Programming type-safe transformations using higher-order abstract syntax

Olivier Savary Belanger

Masters of Science

School of Computer Science

McGill University

Montreal, Canada

April 2014

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Masters of Science

Acknowledgements

I would like to thank my supervisors, Prof. Brigitte Pientka and Prof. Stefan Monnier, for their guidance and support throughout my studies. I am deeply indebted to Prof. Laurie Hendren, who first welcomed me in the field of Computer Science, to Dr. Joshua Dunfield, who introduced me to the world of typed functionnal programming, and to Dr. Kaustuv Chaudhuri, with whom I learned immensely over the course of a summer internship. Finally, I would like to thank Prof. Jacques Carrette for his valuable feedback, Dr. Mathieu Boespflug for helping in fleshing out the background chapter, and Prof. Andrew Appel and the anonymous reviewers of CPP 2013 for their helpful comments on this work and on future extensions.

This work was supported in part by a Master's research scholarship (B1) of the Fonds de recherche du Québec - Nature et technologies (FQRNT).

Earlier, abridged versions of Chapters 6 and 7 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]. Contribution of the Authors: Olivier Savary Belanger is responsible for the implementation and the proofs of type preservation for the included code transformations. Prof. Brigitte Pientka and Prof. Stefan Monnier have supervised the project, advising on the algorithms used and on the presentation of the work.

Abstract

Compiling high-level languages requires complex code transformations which rearrange the abstract syntax tree. Doing so can be particularly challenging for languages containing binding constructs, and often leads to subtle errors. In this thesis, we demonstrate that higher-order abstract syntax (HOAS) encodings coupled with support for contextual objects offer substantial benefits to certified programming. We implement a type-preserving compiler for the simply-typed lambda-calculus, including transformations such as closure conversion and hoisting, in the dependently-typed language Beluga with first-class support for HOAS and contextual objects. Contextual objects allow us to directly enforce contextual invariants which would otherwise be difficult to express when variable contexts exist implicitly on a meta-level, as it is the case with other HOAS encodings. 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, thereby avoiding having to implement any lemma about binder manipulation. Scope and type safety of the code transformations are statically guaranteed, and our implementation directly mirrors the proofs of type preservation.

Résumé

La compilation de langage de haut niveau demande l'application de transformations complexes réorganisant l'arbre de syntaxe abstrait (AST). Une telle réorganisation de l'AST peut être particulierement difficile lorsque le language contient des constructions de liaison de variable, d'où peuvent découler nombre d'erreurs. Dans ce mémoire, nous démontrons l'utilité de l'encodage d'opérateurs de liaison par syntaxe abstraite d'ordre supérieure (HOAS) et de terme contextuel (CMTT) pour le développement de programmes certifiés formellement. Pour ce faire, nous avons implémenté un compilateur préservant les types pour un lambda-calcul typé dans Beluga, un langage de programmation supportant les types dépendants et une notion de terme contextuel. Les termes contextuels nous permettent d'imposer directement des propriétés constantes de nature contextuelle qui seraient autrement difficile à exprimer dû à la nature du contexte d'hypothèse, dictée par l'encodage choisi. Contrairement aux développements précédents, qui abandonnent l'encodage d'opérateur de liaison par syntax abstraite d'ordre supérieur pour un encodage de premier ordre, il nous est possible de conserver notre encodage d'orde supérieur tout au long de la compilation, évitant par ce fait l'implémentation manuelle d'opérations pour la gestion d'opérateurs de liaison. Notre compilateur est formellement vérifié pour le respect de la portée et la préservation des types. De plus, notre implémentation est en correspondance directe avec une preuve de préservation des types, résultant en un minimum de code superflu.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
Introduction

When programming critical software, an important issue resides in determining that the software corresponds to the specification from which it is built. Certified software [Shao, 2010] is software whose source code is accompanied by a formal, machine-checkable proof that they are well-behaved. That a software is well-behaved is established through the correspondence between the implementation and its functional specification, but also through a series of dependability claims, properties of the program taken as necessary conditions for the software to be considered trustworthy. Properties include safety, namely that the software will not crash when being executed, but also domain-specific claims, going from information-theoretic security for certified encryption software to properties from social choice theory for electronic voting technology [Schürmann, 2013].

Modern programmers benefit from a high level language providing abstraction mechanisms and facilities to ease the development of complex software. Source code developed in a high level language has to be translated to an executable language using a *compiler*. The advantages that programmers appreciate in high level programs are also found in certification languages and tools, where layers of abstraction allow for proofs of correctness with no concerns for low level details such as specifics of memory manipulation. However, proofs of dependability claims must still be valid of the code which is being executed, generally in a low level bytecode or an assembly language. As such, certified compilers, which guarantee that claims made about the source code are still valid for the compiled program, are

highly desirable and arguably imperative for large applications of certified software.

The field of certified compilers has seen important breakthroughs in the last decade, for example with Compcert [Leroy, 2006], a fully certified compiler for the C programming language. When certifying compilers, we are concerned with the preservation of the semantics of programs from the source code to the compiled target. By showing that the possible executions (or the execution, in the case of a deterministic language) of compiled targets refines the possible executions dictated by the source language semantics, any claim which is true of all possible executions on the source level will be true of the compiled target, and as such compilation is considered secure. However, demonstrating semantics preservation is an intricate task. Indeed, Leroy [2006] reports certification overheads in the order of 8 times the source code. A significant portion of this effort might be avoidable if the claims made at the source level depend on weaker properties of the programs, or can be satisfied with an approximation of the semantics such as types.

Compiler developers have long recognized the power of types to establish key properties about complex code transformations. However, the standard approach is to type-check the intermediate representations produced by compilation. This amounts to *testing* the result of compilation via type-checking. Type-based verification methods support building correct-by-construction software, and hold the promise of dramatically reducing the costs of quality assurance. By encoding desired properties into the types of programs, verification is reduced to type-checking the developments, resulting in little to no certification overhead. Moreover, as type information follows the structure of the program, type-based verification is easier to

maintain than stand-alone proofs, and scale gracefully. This contrasts with a posteriori verification, where a small modification in the program may necessitate a complete overhaul of its certification proof.

In the present work, we explore the use of sophisticated type systems to certify compilers by implementing a type safe compiler for the simply typed lambda-calculus, including transformation to continuation-passing style (CPS), closure conversion and hoisting.

Code transformations are programs which themselves manipulate programs. We will be using the term *meta-language* to refer to the programming language in which our code transformations are developed (in this case, Beluga), and *object-language* for languages in which programs manipulated by our code transformations are written. The object-language for the input of code transformations will be referred to as *source language*, and for programs produced by code transformations as *target language*.

There are three key ingredients crucial to the success: First, we embed our object-languages in a dependently-typed language, the LF logical framework [Harper et al., 1993], and reason about them using a dependently-typed meta-language, Beluga, such that we can channel type checking to verify precise properties about our code transformations. Second, we encode our object-languages using higher-order abstract syntax (HOAS), reusing the LF function space to model object-level binders. Third, we represent open code fragments using the notion of contextual objects and first-class contexts. A contextual object, written as $[\Psi.M]$, characterizes an open LF object M which may refer to the bound variables listed in the context Ψ [Nanevski et al., 2008]. 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.

A central question when implementing code transformations is the representation of the source and target languages. HOAS is one of the most sophisticated representation techniques, eliminating the need to deal with common and notoriously tricky aspects such as α-renaming, fresh name generation and capture-avoiding substitution. However, while the power and elegance of HOAS encodings have been demonstrated in representing proofs, for example in the Twelf system [Pfenning and Schürmann, 1999], it has been challenging to exploit its power in program transformations which rearrange abstract syntax trees and move possibly open code fragments. Previous implementations (for example Chlipala [2008]; Guillemette and Monnier [2008]) have been unable to take advantage of HOAS throughout the full compiler pipeline and had to abandon HOAS in closure conversion and hoisting. We rely on the rich type system and abstraction mechanisms of the dependently-typed language Beluga [Pientka and Dunfield, 2010; Cave and Pientka, 2012] to implement a type preserving compiler using HOAS for all stages.

Our implementation of a type-preserving compiler is very compact, avoiding tedious infrastructure for manipulating binders. Our code directly manipulates intrinsically typed terms, corresponding to type derivations, and can be interpreted as an *executable* version of the proof that the compiler is type-preserving.

The main contribution of this thesis is the implementation of the *first* type-preserving closure conversion and hoisting implementation using HOAS.

Other contributions of this thesis include:

- The presentation of a type preservation proof for a continuation-passing style transformation based on Danvy and Filinski [1992] and of its implementation in Beluga over the simply typed lambda-calculus (STLC) encoded with HOAS.
- The development and presentation of a closure conversion algorithm, of its type preservation proof and of its implementation in Beluga over the STLC encoded with HOAS.
- The development and presentation of a hoisting algorithm, of its type preservation proof and of its implementation (together with closure conversion) in Beluga over the STLC encoded with HOAS.

Our work shows that programming with contextual objects offers significant benefits to certified programming. The full development is available online at `http://complogic.cs.mcgill.ca/beluga/cc-code`.

1.1 Structure of the Thesis

The structure of this thesis is as follows: We start with an overview of the field of certified compilers (Chapter 2). We then introduce the features and syntax of Beluga (Chapter 3), the proof and programming environment used to implement this work. Next, we show the source language of our compiler and give its encoding in Beluga (Chapter 4). We then present a mechanically checked type preserving implementation of a CPS transformation (Chapter 5), closure conversion (Chapter 6) and hoisting (Chapter 7). Each of these sections consists of a brief introduction to the transformation, followed by a description of the algorithm used and the language it targets, and by a sketch of the type-preservation proof. We include detailed walkthroughs of the implementation in Beluga, guided by the correspondence between the type-preservation proof and the

Beluga development. Finally, we outline future work (Chapter 8) before turning to the conclusion (Chapter 9).

Throughout this thesis, `typewriter` font is used for Beluga source code, and *italic* for $\lambda^{\rightarrow}$ and idealized code.

CHAPTER 2
Background

2.1 From Typed to Certified Code Transformations

Compilers have historically adopted a strong typing discipline in the surface language as a means to rule out a large class of bugs in the user's program, only to erase these types before many of the later stages of compilation. Interest in type preserving transformations has been spurred by the development of compilers that maintain types throughout the successive stages of the compilation pipeline [Benton et al., 1998; Shao and Appel, 1995], resulting in, for example, better optimizations [Leroy, 1992] and support for proof carrying code [Necula, 1997].

One of the most important papers on typed code transformations, Morrisett et al. [1999] defines a series of typed code transformations that take programs written in a richly typed high-level programming language such as System F to a typed assembly language (TAL). In particular, the authors define a typed CPS transform based on an earlier one proposed by Harper and Lillibridge [1993], a typed closure conversion and typed hoisting. Minamide et al. [1996] give an earlier account of closure conversion for a type passing polymorphic language. Chlipala [2007] follows the same blueprint as the compilation to TAL for the implementation of a compiler over a higher-order language with polymorphism and side-effects in Coq, all the while mechanically proving semantic preservation in addition to type preservation throughout the passes.

An alternative to closure conversion that we do not investigate in the current work is defunctionalization [Reynolds, 1972]. Defunctionalization

is a whole-program transformation. Each abstraction is assigned a unique tag and applications are interpreted using a special function that dispatches upon all the tags in the program. Pottier and Gauthier [2004] observe that closure conversion may in fact be viewed as a particular implementation of defunctionalization, where tags happen to be code pointers and dispatching on a tag is replaced with a single indirect jump. Nielsen [2000]; Banerjee et al. [2001] give a simply typed version of defunctionalization, while Pottier and Gauthier [2004] generalize the source language to System F extended with GADTs, which they observe to be closed under defunctionalization.

Before our work, other typed transformations have been implemented in host languages in such a way as to obtain type safety as a direct consequence of type checking the implementation. For example, Chen and Xi [2003] present an implementation of a CPS transformation in Dependent ML (DML) which statically guarantees to be type safe, using GADTs and over a first-order representation of terms using de Bruijn indices. Guillemette and Monnier [2006] achieve similar results in Haskell but with a term representation based on HOAS. Linger and Sheard [2004] use Ωmega's computational facilities on the level of types to implement a statically guaranteed CPS transformation using a first-order representation, which, unlike previously mentioned work, does not rely on explicitly passing around proofs encoded at the term level. It is these lines of work that are most closely related to ours, with the proviso that authors usually make do with first-order encodings. In particular, we are not aware of any static type safe closure conversion operating on higher-order encodings.

Code transformations have also been mechanically proven to be semantics preserving, which is a stronger statement than merely preserving types. Dargaye and Leroy [2007] prove the correctness of a CPS transform

in Coq. Chlipala [2008] proves semantic preservation of CPS, closure conversion and hoisting transformations using parametric weak higher order abstract syntax (PHOAS). However, the author had, in the case of closure conversion, to revert locally to a first-order representation, equivalent to de Bruijn levels encoding, in order to distinguish variables.

2.2 Theorem Provers Using Higher-Order Abstract Syntax

Higher-order abstract syntax (HOAS) is an encoding where binders of an object language are represented using the function space of the meta-language. A description of this encoding technique is included in Chapter 3 of this thesis. Although HOAS is one of the most sophisticated encoding techniques for structures with binders and holds the promise of dramatically reducing the proof overhead related to manipulating abstract syntax trees with binders, the implementation of a certified compiler using HOAS has been elusive.

Abella [Gacek, 2008] is an interactive theorem prover which supports HOAS, but not dependent types at the specification level. The standard approach would be to specify source terms, typing judgments, and the closure conversion algorithm, and then prove that it is type-preserving. However, in Abella one cannot obtain an executable program from the proof. Moreover, the separation between algorithms and proofs of type-preservation would lead to a duplication of code. This is in contrast to our work, where we manipulate intrinsically typed representations of our object languages (see Sec. 3.1.2), which corresponding to type-derivations rather than terms. We thus implement, in place of each transformation, their respective type preservation proof.

Twelf [Pfenning and Schürmann, 1999] is a theorem prover which shares with Beluga its data level language, the LF logical framework. However,

where LF objects are reasoned about in Beluga using indexed datatypes and functions, Twelf provides a logic programming engine, employing proof search to construct derivations. CPS transformations over an object language using HOAS as binder encoding is simple to implement, and known to be an example where the substitution principle provided by LF makes for a compact and direct implementation of the transformation.[1] Closure conversion in Twelf using higher-order abstract syntax is still an open problem. This is because non-naive closure conversion algorithms require discriminating between bound variables, which cannot be done directly in Twelf. In Twelf, hypotheses are grouped in a single, implicit context. Hoisting, whose safety heavily relies on static contextual information, would require significant work, which may include reifying contexts into LF and carrying witnesses of variable occurrences to reason about contextual dependencies.

Closer to Beluga is Delphin [Poswolsky and Schürmann, 2009], a functional programming language manipulating data encoded in the LF logical framework. Having a functional programming language as mode of interaction with the prover allows for certified implementations to be closer to regular implementations based on function specifications. This is not the case when using logic programming or tactic-based provers such as Twelf and Abella respectively. However, Delphin, which is not based on contextual type theory, does not provide the necessary abstraction to distinguish between multiple contexts [Pientka, 2008], which is essential to our closure conversion and hoisting implementations (see Chapters 6 and 7).

[1] CPS was used as an example in the Twelf tutorial at POPL 2009 [Twe, 2009]

2.3 Time Regained

The work presented in this thesis builds on a number of previous work in the field of typed and certified compilation. While the tools available, both in term of theorem provers and of programming languages, have considerably progressed through the years, the objectives are roughly the same. We embrace a rich programming environment, Beluga, providing us with facilities such as explicit substitutions, first-class contextual objects and support for HOAS. We follow the pipeline of code transformations from TAL [Morrisett et al., 1999], exploring the use of modern theorem provers to compiler verification and adapting the algorithms found in the work mentioned in this chapter to harness the benefits of Beluga's proof and programming environment. While we do not verify that our transformations preserve semantics, our work is in line with a number of advancement in tools for certifying software, in particular in describing how to use the abstractions provided by Beluga to achieve lower verification overhead.

CHAPTER 3
Beluga

Beluga [Pientka and Dunfield, 2010; Pientka, 2008; Cave and Pientka, 2012] is a dependently-typed proof and programming environment based on contextual type theory [Nanevski et al., 2008]. It uses a two-level approach, segregating data from computation. Data is defined in the logical framework LF [Harper et al., 1993], a dependently-typed language with support for higher-order abstract syntax (HOAS). At the computational level, functions and datatypes are defined over contextual objects, which represent potentially open LF terms within their contexts.

In this chapter, we establish, through a series of examples, the syntax of Beluga, foreshadowing the use of its various facilities in our compiler. More thorough introductions to Beluga and to programming with dependent types are available in Pientka [2010] and on the website of the Beluga project[1] .

3.1 LF Logical Framework

The LF logical framework provides a meta-language to define higher-order, dependently-typed specifications of formal systems. The object languages of our compiler are encoded in LF, representing programs as abstract syntax trees built from LF data constructors. In Beluga, the *kind* **type** declares an LF type family. For readability, we will display $\rightarrow$ for `->` in Beluga code.

[1] http://complogic.cs.mcgill.ca/beluga/tutorial.pdf

```
datatype tp: type =
| nat : tp
| arr : tp → tp → tp
;
```

Figure 3–1: Example of a Simple LF Datatype

`tp`, from Fig. 3–1, is an example of a simple LF type family encoding a type language with a base type for natural numbers and a type $A \to B$ for functions. It has two constructors, `nat` and `arr`. The former has arity 0, and represents the type of natural numbers. `arr`, the LF representation of $\to$, combines two objects of type `tp` into a single object of type `tp`. An example of an LF term of type `tp` would be `arr nat nat`, which would be interpreted as $\mathbb{N} \to \mathbb{N}$, the type of functions from natural numbers to natural numbers.

3.1.1 Higher-Order Abstract Syntax

LF supports encodings based on higher-order abstract syntax (HOAS), a representation technique where hypothetical derivations are represented as functions of the meta-language.

```
datatype term : type =
| app : term → term → term
| lam : (term → term) → term
| z : term
| s : term → term
;
```

Figure 3–2: The Lambda-Calculus with $\mathbb{N}$ as an LF Datatype

`term` (see Fig. 3–2) is an LF datatype representing an untyped lambda calculus with natural numbers ($\mathbb{N}$). The lambda term λ`x.x 1` would be represented in LF `term` as `lam \x.app x (s z)`. By using higher-order abstract syntax to represent binders in our object language, we reuse the variable facilities from the meta-language, providing us with, amongst other

things, fresh-name generation, selecting names which do not appear in the scope for variables, and α-renaming, which renames variables while avoiding their capture. Substitution is also obtained for free, reducing to function application. This contrasts with first-order encodings such as de Bruijn indices and named representations [Gabbay and Pitts, 1999], where the programmer has to provide his own implementation of capture-avoiding substitution and naming facilities such as fresh-name generation.

3.1.2 Dependent Types

LF is dependently typed, which means that types can be indexed by terms and as such *depend* on them. This allows us to encode precise invariants in types, which are enforced by type-checking the program.

```
datatype source: tp → type =
| app : source (arr T S) → source T → source S
| lam : (source T → source S) → source (arr T S)
| z : source nat
| s : source nat → source nat
;
```

Figure 3–3: The STLC as an LF Datatype

We can use dependent types and index the LF datatype `source` by `tp`, as shown in Fig. 3–3. This indexing, resulting in an intrinsically typed representation, internalizes typing derivations – a `source` object can only represent well-typed terms in the source language it represents. Reusing our previous example, λ`x.x 1` is represented in `source` as `lam \x.app x (s z)`, the same as in `term`, but its type would now be `source (arr (arr nat T)T)` rather than `term`. However, it would be impossible to represent the ill-typed term `(`λ`x.x 1)1` in `source`. This is because `app (lam \x.app x (s z))(s z)`, the LF `term` in direct correspondence with `(`λ`x.x 1)1`, is not derivable from the rules of `source`. Indeed, the second argument to the `app` constructor should be of type `source (arr nat T)`, as dictated by the

type of `(lam \x.app x (s z))`, `source (arr (arr nat T)T)`, however `suc z` is of type `source nat`. In the remainder of this thesis, we display λ in Beluga code as a pretty-printed version of \.

3.2 Computational Level

Once data is defined, we can analyze it recursively on the computational level. In Beluga, this is done by programming, using pattern matching on contextual objects in dependently-typed recursive functions. This section introduces each of these concepts in turn. We use as an example the development of a substitution function over `source` terms to highlight the underlying ideas.

3.2.1 Contextual Objects and Contexts

A contextual object $[\Psi.M]$ of contextual type $[\Psi.A]$ [Nanevski et al., 2008] is composed of an open LF term M which has type A within a context Ψ. Its contextual type precisely characterizes what variables are allowed to occur in M and A, internalizing the concept of a well-scoped open term. In Beluga, when using the modality `[ ]`, the `.` separates the context of assumptions from the conclusion. Pattern matching on a contextual object refines our knowledge of its type and of the form of its context, allowing us to statically reason about open objects.

The context Ψ represents the pool of hypotheses which can appear in the typing derivation of M. The substitution principle is admissible in contextual type theory, meaning that we can always substitute a concrete derivation for a hypothesis of the same type.

In Beluga, contexts consist of zero or one context variables, followed by zero or more concrete variable declarations. Context variables are classified by schemas (keyword **schema**), which describe what can be contained in a context. Schemas are formed by a list of clauses, each

consisting of a tuple of LF terms, grouped with the keyword `block`. Free variables appearing in each clauses are existentially bound at the head of the clause, explicitly using the construct **some** `T1`... `T`n for variables T_1 ... T_n, or implicitly, exploiting Beluga's type reconstruction engine.

```
schema sctx = source s
schema tsctx = block t1:source s, t2:source s
```

Figure 3–4: Example of Schema Definitions in Beluga

In Fig. 3–4, we present two schema definitions as example. The schema `sctx` characterizes contexts containing `source` variables of arbitrary `tp`, while the schema `tsctx` characterizes contexts pairing two `source` assumptions of the same type as a hypothesis.

3.2.2 Inductive Datatypes

Beluga supports inductive datatypes indexed by contextual objects [Cave and Pientka, 2012]. They can be used to assert relations between contexts and contextual objects. The syntax to write such a datatype is similar to the one of an LF datatype, with the exception that its type family is defined in *kind* **ctype** rather than in **type**. Inductive datatypes may be formed of other inductive datatypes, of contexts and of contextual objects, guarded by a constructor.

```
datatype Subst: {Δ:sctx}{Γ:sctx} ctype =
| Emp : Subst [Δ] []
| Dot : Subst [Δ] [Γ] → [Δ. source T]
     → Subst [Δ] [Γ,x:source T]
;
```

Figure 3–5: An Inductive Datatype for `source` Substitutions

`Subst`, given in Fig. 3–5, is an indexed datatype representing a substitution for `source` variables. `Subst` is indexed by two contexts, Γ and

Δ, both of schema `sctx`, respectively the domain and the codomain of the substitution. The first constructor of `Subst`, `Emp`, establish the existence of a substitution from the empty context to any context of schema `sctx`. The second constructor of `Subst`, `Dot`, extends the domain of a substitution from Γ to `[`Γ`,x:source T]` by adding a source term of type `T` in the codomain of the substitution Δ. As an example, the subsitution exchanging `x` and `y` in the context `[x:source S,y:source T]` would be encoded as `Dot (Dot Emp [y:source T,x:source S. x])[y:source T,x:source S. y]`, which has type `Subst [y:source T,x:source S] [x:source S,y:source T]`.

3.2.3 Functions

On the computational level of Beluga, functions can be defined over contextual objects, contexts and indexed datatypes. The syntax of Beluga distinguishes between two function spaces, dependent functions, corresponding to Π, whose domain are named and may appear in types, and simply-typed functions, corresponding to $\rightarrow$. On the computational level, $\rightarrow$ is taken as the simply-typed function space, overloading the notation for the LF function space. A function signature `T`, in Beluga, is formed from inductive datatypes `a` $\vec{C}$, contextual objects `U`, functions types `T->T'`, and of Π quantifiers `{X:U}T` binding a variable of contextual type `U` in type `T`.

Within the body of a function, a variable `e:T` is bound with the construct `fne =>` if emanating from the function type $T \rightarrow T'$, and with `mlame =>` if Π-quantified. For readability, we will display $\lambda^{\square}$ for `mlam` and $\Rightarrow$ for `=>` in Beluga code.

Beluga functions manipulate their inputs through pattern matching, destructing them into their components and reasoning with them before constructing an output. Pattern matching over contextual objects recovers

information about the *shape* of an object, but also about its context and dependencies to its context, both of which are explicit in Beluga's contextual object representation.

A case construct, in Beluga, is formed of an expression being matched, `e` in the example given in Fig. 3–6, and of a series of patterns which will be tested sequentially. Execution chooses the first branch whose pattern matches the expression, with all the variables appearing in the patterns bound as their respective matchee in `e`. In Beluga, terms in

```
case e of
| [Γ. M... ] ⇒ ...
| [Γ,x:term, y:term. M... y] ⇒ ...
| [Γ,x:term, y:term. M ] ⇒ ...
```

Figure 3–6: Example of Case-Construct and Patterns in Beluga

contextual objects are associated with an explicit substitution, carrying information about which variables from the context may appear in them. A ... represents an identity substitution over a context variable, indicating that the term may depend on variables from the context variables appearing in the context portion of the contextual object. Concrete variables from the context may only appear in terms if they are mentioned explicitly in the substitution. For example, in `[Γ,x:term, y:term. M... y]`, `M` may depend on variables from Γ or on `y`, but not on `x`, while in `[Γ,x:term, y:term . M]`, `M` can't refer to any variables – we will say that it is *closed*. In the remainder of this thesis, we display ... for `..` in Beluga code.

Where a stronger function space would treat terms under binders as *black boxes*, we can observe the internals of LF functions. As such, we may use the pattern `λx. M`, where `λx.` is an LF binder for `x`, and `M` is an LF term which can be further refined by observation. `[Γ. lam (λx. M... x)]` is a well-formed contextual object of contextual type `[Γ. source (arr S T)]`

for some `tp S` and `T`. `M... x` itself lives in the context Γ extended with a source variable of type `S`, forming the contextual object `[Γ,x:source S. M ... x]` of contextual type `[Γ,x:source S. source T]`.

```
case sub of
| Emp ⇒ ...
| Dot sub' [Δ. M... ] ⇒ ...
```

Figure 3–7: Example of a Case-Construct on an Inductive Datatype

We can also use pattern matching to observe inductive datatypes. In the example included in Fig. 3–7, where `sub` is a `Subst [Δ] [Γ]`, the `Emp` branch would have Γ being empty, while the `Dot` branch would have `Γ= [Γ',x:source T]`, `sub'` of type `Subst [Δ] [Γ']` and `[Δ. M... ]` of contextual type `[Δ. source T]`.

As an example, we give in Fig. 3–9 the definition of a type-preserving substitution function, taking as input a `source` term of type `T` in the context Γ and a substitution (`Subst`) from context Γ to Δ, and producing a `source` term of the same type in context Δ. We will consider here only three cases: patterns matching on variables, applications and lambda-abstractions, as they are representative of the other cases forming the full function. As mentioned in Subsection 3.1.1, the encoding of our object language `source T` in LF using HOAS already provides us with a substitution principle, such that `Subst` and `applySubst` would be superfluous except when manipulating explicit substitutions is required. We include it here solely to demonstrate the syntax and different features of Beluga functions.

We first present, in Fig. 3–8, a simplified version of the code, with the contexts and contextual dependencies omitted.

```
rec applySubst:
fn (e:source T) ⇒ fn (sub:Subst) ⇒ case e of
| app M N ⇒
 let M' = applySubst M sub in
 let N' = applySubst N sub in
  app M' N'
| lam (λx. M x) ⇒
  let (M' x) = applySubst (M x) (Dot sub x) in
   lam (λx. M' x)
| #p ⇒ ... % look #p up in sub
;
```

Figure 3–8: Simplified Beluga Code of a Substitution Function

In the application case, matched by the pattern `app M N`, we recursively apply the substitution `sub` to the subterms `M` and `N` before reconstructing the application with the updated terms `M'` and `N'`.

In the lambda-abstraction case, matched by `lam (λx. M x)`, we extend the substitution with the identity substitution on `x` using the constructor `Dot`. `x` is re-abstracted over in the returned term `M'` using the `lam` constructor.

The last pattern, `#p`, matches `source` term variables from the domain of `sub`. `#` is a special pattern prefix in Beluga that may only match assumptions of the right type from the context rather than general terms. It would, for example, match `x`, but not `lam (λx. x)`. We can then unfold the constructors of `sub` to reveal the term associated to `#p` in the codomain of `sub`. For example, if `#p` corresponds to one of the variables extending `sub` as we go *under* lambda-binders in `applySubst`, we would return the same variable in the codomain of the substitution.

In Fig. 3–9, we present the Beluga code of `applySubst`, revisiting each cases in more detail.

```
rec applySubst:
[Γ. source T] → Subst [Δ] [Γ] → [Δ. source T] =
fn e ⇒ fn sub ⇒ case e of
| [Γ. app (M… ) (N… )] ⇒
 let [Δ. M'… ] = applySubst [Γ. M… ] sub in
 let [Δ. N'… ] = applySubst [Γ. N… ] sub in
 [Δ. app (M'… ) (N'… )]
| [Γ. lam (λx. M… x)] ⇒
  let (sub:Subst [Δ] [Γ]) = sub in
  let [Δ,x:source S. M'… x] = applySubst [Γ,x:source _. M… x]
      (Dot sub [Δ,x:source _. x]) in
  [Δ. lam (λx. M'… x)]
| [Γ. #p… ] ⇒ lookupSubst [Γ. #p… ] sub
…
;
```

Figure 3–9: Extract from the Implementation of a Substitution Function

Its first pattern, `[Γ. app (M… )(N… )]`, matches applications. We use let-constructs to recurse on its subterms `M` and `N` and bind the resulting terms before combining them as an application in context Δ.

The second pattern, `[Γ. lam (λx. M… x)]`, matches lambda-abstractions. As previously mentioned, we can manipulate terms under an LF binder by adding the local variable to the context. An annotated let-construct is used to recover the names for contexts Γ and Δ. After extending the substitution `sub` with the identity on `x`, we recurse on the body of the lambda-abstraction. Explicit contexts should here make assumptions manipulation clearer than in the code of Fig. 3–8: to observe the body of the lambda-abstraction, we extend the typing context of the lambda-abstraction with a typing assumption on the local variable `x:source _`. Underscores (`_`) can stand in for unnamed variables and are filled automatically by Beluga during *reconstruction*. On return, we bind back the local variable while reconstructing a lambda-abstraction of the right contextual type.

The last pattern, `[Γ. #p... ]`, matches variables from the context Γ, which corresponds to the domain of the substitution `sub`. We can use the function `lookupSubst`, defined for all variables of Γ, to find the `source` term in context Δ corresponding, in our substitution `sub`, to the variable `#p`. The signature of `lookupSubst`, given in Fig. 3–10, is a dependent

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

Figure 3–10: Implementation of the Substitution Function for Variables

function over variables of type `source T` within context Γ, restricting the first argument of the function to be a variable (we note here that we use the dependent function space Π instead of the simply-typed one → even though `#p` doesn't occur in the rest of the signature as Beluga doesn't provide a syntax for the latter over variables). The second argument and the return type are the same as the ones from `applySubst`. We pattern match on `#p` to see if it is the first variable of the context, in which case we return the term `M`. If it is not the first variable of the context, we recurse with the inner substitution `sub'` on the rest of the context. In both cases, as `#p` is a variable from Γ and we know that Γ is not empty. Thus, `sub` is of the form `Dot sub' [Δ. M... ]`.

CHAPTER 4
Source Language

Before dwelling into the intricacies of our type preserving transformations, we present, in this chapter, the source language for our compiler.

4.1 The Simply Typed Lambda-Calculus

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

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

Figure 4–1: Syntax of the source language

Each of our type-preserving algorithms transforms the source language of Figure 4–1 to a separate target language, where all our object languages share the same language for types. N-ary products are constructed using the binary product $S \times T$ and unit. In closure conversion, we will use n-ary tuples to describe the environment. Foreshadowing closure conversion, and inspired by the type language of Guillemette and Monnier [2007], we add a special type $\mathsf{code}\ S\ T$. This type only arises as a result of closure conversion; while it is present in the type language, no `source` terms of type λ`codety S T` may be constructed.

The typing rules for the language presented in Fig. 4–1 are given in Fig. 4–2. The `source` term for lambda-abstraction λ`x.M` is well-typed when

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

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

$$\frac{\Gamma \vdash M : T \quad \Gamma, x : T \vdash N : S}{\Gamma \vdash \mathsf{let}\ x\ =\ M\ \mathsf{in}\ N : S}\ \mathsf{t\_let} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \mathsf{t\_var}$$

$$\frac{\Gamma \vdash M : T \times S}{\Gamma \vdash \mathsf{fst}\ M : T}\ \mathsf{t\_first} \qquad \frac{\Gamma \vdash M : T \times S}{\Gamma \vdash \mathsf{rst}\ M : S}\ \mathsf{t\_rest}$$

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

Figure 4–2: Typing rules for the source language

`M` is well-typed in a typing context extended with a typing assumption for `x`. Application `M N` may be formed at type `S` from a well-typed `M` at function type `T` $\to S$ and a well-typed `N` of type `T`. Let-construct $\mathsf{let}\ x = M\ \mathsf{in}\ N$ is well-typed at type `S` if M is well-typed at some type `T` and N is well-typed at type `S` in a context extended with a typing assumption x`:T`. Variables are well-typed if a typing assumption for them is present in the context. Selectors `fst M` and `rst M` are well-typed if `M` is well-typed at a product type `T`$\times$`S`.

4.2 Representing the Source Language in LF

The encoding of the source language presented in Figure 4–3 completes the definition of `source` given in Chapter 3.

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

Figure 4–3: Encoding of the source language in LF

As a reminder, in Beluga's concrete syntax, the *kind* **type** declares an LF type family, as opposed to a computational data type. Binders in our object languages are represented via HOAS, using the LF function space. We use an intrinsically typed representation: by indexing `source` terms with their types, we represent type derivation 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 $\Gamma \vdash M : T$. N-ary tuples are encoded 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.

CHAPTER 5
Continuation Passing Style

In this chapter, we present an algorithm to translate a direct style source language to a continuation-passing style (CPS) target language, prove that it is type-preserving, and describe its implementation in Beluga. 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. Continuation-passing style is used in many compilers for functional languages for it 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].

5.1 Target Language

The language targeted by our CPS transformation is shown in Fig. 5–1.

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

Figure 5–1: Syntax of the Target Language for CPS

The target language is divided into values, expressions, and continuations. We give all the typing rules for this language in Fig 5–2.

The well-typedness of terms of the target language is defined through three judgments, each referring to a typing context Δ. Context Δ contains

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

$$\frac{x : T \in \Delta}{\Delta \vdash x : T}\ \mathsf{t\_var} \qquad \frac{\Delta, x : T, k\ \bot_S \vdash P\ \bot}{\Delta \vdash \lambda(x, k).\, P : T \to S}\ \mathsf{t\_lam}$$

$$\frac{\Delta \vdash V :_1 S \quad \Delta \vdash V :_2 T}{\Delta \vdash (V_1, V_2) : S \times T}\ \mathsf{t\_cons} \qquad \frac{}{\Delta \vdash () : \mathsf{unit}}\ \mathsf{t\_unit}$$

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

$$\frac{\Delta \vdash V_1 : S \to T \quad \Delta \vdash V_2 : S \quad \Delta \vdash K\ \bot_T}{\Delta \vdash V_1\ V_2\ K\ \bot}\ \mathsf{t\_app} \qquad \frac{\Delta \vdash V : T}{\Delta \vdash \mathsf{halt}\ V \bot}\ \mathsf{t\_halt}$$

$$\frac{\Delta \vdash K\ \bot_T \quad \Delta \vdash V : T}{\Delta \vdash K\ V\ \bot}\ \mathsf{t\_kapp} \qquad \frac{\Delta \vdash V : T \quad \Delta, x : T \vdash P\ \bot}{\Delta \vdash \mathsf{let}\ x = V\ \mathsf{in}\ P\ \bot}\ \mathsf{t\_let}$$

$$\frac{\Delta \vdash V : S \times T \quad \Delta, x : S \vdash P\ \bot}{\Delta \vdash \mathsf{let{-}fst}\ x = V\ \mathsf{in}\ P\ \bot}\ \mathsf{t\_first} \qquad \frac{\Delta \vdash V : S \times T \quad \Delta, x : T \vdash P\ \bot}{\Delta \vdash \mathsf{let{-}rst}\ x = V\ \mathsf{in}\ P\ \bot}\ \mathsf{t\_rest}$$

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

$$\frac{x\ \bot_T \in \Delta}{\Delta \vdash x\ \bot_T}\ \mathsf{t\_kvar} \qquad \frac{\Delta, x : T \vdash P\ \bot}{\Delta \vdash \lambda x.\, P\ \bot_T}\ \mathsf{t\_klam}$$

Figure 5–2: Typing Rules for the Target Language of CPS

typing assumptions for values `x:T`, where variable `x` stands in for a value of type `T`, and for continuations `k` $\bot_T$, where variable `k` stands in for a well-typed continuation of the form $\bot_T$.

Values consist of variables, lambda-abstractions and tuples of values. They are typed using judgement $\Delta \vdash V : T$, signifying that a value V has type T in the typing context Δ. The typing rules for values are straightforward, being the same as the ones for typing `source` terms, with the exception of t_lam, with lambda-abstractions now taking an explicit continuation as argument. $\lambda(x, k).P$ 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 as input values of type S.

Expressions P are typed using judgment $\Delta \vdash P \perp$ for a well-typed P in context Δ, representing that they do not return anything by themselves but instead rely 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 base expression $\mathsf{halt}\ V$, a general let-construct, $\mathsf{let}\ x = V\ \mathsf{in}\ P$ and two let-constructs to observe lists, $\mathsf{let{-}fst}\ x = V\ \mathsf{in}\ P$ and $\mathsf{let{-}rst}\ x = V\ \mathsf{in}\ P$.

Finally, we define the well-typedness of a continuation through the judgment $\Delta \vdash K \perp_T$, where K is a continuation, either a continuation variable or a well-typed expression expecting an input values of type S in the context Δ.

5.2 CPS Algorithm

We give the definition of the translation to continuation-passing style in Figure 5–3. $[\![M]\!]_k = P$ takes as input a source term M and produces a target term P depending on k, where k is a (fresh) variable standing in for the top-level continuation in the translated expression.

The algorithm used, adapted from Danvy and Filinski [1992], eliminates administrative redexes *on the fly*, which is to say that all redexes created during the translation when substituting concrete continuations in place of continuation variables in a continuation application as typed by $\mathsf{t\_kapp}$ are reduced eagerly by the algorithm.

Lemma 1. ***Substitution (Continuation in Expression)***

If $\Gamma, k\ \perp_T \vdash P\ \perp$ *and* $\Gamma \vdash K\ \perp_T$ *then* $\Gamma \vdash [K/k]P\ \perp$

Proof. Proof by induction on the derivation $\Gamma, k\ \perp_T \vdash P\ \perp$.

$$
\begin{array}{lcll}
[\![x]\!]_k & = & k\ x & \\
[\![\lambda x.\, M]\!]_k & = & k\ (\lambda(x, k_1).\, P) & \text{where} \quad [\![M]\!]_{k_1} = P \\
[\![M\ N]\!]_k & = & [(\lambda p.[(\lambda q.p\ q\ k)/k_2]Q)/k_1]P & \text{where} \quad [\![M]\!]_{k_1} = P \\
 & & & \text{and} \quad [\![N]\!]_{k_2} = Q \\
[\![(M, N)]\!]_k & = & [(\lambda p.[(\lambda q.k\ (p, q))/k_2]Q)/k_1]P & \text{where} \quad [\![M]\!]_{k_1} = P \\
 & & & \text{and} \quad [\![N]\!]_{k_2} = Q \\
[\![\mathsf{let}\ x = N\ \mathsf{in}\ M]\!]_k & = & [(\lambda q.\mathsf{let}\ x = q\ \mathsf{in}\ P)/k_2]Q & \text{where} \quad [\![M]\!]_k = P \\
 & & & \text{and} \quad [\![N]\!]_{k_2} = Q \\
[\![\mathsf{fst}\ \ M]\!]_k & = & [(\lambda p.\mathsf{let}{-}\mathsf{fst}\ x = p\ \mathsf{in}\ k\ x)/k_1]P & \text{where} \quad [\![M]\!]_{k_1} = P \\
[\![\mathsf{rst}\ \ M]\!]_k & = & [(\lambda p.\mathsf{let}{-}\mathsf{rst}\ x = p\ \mathsf{in}\ k\ x)/k_1]P & \text{where} \quad [\![M]\!]_{k_1} = P \\
[\![()]\!]_k & = & k\ () &
\end{array}
$$

Figure 5–3: CPS Algorithm

While we could prove a number of substitution properties between our judgments (see Fig. 5–2), in the proof of preservation, we only need to substitute a continuation in an expression. Lemma 1 states that we can substitute a continuation for a continuation variable of the same type in a well-formed expression.

Theorem 1. ***Type Preservation***

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

Our main theorem for this transformation, Theorem 1, states that a source expression of type T will be transformed by the algorithm given in Figure 5–3 to a 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 1 follows by structural induction on the typing derivation of the source term. The substitution prefixing terms in the proof are performed during the algorithm, and should be read as the term resulting from the

substitution, reducing continuation applications eagerly, rather than as a term together with a delayed substitution.

Proof. By induction on the typing derivation $\Gamma \vdash M : T$.

Case. $\Gamma \vdash x : T$

hence $M = x$ 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_kvar
$\Gamma, k\ \bot_T \vdash k\ x\ \bot$	by t_kapp
$\Gamma, k\ \bot_T \vdash [\![M]\!]_k\ \bot$	by definition

Case. $\Gamma \vdash \lambda x.M' : T \to S$

hence $M = \lambda x.M'$ and $[\![M]\!]_k = k\ (\lambda(x, k).[\![M']\!]_k)$

$\Gamma \vdash \lambda x.M' : T \to S$	by assumption
$\Gamma, x : T \vdash M' : S$	by inversion on t_lam
$\Gamma, x : T, k\ \bot_S \vdash [\![M']\!]_k\ \bot$	by i.h.
$\Gamma \vdash \lambda(x, k).[\![M']\!]_k : T \to S$	by t_lam
$\Gamma, k\ \bot_{T\to S} \vdash \lambda(x, k).[\![M']\!]_k : T \to S$	by weakening
$\Gamma, k\ \bot_{T\to S} \vdash k\ \bot_{T\to S}$	by t_kvar
$\Gamma, k\ \bot_{T\to S} \vdash k\ (\lambda(x, k).[\![M']\!]_k)\ \bot$	by t_kapp
$\Gamma, k\ \bot_{T\to S} \vdash [\![\lambda x.M']\!]_k\ \bot$	by definition

Case. $\Gamma \vdash M'\ N : T$

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

$\Gamma \vdash M'\ N : T$	by assumption
$\Gamma \vdash M' : S \to T$ and $\Gamma \vdash N : S$	by inversion on t_app
$\Gamma, k_2\ \bot_S \vdash [\![N]\!]_{k_2}\ \bot$	by i.h.

$\Gamma, k\ \bot_T, p : S \to T, q : S \vdash p : S \to T$ by t_var

$\Gamma, k\ \bot_T, p : S \to T, q : S \vdash q : S$ by t_var

$\Gamma, k\ \bot_T, p : S \to T, q : S \vdash k\ \bot_T$ by t_kvar

$\Gamma, k\ \bot_T, p : S \to T, q : S \vdash p\ q\ k\ \bot$ by t_app

$\Gamma, k\ \bot_T, p : S \to T \vdash \lambda q.p\ q\ k\ \bot_S$ by t_klam

$\Gamma, k\ \bot_T, p : S \to T, k_2\ \bot_S \vdash [\![N]\!]_{k_2}\ \bot$ by weakening, exchange

$\Gamma, k\ \bot_T, p : S \to T \vdash [\lambda q.p\ q\ k/k_2][\![N]\!]_{k_2}\ \bot$ by substitution (Lemma 1)

$\Gamma, k\ \bot_T \vdash \lambda p.[\lambda q.p\ q\ k/k_2][\![N]\!]_{k_2}\ \bot_{S\to T}$ by t_klam

$\Gamma, k_1\ \bot_{S\to T} \vdash [\![M']\!]_{k_1}\ \bot$ by i.h.

$\Gamma, k\ \bot_T, k_1\ \bot_{S\to T} \vdash [\![M']\!]_{k_1}\ \bot$ by weakening, exchange

$\Gamma, k\ \bot_T \vdash [\lambda p.[\lambda q.p\ q\ k/k_2][\![N]\!]_{k_2}/k_1][\![M']\!]_{k_1}\ \bot$

by substitution (Lemma 1)

$\Gamma, k\ \bot_T \vdash [\![M'\ N]\!]_k\ \bot$ by definition

Case. $\Gamma \vdash (M', N) : T \times S$

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

$\Gamma \vdash (M', N) : T \times S$ by assumption

$\Gamma \vdash M' : T$ and $\Gamma \vdash N : S$ by inversion on t_cons

$\Gamma, k_2\ \bot_S \vdash [\![N]\!]_{k_2}\ \bot$ by i.h.

$\Gamma, k\ \bot_{T\times S}, p : T, q : S \vdash p : T$ by t_var

$\Gamma, k\ \bot_{T\times S}, p : T, q : S \vdash q : S$ by t_var

$\Gamma, k\ \bot_{T\times S}, p : T, q : S \vdash (p, q) : T \times S$ by t_cons

$\Gamma, k\ \bot_{T\times S}, p : T, q : S \vdash k\ \bot_{T\times S}$ by t_kvar

$\Gamma, k\ \bot_{T\times S}, p : T, q : S \vdash k\ (p, q)\ \bot$ by t_kapp

$\Gamma, k\ \bot_{T\times S}, p : T \vdash \lambda q.k\ (p, q)\ \bot_S$ by t_klam

$\Gamma, k\ \bot_{T\times S}, p : T, k_2\ \bot_S \vdash [\![N]\!]_{k_2}\ \bot$ by weakening, exchange

$\Gamma, k\ \bot_{T\times S}, p : T \vdash [\lambda q.k\ (p, q)/k_2][\![N]\!]_{k_2}\ \bot$ by substitution (Lemma 1)

$\Gamma, k_1 \perp_T \vdash [\![M']\!]_{k_1} \perp$ by i.h.

$\Gamma, k \perp_{T\times S}, k_1 \perp_T \vdash [\![M']\!]_{k_1} \perp$ by weakening, exchange

$\Gamma, k \perp_{T\times S} \vdash \lambda p.[\lambda q.k\ (p,q)/k_2][\![N]\!]_{k_2} \perp_T$ by t_klam

$\Gamma, k \perp_{T\times S} \vdash [\lambda p.[\lambda q.k\ (p,q)/k_2][\![N]\!]_{k_2}/k_1][\![M']\!]_{k_1} \perp$

by substitution (Lemma 1)

$\Gamma, k \perp_{T\times S} \vdash [\![(M', N)]\!]_k \perp$ by definition

Case. $\Gamma \vdash \mathsf{let}\ x = N\ \mathsf{in}\ M' : T$

hence $M = \mathsf{let}\ x = N\ \mathsf{in}\ M'$

and $[\![M]\!]_k = [\lambda q.\mathsf{let}\ x = q\ \mathsf{in}\ [\![M]\!]_k/k_2][\![N]\!]_{k_2}$

$\Gamma \vdash \mathsf{let}\ x = N\ \mathsf{in}\ M' : T$ by assumption

$\Gamma \vdash N : S$ and $\Gamma, x : S \vdash M' : T$ by inversion on t_let

$\Gamma, x : S, k \perp_T \vdash [\![M']\!]_k \perp$ by i.h.

$\Gamma, k \perp_T, q : S, x : S \vdash [\![M']\!]_k \perp$ by weakening, exchange

$\Gamma, k \perp_T, q : S \vdash q : S$ by t_var

$\Gamma, k \perp_T, q : S \vdash \mathsf{let}\ x = q\ \mathsf{in}\ [\![M']\!]_k \perp$ by t_let

$\Gamma, k \perp_T \vdash \lambda q.\mathsf{let}\ x = q\ \mathsf{in}\ [\![M']\!]_k \perp_S$ by t_klam

$\Gamma, k_2 \perp_S \vdash [\![N]\!]_{k_2} \perp$ by i.h.

$\Gamma, k \perp_T, k_2 \perp_S \vdash [\![N]\!]_{k_2} \perp$ by weakening, exchange

$\Gamma, k \perp_T \vdash [\lambda q.\mathsf{let}\ x = q\ \mathsf{in}\ [\![M']\!]_k/k_2][\![N]\!]_{k_2} \perp$ by substitution (Lemma 1)

$\Gamma, k \perp_T \vdash [\![\mathsf{let}\ x = N\ \mathsf{in}\ M']\!]_k \perp$ by definition

Case. $\Gamma \vdash \mathsf{fst}\ \ M' : T$

hence $M = \mathsf{fst}\ \ M'$ and $[\![M]\!]_k = [\lambda p.\mathsf{let}{-}\mathsf{fst}\ x = p\ \mathsf{in}\ k\ x/k_1][\![M']\!]_{k_1}$

$\Gamma \vdash \mathsf{fst}\ \ M' : T$ by assumption

$\Gamma \vdash M' : T \times S$ by inversion on t_fst

$\Gamma, k_1 \perp_{T\times S} \vdash [\![M']\!]_{k_1} \perp$ by i.h.

$\Gamma, k\ \bot_T, p : T \times S, x : T \vdash k\ \bot_T$ by t_kvar

$\Gamma, k\ \bot_T, p : T \times S, x : T \vdash x : T$ by t_var

$\Gamma, k\ \bot_T, p : T \times S, x : T \vdash k\ x\ \bot$ by t_kapp

$\Gamma, k\ \bot_T, p : T \times S \vdash p : T \times S$ by t_var

$\Gamma, k\ \bot_T, p : T \times S \vdash \mathsf{let{-}fst}\ x = p\ \mathsf{in}\ k\ x\ \bot$ by t_first

$\Gamma, k\ \bot_T \vdash \lambda p.\mathsf{let{-}fst}\ x = p\ \mathsf{in}\ k\ x\ \bot_{T\times S}$ by t_klam

$\Gamma, k\ \bot_T, k_1\ \bot_{T\times S} \vdash [\![M']\!]_{k_1}\ \bot$ by weakening, exchange

$\Gamma, k\ \bot_T \vdash [\lambda p.\mathsf{let{-}fst}\ x = p\ \mathsf{in}\ k\ x/k_1][\![M']\!]_{k_1}\ \bot$

by substitution (Lemma 1)

$\Gamma, k\ \bot_T \vdash [\![\mathsf{fst}\ \ M']\!]_k\ \bot$ by definition

Case. $\Gamma \vdash \mathsf{fst}\ \ M' : T$

hence $M = \mathsf{rst}\ \ M$ and $[\![M]\!]_k = [\lambda p.\mathsf{let{-}rst}\ x = p\ \mathsf{in}\ k\ x/k_1][\![M']\!]_{k_1}$

$\Gamma \vdash \mathsf{rst}\ \ M' : T$ by assumption

$\Gamma \vdash M' : S \times T$ by inversion on t_snd

$\Gamma, k_1\ \bot_{S\times T} \vdash [\![M']\!]_{k_1}\ \bot$ by i.h.

$\Gamma, k\ \bot_T, p : S \times T, x : T \vdash k\ \bot_T$ by t_kvar

$\Gamma, k\ \bot_T, p : S \times T, x : T \vdash x : T$ by t_var

$\Gamma, k\ \bot_T, p : S \times T, x : T \vdash k\ x\ \bot$ by t_kapp

$\Gamma, k\ \bot_T, p : S \times T \vdash p : S \times T$ by t_var

$\Gamma, k\ \bot_T, p : S \times T \vdash \mathsf{let{-}rst}\ x = p\ \mathsf{in}\ k\ x\ \bot$ by t_rest

$\Gamma, k\ \bot_T \vdash \lambda p.\mathsf{let{-}rst}\ x = p\ \mathsf{in}\ k\ x\ \bot_{S\times T}$ by t_klam

$\Gamma, k\ \bot_T, k_1\ \bot_{S\times T} \vdash [\![M']\!]_{k_1}\ \bot$ by weakening, exchange

$\Gamma, k\ \bot_T \vdash [\lambda p.\mathsf{let{-}rst}\ x = p\ \mathsf{in}\ k\ x/k_1][\![M']\!]_{k_1}\ \bot$

by substitution (Lemma 1)

$\Gamma, k\ \bot_T \vdash [\![\mathsf{rst}\ \ M']\!]_k\ \bot$ by definition

Case. $\Gamma \vdash () : \mathsf{unit}$

hence $M = ()$ and $[\![M]\!]_k = k\ ()$

$\Gamma, k\ \bot_{\mathsf{unit}} \vdash k\ \bot_{\mathsf{unit}}$ by t_kvar

$\Gamma, k\ \bot_{\mathsf{unit}} \vdash k\ ()\ \bot$ by t_kapp

$\Gamma, k\ \bot_{\mathsf{unit}} \vdash [\![()]\!]_k\ \bot$ by definition

5.3 Representing the Target Language in LF

As with the source language (see Chapter 4), we encode the target language in the LF logical framework as intrinsically well-typed terms (see Fig. 5–4). We reuse `tp`, the type index of the `source` language, to index `value`, which correspond to values defined in Figure 5–1. 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. Datatype `exp` is used to represent well-typed expressions (Figure 5–1), where an LF term of type `exp` corresponds to a typing derivation $\Gamma \vdash E\ \bot$ for a (unique) expression E. Continuations are not present at this level, being represented directly as LF functions from values to expressions on the computational level of Beluga.

```
datatype exp : type =
 | kapp : value (arr S T) → value S
       → (value T → exp) → exp
 | klet : value S → (value S → exp) → exp
 | kfst : value (cross S T) → (value S → exp) → exp
 | krst : value (cross S T) → (value T → exp) → exp
 | halt : value S → exp

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

Figure 5–4: Encoding of the Target Language of CPS in LF

5.4 Implementation of the Main Theorem

The Beluga function `cpse` implements the translation $[\![-]\!]_k$ together with its type preservation proof. Our description of CPS (see Fig. 5–3) is

defined on open terms M, meaning that they may contain free variables. Moreover, if we assume that M is well-typed in the context Γ, then it must be well-scoped, which is to say that all the free variables in M must be declared in Γ. Similarly, P is open and its typing context contains, for each free `source` variable in M, a corresponding target free variable.

Our CPS translation function `cpse`, presented in Fig. 5–7, takes as input a term M of type `source T` in the context Γ and produces an expression P of type `exp T`. As there is a one to one correspondence between source variables and target variables, we define the context Γ as a joint context, where we store pairs of source and target variables of the same type.

```
schema ctx = block x:source t, y:value t;
```

Figure 5–5: Definition of Schema `ctx`

The **schema** `ctx` (see Fig. 5–5) classifies contexts containing pairs of `source` variables of type `T` and `value` variables of the same type using the `block` keyword.

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

Figure 5–6: Signature of the Main Function `cpse`

The signature of `cpse`, given in Fig. 5–6, can be read as follows: For all contexts Γ, given a source term of type `T` in the context Γ, `cpse` returns a well-typed expression which depends on the continuation `k` of type $\bot_T$ in the context Γ.

Our implementation of the CPS translation follows closely the proof of Type Preservation (Theorem 1) for the algorithm presented in Figure 5–3, with, for example, recursive calls to `cpse` corresponding to appeals to the

induction hypothesis. It consists of a single downward pass on the input program, relying on LF application to reduce administrative redexes. The substitution principle presented in Lemma 1 is provided by Beluga, due to our use of contextual objects to model the judgment $\Gamma \vdash P \perp$. Similarly, appealing to structural lemmas such as exchange and weakening is implicit in the implementation. Beluga recognizes, for example, the term `[Γ,y,x. M… x y]` whenever `[Γ,x,y. M… x y]` is valid.

Figure 5–7 presents the implementation of the transformation in Beluga. The first pattern, `#p.1` matches the first field of members of Γ, corresponding to source variables. Given a continuation `c`, we return the second field of the same member, corresponding to a target variable of the same type, applied to `c`. Type reconstruction fills in the `_` in the type of `c` with the type of the matched variable.

In the application case, we match on the pattern `app (M… )(N… )` and recursively transform `M` and `N`. We then substitute for the continuation variable `k` 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 `k` in `P`, producing a well-typed expression if a continuation for the resulting type `S` is provided. Redexes formed by the substitution of the built continuation for `k` in `P` are reduced automatically, as continuation application is modeled directly using LF function application. The rest of the cases are similar and in direct correspondence with the proof of Theorem 1. In the lambda-abstraction and the let-construct case, the typing annotation `{N:[Γ, x:source S. source T]}` is used to name `S`, the type of `x`, which extends the context in the recursive call to `cpse`.

```
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, k:value (arr T S) → exp. P… k] = cpse [Γ. M… ] in
 let [Γ, k:value T → exp. Q… k] = cpse [Γ. N… ] in
  [Γ, k:value S → exp. P… (λf. Q… (λx. kapp f x k))]
| {M:[Γ, x:source S. source T]}
    [Γ. lam (λx. M… x)] ⇒
  let [Γ, b:block (x:source S, _t:value S), c:value T → exp.
      P… b.2 c] =
   cpse [Γ, b:block (x:source S, _t:value S). M … b.1 ] in
    [Γ, k:value (arr S T) → exp. k (klam (λx.λc. P… x c))]
| [Γ. cons (M… ) (N… )] ⇒
  let [Γ, k1:value S → exp. P… k1] = cpse [Γ. M… ] in
  let [Γ, k2:value T → exp. Q… k2] = cpse [Γ. N… ] in
   [Γ, k:value (cross S T) → exp.
    P… (λp. Q… (λq. k (kcons p q)))]
| [Γ. nilv] ⇒ [Γ, c:value unit → exp. c knil]

| {N:[Γ, x:source S. source T]}
 [Γ. letv (M… ) (λx. N… x)] ⇒
  let [Γ, k1:value S → exp. P… k1] = cpse [Γ. M… ] in
  let [Γ, b:block (x:source S, _t:value S), c:value T → exp.
      Q… b.2 c] =
   cpse [Γ, b:block (x:source S, _t:value S). N … b.1 ] in
   [Γ, k:value T → exp. P… (λp. klet p (λx. Q… x k))]
| [Γ. fst (E… )] ⇒
    let [Γ, c:value (cross S T) → exp. E'… c] = cpse [Γ.
                                                    E… ] in
     [Γ, c:value S → exp. E'… (λx. kfst x c)]
| [Γ. rst (E… )] ⇒
    let [Γ, c:value (cross S T) → exp. E'… c] = cpse [Γ.
                                                    E… ] in
     [Γ, c:value T → exp. E' … (λx. krst x c)]
;
```

Figure 5–7: Implementation of CPS in Beluga

5.5 Discussion and Related Work

The implementation of the continuation-passing style type-preservation transformation in Beluga, including the definition of the type, source and target languages, amounts to less than 65 lines of code. The fact that this algorithm is simple to implement on a language using HOAS is not a new result, nor is it surprising, as it relies heavily on substitution, which

is provided by the meta-language in HOAS encodings. 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. Little change is required for our simply typed implementation to support System F: adding the identity substitution ... to types appearing in annotations and function signatures 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 thesis (see Ch. 6 and Ch. 7), but would 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.

As demonstrated in the included program (Figure 5–7), features of Beluga 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.

CHAPTER 6
Closure conversion

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

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 (which is to say, not in continuation passing style) on the lambda-calculus.

6.1 Target Language

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

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

Figure 6–1: Syntax of the Target Language for Closure Conversion

The essential idea of closure conversion is that it makes 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 $\lambda x.M$ of type $T \to S$, we return a closure $\langle \lambda y_c.P, Q \rangle$ consisting of a closed function $\lambda y_c.P$, where y_c pairs the local argument y, standing in for x, and an environment variable y_{env} whose projections replace free variables of M, and its environment Q, containing all its free variables. Such packages are traditionally given an existential type such as $\exists l.(\mathsf{code}\ (T \times l)\ S) \times l$ where l is the type of the environment. We instead use $T \to S$ to type the closure packages, hiding l and saving us from having to handle existential types in their full generality.

We use the following judgements for our typing rules (see Fig. 6–2):

$$\Delta \vdash P : T \qquad \text{Target } P \text{ has type } T \text{ in context } \Delta$$

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 $\lambda x.P$ is closed. The remaining typing rules are similar to the typing rules for the source language.

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

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

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

$$\frac{\Delta \vdash P : T \times S}{\Delta \vdash \mathsf{fst}\ P : T}\ \mathsf{t\_first} \qquad \frac{\Delta \vdash P : T \times S}{\Delta \vdash \mathsf{rst}\ P : S}\ \mathsf{t\_rest} \qquad \frac{}{\Delta \vdash () : \mathsf{unit}}\ \mathsf{t\_unit}$$

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

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

Figure 6–2: Typing Rules for the Target Language of Closure Conversion

6.2 Closure Conversion Algorithm

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

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

When evaluated, the program reduces to:

$$\mathsf{let}\ \langle f_1, c_1\rangle = \langle\ \lambda(y, e_1).\mathsf{fst}\ e_1 + y\ \ ,\ \ (\,5\,,\,()\,)\,\rangle\ \ \mathsf{in}\ \ f_1\ (2\,,\,c_1)$$

and then further to:

$$(\lambda(y, e_1).\mathsf{fst}\ e_1 + y)\ (2\ ,\ (5\ ,\ ()))$$

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 Figure 6–4 using $[\![M]\!]_\rho$, where M is a source term which is well-typed in the context Γ and ρ a mapping of source variables in Γ to target terms in the context Δ.

Intuitively, ρ maps source variables to the corresponding projection of the environment. It is defined as follows:

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

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

Figure 6–3: Formation Rules for Mappings

For convenience, we write π_i for the i-th projection instead of using the selectors fst and rst . As an example, $\pi_2\ M$ would correspond to the term $\mathsf{fst}\ (\mathsf{rst}\ M)$.

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

Figure 6–4: Closure Conversion Algorithm

To translate a source variable, we look up its binding in the map ρ. To translate tuples and projection, we translate the subterms before reassembling the result using target language constructs. () is directly translated to its target equivalent. Translating let-expression $\mathsf{let}\ x = M\ \mathsf{in}\ N$ involves translating M using the mapping ρ and translating N with the extended map $\rho, x \mapsto x$, 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 lambda-abstraction and application.

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

• A term P, obtained by converting M with the new map ρ' which maps variables $x_1, \ldots, x_n$ to their corresponding projection of the environment variable and x to itself, thereby eliminating all free variables in M.

• An environment tuple P_{env}, obtained by applying ρ to each variable in $(x_1, \ldots, x_n)$.

When translating an application $M\ N$, we first translate M and N to target terms P and Q. Since the source term M denotes a function, the target term P will denote a closure. We extract the two parts of the closure using a let-pack construct, obtaining x_f, the function, and x_{env}, the environment, before applying the extended environment (Q, x_{env}) to x_f.

We implement the described algorithm in Beluga as a recursive program which manipulates intrinsically well-typed source terms. As these encodings represent typing derivations rather than terms, as discussed in the presentation of the `source` language, our program represent a transformation over typing derivation. To understand the general idea behind our implementation, we discuss how to prove that given a well-typed source term M we can produce a well-typed target term which is the result of converting M. The proof relies on several straightforward lemmas which correspond exactly to auxiliary functions needed in our implementation.

Lemma 1. ***Term Strengthening***

If $\Gamma \vdash M : T$ *and* $\Gamma' = FV(M)$ *then* $\Gamma' \vdash M : T$ *and* $\Gamma' \subseteq \Gamma$

Proof. Proof using an auxiliary lemma: if $\Gamma_1, \Gamma_2 \vdash M : T$ then $\Gamma_1', \Gamma_2 \vdash M : T$ for some $\Gamma_1' \subseteq \Gamma_1$ which is proven by induction on Γ_1.

Term Strengthening (Lemma 1) says that the set of typing assumption Γ' as computed by FV is sufficient to type any term `M` if this term is well-typed under some context Γ, and that Γ contains at least the same assumptions as Γ'.

Lemma 2. ***Term Weakening***

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

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

Term Weakening (Lemma 2) says that a term `M` stays well-typed at type `T` if we substitute its typing context Γ for a context Γ' containing all of the assumptions in Γ.

Lemma 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* $\rho = x_1 \mapsto \pi_1\ x_{env}, \ldots, x_n \mapsto \pi_n\ x_{env}$ *s.t.* $x_{env} : T_\Gamma \vdash \rho : \Gamma$ *and* $\Gamma \vdash (x_1, \ldots x_n) : T_\Gamma$.

Proof. By induction on Γ

Context Reification (Lemma 3) says that it is possible to represent an arbitrary context of typing assumptions Γ as a single typing assumption $\mathtt{x}_{env} : \mathtt{T}_\Gamma$. The type $\mathtt{T}_\Gamma$ corresponds to the type of a tuple formed from the term component of each assumption in Γ. As a witness to this claim, a substitution ρ is produced, taking any terms referring to variables from Γ to one solely referring to $\mathtt{x}_{env}$.

Lemma 4. ***Map Extension***

If $\Delta \vdash \rho : \Gamma$ *then* $\Delta, x : T \vdash \rho, x \mapsto x : \Gamma, x : T$.

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

Map Extensions (Lemma 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 application of rules of formation for substitutions (see Fig. 6–3). 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 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 5) states, intuitively, that substitutions as encoded by our mapping judgment (see Fig. 6–3) work as intended: for any variable in Γ, the substitution ρ produces a term of the same type in its target context, Δ.

Lemma 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 5 and inversion on the typing rules.

Map Lookup (Tuple) (Lemma 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.

Theorem 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 $\Gamma \vdash M : T$

Case. $\Gamma \vdash x : T$

hence $M = x$ and $[\![M]\!]_\rho = \rho(x)$.

$\Gamma \vdash x : T$ and $\Delta \vdash \rho : \Gamma$ by assumption

$\Delta \vdash \rho(x) : T$ by Map lookup (Lemma 5)

$\Delta \vdash [\![x]\!]_\rho$: T by definition

Case. $\Gamma \vdash \lambda x.M' : T \to S$

hence $M = \lambda x.M'$

and $[\![M]\!]_\rho = \langle \lambda c.\, \mathsf{let}\ x = \mathsf{fst}\ c\ \mathsf{in}\ \mathsf{let}\ x_{env} = \mathsf{rst}\ c\ \mathsf{in}\ P\ ,\ P_{env} \rangle$

$\Gamma \vdash \lambda x.M' : T \to S$ and $\Delta \vdash \rho : \Gamma$ by assumption

$\Gamma' \vdash \lambda x.M' : T \to S$ and $\Gamma' \subseteq \Gamma$

where $\Gamma' = FV(\lambda x.M')$ by Term strengthening (Lemma 1)

$\Gamma', x : T \vdash M' : S$ by inversion on $\mathsf{t\_lam}$

$\Gamma' \vdash (x_1, \ldots, x_n) : T_{\Gamma'}$ and

$x_{env} : T_{\Gamma'} \vdash \rho' : \Gamma'$ by Context reification (Lemma 3)

$\Gamma \vdash (x_1, \ldots, x_n) : T_{\Gamma'}$ by Term Weakening (Lemma 2)

$\Delta \vdash \rho : \Gamma$ by assumption

$(\rho(x_1), \ldots, \rho(x_n)) = P_{env}$ by assumption

$\Delta \vdash P_{env} : T_{\Gamma'}$ by Map lookup (tuple) (Lemma 6)

$x_{env} : T_{\Gamma'}, x : T \vdash \rho', x \mapsto x : \Gamma', x : T$ By Map extension (Lemma 4)

$x_{env} : T_{\Gamma'}, x : T \vdash P : S$

where $P = [\![M]\!]_{\rho', x \mapsto x}$ by i.h. on M

$c : T \times T_{\Gamma'}, x : T, x_{env} : T_{\Gamma'} \vdash P : S$ by Term weakening (Lemma 2)

$c : T \times T_{\Gamma'}, x : T \vdash \mathsf{let}\ x_{env} = \mathsf{rst}\ c\ \mathsf{in}\ P : S$ by rule $\mathsf{t\_let}$

$c : T \times T_{\Gamma'} \vdash \mathsf{let}\ x = \mathsf{fst}\ x\ \mathsf{in}\ \mathsf{let}\ x_{env} = \mathsf{rst}\ c\ \mathsf{in}\ P : S$ by rule $\mathsf{t\_let}$

$\cdot \vdash \lambda c.\, \mathsf{let}\ x = \mathsf{fst}\ c\ \mathsf{in}\ \mathsf{let}\ x_{env} = \mathsf{rst}\ c\ \mathsf{in}\ P : \mathsf{code}\ (T \times T_{\Gamma'})\ S$ by rule $\mathsf{t\_lam}$

$\Delta \vdash \langle \lambda c.\, \mathsf{let}\ x = \mathsf{fst}\ c\ \mathsf{in}\ \mathsf{let}\ x_{env} = \mathsf{rst}\ c\ \mathsf{in}\ P\ ,\ P_{env} \rangle : T \to S$

by rule t_pack

$\Delta \vdash [\![\lambda x.M']\!]_\rho : T \to S$ by definition

Case. $\Gamma \vdash M'\ N : T$

hence $M = M'\ N$

and $[\![M]\!]_\rho = \mathsf{let}\ \langle x_f, x_{env}\rangle = [\![M']\!]_\rho\ \mathsf{in}\ x_f\ ([\![N]\!]_\rho\ ,\ x_{env})$

$\Gamma \vdash M'\ N : T$ and $\Delta \vdash \rho : \Gamma$ by assumption

$\Gamma \vdash M' : S \to T$ and $\Gamma \vdash N : S$ by inversion on t_app

$\Delta \vdash [\![M']\!]_\rho : S \to T$ by i.h.

$\Delta \vdash [\![N]\!]_\rho : S$ by i.h.

$\Delta, x_{env} : l \vdash [\![N]\!]_\rho : S$ by term weakening

$\Delta, x_{env} : l \vdash x_{env} : l$ by rule t_var

$\Delta, x_{env} : l \vdash ([\![N]\!]_\rho\ ,\ x_{env}) : (S \times l)$ by rule t_cons

$\Delta, x_f : \mathsf{code}\ (S \times l)\ T, x_{env} : l \vdash ([\![N]\!]_\rho\ ,\ x_{env}) : (S \times l)$

by term weakening, exchange

$\Delta, x_f : \mathsf{code}\ (S \times l)\ T \vdash x_f : \mathsf{code}\ (S \times l)\ T$ by rule t_var

$\Delta, x_f : \mathsf{code}\ (S \times l)\ T, x_{env} : l \vdash x_f : \mathsf{code}\ (S \times l)\ T$ by term weakening

$\Delta, x_f : \mathsf{code}\ (S \times l)\ T, x_{env} : l \vdash x_f\ ([\![N]\!]_\rho\ ,\ x_{env}) : T$ by rule t_app

$\Delta \vdash \mathsf{let}\ \langle x_f, x_{env}\rangle = [\![M']\!]_\rho\ \mathsf{in}\ x_f\ ([\![N]\!]_\rho\ ,\ x_{env}) : T$ by rule t_letpack

$\Delta \vdash [\![M'\ N]\!]_\rho : T$ by definition

Case. $\Gamma \vdash \mathsf{let}\ x = N\ \mathsf{in}\ M' : T$

hence $M = \mathsf{let}\ x = N\ \mathsf{in}\ M'$

and $[\![M]\!]_\rho = \mathsf{let}\ x = [\![N]\!]_\rho\ \mathsf{in}\ [\![M']\!]_{\rho, x \mapsto x}$

$\Gamma \vdash \mathsf{let}\ x = N\ \mathsf{in}\ M' : T$ and $\Delta \vdash \rho : \Gamma$ by assumption

$\Gamma \vdash N : S$ and $\Gamma, x : S \vdash M' : T$ by inversion on t_let

$\Delta \vdash [\![N]\!]_\rho : S$ by i.h.

$\Delta, x : S \vdash \rho, x \mapsto x : \Gamma, x : S$ by Map extension (Lemma 4)

$\Delta, x : S \vdash [\![M']\!]_{\rho, x \mapsto x} : T$ by i.h.

$\Delta \vdash \mathsf{let}\ x = [\![N]\!]_\rho\ \mathsf{in}\ [\![M']\!]_{\rho, x \mapsto x} : T$ by rule t_let

$\Delta \vdash [\![\mathsf{let}\ x = N\ \mathsf{in}\ M']\!]_\rho : T$ by definition

Case. $\Gamma \vdash (M', N) : T$

hence $M = (M', N)$ and $[\![M]\!]_\rho = ([\![M']\!]_\rho, [\![N]\!]_\rho)$

$\Gamma \vdash (M', N) : T$ and $\Delta \vdash \rho : \Gamma$ by assumption

$\Gamma \vdash M' : T_1$ and $\Gamma \vdash N : T_2$

where $T = T_1 \times T_2$ by inversion on t_cons

$\Delta \vdash [\![M']\!]_\rho : T_1$ by i.h.

$\Delta \vdash [\![N]\!]_\rho : T_2$ by i.h.

$\Delta \vdash ([\![M']\!]_\rho, [\![N]\!]_\rho) : T_1 \times T_2$ by rule t_cons

$\Delta \vdash [\![(M', N)]\!]_\rho : T$ by definition

Case. $\Gamma \vdash \mathsf{fst}\ M' : T$

hence $M = \mathsf{fst}\ M'$ and $[\![M]\!]_\rho = \mathsf{fst}\ [\![M']\!]_\rho$

$\Gamma \vdash \mathsf{fst}\ M' : T$ and $\Delta \vdash \rho : \Gamma$ by assumption

$\Gamma \vdash M' : T \times S$ by inversion on t_first

$\Delta \vdash [\![M']\!]_\rho : T \times S$ by i.h.

$\Delta \vdash \mathsf{fst}\ [\![M']\!]_\rho : T$ by rule t_first

$\Delta \vdash [\![\mathsf{fst}\ \ M']\!]_\rho : T$ by definition

Case. $\Gamma \vdash \mathsf{rst}\ M' : T$

hence $M = \mathsf{rst}\ M'$ and $[\![M]\!]_\rho = \mathsf{rst}\ [\![M']\!]_\rho$

$\Gamma \vdash \mathsf{rst}\ M' : T$ and $\Delta \vdash \rho : \Gamma$ by assumption

$\Gamma \vdash M' : S \times T$ by inversion on t_rest

$\Delta \vdash [\![M']\!]_\rho : S \times T$ by i.h.

$\Delta \vdash \mathsf{rst}\ [\![M']\!]_\rho : T$ by rule t_rest

$\Delta \vdash [\![\mathsf{rst}\ M']\!]_\rho : T$ by definition

Case. $\Gamma \vdash () : \mathsf{unit}$

hence $M = ()$ and $[\![M]\!]_\rho = ()$

$\Gamma \vdash () : \mathsf{unit}$ by assumption

$\Delta \vdash () : \mathsf{unit}$ by rule t_unit

$\Delta \vdash [\![()]\!]_\rho : \mathsf{unit}$ by definition

□

6.3 Representating the Target Language in LF

In this section, we describe the implementation of the closure conversion algorithm in Beluga. The representation of the target language from Figure 6–1 in LF is given in Figure 6–5.

```
datatype target: tp → type =
| clam : (target T → target S) → target (code T S)
| capp : target (code T S) → target T → target S
| cpack : target (code (cross T L) S) → target L
      → target (arr T S)
| cletpack: target (arr T S)
      → ({l:tp} target (code (cross T l)) S)
            → target l → target S)
      → target S
| cfst : target (cross T S) → target T
| crst : target (cross T S) → target (prod S)
| ccons: target T → target S → target (cross T S)
| cnil : target unit
| clet : target T → (target T → target S) → target S;
```

Figure 6–5: Encoding of the Target Language of Closure Conversion in LF

An LF term of type `target` T in a context Δ, where T is a `tp`, corresponds to a typing derivation $\Delta \vdash P : T$. The data-type definition directly reflects the typing rules with one exception: our typing rule t_pack enforced that P was closed. This cannot be achieved in the LF encoding, since the context of assumptions is ambient. As a consequence, hoisting, which relies

on the fact that the closure converted functions are closed, cannot be implemented as a separate phase after closure conversion. We will come back to this issue in Chapter 7.

6.4 Type Preserving Closure Conversion in Beluga: an Overview

Our main function for the Beluga implementation of closure conversion is `cc`, whose signature, given in Fig. 6–6, corresponds to the type preservation theorem (Theorem 1) for our closure conversion algorithm (Fig. 6–4). `cc` translates well-typed source terms in a source context Γ to well-typed target terms in the target context Δ given a map of the source context Γ to the target context Δ. Δ will consists of an environment variable $\mathtt{x}_{env}$ and the variable `x` bound by the last abstraction, along with variables introduced by let bindings. Just as types classify terms, schemas

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

Figure 6–6: Signature of the Main Function `cc`

classify contexts in Beluga, similarly to world declarations in Twelf [Pfenning and Schürmann, 1999]. We include the definition of the two schemas used for this transformation in Fig. 6–7. The schema `tctx` describes contexts where each declaration is an instance of type `target T`, corresponding to Δ in Fig. 6–1; similarly the schema `sctx` describes contexts where each declaration is an instance of type `source T`, and it corresponds to Γ in Fig. 4–1. In the remainder of this chapter and in Chapter 7, we will use Γ to name contexts characterized by the schema `sctx`, and Δ for contexts of schema `tctx`. While type variables appear in the typing rule `t_letpack`, they only occur locally and are always bound before the term is returned by our functions, such that they do not appear in the schema.

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

Figure 6–7: Definition of Schemas `tctx` and `sctx`

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

Figure 6–8: Definition of `Map` as an Inductive Datatype

We use the indexed recursive type `Map`, whose definition is given in Fig. 6–8, to relate the target context Δ and source context Γ [Cave and Pientka, 2012]. As a reminder, in Beluga's concrete syntax, the *kind* **ctype** indicates that we are not defining an LF datatype, but a recursive type on the level of computations. $\rightarrow$ 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.

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

The next section presents the implementation of the necessary auxiliary functions, followed by `cc`, our main function.

6.5 Implementation of Auxiliary Lemmas

Term strengthening and weakening. We begin by implementing functions for strengthening and weakening source terms, corresponding to Lemmas 1 and 2 respectively. Both operations rely on an inclusion relation $\Gamma' \subseteq \Gamma$, which is defined using the indexed recursive computation-level data-type `SubCtx`, presented in Fig. 6–9.

```
datatype SubCtx: {Γ′:sctx} {Γ:sctx} ctype =
| WInit: SubCtx [] []
| WDrop: SubCtx [Γ′] [Γ] → SubCtx [Γ′] [Γ, x:source T]
| WKeep: SubCtx [Γ′] [Γ]
      → SubCtx [Γ′, x:source T] [Γ, x:source T];
```

Figure 6–9: Definition of `SubCtx` as an Inductive Datatype

Given a source term `M` in Γ the function `strengthen`, whose type signature is included in Fig. 6–10, computes the strengthened version of `M`, which is well-typed in the source context Γ′ characterizing the free variables in `M`, together with the proof `SubCtx [Γ′] [Γ]`. Our implementation slightly deviates from the statement of Term strengthening (Lemma 1) given earlier. Instead of computing the free variables of `M` separately, we simply construct Γ′ such that it characterizes the free variables of `M`. We represent the result using the indexed recursive type `StrTerm'` encoding the existential in the specification as a universal quantifier using the constructor `STm'`. The fact that `Γ'` describes exactly the free variables of `M` is not captured by the type definition. This is because `SubCtx [Γ] [`*Gamma*`]` is valid for all context Γ, such that we could implement a `strengthen` function which would not removed unused variables from the context.

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

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

Figure 6–10: Signature of the Function `strengthen`

Just as in the proof of the term strengthening lemma, we cannot implement the function `strengthen` directly. This is because, while we would like to perform induction on the size of Γ, we cannot appeal to the induction hypothesis while maintaining a well-scoped source term in the case

of an occurring variable in front of Γ. Instead, we implement `str`, which, intuitively, implements the lemma

> *If* $\Gamma_1, \Gamma_2 \vdash M : T$ *and* $\Gamma_1', \Gamma_2 = \mathsf{FV}(M)$, *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 `[`Γ_1`, `Γ_2`. source T]`. Instead, to express this, we use a data-type `wrap` which abstracts over all the variables in Γ_2. `wrap` is indexed by the type `T` of the source term and the size of Γ_2. The function `str` then recursively analyses Γ_1, adding variables occurring in the input term to Γ_2. The type of `str` asserts, through its index `N`, the size of Γ_2, which stays constant on input and output of a call to `str`. This is crucial to verify the totality of the function; in the case where we call `str` with an argument of the form `ainit M`, this ensures that the returned `wrap` will be of the same form, and similarly when called with some `add (`λ`x. M ..x)`.

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

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

rec str: [Γ. wrap T N] → StrTerm [Γ] [.T] [.N] =
fn e ⇒ case e of
| [ . M] ⇒ STm [. M] WInit
| [Γ, x:source T. M... ] ⇒
  let STm [Γ'. M'... ] rel = str [Γ. M... ] in
    STm [Γ'. M'... ] (WDrop rel)
| [Γ, x:source T. M... x] ⇒
  let STm [Γ'. add (λx.M'... x)] rel =
      str [Γ.add (λx. M... x)] in
    STm [Γ', x:source T. M'... x] (WKeep rel);
```

Figure 6–11: Implementation of the Function `str`

The function `str`, given in Fig. 6–11, 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` and test whether `x` occurs in it, as previously demonstrated in Fig. 3–6.

The first case, `[. M]`, is only matched if $\Gamma = \cdot$. Then, $FV(M) = \cdot$ and we initialize the subcontext relation with the `WInit` constructor.

The second case, `[Γ, x:source T. (M... )]`, means that `x`, the variable on top of the context, does not appear in `M`. We can hence strenghten `M` to context Γ, recursively call `str` on this subcontext, and use the subcontext relation constructor `WDrop` to relate Γ and Γ`, x:source T`.

Finally, the last case, `[Γ, x:source T. (M... x)]`, means that `x` may occur in `M`. We know, as the term did not match the previous pattern, that `x` does indeed occur in `M`, and we must keep it as part of `FV(M)`. We use the `add` constructor of the `wrap` datatype to add `x` the accumulator representing Γ_2, and recursively call `str` on the Γ subcontext. The type index `N`, which stays constant in calls to `str`, asserts that the `wrap` in the output will be of the form `add (λx.M'... x)`. We then use the `WKeep` subcontext relation constructor to signify that we kept `x` as part of `FV(M)`.

While one can implement term weakening following similar ideas, we incorporate it into the variable lookup function defined next.

Map extension and lookup. The function `lookup`, given in Fig. 6–12, takes a source variable of type `T` in the source context Γ and `Map [Δ] [Γ]` and returns the corresponding target expression of the same type.

We quantify over all variables in a given context by `{#p:[Γ. source T]}` where `#p` denotes a variable of type `source T` in the context Γ. In the

```
rec lookup:{#p:[Γ.source T]} Map [Δ] [Γ] → [Δ. target T] =
λ□#p ⇒ fn ρ ⇒ let (ρ: Map [Δ] [Γ]) = ρ in case [Γ. #p... ] of
| [Γ′, x:source T. x] ⇒ let Dot ρ′ [Δ. M... ] = ρ in [Δ. M... ]
| [Γ′, x:source S. #q... ] ⇒ let  Dot ρ′ [Δ. M... ] = ρ in
                              lookup [Γ′. #q... ] ρ′;
```

Figure 6–12: Implementation of the Function `lookup`

function body, $\lambda^{\square}$-abstraction introduces an explicitly quantified contextual object and **fn**-abstraction introduces a computation-level function. The function `lookup` is implemented by pattern matching on the context Γ and the parameter variable `#p`, after observing the type of ρ to introduce Γ to the namespace.

To guarantee coverage and termination, it is pertinent that we know that an n-ary tuple is composed solely of source variables from the context Γ, in the same order. We therefore define `VarTup` in Fig 6–13 as a computational datatype for such variable tuples. `Nex v` of type `VarTup [Γ] [.L`$_\Gamma$`]`, where $\Gamma =$ `x`$_1$`:T`$_1$`,..., x`$_n$`:T`$_n$, is taken to represent the source language tuple `(x`$_1$`,..., x`$_n$`)` of type $\mathtt{T}_1 \times \ldots \times \mathtt{T}_n$ in the context Γ.

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

Figure 6–13: Definition of `VarTup` as an Inductive Datatype

The function `lookupVars` (see 6–14) applies a map ρ to every variable in a variable tuple.

`lookupVars` allows the application of a `Map` defined on a more general context Γ provided that $\Gamma' \subseteq \Gamma$. This corresponds, in the theoretical presentation, to weakening a variable tuple with Lemma 2 before applying

```
rec lookupVars: VarTup [Γ′] [.L_Γ′] → SubCtx [Γ′] [Γ]
  → Map [Δ] [Γ] → [Δ. target L_Γ′] =
fn vt ⇒ fn sub ⇒ fn σ ⇒ let (σ : Map [Δ] [Γ]) = σ in
case sub of
| WInit ⇒
 let Emp = vt in % Γ′ = Γ = .
  [Δ. cnil]
| WDrop sub' ⇒
 let DotMap σ′ [Δ. P... ] = σ in
  lookupVars vt sub' σ′
| WKeep sub' ⇒
 let Nex vt' = vt in
 let DotMap σ′ [Δ. P... ] = σ in
 let [Δ. M... ] = lookupVars vt' sub' σ′ in
  [Δ. ccons (P... ) (M... )]
;
```

Figure 6–14: Implementation of the Functione `lookupVars`

a mapping on it with Lemma 6. In the first case, we learn that $\Gamma' = \Gamma = \cdot$, such that the corresponding tuple is $(\cdot)$ 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` $\mapsto$ `P` from σ.

In the third case, the top variable of Γ appears on top of Γ' as well, and we have `x` $\mapsto$ `P` in σ. We recursively construct to tuple for the rest of Γ', before adding `P` in front to get a tuple of type $\mathtt{L}_{\Gamma'}$.

`extendMap` (see Fig. 6–15), which implements Map extension (Lemma 4), weakens a mapping with the identity on a new variable `x`. It is used to extend the `Map` with local variables, for example when we encounter a **let**-binding construct.

To extend a map, we must first weaken Δ to Δ, `x:target S`, which can be done directly if our mapping is empty (constructor `IdMap`). Otherwise, we use `extendMap'` which crawls down the mapping, weakening its

```
rec extendMap': Map [Δ] [Γ]
  → Map [Δ, x:target S] [Γ] =
fn σ ⇒ case σ of
| IdMap [Δ] ⇒ IdMap [Δ, x:target _]
| DotMap σ' [Δ. M… ] ⇒
 let ρ' = extendMap' σ' in
  DotMap ρ' [Δ, x:target _. (M… )]
;

rec extendMap: Map [Δ] [Γ]
  → Map [Δ, x:target S] [Γ, x:source S] =
fn σ ⇒ case σ of
| IdMap [Δ] ⇒
      DotMap (IdMap [Δ, x:target _]) [Δ, x:target _. x]
| DotMap σ' [Δ. M… ] ⇒
 let ρ' = extendMap' σ' in
 let ρ = DotMap ρ' [Δ, x:target _. M… ] in
  DotMap ρ [Δ, x:target _. x];
```

Figure 6–15: Implementation of `extendMap`

constituents by `x`. We then add `x` $\mapsto$ `x` on top of ρ to optain a mapping from Γ, `x:source S` to Δ, `x:target S`.

A Reification of the Context as a Term Tuple. The context reification lemma (Lemma 3) is proven by induction on Γ; in Beluga, we can enable pattern matching on contexts by defining an indexed data-type `Ctx` which wraps a context Γ in a constructor.

The function `reify`, given in Fig. 6–16, translates the context Γ to a source term. It produces a tuple containing variables of Γ in order, along with `Map [x:target` $\mathtt{T}_\Gamma$`] [Γ]` describing the mapping between those variables and their corresponding projections. The type of `reify` enforces that the returned `Map` contains, for each of the variables in Γ, a target term of the same type referring solely to a variable `x` of type $\mathtt{T}_\Gamma$. This means the tuple of variables of type $\mathtt{T}_\Gamma$ also returned by `reify` contain enough *information* to replace occurrences of variables in any term in context Γ

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

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

rec reify: Ctx [Γ] → CtxAsTup [Γ] =
fn Γ ⇒ case Γ of
| Ctx [ ] ⇒ CTup Emp (IdMap [x:target unit])
| Ctx [Γ, x:source S] ⇒
 let CTup vt σ = reify (Ctx [Γ]) in
 let ρ' = pushMap σ in
 let rho = DotMap ρ' [x_env:target (cross S _). cfst x_env] in
  CTup (Nex vt) ρ
;
```

Figure 6–16: Implementation of the Function `reify`

perserving types - it contains either the variables themselves or terms of the same type.

6.6 Implementation of the Main Theorem

The function `cc`, given in Fig. 6–17 and 6–18, implements our closure conversion algorithm (see Fig. 6–4) recursively by pattern matching on objects of type `[Γ. source T]` . It follows closely the earlier proof (Thm. 1). We describe here the cases for variables and lambda-abstractions omitting the case for applications. When we encounter a variable, we simply lookup its corresponding binding in ρ.

Given a lambda abstraction in context Γ and ρ which represents the map from Γ to Δ, we begin by strengthening the term to some context Γ'. We then reify the context Γ' to obtain a tuple `E` together with the new map ρ'' of type `Map [x_env:target T_Γ'] [Γ']`. We use a type-annotation on ρ'' in the code to resurrect implicit information present in the types. Next, we extend ρ'' with the identity on the lambda-abstraction's local variable to obtain ρ', and recursively translate `M` using ρ', obtaining a `target` term in context `x_env, x`. Abstracting over `x_env` and `x` gives us the

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

| [Γ. #p... ] ⇒ lookup ρ [Γ. #p... ]

| [Γ. lam λx.M... x] ⇒
 let STm [Γ'. add (λx. ainit (M' ... x))] rel =
                          str [Γ. add λx.ainit (M ... x)] in
 let CTup [Γ'. E... ] (ρ'':Map [x_env:target T_Γ'] [Γ']) =
                          reify (Ctx [Γ']) in
 let ρ' = extendMap ρ'' in
 let [x_env:target T_Γ', x:target T. (P x_env x)] =
                          cc ρ' [Γ', x:source _. M... x] in
 let [Δ. P_env... ] = lookupVars [Γ'. E... ] rel ρ in

   [ Δ. cpack (clam (λc. (clet (cfst c)
                       (λx.(clet (crst c)
                             (λx_env. P x_env x))))))
          (P_env... ) ]
| [Γ. z] ⇒ [Δ. cz]
| [Γ. suc (M ... )] ⇒
 let [Δ. P'... ] = cc ρ [Γ. M... ] in
 [Δ. csuc (P'... )]
| [Γ. app (M... ) (N... )] ⇒
 let [Δ. P... ] = cc ρ [Γ. M... ] in
 let [Δ. Q... ] = cc ρ [Γ. N... ] in
    [Δ. copen (P... )
      λe.λx_f.λx_env. capp x_f (ccons (Q... ) x_env)
...
;
```

Figure 6–17: Implementation of Closure Conversion in Beluga

desired closure-converted lambda-abstraction. To obtain the environment $\texttt{P}_{env}$, we apply ρ on each variables in `E` using `lookupVars`. Finally, we pack the converted lambda-abstraction and the environment $\texttt{P}_{env}$ as a closure, using the constructor `cpack`.

The rest of the cases are straightforward. `z` and `nilv` are replaced by the appropriate constant of the target language, weakened to Δ given by ρ. Unary `fst`,`rst` and `suc`, and binary `app` and `cons` are translated directly by recursively calling the function on their subargument(s) using the input map ρ, before recombining them using target term constructors. For **let**, we

```
rec cc: Map [Δ] [Γ] → [Γ. source T] → [Δ. target T] =
fn ρ ⇒ fn m ⇒ case m of
...
| [Γ. letv (M... ) (λx.(N... x))] ⇒
  let [Δ. P... ] = cc ρ [Γ. M... ] in
  let [Δ, x:target S . Q... x] =
    cc (extendMap ρ) [Γ, x. N... x]
  in
    [Δ. clet (P... ) (λx.Q... x)]
| [Γ. cons (M... ) (N... )] ⇒
  let [Δ. M'... ] = cc ρ [Γ. M... ] in
  let [Δ. N'... ] = cc ρ [Γ. N... ] in
    [Δ. ccons (M'... ) (N'... )]
| [Γ. nilv] ⇒ [Δ. cnil]
| [Γ. fst (M... )] ⇒
  let [Δ. M'... ] = cc ρ [Γ. M... ] in
    [Δ. cfst (M'... )]
| [Γ. rst (M... )] ⇒
  let [Δ. M'... ] = cc ρ [Γ. M... ] in
    [Δ. crst (M'... )];
```

Figure 6–18: Implementation of Closure Conversion in Beluga (Continued)

extend the map with `x` ↦ `x` using `extendMap` before translating the body of the construct with a recursive call to `cc`.

6.7 Discussion and Related Work

Our implementation of closure conversion, including all definitions and auxiliary functions, consists of approximately 200 lines of code.

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

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.

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] extends the closure conversion implementation to System F.

Chlipala [2008] presents a certified compiler for STLC in Coq using parametric higher-order abstract syntax (PHOAS), a variant of weak HOAS. He however annotates his binders with de Bruijn level before the closure conversion pass, thus degenerating to a first-order representation. His closure conversion is hence similar to the one of Guillemette and Monnier [2007].

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.

CHAPTER 7
Hoisting

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

As alluded to in Sec. 6.3, 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, as we observe the contextual objets, 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 hoisting algorithm and its type preservation proof, highlighting the main ideas of the transformation.

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.

Performing hoisting on the closure-converted program presented in Sec. 6.2

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

will result in

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

7.1 The Target Language Revisited

We define hoisting on the target language of closure conversion and keep the same typing rules (see Fig. 6–2) with one exception: the typing rule for t_pack is replaced by the one included in Fig. 7–1.

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

Figure 7–1: Alternative Typing Rule for Hoisting

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_pack, as we request the code portion to be closed; t_pack' specifically allows the code portion to depend on the function environment.

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

7.2 Hoisting Algorithm

We now define the hoisting algorithm in Figure 7–2 as $[\![P]\!]_l = Q \bowtie E$, 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 target term 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 functions environment. As $\circ$ is only defined on n-ary tuples and product types and not on general terms and types, we enforce that the returned environment E and its type L are of the right form. We take $\Delta \vdash_l E : L$ as restricting $\Delta \vdash E : L$ for a n-ary tuple E of product type L.

Lemma 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 $\Delta \vdash_l E_1 : L_1$.

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

Figure 7–2: Hoisting Algorithm

Append Function Environments (Lemma 1) says that the append operator on tuples $\circ$ corresponds, at the level of terms, to the append operator on product types $\circ$. Thus, appending two tuples E_1 and E_2 of type L_1 and L_2 respectively will result in a tuple of type $L_1 \circ L_2$.

Lemma 2. ***Function Environment Weakening (1)***

If $\Delta, l : L_{f_1} \vdash P : T$ *and* $L_{f_1} \circ L_{f_2} = L_f$*, then* $\Delta, l : L_f \vdash P : T$*.*

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

Lemma 3. ***Function Environment Weakening (2)***

If $\Delta, l : L_{f_2} \vdash P : T$ *and* $L_{f_1} \circ L_{f_2} = L_f$*, then* $\Delta, l : L_f \vdash P : T$

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

Function Environment Weakening (1) (Lemma 2) and (2) (Lemma 3) say that we can replace a variable of a product type by a product type which extends the original one. Lemma 2 proves this property when constructing

the new product type by appending the original product type to another, while Lemma 3 proves it when appending a product type onto the original one. While we could prove this property more generally, for example, that we can perform Function Environment Weakening as long as the targeted product type contains all the components of the original product types, with no respect for order, the two lemmas given here are sufficient for the proof of the main theorem that follows, as they corresponds to our usage of the $\circ$ operator.

Theorem 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 $\Delta \vdash P : T$.

Case. $\Delta \vdash_l x : T$

$\Delta \vdash_l x : T$	by assumption
$[\![x]\!]_l = x \bowtie ()$	by definition
$\Delta \vdash_l () : \mathsf{unit}$	by rule t_unit
$\Delta, l_f : \mathsf{unit} \vdash x : T$	by term weakening

Case. $\Delta \vdash \langle P_1, P_2 \rangle : T \to S$

$\Delta \vdash \langle P_1, P_2 \rangle : T \to S$	by assumption
$\cdot \vdash P_1 : \mathsf{code}\ (T \times L_x)\ S$ and $\Delta \vdash P_2 : L_x$	by inversion
$[\![\langle P_1, P_2 \rangle]\!]_l = \langle (\mathsf{fst}\ l)\ (\mathsf{rst}\ l), Q_2 \rangle \bowtie (\lambda l.Q_1, E)$	by definition
where $[\![P_1]\!]_l = Q_1 \bowtie E_1$, $[\![P_2]\!]_l = Q_2 \bowtie E_2$ 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 P_1
$\Delta, l : L_2 \vdash Q_2 : L_x$ and $\cdot \vdash_l E_2 : L_2$	by i.h. on P_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 1)
$l : L_f \vdash Q_1 : \mathsf{code}\ (T \times L_x)\ S$	by F. env. weaken. (1) (Lemma 2)

$\Delta, l : L_f \vdash Q_2 : L_x$ by F. env. weaken. (2) (Lemma 3)

$\cdot \vdash \lambda l.Q_1 : \mathsf{code}\ L_f\ (\mathsf{code}\ (T \times L_x)\ S)$ by rule t_lam

$\cdot \vdash (\lambda l.Q_1, E) : (\mathsf{code}\ L_f\ (\mathsf{code}\ (T \times L_x)\ S))\ \times\ L_f$ by rule t_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 rule t_first

$l : (\mathsf{code}\ L_f\ (\mathsf{code}\ (T \times L_x)\ S))\ \times\ L_f \vdash \mathsf{rst}\ l : L_f$ by rule t_rest

$l : (\mathsf{code}\ L_f\ (\mathsf{code}\ (T \times L_x)\ S))\ \times\ L_f \vdash (\mathsf{fst}\ l)\ (\mathsf{rst}\ l) : \mathsf{code}\ (T \times L_x)\ S$

by rule t_app

$\Delta, l : (\mathsf{code}\ L_f\ (\mathsf{code}\ (T \times E)\ S))\ \times\ L_f \vdash \langle(\mathsf{fst}\ l)\ (\mathsf{rst}\ l), Q_2\rangle : T \to S$

by rule t_pack'

Case. $\Delta \vdash \mathsf{let}\ x = P_1\ \mathsf{in}\ P_2 : T$

$\Delta \vdash \mathsf{let}\ x = P_1\ \mathsf{in}\ P_2 : T$ by assumption

$\Delta \vdash P_1 : S$ and $\Delta, x : S \vdash P_2 : T$ by inversion on t_let

$[\![\mathsf{let}\ x = P_1\ \mathsf{in}\ P_2]\!]_l = \mathsf{let}\ x = Q_1\ \mathsf{in}\ Q_2\ \bowtie\ E_1 \circ E_2$ by definition

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

$\Delta, l : L_1 \vdash Q_1 : S$ and $\cdot \vdash_l E_1 : L_1$ by i.h. on P_1

$\Delta, x : S, l : L_2 \vdash Q_2 : T$, $\cdot \vdash_l E_2 : L_2$ by i.h. on P_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 1)

$\Delta, l : L_f \vdash Q_1 : S$ by F. env. weaken. (1) (Lemma 2)

$\Delta, x : S, l : L_f \vdash Q_2 : T$ by F. env. weaken. (2) (Lemma 3)

$\Delta, l : L_f, x : S \vdash Q_2 : T$ by exchange

$\Delta, l : L_f \vdash \mathsf{let}\ x = Q_1\ \mathsf{in}\ Q_2 : T$ by rule t_let

Case. $\Delta \vdash \mathsf{let}\ \langle x_f, x_{env}\rangle = P_1\ \mathsf{in}\ P_2 : T$

$\Delta \vdash \mathsf{let}\ \langle x_f, x_{env}\rangle = P_1\ \mathsf{in}\ P_2 : T$ by assumption

$\Delta \vdash P_1 : S \to T$ and

$\Delta, x_f : \mathsf{code}\ (S \times L)\ T, x_{env} : L \vdash P_2 : T$ by inversion on t_letpack

$[\![\mathsf{let}\ \langle x_f, x_{env}\rangle = P_1\ \mathsf{in}\ P_2]\!]_l = \mathsf{let}\ \langle x_f, x_{env}\rangle = Q_1\ \mathsf{in}\ Q_2 \bowtie E$ by definition

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 P_1

$\Delta, x_f : \mathsf{code}\ (S \times L)\ T, x_{env} : L, l : L_1 \vdash Q_2 : T$ and $\cdot \vdash_l E_2 : L_2$

by i.h. on P_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 1)

$\Delta, l : L_f \vdash Q_1 : S \to T$ by F. env. weaken. (1) (Lemma 2)

$\Delta, x_f : \mathsf{code}\ (S \times L)\ T, x_{env} : L, l : L_f \vdash Q_2 : T$

by F. env. weaken. (2) (Lemma 3)

$\Delta, l : L_f, x_f : \mathsf{code}\ (S \times L)\ T, x_{env} : L \vdash Q_2 : T$ by exchange

$\Delta, l : L_f \vdash \mathsf{let}\ \langle x_f, x_{env}\rangle = Q_1\ \mathsf{in}\ Q_2 : T$ by rule t_letpack

Case. $\Delta \vdash \lambda x.P : \mathsf{code}\ S\ T$

$\Delta \vdash \lambda x.P : \mathsf{code}\ S\ T$ by assumption

$\Delta, x : S \vdash P : T$ by inversion on t_lam

$[\![\lambda x.P]\!]_l = \lambda x.Q \bowtie E$ by definition

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 P

$\Delta, l : L_f, x : S \vdash Q : T$ by exchange

$\Delta, l : L_f \vdash \lambda x.Q : \mathsf{code}\ S\ T$ by rule t_lam

Case. $\Delta \vdash P_1\ P_2 : T$

$\Delta \vdash P_1\ P_2 : T$ by assumption

$\Delta \vdash P_1 : \mathsf{code}\ S\ T$ and $\Delta \vdash P_2 : S$ by inversion on t_app

$[\![P_1\ P_2]\!]_l = Q_1\ Q_2 \bowtie E$ by definition

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 P_1

$\Delta, l : L_2 \vdash Q_2 : S$ and $\cdot \vdash E_2 : L_2$ by i.h. on P_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 1)

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

$\Delta, l : L_f \vdash Q_2 : S$ by F. env. weaken. (2) (Lemma 3)

$\Delta, l : L_f \vdash Q_1\ Q_2 : T$ by rule t_app

Case. $\Delta \vdash (P_1, P_2) : T$.

$\Delta \vdash (P_1, P_2) : T$ by assumption

$\Delta \vdash P_1 : T_1$ and $\Delta \vdash P_2 : T_2$ by inversion on t_cons

where $T = T_1 \times T_2$

$[\![(P_1, P_2)]\!]_l = (Q_1, Q_2) \bowtie E$ by definition

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

$\Delta, l : L_1 \vdash Q_1 : T_1$ and $\cdot \vdash E_1 : L_1$ by i.h. on P_1

$\Delta, l : L_2 \vdash Q_2 : T_2$ and $\cdot \vdash E_2 : L_2$ by i.h. on P_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 1)

$\Delta, l : L_f \vdash Q_1 : T_1$ by F. env. weaken. (1) (Lemma 2)

$\Delta, l : L_f \vdash Q_2 : T_2$ by F. env. weaken. (2) (Lemma 3)

$\Delta, l : L_f \vdash (Q_1, Q_2) : T$ by rule t_cons

Case. $\Delta \vdash \mathsf{fst}\ P : T$.

$\Delta \vdash \mathsf{fst}\ P : T$ by assumption

$\Delta \vdash P : T \times S$ by inversion on t_first

$[\![\mathsf{fst}\ P]\!]_l = \mathsf{fst}\ Q \bowtie E$ by definition

where $[\![P]\!]_l = Q \bowtie E$

$\Delta, l : L_f \vdash Q : T \times S$ and $\cdot \vdash E : L_f$ by i.h. on P

$\Delta, l : L_f \vdash \mathsf{fst}\ Q : T$ by rule t_first

Case. $\Delta \vdash \mathsf{rst}\ P : T$.

$\Delta \vdash \mathsf{rst}\ P : T$	by assumption
$\Delta \vdash P : S \times T$	by inversion on t_rest
$[\![\mathsf{rst}\ P]\!]_l = \mathsf{rst}\ Q \bowtie E$	by definition
where $[\![P]\!]_l = Q \bowtie E$	
$\Delta, l : L_f \vdash Q : S \times T$ and $\cdot \vdash E : L_f$	by i.h. on P
$\Delta, l : L_f \vdash \mathsf{rst}\ Q : T$	by rule t_rest

Case. $\Delta \vdash () : \mathsf{unit}$.

$\Delta \vdash () : \mathsf{unit}$	by assumption
$\Delta \vdash_l () : \mathsf{unit}$	by rule t_unit
$\Delta, l : L_f \vdash () : \mathsf{unit}$	by rule t_unit

□

7.3 Implementation of Auxiliary Lemmas

Defining functions environments. The functions 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.

Our hoisting algorithm uses aggregate operations such as $\circ$, which are only defined on n-ary tuples and on product types. To guarantee coverage, we define the indexed datatype `Env`, given in Fig. 7–3, encoding the judgement $\Delta \vdash_l E : L_f$ which asserts that environment E and its type L_f are of the right form.

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

Figure 7–3: Definition of `Env` as an Inductive Datatype

`Env` restricts `target` and `tp` to only form tuples of product types. Every object of type `Env` begins with `EnvNil`, corresponding to the empty tuple of type `cnil`. Bigger tuples are formed by adding target terms `P` of type `T` on top of `Env` representing a tuple `E` of product type `L` with the constructor `EnvCons`.

Appending function environments. When hoisting terms with more than one subterm, each recursive call on those subterms results in a different function environment; they need to be merged before combining the subterms again. This is accomplished in our hoisting algorithm by the operation ∘, and represented in Beluga as the function `append`, included in Fig. 7–5. `append` corresponds to the application of Lemma 1 in the proof of Type Preservation (Theorem 1). It takes as input two function environments of type `Env [.L1] [.E1]` and and `Env [.L2] [.E2]`, and constructs a function environment of type `Env [.L1 ∘ L2] [.E1 ∘ E2 ]`. As Beluga does not support computation in types, we return some function environment `E` of type `L`, and a proof that `E` and `L` are the results of concatenating `E1` and `E2`, and respectively `L1` and `L2`.

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

Figure 7–4: Definition of `App` as an Inductive Datatype

App [.L_1] [.L_2] [.L] [.E_1] [.E_2] [.E], whose definition is given in Fig. 7–4 can be read as E_1 and E_2 being tuples of type L_1 and L_2, and concatenating them yields the tuple E of type L. The first constructor, AStart, encode the fact that *unit* (and *cnil* at the level of type) is left identity for ∘. ACons can then be used to relate ∘ when the left constituent is not empty, building it up one element at a time on top of the right constituent.

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

rec append: Env [.L1] [.E1]→ Env [.L2] [.E2]
  → ExApp [.L1] [.L2] [.E1] [.E2]
fn m ⇒ fn n ⇒ case m of
| EnvNil ⇒
 AP (AStart n) n
| EnvCons [.P] e ⇒
 let AP p e' = append e n in
  AP (ACons p) (EnvCons [. _] e')
;
```

Figure 7–5: Implementation of the Function append

Next, we show the implementation of the two lemmas about function environment weakening (Lemma 2 and 3).

```
rec weakenEnv1: (Δ:tctx) App [.L1] [.L2] [.L] [.E1] [.E2] [.E]
  → [Δ, l:target L2. target T] → [Δ, l:target L. target T] =
fn prf ⇒ fn m ⇒ case prf of
| AStart e ⇒ m
| ACons p ⇒
 let [Δ, l: target L3' . M'… l] = weakenEnv1 p m in
   [Δ, l:target _ . M'… (crst l)]
;
```

Figure 7–6: Implementation of the Function weakenEnv1

weakenEnv1 (see Fig. 7–6) is a direct encoding of its specification. In the first case, we learn that L_2 = L, such that we return the original

target term. In the other case, `L = cross T' T` for some `T'`. We recursively weaken L_1 to `T`, before replacing occurences of `l` by `crst l` in the target term `M'`.

```
rec appL1Unit:
App [. L1] [. unit] [. L3] [. E1] [. cnil] [. E3]
  → [. eqlt L1 L3] =
fn a ⇒ case a of
| AStart EnvNil ⇒ [. refllt]
| ACons a' ⇒
 let [. refllt] = appL1Unit a' in
  [. refllt]
;

rec weakenEnv2: (Δ:tctx) App [.L1] [.L2] [.L] [.E1] [.E2] [.E]
 → [Δ, l:target L1. target T] → [Δ, l:target L. target T] =
fn prf ⇒ fn n ⇒ case prf of
| AStart e ⇒
 let AP a e' = append e EnvNil in
 let [. refllt] = appL1Unit a in
  weakenEnv1 a n
| ACons p ⇒
 let [Δ, l:target (cross S L1'). N… l] = n in
 let [Δ, x:target S, l:target L3'. N'… x l] =
    weakenEnv2 p [Δ, x:target S, l:target L1'. N… (ccons x l)]
       in
   [Δ, l:target (cross S L3'). N'… (cfst l) (crst l)];
```

Figure 7–7: Implementation of the Functione `weakenEnv2`

Due to the definition of `Map`, which builds $\mathtt{L}_1$ on top of a fixed $\mathtt{L}_2$, `weakenEnv2`, given in Fig. 7–7, is not as direct to implement as `weakenEnv1`. Intuitively, for given concatenation relation $\mathtt{L}_1 \circ \mathtt{L}_2 = \mathtt{L}$, we peel variables from $\mathtt{L}_1$ until it is empty, in which case we have $\mathtt{L}_2 = \mathtt{L}$. We then use `appL1Unit` which proves that $\mathtt{L}_2 \circ \cdot = \mathtt{L}_2$, which we use, with `weakenEnv1`, to weaken our term by $\mathtt{L}_2$, before adding back the variables from $\mathtt{L}_1$.

7.4 Implementation of the Main Theorem

The top-level function `hcc` performs hoisting at the same time as closure conversion on closed terms. It relies on the function `hcc'`, included in Fig. 7–8, to closure convert and hoist open terms when given a map

between the source and target context. As illustrated in the included code, only small changes are necessary to integrate hoisting within the closure conversion implementation from Section 6.6.

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

`hcc'` transforms a source term in the context Γ given a map between the source context Γ and the target context Δ following the algorithm described in Fig. 7–2. It returns a target term of type `T` which depends on a function environment `l` of some product type $\mathtt{L}_f$ together with a concrete function environment of type $\mathtt{L}_f$. The result of `hcc'` is described by the datatype `HCCRet` which is indexed by the target context Δ and the type `T` of the target term.

`hcc'`, given in Fig. 7–8 and 7–9, follows closely the structure of `cc'`. When encountering a variable, we look it up in ρ and return the corresponding target term with the empty function environment `EnvNil`. When reaching a lambda-abstraction of type `arr S T`, we again strengthen the body `lam` λ`x.M ... x` to some context Γ'. We then reify Γ' to obtain a variable tuple $(x_1, \ldots, x_n)$ and convert the strengthened `M` recursively using the map ρ extended with the identity. As a result, we obtain a closed target term `Q` together with a well-formed function environment `e` containing the functions collected so far. We then build the variable environment $(\rho(x_1), \ldots, \rho(x_n))$, extend the function environment with the converted result of `M` which is known to be closed, and return `capp (cfst l)(crst l)` where `l` abstracts over the current function environment.

While constant and unary constructor cases are unchanged by hoisting, cases with more than one subterm demand more care with respect to the

```
datatype HCCRet:{Δ:tctx} [.tp] → ctype =
| HRet: [Δ, l:target L_f. target T]
        → Env [.L_f] [.E] → HCCRet [Δ] [.T];

rec hcc': Map [Δ] [Γ] → [Γ. source T] → HCCRet [Δ] [.T] =
fn ρ ⇒ fn m ⇒ case m of
| [Γ. #p... ] ⇒
 let [Δ. Q... ] = lookup [Γ] [Γ. #p... ] ρ in
  HRet [Δ, l:target (prod unit). Q... ] EnvNil

| [Γ. lam λx.M... x] ⇒
 let STm [Γ'. add λx.ainit (M'... x)] rel =
                              str [Γ. add λx. ainit (M ... x)] in
 let CTup vt (ρ'':Map [x_env:target T_Γ'] [Γ']) = reify (Ctx [Γ']) in
 let [Δ. P_env... ] = lookupTup vt rel ρ in
 let HRet r e = hcc' (extendMap ρ'') [Γ', x:source _. M'... x] in
 let [x_env:target T_Γ', x:target T, l:target T_f. (Q x_env x l)] = r
    in
 let e' = EnvCons [.clam λl. clam λc.
       clet (cfst c) (λx.clet (crst c) (λx_env. Q x_env x l))]
            e in
 let [.T'] = [.cross (code T_f (code (cross T T_Γ') S)) T_f]
 in HRet
   [Δ, l:target T'. cpack (capp (cfst l) (crst l)) (P_env... )] e'
| [Γ. z] ⇒ HRet [Δ, l:target unit. cz] EnvNil

| [Γ. suc (M ... )] ⇒
 let HRet r e = hcc' ρ [Γ. M... ] in
 let [Δ, l:target L. P'... l] = r in
  HRet [Δ, l:target L. csuc (P'... l)] e

| [Γ. app (M... ) (N... )] ⇒
 let HRet r1 (e1 : Env [. L1] [.E]) = hcc' ρ [Γ. M... ] in
 let {P:[Δ, l:target L1. target (arr T S)]}
    [Δ, l:target L1. P... l] = r1 in
 let HRet r2 e2 = hcc' ρ [Γ. N... ] in
 let AP a12 e12 = append e1 e2 in
 let [Δ, l:target L2 . Q... l] = r2 in
 let [Δ, l:target L. P'... l] =
                      weakenEnv2 a12 [Δ, l:target L1. P... l ] in
 let [Δ, l:target L. Q'... l] =
                      weakenEnv1 a12 [Δ, l:target L2. Q... l ] in
  HRet [Δ, l:target L. copen (P'... l) λe.λx_f.λx_env.
                           capp x_f (ccons (Q'... l) x_env)] e12
...
;
```

Figure 7–8: Implementation of Closure Conversion and Hoisting in Beluga

```
rec hcc': Map [Δ] [Γ] → [Γ. source T] → HCCRet [Δ] [.T] =
fn ρ ⇒ fn m ⇒ case m of
...

| [Γ. letv (M... ) (λx.(N... x))] ⇒
 let HRet r1 e1 = hcc' ρ [Γ. M... ] in
 let [Δ, l:target L1. P... l] = r1 in
 let HRet r2 e2 = hcc' (extendMap ρ) [Γ, x. N... x] in
 let [Δ, x:target T1, l:target L2 . Q... x l] = r2 in
 let AP a12 e12 = append e1 e2 in
 let [Δ, l:target L. P'... l] =
    weakenEnv2 a12 [Δ, l:target L1. P... l] in
 let [Δ, x:target T1, l:target L. Q'... x l] =
    weakenEnv1 a12 [Δ, x:target _, l:target L2. Q... x l] in
  HRet [Δ, l:target L. clet (P'... l) (λx. Q'... x l)] e12

| [Γ. cons (M... ) (N... )] ⇒
 let HRet r1 e1 = hcc' ρ [Γ. M... ] in
 let [Δ, l:target L1. M'... l] = r1 in
 let HRet r2 e2 = hcc' ρ [Γ. N... ] in
 let [Δ, l:target L2. N'... l] = r2 in
 let AP a12 e12 = append e1 e2 in
 let [Δ, l:target L. M''... l] =
    weakenEnv2 a12 [Δ, l:target L1. M'... l] in
 let [Δ, l:target L. N''... l] =
    weakenEnv1 a12 [Δ, l:target L2. N'... l] in
  HRet [Δ, l:target L. ccons (M''... l) (N''... l)] e12

|[Γ. fst (M... )] ⇒
 let HRet r e = hcc' ρ [Γ. M... ] in
 let [Δ, l:target L. P... l] = r in
  HRet [Δ, l:target L. cfst (P... l)] e

|[Γ. rst (M... )] ⇒
 let HRet r e = hcc' ρ [Γ. M... ] in
 let [Δ, l:target L. P... l] = r in
  HRet [Δ, l:target L. crst (P... l)] e
|[Γ. nilv] ⇒ HRet [Δ, l:target unit. cnil] EnvNil
;

rec hcc: [. source T] → [. target T] =
fn m ⇒ let HRet r (e: Env [._] [.E]) = hcc' (IdMap []) m in
     let [l:target S. Q l] = r in
       [. clet E (λl. Q l)];
```

Figure 7–9: Implementation of Closure Conversion and Hoisting in Beluga (Continued)

function environment `l`. This is because recursive calls to each subterm result in a different function environment, which must then be merged, using `append`. Moreover, the references to the function environment in subterms must be adjusted to the merged function environment, using functions `weakenEnv1` and `weakenEnv2`. This is implicit in the algorithmic presentation, but appears in the proof as, respectively, Lemma 2 and 3.

7.5 Discussion and Related Work

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

An alternative to the presented algorithm would be to thread through the function environment as an additional argument to `hcc`. This avoids the need to append function environments and obviates the need for certain auxiliary functions such as `weakenEnv1`. Other properties around `App` 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 is 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 can't be hoisted out.

CHAPTER 8
Future Work

A number of extensions are foreseeable as future work, both on the compiler and on our programming environment, Beluga. We list a few of them in this section, noting their relation to the subject of this thesis and sketching plausible approaches.

8.1 On the Compiler

System F. While the simply typed lambda calculus is a prototypical formal system for typed functional programming languages, contemporary popular languages are based on more expressive typed lambda calculi (see OCa [2013]; Has [2013]). In System F [Reynolds, 1974], the core calculus for many languages, types can be quantified over types, corresponding to polymorphism in programming languages. We have already extended the source language of the hoisting transformation to System F. Extending our source language for the rest of the compiler would follow extensions to comparable algorithms presented in related work [Guillemette and Monnier, 2008; Chlipala, 2008]. While the algorithms seldom change from STLC to System F, open types pose a significant challenge. This is because our implementation of closure conversion and hoisting reuse types in different contexts. This would not be possible to do with types depending on type variables present in the source context without adjusting their type variable references to the target context. We believe that the extension could be performed, albeit rather tediously, in the current version of Beluga, by carrying a relation effectively copying the type from a source context to a target contexts, ignoring the term variables while carrying the same type

variables. Extensions to Beluga such as explicit substitutions, mentioned later in this section, would facilitate developments with multiple contexts such as this one, exposing fine-grained dependencies in contextual objects.

Other transformations and optimizations. A number of transformations and optimizations could supplement our compilation pipeline, making it closer to realistic modern compilers.

Function inlining is an optimization where function calls are selectively expanded with the definition of the function. Functions which are not called in the program can have their definition removed, a process called dead-function elimination, while functions which are called only once can be replaced by their definition and their top-level definition removed, reducing the size of the program and, on a lower level language, avoiding costly procedure calls. Appel and Jim [1997] present a quasi-linear algorithm to perform function inlining, reducing all eta redexes and most shrink redexes in linear time. Dead-function elimination can be performed in Beluga by pattern matching on the substitution affixed to terms within contextual objects, as exemplified in `str` in Chapter 6. This substitution tells us if a given variable does not occur in a term, or if it may occur, but not how many time it occurs. To know that a function is called only once in the body of the program, we would have to traverse the term and keep a count of the occurrences of function variables, as an upper-bound to the function calls performed at run-time. The absence of references and pointers in Beluga prevents us from implementing the data-structure used in the final algorithm presented in the paper effectively, and thus from reaching the desired time-complexity.

Common subexpression elimination is an optimization where identical portions of the programs are replaced by references to a single expression,

reducing the amount of duplicated computation. It is commonly found in compilers for both functional and imperative languages. Non-linear patterns, which are supported in Beluga, could be used to test multiple subexpressions for both structural and variable equality. However, as with function inlining, some algorithms might be out of reach of our programming environment, the lack of references and pointers preventing us from implementing efficient data-structures.

8.2 On Beluga

Interactive Development and Automation. In Beluga, we write proofs directly, by programming, on the computational level, with its functional language. This contrasts with other proof assistant such as Coq and Abella, where proof terms are constructed interactively using a language of tactics. Both approaches have their benefits, a complete discussion of which is outside of the scope of this thesis (see, for example, Gonthier et al. [2011]). However, for programming in Beluga to scale to larger project, support for interactive development and automation in Beluga would be desirable, for example automatically generating the different cases of a case construct, or using proof search to fill trivial portions of programs.

Inspiration could be taken from the agda-mode [Cha, 2013], an emacs mode for the Agda theorem prover where terms are constructed both manually and interactively using predefined commands. Progress has been made in Beluga towards a more responsive programming environment, with the supports of *holes* in Beluga programs, meta-constructs of the computational level representing a meta-variable, which can be substituted by contextual objects of the right contextual type. Programs with holes can be type-checked, but not executed; this assists developers by exposing the type and available information at different points in the program.

Work has also been done to support a certain number of tactics to interactively *fill* the holes, for example by automatically generating covering case constructs. We believe that our rich type system, exposing complex invariants about the contexts of terms, could be exploited for program synthesis. As programs are represented as proofs over LF derivations in our system, program synthesis would corresponds to proof search. While proof search is a well-researched area, it is often used with sole concern to confirm the validity of the theorem, where, as we are interested in *executing* the proof found, we also have programmatic concerns such as algorithmic complexity. For this, careful ordering of the goals and being able to provide hints to direct proof search might be beneficial.

It might then be fruitful to consider supporting an extensible tactics language in the style of Ltac [Delahaye, 2000], where multiple tactics may be combined, discriminating between state of the prover, which would correspond to the contextual type and meta-context of holes.

Theoretical extensions. Theoretical extensions to Beluga could also be envisioned to make it a more suitable environment for the development of certified software.

Support for first-class substitution would make it easier to reuse terms in different contexts, appending to them a substitution from their original context to another. This would help in extending our compiler to System F, where type arguments could be represented in a context containing only type variables and substitution variable would transport them into contexts containing both term and type variables. This extension has already been described in Cave and Pientka [2014] and should be present in future versions of Beluga.

Looking further ahead, Beluga's restricted form of dependent type, which does not allow computations to appear in types, prevents us from proving semantical properties about Beluga functions. Extending Beluga to support full dependent types would open the possibility to prove not only type preservation, but also semantics preservation for transformations implemented, as ours, on the computational level.

CHAPTER 9
Conclusion

In this thesis, we present a series of type-preserving transformations over the simply typed lambda calculus as implemented in Beluga.

Our compiler not only type checks, but also coverage checks. Termination can be verified straightforwardly by the programmer, as every recursive call is made on a structurally smaller argument, such that all our functions are total. The fact that we are not only preserving types but also the scope of terms guarantees that our implementation is *essentially* correct by construction.

While we cannot, as discussed in Chapter 8, guarantee the preservation of semantics for each translation, which would imply the full correctness of our compiler, the aim of our work is different. In our presentation, Beluga functions correspond to lemmas and theorem in type preservation proofs, whose execution wields transformation algorithms. We show that the infrastructure provided by Beluga, as a programming environment, can be used to reduce the overhead of developing reliable software.

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. In addition, by supporting computation-level inductive datatypes, Beluga provides us with an elegant tool to encode properties about contexts and contextual object. 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.

Our work clearly demonstrates the elegance of developing certified software in Beluga. As we refine the understanding of programming with contextual objects as supported by Beluga, we further motivate extensions to the system, towards better support for safe realistic developments, and a democratization of certified programming.

REFERENCES

POPL Tutorial: Mechanizing meta-theory with LF and Twelf, 2009. URL `http://twelf.org/wiki/POPL_Tutorial`.

The Haskell Wiki, 2013. URL `http://www.haskell.org/haskellwiki/Haskell`.

The OCaml Website, 2013. URL `http://ocaml.org/`.

A. W. Appel and T. Jim. Shrinking lambda expressions in linear time. *J. Funct. Program.*, 7(5):515–540, 1997.

A. Banerjee, N. Heintze, and J. Riecke. Design and correctness of program transformations based on control-flow analysis. In *Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science, pages 420–447. Springer, 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.

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 *Logical Frameworks and Meta-Languages: Theory and Practice*, 2014.

The Agda wiki. Chalmers University of Technology, 2013. URL `http://wiki.portal.chalmers.se/agda/`.

C. Chen and H. Xi. Implementing typeful program transformations. In *Workshop on Partial evaluation and semantics-based program manipulation*, pages 20–28, New York, NY, USA, 2003. ACM.

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. A certified type-preserving compiler from lambda calculus to assembly language. In *Symposium on Programming Language Design and Implementation*, pages 54–65. ACM, 2007.

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, 1992.

Z. Dargaye and X. Leroy. Mechanized verification of cps transformations. In *LPAR*, volume 4790 of *Lecture Notes in Computer Science*, pages 211–225. Springer, 2007.

D. Delahaye. A tactic language for the system coq. In *international conference on Logic for Programming and Automated Reasoning*, LPAR, pages 85–95, 2000.

M. Gabbay and A. Pitts. A new approach to abstract syntax involving binders. In *Symposium on Logic in Computer Science*, pages 214–224. IEEE, 1999.

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

G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. In *International Conference on Functional Programming*, pages 163–175. ACM, 2011.

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*, 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 M. Lillibridge. Explicit polymorphism and cps conversion. In *Symposium on Principles of programming Languages*, POPL '93, pages 206–219, New York, NY, USA, 1993. ACM.

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

X. Leroy. Unboxed objects and polymorphic typing. In *Symposium on Principles of Programming Languages*, pages 177–188. ACM, 1992.

X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL*, pages 42–54. ACM Press, 2006.

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

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

ACM, 1996.

G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

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

G. C. Necula. Proof-carrying code. In *Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

L. Nielsen. A denotational investigation of defunctionalization. Technical Report RS-00-47, BRICS, 2000.

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

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

B. Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Principles of Programming Languages*, pages 371–382. ACM, 2008.

B. Pientka. Beluga: Programming with dependent types, contextual data, and contexts. In *Functional and LOgic Programming*, pages 1–12, 2010.

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

A. Poswolsky and C. Schürmann. System description: Delphin – a functional programming language for deductive systems. *Electron. Notes Theor. Comput. Sci.*, 228:113–120, Jan. 2009.

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

F. Pottier and N. Gauthier. Polymorphic typed defunctionalization. In *Symposium on Principles of Programming Languages*, POPL '04, pages 89–98. ACM, 2004.

J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, volume 2, pages 717–740. ACM, Aug. 1972.

J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Colloque Sur La Programmation*, pages 408–423, 1974.

O. Savary B., S. Monnier, and B. Pientka. Programming type-safe transformations using higher-order abstract syntax. In *Certified Programs and Proofs*, volume 8307 of *Lecture Notes in Computer Science*. Springer, 2013.

C. Schürmann. Certifying voting protocols. In *ITP*, page 17, 2013.

Z. Shao. Certified software. *Communications of the ACM*, 53(12):56–66, 2010.

Z. Shao and A. W. Appel. A type-based compiler for standard ml. In *Conference on Programming Language Design and Implementation*, pages 116–129, New York, NY, USA, 1995. ACM.