x` occurs in it.
In the function body, λ□-abstraction introduces the explicitly quantified context and `fn`-abstraction introduces a computation-level function. In order to observe the contextual dependencies of `M`, we first pattern match on its context Γ. The first case, $\Gamma = \cdot$ implies that `[ ⊢ M]` and hence $\mathsf{FV}(M) = \cdot$. As a witness we return the subcontext relation with the `WInit` constructor.
In the second case where `Γ = (Γ', x:source _ )` we pattern match on `e` to see, if `x` occurs. If `e` matches the pattern `[Γ', x:source _ ⊢M[…]]`, then we are guaranteed that `x`, the rightmost variable in the context, does not appear in `M`. Recall that associating `M` with the weakening substitution `[…]` means that `M` may only refer to

Γ'. The type of M will be [Γ′ ⊢ source _]. We can hence strengthen M to the context Γ' by recursively calling str on the subcontext Γ', and use the subcontext relation constructor WDrop to relate Γ and Γ′,x:source _ .
Otherwise [Γ′, x:source _ ⊢ M] matches, meaning that x may occur in M. Recall that M is implicitly associated with the identity substitution and hence may refer to Γ' and x. As the term did not match the first pattern, we know that x must occur in M; hence we must keep it as part of $\mathsf{FV}(M)$. We use the abs constructor of the wrap type to add x to the accumulator representing Γ_2, and recursively call str on the Γ subcontext. As the type index in the recursive call is now suc N, we are guaranteed that the wrapped term in the output will be of the form abs λx.M’. Finally we use the WKeep subcontext relation constructor to keep x as part of FV(M).

The function strengthen then simply calls str. This helps to keep the code modular. Because str implicitly reasons about the size of Γ_2, we are guaranteed that the term returned by the function str must be of the form [Γ’⊢ abs λx.M’]. However, this does not guarantee that the actual size of the term M’ is equal to the size of M. We will return to this issue when we discuss totality of the closure conversion function.

Term weakening. In the formal development, we relied on two weakening lemmas: in the first we weaken a source term with respect to a given context relation: Given a well-typed term M in the context Γ' and $\Gamma' \subseteq \Gamma$, the term M remains well-typed in Γ. This form of weakening cannot be obtained for free, since it relies on the context relation $\Gamma' \subseteq \Gamma$. It is a general form of weakening. In our implementation, we incorporated this form of weakening directly into the variable lookup function discussed below.

The second form of weakening allows us to weaken a target term, i.e. if a target term is well-typed in a context Δ, it remains well-typed if we add an additional assumption. In Beluga, we obtain this kind of weakening for free since contextual type theory incorporates weakening by individual declarations.

Map Variable lookup. The function lookup takes as input a Map [Δ] [Γ] together with a source variable of type T in the source context Γ and returns the corresponding target expression of the same type.

```
rec lookup: Map [Δ] [Γ] → {#q:[Γ ⊢source T[]]} [Δ ⊢target T[]] =
fn ρ ⇒ λ□ #q ⇒ case ρ of
| M_dot ρ′ [Δ ⊢M] ⇒
 let (ρ : Map [Δ] [Γ′,x:source _) = ρ in
 (case [Γ′, x:source _ ⊢#q] of
  | [Γ′, x:source _ ⊢ x    ] ⇒  [Δ ⊢ M]
  | [Γ′, x:source _ ⊢#p[…]] ⇒ lookup ρ’ [Γ′ ⊢#p] )
| M_id [Δ] ⇒ impossible [ ⊢#q]
```

We quantify explicitly over all variables in the context Γ by {#q:[Γ ⊢source T[]]} where #q denotes a variable of type source T[] in the context Γ. The function lookup is implemented by pattern matching on the map ρ. This makes establishing the totality of this function straightforward. If ρ is of type Map [Δ] [ ], i.e. Γ is empty, there is no possible variable #q. We can disprove a case by impossible [⊢#q] which tries to split on the variable #q.

If ρ is of type Map [Δ] [Γ′, x:source _], then there is a ρ' of type Map [Δ] [Γ′] and some target term [Δ ⊢M] of target type [Δ ⊢ target _] corresponding to the source variable `x`. We now check whether `x` is the variable we are looking for by pattern matching on `[Γ′, x:source _ ⊢#q]`. There are two cases: either `#q` stands for `x`, in which case we choose the branch with the pattern `[Γ′, x:source _ ⊢ x]` and simply return `[Δ ⊢M]`, or it stands for another variable from Γ and we choose the branch with the pattern `[Γ′, x:source _ ⊢#p […]]` and search the remaining context Γ′.

For the closure conversion algorithm, it is important to 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 which guarantees that the tuple only contains variables in the order they occur in Γ.

```
inductive VarTup: (Γ:sctx) {T:[ ⊢tp]} [Γ ⊢source T[]] → ctype =
| Empty : VarTup [⊢unit] [⊢nil]
| Next : VarTup [⊢ L] [Γ ⊢R]
     →  VarTup [⊢cross T L] [Γ,x:source _ ⊢cons x R[…]]
;
```

The constructor `Empty` stands for an empty tuple. Given a variable tuple `[Γ ⊢R]`, we can form a variable tuple `[Γ, x:source _ ⊢cons x R[…]]` using the constructor `Next`. Note that we must weaken `R` to guarantee it is meaningful in the extended context.

Next, we translate the tuple `VarTup [⊢L] [Γ′ ⊢R]` where `SubCtx [Γ′] [Γ]` to a corresponding target term using a mapping ρ between source and target context. In the type preservation proof, we first weaken the tuple `VarTup [⊢L] [Γ′ ⊢R]` to be meaningful in the context Γ, and then lookup for each x_i the corresponding target term in the mapping ρ. Here, we give a direct implementation which incorporates weakening directly. The function is defined recursively on `SubCtx [Γ′] [Γ]`.

```
rec lookupVars: SubCtx [Γ′] [Γ] → VarTup [Γ′] [⊢L] → Map [Δ] [Γ]
              → [Δ ⊢target L] =
fn r ⇒ fn vt ⇒ fn σ ⇒ let (σ : Map [Δ] [Γ]) = σ  in case r of
| WInit ⇒
  let Empty = vt in [Δ ⊢cnil]
| WDrop r' ⇒
  let M_dot σ′ [Δ ⊢P] = σ in  lookupVars r' vt σ′
| WKeep r' ⇒
  let Next vt' = vt in
  let M_dot σ′ [Δ ⊢P] = σ in
  let [Δ ⊢M] = lookupVars r' vt' σ′ in
    [Δ ⊢ccons P M]
;
```

In the first case, we learn that $\Gamma' = \Gamma = \cdot$ and the variable tuple must be empty; the corresponding target tuple hence can be represented with `cnil` in context Δ.
In the second case, the first variable of Γ does not appear in Γ′, we can thus disregard it in the tuple construction and we recursively call `lookupVars` after removing `x ↦ P` from σ.
In the third case, the top variable of Γ appears on top of Γ′ as well, and we have `x ↦ P` in σ. We recursively construct the tuple representing the rest of Γ′, before adding `P` in front.

Map Extension. In the type preservation proof, we relied on being able to extend our map between source Γ and target context Δ. First, we implement the function `weaken` which allows us to simply weaken the target context; given a map ρ between source context Γ and target context Δ, it is straightforward to construct a map between Γ and `Δ, x:target S[]`. Since our map ρ is only required to provide mappings for all the source variables in Γ, we recursively analyze the given map ρ and weaken each element to be meaningful in the extended target context.

We then use this function to extend a map ρ between the source context Γ and target context Δ with an identity mapping, i.e. a new source variable is mapped to its corresponding target variable. We first retrieve a name for the target context by re-binding ρ to `(ρ : Map [Δ] [Γ])`. We then weaken ρ using the function `weaken` and extend the result with `[Δ, x:target _ ⊢x]`.

```
rec weaken: Map [Δ] [Γ] → Map [Δ, x:target S[]] [Γ] =
fn ρ ⇒ case ρ of
| IdMap [Δ]          ⇒ IdMap [Δ, x:target _]
| DotMap ρ′ [Δ ⊢M] ⇒ DotMap (weaken ρ′) [Δ, x:target _ ⊢M]
;

rec extend: Map [Δ] [Γ] → Map [Δ, x:target S[]] [Γ, x:source S[]] =
fn ρ ⇒ let (ρ : Map [Δ] [Γ]) = ρ in
  M_dot (weakenMap ρ) [Δ,x:target _ ⊢x]
;
```

Context Reification. The context reification lemma is proven by induction on Γ and our function `reify` directly implements the proof by pattern matching on the context Γ. Given a context Γ, the function `reify` produces a tuple containing variables of Γ together with `Map [`x_{env}`:target `T_Γ`] [Γ]`, the mapping between those target variables and their corresponding projections. We call the result an environment closure, written as `EnvClo vt ρ`, since we package the variable tuple `vt` with the environment ρ. 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_{env} of type T_Γ. In particular, we replace a source variable with the corresponding projection on the target variable x_{env}. This replacement is elegantly accomplished by relying on the meta-level substitution. In the function `extendEnv`, we recursively analyze the given map ρ which provides mappings from the source context Γ to the target x_{env}`: target S`; for each element in the map ρ we replace any occurrence of x_{env} with `crst` x_{env} where x_{env} has now the extended type `target (cross T S)`.

```
inductive CtxAsTup: {Γ:sctx} ctype =
| EnvClo: VarTup [ ⊢T_Γ] [Γ ⊢M] → Map [x_env:target T_Γ] [Γ] → CtxAsTup [Γ];

rec extendEnv: Map [x_env:target S] [Γ] → Map [x_env:target (cross T S)] [Γ] =
fn ρ ⇒ case ρ of
| M_id [x_env:target S] ⇒ M_id [x_env:target _]
| M_dot ρ′ [x_env:target S ⊢M] ⇒
    M_dot (extendEnv ρ′) [x_env:target _  ⊢M[crst x_env]]
;

rec reify: {Γ:sctx} CtxAsTup [Γ] =
λ□ Γ ⇒ case [Γ] of
| [ ] ⇒  EnvClo  Emp (IdMap [x:target unit])
| [Γ, x:source S[]] ⇒
  let  EnvClo vt ρ = reify [Γ] in
  let ρ′ = DotMap (extendEnv ρ) [x_env:target (cross S[] _) ⊢cfst x_env] in
    EnvClo (Next vt) ρ′
;
```

```
rec cc: [Γ ⊢source T[]] → Map [Δ] [Γ] → [Δ ⊢target T[]] =
fn m ⇒ fn ρ ⇒ case m of

| [Γ ⊢#p] ⇒ lookup ρ [Γ ⊢#p]

| [Γ ⊢app M N] ⇒
  let [Δ ⊢P] = cc [Γ ⊢M] ρ in
  let [Δ ⊢Q] = cc [Γ ⊢N] ρ in
      [Δ⊢cletpack P λl.λx_f.λx_env. capp x_f (ccons (Q [...]) x_env)]

| [Γ ⊢lam λx.M] ⇒
  let STm [Γ',x:source _ ⊢M'] wk = strengthen [Γ, x:source _⊢M] in
  let EnvClo env ρ' = reify [Γ'] in
  let [Δ ⊢P_env] = lookupVars wk env ρ in
  let [x_env:target _,x:target _⊢P] = cc [Γ',x:source _⊢M'] (extend ρ')
  in  [Δ ⊢cpack (clam λc.clet (cfst c)
                             (λx.clet (crst c) (λx_env.P [x_env, x])))
                 P_env  ]
;
```

Fig. 13. Implementation of Closure Conversion in Beluga

4.6 Type-preserving Closure Conversion Implementation

We now describe the implementation of the type-preserving closure conversion algorithm using the type-preservation proof (Thm. 4.1) as a guide. The top-level function `cc` takes as input a well-typed source term, `[Γ ⊢source T[]]`, together with a map ρ between the source context Γ and the target context Δ, and returns a well-typed target term, `[Δ ⊢target T[]]`. The function recursively analyzes a given source term, `[Γ ⊢source T[]]`. We concentrate here on the cases for variables, λ-abstractions and applications. When we encounter a variable, written as `[Γ ⊢#p]`, we simply lookup its corresponding binding in ρ. When we encounter an application, `[Γ ⊢app M N]`, we recursively translate `[Γ ⊢M]` and `[Γ ⊢N]` and package together their results.

The most interesting case is the one for λ-abstraction, `[Γ ⊢lam λx.M]`. We first strengthen the term to a term `[Γ',x:source _ ⊢M']` where Γ' describes the free variables occurring in `lam λx.M` and `wk` is a witness for `SubCtx [Γ'] [Γ]`, i.e. the fact that Γ' is a sub-context of Γ. Next, we reify the context Γ' into a tuple `env` describing the environment and a mapping ρ' between the source context Γ' and the target variable $\mathtt{x}_{env}$. Recall that ρ' associates each variable in Γ' with the corresponding projection of $\mathtt{x}_{env}$. Moreover, if $\Gamma' = \mathtt{x}_k\mathtt{:source\ T}_k\mathtt{,\ \ldots,\ x}_1\mathtt{:source\ T}_1$, then $\mathtt{x}_{env}$ has type $\mathtt{target\ (cross\ T_1\ (cross\ \ldots\ (cross\ T}_k\mathtt{\ unit)))}$ and `env` is a tuple consisting of variables $\mathtt{x}_1\mathtt{,\ \ldots,\ x}_k$.

Using `lookupVars wk env ρ` we build a corresponding tuple in the target language, called $\mathtt{[\Delta \vdash P}_{env}\mathtt{]}$. By recursively translating `[Γ',x:source _⊢ M']` with the map ρ' extended with the identity, we obtain $\mathtt{[x}_{env}\mathtt{:target\ \_,\ x:target\ \_\vdash P]}$. Finally, we build our result: $\mathtt{clam\ \lambda c.clet\ (cfst\ c)\ (\lambda x.\ clet\ (crst\ c)\ (\lambda x}_{env}\mathtt{.\ P\ [x}_{env}\mathtt{\ ,\ x]))}$ together with its environment $\mathtt{P}_{env}$. Our underlying dependent types guarantee that our implementation is well-typed.

4.7 Discussion

Our implementation of closure conversion, including all definitions and auxiliary functions, consists of approximately 200 lines of code and follows closely the type-preservation proof. By taking advantage of the built-in substitution to replace variables in the source term with their respective projections in the target language, our implementation remains compact avoiding the need to build and manage infrastructure regarding variable bindings and contexts. There is only one instance where we wish Beluga would support richer abstractions: it is the function for strengthening. We come back to this point shortly.

All our auxiliary functions and the main closure conversion function pass the coverage checker [Dunfield and Pientka, 2009]. All the auxiliary functions also are accepted by the totality checker [Pientka and Abel, 2015; Pientka and Cave, 2015] certifying that they are terminating. Certifying that the closure conversion function is terminating fails. The reason is subtle: in the case for lambda-abstraction we strengthen `[Γ, x:source _ ⊢M]` and obtain `[Γ', x:source _ ⊢M']`. We then recursively translate `[Γ', x:source _ ⊢M']`. However, there is no obvious reason why it is structurally smaller than the original term. Therefore, the totality checker flags this recursive call as (possibly) not decreasing.

Note that this is not as dramatic as it seems: while our Beluga code looks very much like a proof, the actual proof of type preservation is not directly given by our code but by the meta-theoretical properties of Beluga, which do not depend on the totality of our code. Instead, type preservation is only guaranteed *under the condition* that the code terminates. So the failure of the totality checker means that we cannot guarantee that the closure conversion will always terminate, but when it does terminate, we still know that it returns a term of the right type. In this context, a proof of termination is not terribly important, especially if we consider that a total function may still fail to terminate within our lifetime.

How could one guarantee that `M'` is indeed equivalent to `M` up to possibly some variable renaming? – Intuitively, when we strengthen a term `[Γ ⊢lam λx.M]`, we would like to return a strengthened term `[Γ' ⊢lam λx.M']` together with a strengthening substitution $\Gamma \vdash \sigma : \Gamma'$. Since σ is a strengthening substitution and only maps variables to variables, we then know that `[Γ ⊢lam λx.M'[σ x]]` has the same size as `[Γ ⊢lam λx.M]`. Therefore, `[Γ',x:source _ ⊢M']` can be considered smaller than `[Γ,x:source _⊢M]` and we can safely convert `[Γ',x:source _⊢M']`.

While Beluga supports substitution variables [Cave and Pientka, 2013], it does not allow us to express and guarantee that these substitutions only map variables to variables. Hence, we cannot exploit and take advantage of substitution variables directly. In the same way that Beluga distinguishes between general meta-variables, written with upper-case letters, and parameter variables, written as lower case letters prefixed with `#`, distinguishing between general substitution variables and variable substitutions is desirable. It would not only allow us to express the strengthening function more abstractly, but also would simplify reasoning about the size of strengthened terms. However, in balance, the current implementation is likely more efficient. Hence, this seems a trade-off between efficiency and guaranteeing correctness properties statically.

Closely related to our work is Guillemette and Monnier [2007], which describes the 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. They then compute the free-variables of a term as a list, and use this list to create a map from the variable to its projection when variable occurs in the term, and to $\perp$ otherwise. Guillemette and Monnier [2008] 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 that of Guillemette and Monnier [2007].

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.

5. HOISTING

The last program transformation we consider is hoisting. It lifts λ-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 alluded to in Sec. 4.3, 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, LF. We therefore extend our closure conversion algorithm to perform hoisting at the same time. Hoisting can however be understood by itself; we present here a standalone type-preserving hoisting algorithm. As before, we revisit the type preservation proof to guide us in explaining the implementation.

When hoisting all functions from a program, each function may 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. We instead merge all the functions in a single tuple, representing the function environment, and bind it as a single variable from which we project individual functions, which ends up being less cumbersome when using Beluga's notion of context variables.

For example, performing hoisting on the closure-converted program presented in Sec. 4.2

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

will result in

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

5.1 The Target Language Revisited

We define hoisting on the target language of closure conversion and keep the same typing rules (see Fig. 9) with one exception: the typing rule for t_cpack is replaced by the following one:

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

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. This is only problematic for t_cpack, as we request the code portion to be closed; t_cpack' specifically allows the code portion to depend on the function environment.

The distinction between t_cpack and t_cpack' is irrelevant in our implementation, as in our representation of the typing rules in LF the context is ambient.

5.2 Hoisting Algorithm

We now define the hoisting algorithm in Fig. 14 using $[\![P]\!]_l = Q \bowtie E$, where P, Q and E are target terms and l is a variable name which does not occur in P. 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 tuple 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. Renaming and adjustment of references to the function environment are performed implicitly in the presentation, and binding l is taken to uniquely name function references.

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 function 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 environment E and its type L are of the right form. We define separately $\Delta \vdash_l E : L$ restricting $\Delta \vdash E : L$ to a n-ary tuple E of product type L.

$\boxed{\Delta \vdash_l E : L}$ E is a well-formed tuple of type L in target context Δ

$$\frac{}{\Delta \vdash_l () : \mathsf{unit}}\ \mathsf{env\_nil} \qquad \frac{\Delta \vdash P : T \quad \Delta \vdash_l E : L}{\Delta \vdash_l (P, E) : T \times L}\ \mathsf{env\_cons}$$

$$
\begin{array}{lcll}
[\![x]\!]_l & = & x \bowtie () & \\
[\![\langle P_1, P_2\rangle]\!]_l & = & \langle (\mathsf{fst}\ l)\ (\mathsf{rst}\ l), Q_2\rangle \bowtie E & \text{where } [\![P_1]\!]_l = Q_1 \bowtie E_1 \\
 & & & \quad \text{and } [\![P_2]\!]_l = Q_2 \bowtie E_2 \\
 & & & \quad \text{and } E = (\mathsf{lam}\, l.\, Q_1,\ E_1 \circ E_2) \\
\left[\!\!\left[\begin{array}{l}\mathsf{let}\ \langle x_f, x_{env}\rangle = P_1 \\ \mathsf{in}\ P_2\end{array}\right]\!\!\right]_l & = & \begin{array}{l}\mathsf{let}\ \langle x_f, x_{env}\rangle = Q_1 \\ \mathsf{in}\ Q_2\end{array} \bowtie E & \text{where } [\![P_1]\!]_l = Q_1 \bowtie E_1 \\
 & & & \quad \text{and } [\![P_2]\!]_l = Q_2 \bowtie E_2 \\
 & & & \quad \text{and } E = E_1 \circ E_2 \\
[\![\mathsf{lam}\, x.\, P]\!]_l & = & \mathsf{lam}\, x.\, Q \bowtie E & \text{where } [\![P]\!]_l = Q \bowtie E \\
[\![P_1\ P_2]\!]_l & = & Q_1\ Q_2 \bowtie E_1 \circ E_2 & \text{where } [\![P_1]\!]_l = Q_1 \bowtie E_1 \\
 & & & \quad \text{and } [\![P_2]\!]_l = Q_2 \bowtie E_2 \\
[\![\mathsf{let}\ x = P_1\ \mathsf{in}\ P_2]\!]_l & = & \mathsf{let}\ x = Q_1\ \mathsf{in}\ Q_2 \bowtie E_1 \circ E_2 & \text{where } [\![P_1]\!]_l = Q_1 \bowtie E_1 \\
 & & & \quad \text{and } [\![P_2]\!]_l = Q_2 \bowtie E_2 \\
[\![(P_1, P_2)]\!]_l & = & (Q_1, Q_2) \bowtie E_1 \circ E_2 & \text{where } [\![P_1]\!]_l = Q_1 \bowtie E_1 \\
 & & & \quad \text{and } [\![P_2]\!]_l = Q_2 \bowtie E_2 \\
[\![\mathsf{fst}\ P]\!]_l & = & \mathsf{fst}\ Q \bowtie E & \text{where } [\![P]\!]_l = Q \bowtie E \\
[\![\mathsf{rst}\ P]\!]_l & = & \mathsf{rst}\ Q \bowtie E & \text{where } [\![P]\!]_l = Q \bowtie E \\
 & & & \\
[\![()]\!]_l & = & () \bowtie () &
\end{array}
$$

Fig. 14. Hoisting Algorithm

The type correctness of hoisting depends on a number of lemmas. In particular, we rely on a number of properties about those environments represented as n-ary tuples containing target terms; for example, weakening states that given a target term P which depends on an environment $l : L_f$, it remains well-typed given an extended environment $l : L$ where either $L = L_f \circ L'$ or $L = L' \circ L_f$; also, appending two well-typed environments yields a well-typed environment.

Lemma 5.1. ***Append Function Environments***
If $\Delta \vdash_l E_1 : L_1$ *and* $\Delta \vdash_l E_2 : L_2$, *then* $\Delta \vdash_l E_1 \circ E_2 : L_1 \circ L_2$.

Proof. By induction on the derivation $\Delta \vdash_l E_1 : L_1$. □

Append Function Environments (Lemma 5.1) states that appending two environments remains well-typed. Thus, appending two tuples E_1 and E_2 of type L_1 and L_2 respectively will result in a tuple of type $L_1 \circ L_2$. We overload here the append operator written as $\circ$ using it on the one hand to append terms and on the other to append types.

Lemma 5.2. ***Function Environment Weakening (1)***
If $\Delta, l_2 : L_{f_2} \vdash P : T$ *and* $L_{f_1} \circ L_{f_2} = L_f$, *then* $\Delta, l : L_f \vdash [l_2 \mapsto \pi_{n+1}\, l]P : T$ *where* $n = |L_{f_1}|$.

Proof. By induction on the relation $L_{f_1} \circ L_{f_2} = L_f$. □

Lemma 5.3. ***Function Environment Weakening (2)***
If $\Delta, l : L_{f_1} \vdash P : T$ *and* $L_{f_1} \circ L_{f_2} = L_f$, *then* $\Delta, l : L_f \vdash P : T$.

Proof. By induction on the relation $L_{f_1} \circ L_{f_2} = L_f$. □

Function Environment Weakening (1) (Lemma 5.2) and (2) (Lemma 5.3) say that we can replace a variable of a product type by a product type which is larger, i.e. it has more elements. Lemma 5.2 proves this property when we extend the product type L_{f_2} by appending the product type L_{f_1} to the left, i.e. we compute $L_{f_1} \circ L_{f_2}$. If L_{f_1} denotes a product type $T_1 \times (T_2 \times (\ldots (T_n \times \mathsf{unit}) \ldots))$, then $L_{f_1} \circ L_{f_2}$ essentially results in replacing unit with the product type L_{f_2}. Hence, any occurrence of l_2 in the term P must be replaced by $\pi_{n+1}\, l$ in the statement of Lemma 5.2. When we append L_{f_2} to L_{f_1} in the statement of Lemma 5.3 however, we do not need to renumber the projections in P.

THEOREM 5.1. ***Type Preservation***
If $\Delta \vdash P : T$ *and* $[\![P]\!]_l = Q \bowtie E$ *then* $\cdot \vdash_l E : L_f$ *and* $\Delta, l : L_f \vdash Q : T$ *for some* L_f.

PROOF. By induction on the typing derivation $\mathcal{D}_0 :: \Delta \vdash P : T$. We only show a few cases in detail. The other cases are similar.

$$\textit{Case } \mathcal{D}_0 = \dfrac{\begin{matrix}\mathcal{D} \\ x : T \in \Delta\end{matrix}}{\Delta \vdash x : T}\ \mathsf{t\_cvar}$$

where $[\![x]\!]_l = x \bowtie ()$.

$\cdot \vdash_l () : \mathsf{unit}$ — by env_nil
$\Delta, l_f : \mathsf{unit} \vdash x : T$ — by t_cvar

$$\textit{Case } \mathcal{D}_0 = \dfrac{\begin{matrix}\mathcal{D}_1 \\ \cdot \vdash P_1 : \mathsf{code}\ (T \times L_x)\ S\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ \Delta \vdash P_2 : L_x\end{matrix}}{\Delta \vdash \langle P_1, P_2 \rangle : T \to S}\ \mathsf{t\_cpack}$$

where $[\![\langle P_1, P_2\rangle]\!]_l = \langle (\mathsf{fst}\ l)\ (\mathsf{rst}\ l), Q_2\rangle \bowtie (\mathsf{lam}\, l.\, Q_1, E)$
and $[\![P_1]\!]_l = Q_1 \bowtie E_1$, $[\![P_2]\!]_l = Q_2 \bowtie E_2$ and $E = E_1 \circ E_2$.

$l : L_1 \vdash Q_1 : \mathsf{code}\ (T \times L_x)\ S$ and $\cdot \vdash_l E_1 : L_1$ — by i.h. on $\mathcal{D}_1$
$\Delta, l : L_2 \vdash Q_2 : L_x$ and $\cdot \vdash_l E_2 : L_2$ — by i.h. on $\mathcal{D}_2$
$\cdot \vdash_l E_1 \circ E_2 : L_1 \circ L_2$ and $L_1 \circ L_2 = L_f$ — by Append f. env. (Lemma 5.1)
$l : L_f \vdash Q_1 : \mathsf{code}\ (T \times L_x)\ S$ — by F. env. weaken.(1)(Lemma 5.2)
$\Delta, l : L_f \vdash Q_2 : L_x$ — by F. env. weaken.(2)(Lemma 5.3)
$\cdot \vdash \mathsf{lam}\, l.\, Q_1 : \mathsf{code}\ L_f\ (\mathsf{code}\ (T \times L_x)\ S)$ — by t_clam
$\cdot \vdash_l (\mathsf{lam}\, l.\, Q_1, E_1 \circ E_2) : (\mathsf{code}\ L_f\ (\mathsf{code}\ (T \times L_x)\ S)) \times L_f$ — by env_cons
$l : (\mathsf{code}\ L_f\ (\mathsf{code}\ (T \times L_x)\ S)) \times L_f \vdash \mathsf{fst}\ l : \mathsf{code}\ L_f\ (\mathsf{code}\ (T \times L_x)\ S)$ — by t_cfst
$l : (\mathsf{code}\ L_f\ (\mathsf{code}\ (T \times L_x)\ S)) \times L_f \vdash \mathsf{rst}\ l : L_f$ — by t_csnd
$l : (\mathsf{code}\ L_f\ (\mathsf{code}\ (T \times L_x)\ S)) \times L_f \vdash (\mathsf{fst}\ l)\ (\mathsf{rst}\ l) : \mathsf{code}\ (T \times L_x)\ S$ — by t_capp
$\Delta, l : (\mathsf{code}\ L_f\ (\mathsf{code}\ (T \times E)\ S)) \times L_f \vdash \langle (\mathsf{fst}\ l)\ (\mathsf{rst}\ l), Q_2\rangle : T \to S$ — by t_cpack'

$$\textit{Case } \mathcal{D}_0 = \dfrac{\begin{matrix}\mathcal{D}_1 \\ \Delta \vdash P_1 : S \to T\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ \Delta, x_f : \mathsf{code}\ (S \times l_0)\ S, x_{env} : l_0 \vdash P_2 : T\end{matrix}}{\Delta \vdash \mathsf{let}\ \langle x_f, x_{env}\rangle = P_1\ \mathsf{in}\ P_2 : T}\ \mathsf{t\_cletpack}^{l_0}$$

where $[\![\mathsf{let}\ \langle x_f, x_{env}\rangle = P_1\ \mathsf{in}\ P_2]\!]_l = \mathsf{let}\ \langle x_f, x_{env}\rangle = Q_1\ \mathsf{in}\ Q_2 \ \bowtie\ E$
where $[\![P_1]\!]_l = Q_1 \ \bowtie\ E_1$, $[\![P_2]\!]_l = Q_2 \ \bowtie\ E_2$ and $E = E_1 \circ E_2$.

$\Delta, l : L_1 \vdash Q_1 : S \to T$ and $\cdot \vdash_l E_1 : L_1$ by i.h. on $\mathcal{D}_1$
$\Delta, x_f : \mathsf{code}\ (S \times l_0)\ T, x_{env} : l_0, l : L_1 \vdash Q_2 : T$ and $\cdot \vdash_l E_2 : L_2$ by i.h. on $\mathcal{D}_2$
$\cdot \vdash_l E_1 \circ E_2 : L_1 \circ L_2$ and $L_1 \circ L_2 = L_f$ by Append f. env.(Lemma 5.1)
$\Delta, l : L_f \vdash Q_1 : S \to T$ by F. env. weaken.(1)(Lemma 5.2)
$\Delta, x_f : \mathsf{code}\ (S \times l_0)\ T, x_{env} : l_0, l : L_f \vdash Q_2 : T$ by F. env. weaken.(2)(Lemma 5.3)
$\Delta, l : L_f, x_f : \mathsf{code}\ (S \times L)\ T, x_{env} : l_0 \vdash Q_2 : T$ by exchange
$\Delta, l : L_f \vdash \mathsf{let}\ \langle x_f, x_{env}\rangle = Q_1\ \mathsf{in}\ Q_2 : T$ by t_cletpack

$$\textit{Case}\ \mathcal{D}_0 = \cfrac{\begin{matrix}\mathcal{D}\\ \Delta, x : S \vdash P : T\end{matrix}}{\Delta \vdash \mathsf{lam}\, x.\, P : \mathsf{code}\ S\ T}\ \mathsf{t\_clam}$$

where $[\![\mathsf{lam}\, x.\, P]\!]_l = \mathsf{lam}\, x.\, Q \ \bowtie\ E$ where $[\![P]\!]_l = Q \ \bowtie\ E$.

$\Delta, x : S, l : L_f \vdash Q : T$ and $\cdot \vdash_l E : L_f$ by i.h. on $\mathcal{D}$
$\Delta, l : L_f, x : S \vdash Q : T$ by exchange
$\Delta, l : L_f \vdash \mathsf{lam}\, x.\, Q : \mathsf{code}\ S\ T$ by t_clam

$$\textit{Case}\ \mathcal{D}_0 = \cfrac{\begin{matrix}\mathcal{D}_1\\ \Delta \vdash P_1 : \mathsf{code}\ S\ T\end{matrix} \qquad \begin{matrix}\mathcal{D}_2\\ \Delta \vdash P_2 : S\end{matrix}}{\Delta \vdash P_1\ P_2 : T}\ \mathsf{t\_capp}$$

where $[\![P_1\ P_2]\!]_l = Q_1\ Q_2 \ \bowtie\ E$
where $[\![P_1]\!]_l = Q_1 \ \bowtie\ E_1$, $[\![P_2]\!]_l = Q_1 \ \bowtie\ E_2$ and $E = E_1 \circ E_2$.

$\Delta, l : L_1 \vdash Q_1 : \mathsf{code}\ S\ T$ and $\cdot \vdash E_1 : L_1$ by i.h. on $\mathcal{D}_1$
$\Delta, l : L_2 \vdash Q_2 : S$ and $\cdot \vdash E_2 : L_2$ by i.h. on $\mathcal{D}_2$

$\cdot \vdash_l E_1 \circ E_2 : L_1 \circ L_2$ and $L_1 \circ L_2 = L_f$ by Append f. env.(Lemma 5.1)

$\Delta, l : L_f \vdash Q_1 : \mathsf{code}\ S\ T$ by F. env. weaken. (1) (Lemma 5.2)
$\Delta, l : L_f \vdash Q_2 : S$ by F. env. weaken. (2) (Lemma 5.3)
$\Delta, l : L_f \vdash Q_1\ Q_2 : T$ by t_capp
□

While we have so far concentrated on describing hoisting as a separate transformation on the result of closure conversion, our Beluga implementation of hoisting is incorporated into closure conversion. Certain cases had to be adapted to this end. For example, as the closure case $\langle P_1, P_2\rangle$ corresponds in the implementation to the translation of a source lambda-abstraction $\mathsf{lam}\, x.\, M$, the need to merge function environments is eliminated due to applying the induction hypothesis to a single subterm.

5.3 Implementation of Auxiliary Lemmas

Defining function environments. The function environment represents the collection of functions hoisted out of a program. Since our context keeps track both of variables which represent functions bound at the top-level and of those representing local arguments, extra machinery would be required to separate them. For this reason, we represent the function environment as a single term in the target language rather than multiple terms with individual binders, maintaining as an additional proposition its form as a tuple of product type.

As mentioned in our description of the hoisting algorithm, operations such as $\circ$ are only defined on n-ary tuples and on product types. To guarantee coverage, we hence define a data-type `env` which guarantees that a given target term `P` is a tuple of type L_f encoding the judgement $\cdot \vdash_l$ `P` : L_f given earlier. To put it differently, `env` L_f `P` guarantees that the tuple `P` is a well-formed environment of type L_f and stands for the derivation $\cdot \vdash_l$ `P` : L_f.

```
LF env:  {Lf:tp} target Lf → type =
| env_nil:  env unit cnil
| env_cons: {P:target T} env Lf E → env (cross T Lf) (ccons P E)
;
```

The constructor `env_nil` describes an empty tuple of type `unit`. The constructor `env_cons` allows us to build a well-formed environment given a target `P` of type `T` and an environment consisting of the tuple `E` together with its type L_f.

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. In our hoisting algorithm we rely on the operation $\circ$ which allows us to append environments together with several auxiliary lemmas regarding environments.

To append two environments, we first define in BELUGA the function `append` which when given two environments `Env1` and `Env2` returns as a result a new environment `Env3` together with the proof of `Append [⊢ Env1] [⊢ Env2] [⊢ Env3]`. Knowing the relationship between `Env1`, `Env2` and `Env3` is important, since we exploit it later to prove two weakening lemmas about environments. Our `append` corresponds to Lemma 5.1 about appending function environments.

```
inductive Append: [⊢ env L1 E1] → [⊢ env L2 E2] → [⊢ env L3 E3] → ctype =
| App_nil : Append [⊢ env_nil] [⊢ E] [⊢ E]
| App_cons: Append [⊢ E1] [⊢ E2] [⊢ E3]
          → Append [⊢ env_cons P E1] [⊢ E2] [⊢ env_cons P E3]
;

inductive ExAppend: [⊢ env L1 E1] → [⊢ env L2 E2] → ctype =
| ExEnv: Append [⊢ Env1] [⊢ Env2] [⊢ Env3] → ExAppend [⊢ Env1] [⊢ Env2]
;

rec append: {Env1:[⊢ env L1 E1]}{Env2:[⊢ env L2 E2]}
            ExAppend [⊢ Env1] [ ⊢ Env2] =
λ□ Env1, Env2 ⇒  case [⊢ Env1] of
| [⊢ env_nil] ⇒  ExEnv  App_nil
| [⊢ env_cons P Env'1] ⇒
  let ExEnv a = append [⊢ Env'1] [⊢ Env2] in  ExEnv (App_cons a)
;
```

Let us look at our implementation more closely. The data-type definition of `Append` as an indexed recursive type is straightforward. The constructor `App_nil` describes the fact that appending the empty environment to an environment `E` returns `E`. The constructor `App_cons` states given that appending E_1 to E_2 returns E_3, we know that appending `(env_cons P E1)` to E_2 returns `(env_cons P E3)`.

As mentioned earlier, the function `append` states that given two well-typed environments $\texttt{Env}_1$ and $\texttt{Env}_2$ there exists a well-typed environment $\texttt{Env}_3$ together with the witness that `Append [⊢ Env₁] [⊢ Env₂] [⊢ Env₃]`. As BELUGA does not directly support existential types, we define a recursive type `ExAppend [⊢ Env₁] [⊢ Env₂]` to describe the result.

The implementation of the function `append` is then straightforward by recursion on the first environment $\texttt{Env}_1$.

Next, we consider the implementation of the two weakening lemmas for function environment (Lemma 5.2 and 5.3). The lemmas do in fact not talk about appending environments, but only about appending their types. Since in our implementation environments are intrinsically typed, we re-formulate these lemmas slightly. The first can be read as:

> Let $\texttt{Env}_1$, $\texttt{Env}_1$ and $\texttt{Env}_3$ be well-typed environments describing $\cdot \vdash_l \texttt{E}_1 : \texttt{L}_1$, $\cdot \vdash_l \texttt{E}_2 : \texttt{L}_2$, and $\cdot \vdash_l \texttt{E}_3 : \texttt{L}_3$ respectively.
> if `Append [⊢ Env₁] [⊢ Env₂] [⊢ Env₃]` and `[Δ, l:target L₂[] ⊢ target T[]]`
> then `[Δ, l:target L₃[] ⊢ target T[]]`

It is translated directly into the following function:

```
rec weakenEnv1:(Δ:tctx)
              {Env₁:[⊢env L₁ E₁]}{Env2:[⊢env L₂ E₂]}{Env₃:[⊢env L₃ E₃]}
              Append [⊢Env₁] [⊢Env₂] [⊢Env₃]  → [Δ, l:target L₂[]⊢target T[]]
              → [Δ, l:target L₃[]⊢target T[]] =
λ□ Env₁, Env₂, Env₃ ⇒ fn a ⇒ fn m ⇒ case prf of
| App_nil     ⇒  m
| App_cons a' ⇒
  let [Δ,l: target _⊢P] = weakenEnv1 [⊢_] [⊢_] [⊢_] a' m in
    [Δ,l:target _⊢P[…,(crst l)] ]
;
```

We made the environment variables $\texttt{Env}_1$, $\texttt{Env}_2$, $\texttt{Env}_3$ explicit, since the property we state crucially depends on the type of the environments. The function `weakenEnv1` is implemented by recursion over the proof that `Append [⊢ Env₁] [⊢ Env₂] [⊢ Env₃]`. We could have also implemented this function recursively over $\texttt{Env}_1$. There are only two cases. If $\texttt{Env}_1$ describes the empty environment and `a` stands for `App_nil`, then we know that $\texttt{L}_2$ and $\texttt{L}_3$ are the same and hence we simply return target term `m` given as input. If $\texttt{Env}_1$ is a well-typed non-empty tuple and `a` stands for `App_cons a'`, we recursively translate `a'` leaving it up to type reconstruction to fill in the concrete instantiations for the corresponding environments. As a result, we obtain `[Δ, l:target L[]⊢M']`. We need to however return a target term in the context `Δ, l:target (cross T[] L[])`. We therefore replace `l` in `M'` with `(crst l)`. We again exploit the built-in substitution for LF objects in BELUGA to model it.

The second weakening lemma states that we can weaken the type standing for an environment by adding additional elements to the right. More precisely it says:

> For all well-typed environments $\texttt{Env}_1$ describing $\cdot \vdash_l \texttt{E}_1 : \texttt{L}_1$, $\texttt{Env}_2$ describing $\cdot \vdash_l \texttt{E}_2 : \texttt{L}_2$, and and $\texttt{Env}_3$ describing $\cdot \vdash_l \texttt{E}_3 : \texttt{L}_3$,
> if `Append [⊢ Env₁] [⊢ Env₂] [⊢ Env₃]` and `[Δ, l:target L₁[] ⊢ target T[]]`
> then `[Δ, l:target L₃[] ⊢ target T[]]`

To prove this lemma, we in fact rely on another simple lemma which states that appending an environment `Env` to an empty environment returns the environment `Env`.

```
rec append_nil: Append [⊢ Env1] [⊢ env_nil] [⊢ Env3] → [ ⊢ eq Env1 Env3] =
fn a ⇒ case a of
| App_nil      ⇒ [⊢ refl]
| App_cons a'  ⇒
  let [⊢ refl] = append_nil a' in  [⊢ refl]
;

rec weakenEnv2:(Δ:tctx)
              {Env1:[⊢env L1 E1]}{Env2:[⊢env L2 E2]}{Env3:[⊢env L3 E3]}
              Append [⊢Env1] [⊢Env2] [⊢Env3]  →
              [Δ, l:target L1[]⊢target T[]] → [Δ, l:target L3[]⊢target T[]] =
λ□ Env1, Env2, Env3 ⇒ fn a ⇒ fn n ⇒ case a of
| App_nil ⇒
  let ExEnv a  = append [⊢ Env3] [⊢ env_nil] in
  let [⊢refl] = append_nil a in  weakenEnv1 [⊢_ ] [⊢_ ] [⊢_ ] a n
| App_cons p ⇒
  let [Δ,l:target (cross S[] L1'[])⊢N]  = n in
  let [Δ,x:target S[], l:target L3'[]⊢N'] =
    weakenEnv2 [⊢_ ] [⊢_ ][⊢_ ] p [Γ,x:target S[],l:target L1'[]⊢N[…, ccons x l]]
  in [Δ,l:target (cross S[] L3'[])⊢N'[…, cfst l, crst l]]
;
```

While the implementation of `weakEnv1` is directly using the definition of `Append`, the function `weakEnv2`, which corresponds to proof of Lemma 5.3 is a bit more involved. The reason is that `Append` is defined recursively on the first environment. When we recursively analyze `a: Append [⊢ Env`$_1$`] [⊢ Env`$_2$`] [⊢ Env`$_3$`]` we learn in the base case that `Env`$_1$ is the empty environment. Therefore, its type is `unit` and we need to build a target term in the context `Δ, l:target L`$_3$`[]` given a target term in the context `Δ, l:target unit`. We also know that `Env`$_2$ and `Env`$_3$ are the same. We first show that there exists an environment `Env'` s.t. `Append [⊢ Env`$_3$`] [⊢ env_nil] [⊢ Env']`. Then we show that `Env'` is uniquely determined, i.e. it is in fact `Env`$_3$. Finally, we use `weakenEnv1` (Lemma 5.2), to show that given `Append [⊢ Env`$_3$`] [⊢ env_nil] [⊢ Env`$_3$`]` and a target term in the context `Δ, l:target unit` there exists a target term in the context `Δ, l:target L`$_3$`[]`.

In the recursive case, the environment `Env`$_1$ stands for a tuple `(env_cons P Env`$_1'$`)` of type `(cross S L`$_1'$`)` and `Env`$_3$ stands for a tuple `(env_cons P Env`$_3'$`)` of type `(cross S L`$_3'$`)`. Hence, we need to construct a target term in the context `Δ,l:target (cross S[] L`$_3'$`[])` given a target term `n` in the context `Δ,l:target (cross S[] L`$_1'$`[])`. By the assumption, we know that `a:Append [⊢ Env`$_1'$`] [⊢ Env`$_2$`] [⊢ Env`$_3'$`]` and we want to recursively weaken `n` which stands for `[Δ,l:target (cross S[] L`$_1'$`[]) ⊢ N]`. The problem however is that `Env`$_1'$ is an environment tuple of type `L`$_1'$`[]`, not `(cross S[] L`$_1'$`[])`.

We therefore employ a clever trick: we replace any occurrence of `l` in `N` where `l` stood for an environment of type `(cross S[] L`$_1'$`[])` with `ccons x l` in the context `Δ,x:target S[],l:target L`$_1'$`[]` and then recursively weaken the term `N[…, ccons x l]`. As a result we obtain a term `N'` in the context `Δ, x:target S[], l:target L`$_3'$`[]`. Note that the type of the function guarantees the shape of the context. We now must translate the term `N'` back to the context `Δ, l:target (cross S[] L`$_3'$`[])`. Again we rely on substitution and replace any occurrence of `x` with the first projection and any occurrence of `l` with the second projection. As is apparent being able to silently substitute for variables in a term plays a crucial role in our implementation and we benefit substantially from the built-in support provided by Beluga.

```
inductive Result:{Δ:tctx} [ ⊢ tp] → ctype =
| Result : [Δ,l:target L_f[] ⊢ target T[]] → [ ⊢ env L_f E]
         → Result [Δ] [ ⊢ T]
;

rec hcc: [Γ⊢source T[]] → Map [Δ] [Γ] → Result [Δ] [⊢T] =
fn m ⇒ fn ρ ⇒ case m of
| [Γ⊢#p] ⇒
  let [Δ⊢Q] = lookup ρ [Γ⊢#p] in
    Result [Δ, l:target unit⊢Q [...]] [⊢env_nil]

| [Γ⊢app M N] ⇒
  let Result r1 [⊢ Env1]  = hcc [Γ⊢M] ρ in
  let Result r2 [⊢ Env2]  = hcc [Γ⊢N] ρ in
  let ExEnv (a : Append [⊢ _] [⊢ _] [⊢Env3])  = append [⊢Env1] [⊢Env2] in
  let [Δ, l:target L[]⊢P] = weakenEnv2 [⊢ _] [⊢ _] [⊢ _] a r1 in
  let [Δ, l:target L[]⊢Q] = weakenEnv1 [⊢ _] [⊢ _] [⊢ _] a r2 in
   Result [Δ,l:target L[]⊢cletpack P λe.λx_f.λx_env.capp x_f (ccons Q[...,l] x_env)]
          [⊢ Env3]

| [Γ⊢lam λx.M] ⇒
  let STm [Γ',x:source S[]⊢M'] wk = strengthen [Γ, x:source _⊢M] in
  let EnvClo env ρ'                 = reify [Γ'] in
  let Result r1 [⊢Env1]              = hcc [Γ',x:source _⊢M'] (extend ρ') in
  let [Δ⊢P_env]                       = lookupVars wk env ρ  in
  let [x_env:target L_Γ'[],x:target T[],l:target L[]⊢P] = r1 in
    Result [Δ,l:target (cross (code L[] (code (cross T[] L_Γ'[]) _ )) L[])
                             ⊢cpack  (capp (cfst l) (crst l)) P_env[...]]
           [⊢ env_cons (clam λl.
                          clam (λc.clet (cfst c) λx.clet (crst c)
                                                      λx_env.P[x_env, x, l]))
                        Env1]

...
;

rec hoist_cc: [⊢source T] → [⊢target T] =
fn m ⇒
 let Result [l:target _ ⊢M] (e : [⊢env _ E]) = hcc m  (M_id []) in
   [⊢clet E (λl. M)]
;
```

Fig. 15. Implementation of Closure Conversion and Hoisting in Beluga

5.4 Implementation of the Main Theorem

We now generalize the closure conversion function such that it not only closure converts a source term but also hoists all converted and closed code to the top level. The top-level function `hoist_cc` performs hoisting at the same time as closure conversion on closed terms. It relies on the generalized function `hcc` (see Fig. 15) for the main work. It takes in a map between the source context Γ and target context Δ in addition to a source term of type `T` in context Γ. Compared to our previous implementation of type-preserving closure conversion, only small changes are necessary to integrate hoisting. The first obvious change is that we return not only the closure converted term, `[Δ, l:target L_f[]⊢target T[]]`, but also the corresponding function environment, `[⊢env L_f E]`. This result is encapsulated in a compile time data-type of type `Result [Δ] [⊢ T]`. How we build the closure converted term remains essentially unchanged. The main change is apparent in the case for application `app M N`. Here, we first translate `M` obtaining the closure converted term r_1 together with an environment Env_1. Similarly, translating `N` returns the closure converted term r_2 together with an environment Env_2. We now append the environments Env_1 and Env_2 obtaining a joint environment Env_3. Finally, we weaken both, the closure converted terms r_1 and r_2, to be meaningful with respect to the environment Env_3 and build our final closure converted term.

The top-level function `hoist_cc` calls `hcc` with the initial map and the source term `m` of type `T`, producing a `Result [ ] [⊢ T]` pairing a function environment and a target term depending solely on a function environment variable `l`. It then binds, with `clet`, the function environment as `l` in the hoisted term, resulting in a closed target term of the same type.

5.5 Discussion

Our implementation of hoisting adds in the order of 100 lines to the development of closure conversion (for a total of approximately 300 lines of code) 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 certain auxiliary functions such as `weakenEnv1`. Other properties around `Append` would however have to be proven, some of which require multiple nested inductions; therefore, the complexity and length of the resulting implementation is similar.

As in our work, hoisting in Chlipala [2008] is performed at the same time as closure conversion, as a consequence of the target language not capturing that functions are closed.

In Guillemette and Monnier [2008], the authors include hoisting as part of their transformation pipeline, after closure conversion. Since the language targeted by their closure conversion syntactically enforces that functions are closed, it would have been possible for them to perform hoisting in a separate phase. In Beluga, we could perform partial hoisting on the target language of closure conversion, only lifting provably closed functions to the top level. To do so, we would use two patterns for the closure case, `[Γ ⊢ cpack M[] N]` where the function part, `M`, is closed and can thus be hoisted out, and `[Γ ⊢ cpack M N]`, where `M` may still depend on the context Γ, and as such cannot be hoisted out.

6. 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 variable equality.

In more recent work, the closure conversion algorithm together with separate typing rules has been specified in Abella [Gacek, 2008], an interactive theorem prover which supports HOAS specifications, but not dependent types at the specification level. Specifying the closure conversion algorithm using HOAS requires extra book-keeping infrastructure for tracking variables, computing the free variables in a term, and describing the mapping ρ. Moreover, when we reason about the algorithm and prove that types are preserved, we need to establish invariants about variable contexts separately, since first-class support for contexts is missing. While this demonstrates, implementing and reasoning about closure conversion using HOAS is feasible, our development is unique as we develop the algorithm and proof at the same time. In fact, our algorithm itself is the proof.

A number of typed transformations to CPS [Harper and Lillibridge, 1993; Barthe et al., 1999, 2001] have been implemented in such a way as to obtain type safety as a direct consequence of type checking the implementation in the host language's type system. This is the case of Chen and Xi [2003], who show an implementation statically guaranteed to be type safe of the CPS transform, using GADTs and over a decidedly first-order representation of terms using de Bruijn indices. Guillemette and Monnier [2006]; Chlipala [2008] achieve similar results in Haskell and Coq respectively, with term representations based on HOAS.

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 very 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.

6.1 Design Options

Beluga's provision of dependent types along with datatypes both at the computation level and the LF level gives a lot of flexibility in terms of implementation choices. Throughout the development, design choices have been made for the purpose of simplifying the implementation while still obtaining a realistic compilation pipeline.

All our object languages are encoded in LF. Alternatively, we could have encoded them as indexed datatypes. This would have allowed us, among other things, to syntactically enforce that the function part of closures is closed. It would however have made it necessary to manually implement capture-avoiding substitution functions for each language. Moreover, as indexed datatypes may not be indexed with other indexed datatypes, closure conversion and hoisting would have required additional facilities and functions to work around this limitation, for example by duplicating the encoding of terms at the LF level. Another alternative would be to construct a datatype indexed by LF terms, effectively duplicating the encoding, *witnessing* the closedness of the function part of closures. This could be used to characterize the output of closure conversion, in which case hoisting could be performed separately.

The CPS algorithm could have been encoded as an indexed datatype relating the source and target languages. This would have made it possible to prove full semantics preservation of the algorithm in Beluga. However, doing so, we would not obtain an executable code transformation. Closure conversion would have been difficult to encode as an indexed datatype as it depends on pattern matching, which is only available within Beluga functions, to maximally strengthen the variable environment (see Fig. 12). A naive closure conversion algorithm can be encoded in LF, using explicit tags to track variables, after which other properties such as semantics preservation could be proven in Beluga.

With the transformations encoded as Beluga functions, we have the option of relating contexts of the source and target languages using a joint context, as we did for CPS, or using a context relation, as we did for closure conversion and hoisting. Differences between these two approaches are discussed thoroughly in Felty and Pientka [2010] and Felty et al. [2015]. We could have used a context relation for CPS with no changes to the length or complexity of the code. It would have been complicated to use a joint context for closure conversion, as many source variables are represented by the same target variable, namely the environment, in the target context.

7. CONCLUSION

Using a representation based on HOAS, we have implemented not only a translation to continuation-passing style but also a closure conversion and hoisting, where those implementations also constitute a proof of their type-preservation.

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 systems 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 develop the program and proof hand-in-hand. In our development, the actual program *is* 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.

Developing this case study has already lead to a number of improvements for programmers wanting to use Beluga for certified programming. In particular, the support to write programs with holes and printing the typing information of the hole has been tremendously beneficial in practice. Support for automatic splitting on variables is also useful. This case study also has highlighted other areas: for example, support for variable substitutions in addition to substitution variables [Cave and Pientka, 2013], would allow us to express and guarantee that these substitutions only map variables to variables and do not change the size of the overal term. This seems an elegant way of justifying why recursing on the strengthened term during closure conversion will terminate.

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. An abridged versions of this paper appeared in the proceedings of the International Conference on Certified Programs and Proofs in December 2013 under the name "Programming type-safe transformations using higher-order abstract syntax" [Savary B. et al., 2013]. We also thank all the reviewers for this article for their careful reading and detailled feedback which helped improve the paper. This research was supported through the Natural Sciences and Engineering Research Council of Canada, the Ministère du Dèveloppement économique, de l'Innovation et de l'Exportation du Quebec and the Fonds de recherche Nature et Technologies Quebec.

8. BIBLIOGRAPHY

G. Barthe, J. Hatcliff, and M. H. Sørensen. CPS Translations and Applications: The Cube and Beyond. *Higher-Order and Symbolic Computation*, 12(2):125–170, 1999.

G. Barthe, J. Hatcliff, and M. H. Sørensen. Weak Normalization Implies Strong Normalization in a Class of Non-Dependent Pure Type Systems. *Theor. Comput. Sci.*, 269(1-2):317–361, 2001.

N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java Bytecodes. In *International Conference on Functional Programming*, pages 129–140. ACM, 1998.

POPL Tutorial: Mechanizing Meta-Theory with LF and Twelf. Carnegie Mellon University, 2009. URL `http://twelf.org/wiki/POPL_Tutorial`.

A. Cave and B. Pientka. Programming with Binders and Indexed Data-Types. In *Symposium on Principles of Programming Languages*, pages 413–424. ACM, 2012.

A. Cave and B. Pientka. First-Class Substitutions in Contextual Type Theory. In *Proceedings of the Eighth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'13)*, pages 15–24. ACM, 2013.

C. Chen and H. Xi. Implementing Typeful Program Transformations. In *Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 20–28, ACM, 2003.

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.

O. Danvy and A. Filinski. Representing Control: A Study of the CPS Transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

J. Dunfield and B. 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.

A. P. Felty and B. Pientka. Reasoning with Higher-Order Abstract Syntax and Contexts: A Comparison. In *International Conference on Interactive Theorem Proving (ITP'10)*, pages 227–242. Springer, 2010.

A. P. Felty, A. Momigliano, and B. Pientka. The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations: Part 2 - a Survey. *Journal of Automated Reasoning*, 2015.

F. Ferreira and B. Pientka. Bidirectional Elaboration of Dependently Typed Languages. In *16th International Symposium on Principles and Practice of Declarative Programming (PPDP'14)*, pages 161–174. ACM, 2014.

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. Statically Verified Type-preserving Code Transformations in Haskell. In *Programming Languages meets Program Verification*, Electronic Notes in Theoretical Computer Science (ENTCS). Elsevier, 2006.

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*, ICFP'08, 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 and D. R. Licata. Mechanizing Metatheory in a Logical Framework. *Journal of Functional Programming*, 17(4-5):613–673, 2007.

R. Harper and M. Lillibridge. Explicit Polymorphism and CPS Conversion. In *Symposium on Principles of programming Languages*, POPL '93, pages 206–219, ACM, 1993.

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

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*, LICS'89, pages 313–322. IEEE, 1989.

F. Pfenning and C. Schürmann. System description: Twelf — a Meta-Logical Framework for Deductive Systems. In *International Conference on Automated Deduction*, CADE'99, pages 202–206. Springer, 1999.

B. Pientka. A Type-theoretic Foundation for Programming with Higher-Order Abstract Syntax and First-Class Substitutions. In *Symposium on Principles of Programming Languages*, POPL'08, pages 371–382. ACM, 2008.

B. Pientka. An Unsider's Look at LF Type Reconstruction: Everything You (N)ever Wanted to Know. *Journal of Functional Programming*, 1(1–37), 2013.

B. Pientka and A. Abel. Structural Recursion over Contextual Objects. In *13th International Conference on Typed Lambda Calculi and Applications*, TLCA'15, pages 273–287. LIPIcs-Leibniz International Proceedings in Informatics, 2015.

B. Pientka and A. Cave. Inductive Beluga: Programming Proofs (System Description). In *25th International Conference on Automated Deduction*, CADE-25, pages 272–281. Springer, 2015.

B. Pientka and J. Dunfield. Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description). In *International Joint Conference on Automated Reasoning*, IJCAR'10, 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*, ESOP'08, pages 93–107. Springer, 2008.

F. Pottier. Static Name Control for FreshML. In *Symposium on Logic in Computer Science*, LICS'07, pages 356–365. IEEE, 2007.

O. Savary B., S. Monnier, and B. Pientka. Programming Type-Safe Transformations using Higher-order Abstract Syntax. In *Certified Programs and Proofs*, CPP'13. Springer, 2013.

Z. Shao and A. W. Appel. A Type-Based Compiler for Standard ML. In *Conference on Programming Language Design and Implementation*, PLDI'95, pages 116–129, 1995. ACM.

Z. Shao, C. League, and S. Monnier. Implementing Typed Intermediate Languages. In *International Conference on Functional Programming*, ICFP'98, pages 313–323. ACM, 1998.

D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *Conference on Programming Languages Design and Implementation*, PLDI'96, pages 181–192, ACM, 1996.

R. Virga. *Higher-Order Rewriting with Dependent Types.* PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 1999. CMU-CS-99-167.