Université de Montréal

A Type-Preserving Compiler from System F to Typed Assembly Language

par

Louis-Julien Guillemette

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures et postdoctorales

en vue de l'obtention du grade de

Philosophiae Doctor (Ph.D.)

en informatique

Avril 2009

© Louis-Julien Guillemette, 2009

Université de Montréal

Faculté des études supérieures et postdoctorales

Cette thèse intitulée:

A Type-Preserving Compiler from System F

to Typed Assembly Language

présentée par:

Louis-Julien Guillemette

a été évaluée par un jury composé des personnes suivantes:

Guy Lapalme
(président-rapporteur)

Stefan Monnier
(directeur de recherche)

Marc Feeley
(co-directeur)

Julie Vachon
(membre du jury)

Tim Sheard
(examinateur externe)

Paul Arminjon
(représentant du doyen de la F.A.S.)

Thèse acceptée le: ....

Résumé

Mots clés: Compilation certifiée, langage assembleur typé, polymorphisme, vérification formelle, liaisons.

L'utilisation des méthodes formelles est de plus en plus courante dans le développement logiciel, et les systèmes de types sont la méthode formelle qui a le plus de succès. L'avancement des méthodes formelles présente de nouveaux défis, ainsi que de nouvelles opportunités. L'un des défis est d'assurer qu'un compilateur préserve la sémantique des programmes, de sorte que les propriétés que l'on garantit à propos de son code source s'appliquent également au code exécutable.

Cette thèse présente un compilateur qui traduit un langage fonctionnel d'ordre supérieur avec polymorphisme vers un langage assembleur typé, dont la propriété principale est que la préservation des types est vérifiée de manière automatisée, à l'aide d'annotations de types sur le code du compilateur. Notre compilateur implante les transformations de code essentielles pour un langage fonctionnel d'ordre supérieur, nommément une conversion CPS, une conversion des fermetures et une génération de code. Nous présentons les détails des représentation fortement typées des langages intermédiaires, et les contraintes qu'elles imposent sur l'implantation des transformations de code.

Notre objectif est de garantir la préservation des types avec un minimum d'annotations, et sans compromettre les qualités générales de modularité et de lisibilité du code du compilateur. Cet objectif est atteint en grande partie dans le traitement des fonctionnalités de base du langage (les "types simples"), contrairement au traitement du polymorphisme qui demande encore un travail substantiel pour satisfaire la vérification de type.

Abstract

Keywords: Certified compilation, typed assembly language, polymorphism, program verification, bindings.

Formal methods are rapidly improving and gaining ground in software. Type systems are the most successful and popular formal method used to develop software. As the technology of type systems progresses, new needs and new opportunities appear. One of those needs is to ensure the faithfulness of the translation from source code to machine code, so that the properties you prove about the code you write also apply to the code you run.

This thesis presents a compiler from a polymorphic higher-order functional language to typed assembly language, whose main property is that type preservation is verified statically, through type annotations on the compiler's code. Our compiler implements the essential code transformations for a higher-order functional language, namely a CPS conversion and closure conversion as well as a code generation. The thesis presents the details of the strongly typed intermediate representations and the constraints they set on the implementation of code transformations.

Our goal is to guarantee type preservation with a minimum of type annotations, and without compromising readability and modularity of the code. This goal is already a reality for simple types, and we discuss the problems remaining for polymorphism, which still requires substantial extra work to satisfy the type checker.

Contents

1 Introduction **1**

1.1 Typed intermediate languages 4

1.2 Challenges 7

1.3 hTAL 8

1.4 Contributions 12

1.5 Related systems 14

1.6 Structure of the document 15

2 Overview and background **17**

2.1 Generalized algebraic datatypes 17

2.2 Abstract syntax 19

2.2.1 Names 19

2.2.2 Higher-order abstract syntax 20

2.2.3 De Bruijn indices 23

2.3 Type families 25

2.4 Compilation phases 25

2.4.1 Type checking 26

2.4.2 CPS conversion 27

2.4.3 Closure conversion 28

2.4.4 Hoisting 31

2.4.5 Code generation 31

3 Encoding of System F **33**

3.1 Types 35

3.2 Higher-order term encoding 36
3.2.1 Washburn and Weirich's encoding of HOAS 38
3.3 First-order term encoding 41
3.4 Substitution 41

4 CPS conversion 45
4.1 Target language 46
4.2 Translation 50
4.3 Implementation 52
4.3.1 Fegaras and Sheard's iterator 52
4.3.2 Danvy and Filinski's CPS transform 55
4.4 Polymorphism 56
4.4.1 Lemma application 59
4.4.2 Type abstraction 60
4.5 Discussion 61

5 Conversion to de Bruijn indices 63
5.1 Type-level reverse de Bruijn indices 65
5.2 Construction of first-order terms 67
5.3 Reverting to type-level de Bruijn indices 71

6 Closure conversion 75
6.1 Closure conversion and de Bruijn indices 77
6.2 Target language 81
6.3 Translation 86
6.4 Polymorphism 87
6.5 Auxiliary functions 90
6.5.1 Free variables 91
6.5.2 Construction of the variables map 92

7 Hoisting 95
7.1 Target language 96
7.2 Translation 99

7.3 Implementation 100

8 Code generation 103

8.1 Typed assembly language 105

8.2 Translation 106

8.3 Type preservation 109

8.4 Implementation 110

8.4.1 Type translation 112

8.4.2 Term translation 114

9 Benchmarks 117

10 Conclusion 125

10.1 Representation of bindings 126

10.2 Implementation language 128

10.3 Related work 130

10.3.1 Representations of bindings 130

10.3.2 Typed intermediate languages 132

10.3.3 Certified compilation 133

10.4 Future work 135

10.4.1 Bootstrap 136

10.5 Summary 137

A Source code 147

À la mémoire de ma mère.

Acknowledgements

First, I have to thank my advisor, Stefan Monnier, for his support and patience during my research. He was always willing to help me out with every aspect of this work. I also thank my co-advisor, Marc Feeley, for his guidance and encouragement, especially while I was finishing the dissertation.

It was a pleasure to work with Tom Schrijvers when he was visiting our group. I thank him for the insightful discussions we had then.

I am very grateful to Tim Sheard for accepting to take part in the thesis committee.

This research was funded in part by NATEQ, *Le Fonds québécois de la recherche sur la nature et les technologies*, and NSERC, the Natural Sciences and Engineering Research Council of Canada.

Chapter 1

Introduction

Type systems are a successful and popular formal method used for developing software. In this dissertation, we will show that the technology of type systems can be applied to improve the reliability of a compiler implementation, with a low impact on common compiler development practices.

The construction of reliable software systems is commonly achieved by incorporating rigorous testing as part of the development cycle. In the best case, a clear specification describes exactly how the software should behave, and a number of test cases can be derived from the specification. By subjecting a program to a large enough array of tests, one gains some degree of confidence in its correct behavior.

In all but the most trivial cases, testing cannot be applied exhaustively. Testing is useful to discover errors, but cannot prove that no bugs remain. Testing can be sufficient for some programming tasks, but mission-critical systems, where failure can be disastrous, require a higher degree of confidence. Formal methods, unlike testing, aim to prove that a program is correct by construction, rather than sampling the program behavior after-the-fact, and can thus give guarantees on its behavior. Formal methods are still not as commonly used in software systems as in digital systems, but they are heavily researched, and are steadily improving and gaining ground.

Given that compilers play a central role in the software development process, it is essential that the compiler produces an executable program which behaves as expected, so that the properties inferred from the source programs also apply to the code that

is executed. Intuitively, compiler correctness states that a compiler should faithfully implement the language definition.

Realistic compilers are highly complex programs, typically made of hundreds of thousands of lines of code, and writing reliable compilers is indeed a major task. In translating programs from a high-level language (such as Java or Eiffel), into a low-level one (such as the Java Virtual Machine), a compiler typically performs a number of transformations on the input program, to progressively turn it into a lower-level one, which makes use of more rudimentary language constructs. A realistic compiler also makes a number of optimizations. As Aho et al. (2006) put it, "Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free! Thus, the most important objective in writing a compiler is that it is correct."

To ensure some degree of reliability in compilers, the common practice is to run the compiler against a suite of test programs, and check that the compiled programs produce the expected output when executed. This scheme is useful for regression testing, and can also be used to benchmark the performance of the compiled code. In addition, it is useful to instrument the compiler with internal consistency checks, which verify some conditions that internal data structures in the compiler, including intermediate representations of the program being compiled, are expected to satisfy. This helps discover a larger class of bugs, and also gives more precise feedback to the compiler implementor when something goes wrong. To support this kind of sanity check, some compilers maintain some type information about the program as it undergoes the various transformation stages, so that it can be type-checked at various points during compilation. This does not guarantee that the source language semantics is preserved, but it means that the generated program still maintains some form of safety.

To overcome the inherent limitations of informal methods such as these, researchers have in recent years tackled the construction of compilers with completely formalized proofs of correctness, or so-called *certified compilers*. Compiler correctness means that the semantic of the compiled program must be equivalent to that of the source program, which presupposes a formal notion of equivalence between the semantic models of the source and target languages. A well-known example is the certified compiler of Xavier Leroy (Leroy 2006; Blazy et al. 2006), where the operational semantics of the source

and target languages, as well as a proof that the target program simulates the source program's semantics, are formalized in the language of a proof assistant.

In general, constructing a certified compiler is a task that requires significantly more effort than common compiler development. The correctness proof will typically be significantly more involved than the actual compiler's code, and this proof will be harder to read and maintain than the code of a conventional compiler. In the aforementioned example of Leroy's compiler, the proof is reportedly eight times the size of the compiler's actual code.

There is thus a large gap between informal methods, which admit the possibility of errors, and full compiler certification, which requires tremendous effort and departs drastically from common compiler implementation practices. Our work explores an intermediate approach, where formal methods are used to enforce some important properties of compilers, while staying in line with common compiler implementation practices.

We focus our efforts on preservation of *static* semantics. The static semantics (or type system) of a language imposes a syntactic discipline on programs, which prevents important classes of errors (such as applying a function with the wrong number or type of arguments). Strongly typed languages enjoy *type soundness*, which establishes a formal connection between the static and dynamic semantics of a program, and intuitively states that "well-typed programs do not go wrong" at run-time. Preservation of static semantics thus means that the compiled program will enjoy the same safety characteristics as the source program.

A sufficiently strong type discipline on the source language will make erroneous, yet type-correct program manipulations, highly unlikely. Type preservation does not give the same degree of assurance as a completely certified compiler, for which preservation of dynamic semantics has been verified as well. For example, mistakingly generating code which applies an operator whose type is identical to that of the intended one (say, performing a multiplication instead of an addition) will not be reflected in the types, so the error will be undetected. In any case, imposing a type discipline on code manipulations will undoubtedly reduce the possibility of errors by a large factor, and should make any remaining error easier to identify.

A type system classifies program expressions by the kind of values their evaluation

will yield at run time. Early type systems distinguished a small number of types, such as integer and floating-point numbers. Modern type systems can capture a wide variety of properties beyond the simple classification of primitive data. For instance, a type system can be used to track the effect of functions on program state (Gifford and Lucassen 1986; Moggi 1989), to assert the compliance of a program to an information flow security policy (Heintze and Riecke 1998; Ørbæk and Palsberg 1997), or to check that invariants of data structures are never violated, see e.g. (Kahrs 2001). General-purpose programming languages incorporate increasingly powerful features in their type system, which enhance the potential to use type systems as the basis for formal program verification (Sheard 2004; Hinze 2003).

In our work, we will employ the type system of the language in which we implement the compiler to enforce preservation of the static semantics of the program we compile. Instead of writing a formal proof as a separate artifact, the type annotations in the compiler's code will constitute the essential elements of the proof, which will be mechanically verified when the compiler is compiled. The use of types in compilers is not a new idea, and we contrast our approach to the customary use of types in the next section.

1.1 Typed intermediate languages

A compiler for a strongly typed language performs type-checking in an early phase, to ensure that the input program satisfies the type discipline dictated by the source language. After this initial type-checking, the compiler is free to discard all type information, and perform subsequent program manipulations on untyped program representations. In modern compilers, however, it is a common practice to keep some form of type information with the program as it undergoes the various transformation phases. This type information can be used as a kind of "sanity check" to help catch errors in the compiler: by type-checking the code at certain points during compilation, one can identify subtle bugs in the compiler, which would otherwise be much harder to find.

Typed intermediate languages can be used for other purposes than sanity checks; type information can also be used to drive optimizations, as in the work of Leroy (1992), Tarditi et al. (1996), and Shao (1997a). Another application is to construct proofs that

the generated code verifies some safety properties (Morrisett et al. 1999; Hamid et al. 2002).

Compilers employing typed intermediate languages typically represent types in the form of data structures which have to be carefully manipulated to keep them in sync with the code they annotate as this code progresses through the various stages of compilation. Despite its obvious advantages, this approach has several drawbacks:

1. It amounts to *testing* the compiler, thus bugs can lurk, undetected. One would normally have to run the compiler on a suite of sample programs to gain assurance that the compiler behaves as expected.

2. A detected type error, reported as an "internal compiler error", will surely annoy the user, who generally holds no responsibility for what went wrong.

3. It incurs space and time overhead, for manipulating the type information and for type-checking the intermediate languages.

4. The manipulation of type as data can obfuscate the compiler's code to a certain degree and imposes extra maintenance burden.

The approach taken in this thesis is to use the type system of the language in which the compiler is implemented to track the type of the compiled program as it progresses through the various compilation stages. In this way, type preservation can be checked statically, i.e. when type checking the compiler's code, rather than every time an object program is compiled.

This scheme has many significant advantages over conventional typed intermediate languages, and has the potential to address all their drawbacks listed above. The main advantages are the following:

1. This approach is formal, and thus exhaustive, in the sense that it completely eliminates the possibility of internal compiler errors resulting from program manipulations which violate the type discipline of the source and intermediate languages. It thus obviates the need for testing the compiler for type preservation on a suite of test programs.

2. It gives earlier detection of errors introduced by an incorrect program transformation, as they are discovered when the compiler is type-checked, rather than when compiling object programs which exercise the buggy code.

3. It gives better precision in error reports about faulty code transformations, as it would normally identify the line of code where that faulty manipulation(s) occur. In contrast, in a conventional compiler employing typed intermediate languages, the cause of a type checking failure can be fairly subtle, especially in a large compiler.

4. It can eliminate the time and space overhead of manipulating type information as data; the compiler itself can then run at full speed without having to manipulate and check any more types.

5. Last but not least, it can in principle eliminate the need for explicit manipulation of type information as data, and to implement separate type checkers for the intermediate languages.

Total *vs* partial correctness Note that the first point above does not imply *total* correctness, which requires that the compiler always succeeds to produce a well-typed program when given a well-typed program as input. This may fail to be true as the type discipline would not prevent the compiler from entering an infinite loop, for example. What is implied is *partial* correctness: the type discipline imposed on the compiler guarantees that if a program is produced, then this program is well typed.

A long term goal of this research is to formally verify that types are preserved in the compiler, with a minimum of type annotations on the compiler's code. It should not compromise the basic qualities of modularity and readability of the compiler's code. That is, the compiler's code should be as close as possible to that of a conventional compiler which manipulates untyped program representations – in short, what we are aiming for is "type preservation for free".

The benefits of our approach are not fully realized in the current implementation of the compiler. The objective of "type preservation for free" is not fully achieved, as we still need to employ program representations that are not as intuitive as those typically used in conventional compilers, and the treatment of the advanced features of the source

language (in particular, parametric polymorphism) still requires substantial work. We still need to manipulate some type information as data and perform dynamic checks, but this is due in large part to current limitations of the language in which we implement the compiler. We summarize our achievements and the current limitations of our compiler implementation in more detail in the conclusion (Chapter 10).

1.2 Challenges

Given the benefits of our approach, one may wonder why it has not been applied in the past. We mention some reasons which make this approach difficult.

Mechanically verifying type preservation using type annotations on the compiler's code is certainly more challenging than manipulating type information as data, as done in conventional compilers employing typed intermediate languages. Instead of simply type-checking the produced code after-the-fact, we must arrange for every function that manipulates code to have a type signature that captures its effect on source-level types. It thus presupposes a more precise understanding of the way code transformations preserve types.

Type preservation for code transformations has been proven for various code transformations such as CPS conversion (Harper and Lillibridge 1993) and closure conversion (Minamide et al. 1996). Our approach is a form of machine verification, and as such, it imposes difficulties that do not appear in pen-and-paper proofs.

The first difficulty is to enforce the type discipline of the source (and intermediate) languages, by somehow imposing constraints on the data structures used to represent abstract syntax trees in the compiler. This requires judicious use of advanced features of the type system of the language in which the compiler is implemented. Retaining the general qualities of a traditional compiler implementation rather than developing a proof as a separate artifact calls for the use of a general-purpose programming language for implementing the compiler, rather than a proof assistant. Such type-based verification is still an emerging discipline, and is certainly not as well developed as the more traditional forms of machine verification, such as theorem proving, where the emphasis is on proofs rather than types.

As in all forms of mechanized proofs involving programs and languages, a central aspect is the representation of program variables and bindings. Several approaches are possible, and many are actively researched in the field of mechanizing programming language meta-theory, without an emerging consensus as to which one is best suited for applications like ours. A seemingly simple change in the representation of bindings in abstract syntax trees can have a drastic impact on the implementation of code transformations.

In the next section, we further discuss these central difficulties and explain the important choices we have made about them in our compiler.

1.3 hTAL

Our compiler translates a variant of System F to typed assembly language. In this section, we briefly discuss this choice of source and target language, and also discuss our choice of the language Haskell for implementing the compiler.

Terminology To avoid confusion, as we are referring to many languages in this text, we first clarify the terms we use to designate them. A complier is a program that translates programs written in some language, called the *source* language, to programs in some other language, called the *target* language. We refer to the language in which the compiler is written as the *implementation* language, and sometimes the *host* language. When we use a data structure in the host language to represent the abstract syntax trees for the programs in some language, we call the language in question an *object* language; in this sense, the source and target language, as well as any intermediate language used in the compiler, is viewed as an object language.

Source language

The source language of our compiler is a variant of System F, that is, a functional programming language with parametric polymorphism. System F is a powerful system which makes a remarkable economy of features, consisting of a very small number of syntactic constructs, and is thus relatively convenient to manipulate in a compiler. System F can

be seen as the core language at the heart of modern (strongly typed) functional programming languages such as Haskell and ML. We will introduce the syntax and type discipline of System F in Chapter 3.

The reason for choosing System F is that it is a common choice of typed intermediate language in compilers for richly typed functional languages. For example, the Standard ML of New Jersey (SML/NJ) compiler internally makes use of a variant of System F (Shao and Appel 1995). The Glasgow Haskell Compiler also uses a variant of System F (Sulzmann et al. 2007) as its main internal representation. System F is a good representative of typed intermediate languages employed in production quality compilers, as most if not all intermediate languages are some sort of derivatives of System F.

Target language

To show that our techniques are indeed applicable to all phases in a compiler, our compiler preserves types all the way down to assembly language. Our typed assembly language models the assembly language of a reduced instruction set computer (RISC).

The compilation of System F to typed assembly language is studied in detail by Morrisett et al. (1999), who show a series of code transformations that preserve types. Our compiler follows the same general structure as theirs. Their paper discusses the type discipline of a number of intermediate languages, define precisely each transformation step, and state a type-preservation theorem for each transformation. They also define the dynamic semantics of their typed assembly language and prove a type soundness theorem. Our compiler can be seen as a type-checked implementation of their type-preserving translation to typed assembly language, as well as a mechanized proof of the type-preservation theorem stated in their paper.

Implementation language

A secondary objective of our research is to identify the language features of the implementation language that best serve the implementation of a type-preserving compiler. We believe that type preservation is the perfect example of the kind of properties that type systems of the future should allow programmers to conveniently express and verify.

The implementation language must have a type system which is powerful enough to

construct a *strongly typed* representation of syntax trees, that is, a representation that enforces the object language's type system. For instance, we should be able to express in the type signature of a function that it can only return syntax trees representing well-typed expressions.

In general, this sort of constraint can be imposed using *dependent* types. Languages with dependent types allow term-level expressions to appear in type-level expressions. For example, a dependent type $List\, n$ could represent lists of n elements. Dependent types are a very powerful notion, but type-checking for general dependent types is non-decidable since we cannot in general determine if two term-level expressions are equivalent.

There are a number of proof assistants based on dependent types which could be used to construct a type-preserving compiler. Coq (Paulin-Mohring 1993) is a mature proof assistant based on the calculus of inductive constructions (CiC). It has a powerful type system with inductive types, polymorphism and dependent types. The system provides an interactive mode for proof development. While Coq "programs" can be given an immediate operational meaning, executable programs are typically obtained by program extraction, which strips down parts of the code required for the proof and produces a simpler ML or Haskell program. Agda (Norell 2007) would be an alternative, with a newer design, but a less mature implementation and with less proof automation.

There is currently much interest in incorporating features from dependently typed languages and proof assistants in general-purpose functional languages. Many experimental languages incorporate some form of dependent types. Languages such as DML (Xi and Scott 1999), Cayenne (Augustsson 1998), and Omega (Sheard 2004), fall in this category. While some of these languages would have potentially served us well as implementation language, we decided to go with a more mainstream system, and benefit from an industrial-strength implementation and plentiful libraries.

We have chosen Haskell (with GHC's extensions) as our implementation language, as it offered what seemed to be the best combination of a robust implementation and modern type system features, while retaining the qualities of a general-purpose programming language. Haskell is a "mainstream" general-purpose functional programming language supported by a production-quality compiler and a plentiful libraries.

The Glasgow Haskell Compiler (GHC) is a state-of-the-art compiler for Haskell. It

offers a number of extensions to the Haskell 98 standard, and in particular supports *generalized abstract datatypes*, or GADTs (cf. Section 2.1). GADTs allow a limited form of dependently typed programming, where the phase distinction between types and terms is maintained. GHC also supports *type families* (cf. Section 2.3), which allow the user to define functions at the level of types. This feature plays an important role in our work, both for proving type preservation, and for enforcing the type discipline of System F.

An advantage of using Haskell rather than a language with full dependent types is that it narrows down the "semantic gap" between the host and object languages. Our implementation relies essentially on GADTs and type families, and all these features can be encoded in a variant of System F with type equality coercions (Sulzmann et al. 2007). This makes more concrete the possibility of bootstrapping a type-preserving compiler, that is, having the source language be the same as the implementation language, so that we could compile our own compiler.

Representation of binders

An important design decision in systems which manipulate programs, such as compilers, is the way to represent variable bindings, such as local variables and function parameters. In compilers, the most common approach is to represent variables concretely by their name. Alternatively, variables can be identified using some sort of numbering scheme, such as de Bruijn indices, with the advantage that terms which differ only in the name of their bound variables have identical representation. We discuss these approaches further in Section 2.2.

The representation of binders is the subject of much active research. In the field of mechanized meta-theory of programming languages, tools such as proof assistants and theorem provers employ and promote a large variety of approaches. Some systems represent object-level bindings abstractly using bindings in the host language, a technique called *higher-order abstract syntax* (HOAS). Twelf (Pfenning and Schurmann 1999) is an example of a system which promotes such higher-order encodings. Some systems use hybrid representations which combine first-order and higher-order representations. We further discuss research in this area in the related work section (see Section 10.3).

Our compiler manipulates strongly typed program representations, and these repre-

sentations must allow the type system of the host language to track the types of the object-language variables. Higher-order abstract syntax is a natural way to accomplish this, as it allows us to re-use the facility for tracking the types of variables from the host language. The initial phase of our compiler employs such a higher-order representation to good effect. We also use representations based on de Bruijn indices in other phases of the compiler. We show how the basic strongly typed representation (both first-order and higher-order) work in Section 2.2.

As our compiler manipulates polymorphic code, we have to deal with the extra complexity of variables at the level of types. Just like ordinary (i.e. term-level) variables, which abstract values, appear in expressions, type variables abstract types and appear in type-level expressions. The type system of System F comprises a notion of reduction at the type level (which is used to assign a type to a polymorphic term instantiated with a specific type.) This creates delicate interaction between binders at the term and type level. We address these issues in Chapter 3 where we discuss our encoding of System F.

1.4 Contributions

We have implemented a proof-of-concept compiler for all of System F, where type preservation is enforced statically, using type annotations on the compiler's code. Our compiler can be seen as a type-checked implementation of the type-preserving translation to typed assembly language of Morrisett et al. (1999). This constitutes the first mechanized argument of type preservation for these transformations over System F. Our source language supports higher-order functions, parametric polymorphism, term-level recursion (i.e. the ability to define recursive functions), and product types, and is thus sufficiently powerful to encode a large variety of features of modern functional languages.

We show in detail the implementation of the essential code transformations for a call-by-value language to assembly language. We show a CPS conversion, closure conversion, and code generation phase over System F (see Section 2.4 for a general presentation of each transformation.) Our implementation precisely captures the way types are preserved as the code undergoes program transformations.

We show strongly typed program representations of abstract syntax trees for a full

language with term-level and type-level bindings. We show both a first-order and higher-order encoding of System F, as well as a first-order encoding of a number of intermediate languages and a typed assembly language. Our higher-order encoding adapts the parametric representation of HOAS using parametric polymorphism of Washburn and Weirich (2003) to a strongly typed program representation based on GADTs and type families (cf. Section 3.2.1). We show a conversion from this higher-order encoding to the first-order one, which clarifies the relationship between the two. We also address subtle issues about the interaction of bindings at the levels of types and terms.

Our experience provides insight into the technical issues of program representations. We argue that none of the existing representations of bindings is suitable in the sense that either they cannot be used, or they introduce significant extra complexity.

To our knowledge, this is the first example of an extensive application of GHC's type families and type equality coercions. Our work should serve as both a showcase and a stress test for these new features. It also feeds the current debate as to which one of type families, associated types, or multiple-parameter type classes with functional dependencies, should make it to the next Haskell standard (Peyton-Jones et al. 2007). Our work also motivates the development of further Haskell extensions to complement and address the current limitations of type families (cf. Section 4.5), which would extend their power and offer a greater degree of static safety.

Publications Progress in the construction of our compiler has been reported in a number of articles. The first transformation step, CPS conversion, was presented at the first Programming Languages meets Program Verification (*PLPV*) meeting (Guillemette and Monnier 2006). We later presented the closure conversion phase at the Haskell Workshop (Guillemette and Monnier 2007). In both cases, the language treated was a simply typed λ-calculus. We subsequently extended the compiler to support parametric polymorphism, and presented our results at *ICFP*, the International Conference on Functional Programming (Guillemette and Monnier 2008b); this article gives a concise technical overview of this thesis. This document is a synthesis of these three articles, and also presents a final phase of code generation which has not been published elsewhere.

By the time we prepared the article for *ICFP*, we had updated our implementation to

use open type families, which were recently implemented in GHC. As a by-product of this research, in an article presented at the Trends in Functional Programming (*TFP*) symposium (Guillemette and Monnier 2008a), we documented our transition to type families, and took a position in favor of type families. In that article, we also suggested equipping Haskell with a means to specify invariants on type families, which would extend their power for static analysis (cf. Section 4.5). Pursuing this goal, I worked in collaboration with Tom Schrijvers to formalize this idea (Schrijvers et al. 2008). Although it is indeed related to our subject, the material of these two articles is not part of this thesis.

1.5 Related systems

The approach taken in this thesis lies somewhere between conventional compilers employing typed intermediate languages and fully certified compilers. Although type preservation has been formally verified for individual code transformations, our work is the first to apply this idea to the scale of an entire compiler. However, a number of certified compilers have been constructed for various languages, and we describe the most closely related ones below. We review other research in certified compilation in the related work section (Section 10.3).

CompCert As part of the CompCert project Xavier Leroy et al. developed a certified compiler from a C-like language to PowerPC assembly code. The proof of correctness is developed in the Coq proof assistant, and an executable compiler is obtained by means of program extraction.

Their source language is relatively low-level, and has no higher-order features. However their source language is fairly large, and supports all the major features of C. The front-end of the compiler (Blazy et al. 2006) is mainly concerned with resolving operator overloading. The back-end (Leroy 2006) performs register allocation and instruction selection, as well as a couple of simple optimizations. All the intermediate languages are given an operational semantics.

The correctness proofs take the form of a simulation argument, relating the operational semantics of the source and target code. The back-end of the compiler is about 35000 line of Coq code. The semantic definitions and correctness proofs account for most of this, so

that it is notably longer (about 8 times longer) than the actual compiler code.

Lambda Tamer Adam Chlipala (2007) developed a certified compiler from higher-order functional language to typed assembly language. The source language does not have polymorphism or term-level recursion, so it is much simpler than ours. Like Leroy's compiler, it is developed in the Coq proof assistant, using program extraction to obtain an executable compiler. A distinctive feature of this work is the use of denotational semantics to characterize the intermediate languages, and logical relations to establish correctness of the translation steps.

The program representations he uses are very similar to ours, using de Bruijn indices and de Bruijn contexts encoded as lists of types. They are constructed using the inductive types of Coq, which are similar to GADTs (but more general, since they can be indexed by terms instead of just types, and more restrictive, since they disallow negative occurrences.) The compilation phases are roughly those of the original work on compilation to typed assembly language by Morrisett et al. (1999), so they are not very different from ours.

1.6 Structure of the document

We introduce key techniques employed in the implementation of the compiler, and explain the compilation phases of our compiler, in Chapter 2. We also explain different ways of representing syntax trees, using techniques such as higher-order abstract syntax and de Bruijn indices. In particular, we show how to construct a strongly typed encoding of a simply typed language using GADTs.

In chapter 3, we show how to extend the program encodings from Chapter 2 to encode a language with parametric polymorphism. We formally define our source language (a variant of System F) which is the input of the compiler, and show its strongly typed representation.

The subsequent chapters (4 through 8) present the individual transformation phases implemented in the compiler. Chapter 4 presents the implementation of CPS conversion over the higher-order representation of our source language from Chapter 3. Chapter 5 shows a conversion from the higher-order program representation to a first-order one, which is done to facilitate closure conversion. Chapter 6 and 7 present the closure conver-

sion phase and the closely related function hoisting phase. Chapter 8 presents the typed assembly language that is our final target language, and the code generation phase. We report benchmarks of compilation times in Chapter 9. We make general comments on our experience, and mention related and future work in Chapter 10.

Appendix A gives the full code listing of the compiler. In large part, the dissertation and the code can be read in parallel (following the indications at the beginning of each chapter, and consulting the introduction to Appendix A which relates the source files to specific sections of the dissertation.)

Chapter 2

Overview and background

This chapter introduces the types and techniques we use to make the compiler type-preserving, and describes the overall structure of the compiler.

2.1 Generalized algebraic datatypes

Algebraic data types Algebraic data types are a central feature of strongly typed functional languages such as Haskell or ML, and the main mechanism by which the user can define new data types. The definition of an algebraic data type introduces a type constructor, and a set of data constructors, which inject values into the type. Every data constructor accepts a number of arguments of specified types.

For example, we can define a data type for representing lists in this way[1]:

$$
\begin{array}{l}
\textbf{data}\ \mathit{List}\ t\ \textbf{where} \\
\quad \mathit{Cons} :: t \rightarrow \mathit{List}\ t \rightarrow \mathit{List}\ t \\
\quad \mathit{Nil} \quad :: \mathit{List}\ t
\end{array}
$$

This definition introduces a type constructor *List*, which takes a type parameter t, representing the type of the elements in the list. It also introduces the data constructors *Cons* and *Nil*, with explicit type signatures. All the data constructors must have the exact same polymorphic return type, in this case, $\mathit{List}\ t$.

[1] The syntax used here is not the standard Haskell, but the one used in GHC for the definition of GADTs.

Functions which operate over data types are defined by case analysis over data constructors. For instance, a function which calculates the length of a list can be written as follows:

$length :: List\ t \rightarrow Int$

$length\ (Cons\ h\ t) = 1 + length\ t$

$length\ Nil = 0$

GADTs Generalized algebraic datatypes (GADTs) eliminate the restriction that all data constructors must have identical polymorphic return types: the return types can vary in the arguments to the type constructor being defined.

This allows us to encode in *the type* of a value additional information about the value. For instance, with GADTs, we can define a type *ListN* which will give information on the length of the list. We define a type $ListN\ t\ n$ for lists containing n elements of type t. Of course, we will need a representation of natural numbers at the type level. For this, we will use a type to represent the number zero (we call it Z), and a type constructor (we call it $S\ n$) to construct the representation of the successor $(n+1)$ given the type corresponding to n. For instance, the number three is represented by the type $S\ (S\ (S\ Z))$. The type *ListN* is defined as follows:

data Z — natural numbers encoded as types

data $S\ i$

data $ListN\ t\ n$ **where**

$ConsN :: t \rightarrow ListN\ t\ n \rightarrow ListN\ t\ (S\ n)$

$NilN\ \ :: ListN\ t\ Z$

For example, we can represent a list of three elements by the following term:

$ConsN\ 'a'\ (ConsN\ 'b'\ (ConsN\ 'c'\ Nil))$

whose type is $ListN\ Char\ (S\ (S\ (S\ Z)))$.

Now, when we define functions that manipulate such lists, we can specify properties of those functions that involve the length of the lists. We can for instance implement a scalar product of two vectors represented as lists, and express the constraint that the two vectors must be of identical dimension:

$$dotProduct :: ListN\ Int\ n \rightarrow ListN\ Int\ n \rightarrow Int$$
$$dotProduct\ NilN\ NilN = 0$$
$$dotProduct\ (ConsN\ h_1\ t_1)\ (ConsN\ h_2\ t_2) = (h_1 * h_2) + dotProduct\ t_1\ t_2$$

The primary use we make of GADTs in our compiler is to construct strongly typed representations of abstract syntax trees, and we discuss the kind of representations we use in the next section.

2.2 Abstract syntax

Consider a simply typed λ-calculus with primitive operations on integers (which we will call L_S) defined in BNF as follows:

$$\begin{array}{ll} (exps) & e ::= \lambda x\ .\ e \mid e_1\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid x \\ & \quad\ \mid n \mid e_1\ p\ e_2 \mid \mathsf{if0}\ e_1\ e_2\ e_3 \\ (primops) & p ::= + \mid - \mid \times \end{array}$$

There exists different ways of representing such an object language as a data structure in the host language, depending on the way we encode variables. We describe a few ways of doing this, which are used in different parts of our compiler. We also show how to construct strongly typed representations (that is, representations that enforce the type discipline of the object language) using GADTs.

2.2.1 Names

By far the most common way to represent variables in compilers is by their name. For example, the language L_S can be represented using the following algebraic data type:

$$
\begin{array}{l}
\textbf{type}\ \mathit{ID} = \mathit{String} \\
\textbf{data}\ \mathit{Exp}\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{Var} & :: \mathit{ID} & \rightarrow \mathit{Exp} \\
\mathit{Lam} & :: \mathit{ID} \rightarrow \mathit{Exp} & \rightarrow \mathit{Exp} \\
\mathit{App} & :: \mathit{Exp} \rightarrow \mathit{Exp} & \rightarrow \mathit{Exp} \\
\mathit{Let} & :: \mathit{ID} \rightarrow \mathit{Exp} \rightarrow \mathit{Exp} & \rightarrow \mathit{Exp} \\
\mathit{Num} & :: \mathit{Int} & \rightarrow \mathit{Exp} \\
\mathit{Prim} & :: \mathit{Op} \rightarrow \mathit{Exp} \rightarrow \mathit{Exp} & \rightarrow \mathit{Exp} \\
\mathit{If0} & :: \mathit{Exp} \rightarrow \mathit{Exp} \rightarrow \mathit{Exp} & \rightarrow \mathit{Exp}
\end{array} \\
\textbf{data}\ \mathit{Op}\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{Add} & :: \mathit{Op} \\
\mathit{Sub} & :: \mathit{Op} \\
\mathit{Mul} & :: \mathit{Op}
\end{array}
\end{array}
$$

Each data constructor encodes a particular production from L_S's grammar. The constructor for variables *Var* explicitly mentions the variable name, as a character string. Similarly, the constructors that introduce variables (*Lam* and *Let*) mention the name of the newly bound variables.

The representation of variables as character strings is very intuitive, but not very well suited to constructing a strongly typed program representation, as we cannot easily associate a type to a variable represented in this way.

2.2.2 Higher-order abstract syntax

Higher-order abstract syntax (HOAS) is a program representation in which object-level variables are represented using variables in the host language (Haskell in our case.) All structures which imply bindings (not only functions, but all declarations which introduce variable names) will be represented using functions in the host language. The following algebraic data type would be used to represent L_S:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x\ .\ e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_1} \qquad \frac{}{\Gamma \vdash n : \mathsf{int}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1\ p\ e_2 : \mathsf{int}} \qquad \frac{\Gamma \vdash v : \mathsf{int} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{if0}\ v\ e_1\ e_2 : \tau}$$

Figure 2.1: Static semantics of L_S.

```
data Exp where
  Lam  :: (Exp → Exp)         → Exp
  App  :: Exp → Exp           → Exp
  Let  :: Exp → (Exp → Exp)   → Exp
  Num  :: Int                 → Exp
  Prim :: Op → Exp → Exp      → Exp
  If0  :: Exp → Exp → Exp     → Exp
```

Note that, in this representation, there is no need for a data constructor for variables. This representation has the advantage that name-handling is inherited from the host language. For example, we can define a capture-avoiding substitution through a simple function application in the host language. Applications which make extensive use of substitutions can take advantage of this representation. For example, an evaluator which reduces an expression to its normal form:

```
eval :: Exp → Exp
eval (Lam f) = Lam f
eval (App e1 e2) = case eval e1 of
                     Lam f → f (eval e2)
                     _ → error "trying to apply something that is not a function"
...
```

Strongly typed encoding Consider the usual typing rules for L_S shown in Figure 2.1. The judgment $\Gamma \vdash e : \tau$ states that expression e has type τ in static context Γ. The static

context simply lists the types of all the variables in scope, so it has the form:

$$\Gamma ::= x_0 : \tau_0,\ \ldots,\ x_{n-1} : \tau_{n-1}$$

and we use $\bullet$ to denote and empty context (i.e. when $n = 0$).

Using GADTs, we can construct a strongly typed encoding of L_S which enforces these typing rules as follows:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Exp}\ t\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{Lam} & :: (\mathit{Exp}\ t_1 \to \mathit{Exp}\ t_2) & \to \mathit{Exp}\ (t_1 \to t_2) \\
\mathit{App} & :: \mathit{Exp}\ (t_1 \to t_2) \to \mathit{Exp}\ t_1 & \to \mathit{Exp}\ t_2 \\
\mathit{Let} & :: \mathit{Exp}\ t_1 \to (\mathit{Exp}\ t_1 \to \mathit{Exp}\ t_2) & \to \mathit{Exp}\ t_2 \\
\mathit{Num} & :: \mathit{Int} & \to \mathit{Exp}\ \mathit{Int} \\
\mathit{Prim} & :: \mathit{Op} \to \mathit{Exp}\ \mathit{Int} \to \mathit{Exp}\ \mathit{Int} & \to \mathit{Exp}\ \mathit{Int} \\
\mathit{If0} & :: \mathit{Exp}\ \mathit{Int} \to \mathit{Exp}\ t \to \mathit{Exp}\ t & \to \mathit{Exp}\ t
\end{array}
\end{array}
$$

The type parameter t reflects the source type (i.e. object-level type) of the expression. That is, a Haskell term of type *Exp* t represents a well-typed L_S expression e satisfying a judgment $\Gamma \vdash e : \tau$, where t is the Haskell type we have chosen to represent τ. Here, we have chosen to use the Haskell type *Int* to stand for the object type int, and $t_1 \to t_2$ to stand for function types. Indeed, this choice is arbitrary, and we could as well use any other types with the same effect. For instance, we could define a type constructor *Arw*, and use the type *Arw* $t_1\ t_2$ instead of $t_1 \to t_2$ to represent the object-level type $\tau_1 \to \tau_2$.

Note that Γ is implicit in this representation: we are implicitly re-using the type context of the implementation language to track the type of variables in scope. This can work as long as the typing contexts of the implementation language behave the same as those of the object language, which is indeed the case as both use static scoping.

The type *Exp* actually encodes type derivations, not just expression syntax, and indeed type derivations and well-typed expressions are in one-to-one correspondence in L_S. Thus it is impossible to construct the representation of an ill-typed term, as there is no corresponding type derivation.

2.2.3 De Bruijn indices

In contrast to HOAS, a first-order representation introduces variables explicitly. With de Bruijn indices, as with HOAS, variable names are irrelevant, and variables are instead represented as numbers, called "indices".

A de Bruijn index indicates the number of variable introductions that take place between the point where a variable is bound, and the point where that variable occurs. For instance, the term $\lambda x.\lambda y.x\ y$ is written with de Bruijn indices as $\lambda\lambda\ 1\ 0$.

The following algebraic data type would be used to represent the language L_S with de Bruijn indices:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Exp}\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{Var} & :: \mathit{Int} & \to \mathit{Exp} \\
\mathit{Lam} & :: \mathit{Exp} & \to \mathit{Exp} \\
\mathit{App} & :: \mathit{Exp} \to \mathit{Exp} & \to \mathit{Exp} \\
\mathit{Let} & :: \mathit{Exp} \to \mathit{Exp} & \to \mathit{Exp} \\
\ldots & &
\end{array}
\end{array}
$$

Strongly typed representation With de Bruijn indices, a strongly typed representation can be constructed as follows:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Exp}\ \mathit{ts}\ t\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{Var} & :: \mathit{Index}\ \mathit{ts}\ t & \to \mathit{Exp}\ \mathit{ts}\ t \\
\mathit{Lam} & :: \mathit{Exp}\ (t_1, \mathit{ts})\ t_2 & \to \mathit{Exp}\ \mathit{ts}\ (t_1 \to t_2) \\
\mathit{App} & :: \mathit{Exp}\ \mathit{ts}\ (s \to t) \to \mathit{Exp}\ \mathit{ts}\ s & \to \mathit{Exp}\ \mathit{ts}\ t \\
\mathit{Let} & :: \mathit{Exp}\ \mathit{ts}\ t_1 \to \mathit{Exp}\ (t_1, \mathit{ts})\ t_2 & \to \mathit{Exp}\ \mathit{ts}\ t_2 \\
\ldots & &
\end{array}
\end{array}
$$

The type associated with an index is drawn from an explicit type argument (ts), which represents the type context (Γ). As there are no variable names in de Bruijn, the context Γ simply takes the form:

$$\Gamma = \tau_{n-1}, \ldots, \tau_1, \tau_0$$

where t_i gives the type associated with variable i (i.e. whose binder is reached by traversing i binders outward.) The parameter ts encodes Γ as a list (in the form of nested

pairs):

$$ts = (t_0, (t_1, \ldots (t_{n-1}, ())))$$

where t_i is the Haskell type that stands for τ_i. Note that we use Haskell's unit type, (), to indicate the end of the list, and to denote the empty environment, $\bullet$. A variable is represented as an index, whose type reflects the type of the corresponding variable. The indices are constructed as Peano numbers:

```
data Index ts t where
  I0 :: Index (t, ts) t
  Is :: Index ts t → Index (t0, ts) t
```

Note that individual indices are polymorphic in *ts* and *t*, as a given index needs to assume different types in different contexts. Specifically, for an index of the form $Is^i\ I0$ of type *Index ts t*, the only relation between *ts* and *t* is that the ith type appearing in *ts* is *t*, which is why t_0 appears free in the type of *Is*, and *ts* appears free in the type of *I0*.

To illustrate the first-order and higher-order encodings, the following expression:

$$\begin{aligned} &\mathsf{let}\ a = 2 \\ &\quad\ \ b = 3 \\ &\mathsf{in}\ a + b \end{aligned}$$

would be represented in HOAS as:

```
Let (Num 2) (λa →
  Let (Num 3) (λb →
    Add a b))
```

and with de Bruijn indices as:

```
Let (Num 2) (
  Let (Num 3) (
    Add (Var (Is I0)) (Var I0)))
```

Now that we have presented GADTs and the basics of our strongly typed program representations, we will discuss another important feature of GHC, namely type families, which we use to capture the effect of functions on the object-level types.

2.3 Type families

Type families (Schrijvers et al. 2007) are a recent addition to GHC that allows programmers to directly define functions over types by case analysis, in a way that resembles term-level function definitions with pattern matching.

For example, we can define a type function *Add* that computes (statically) the sum of two Peano numbers:

```
type family Add n m
type instance Add Z m = m
type instance Add (S n) m = S (Add n m)
```

We can then use this type family to express the fact that an *append* function over length-annotated lists produces a list of the expected length:

```
data List elem len where
  Cons :: elem → List elem n → List elem (S n)
  Nil :: List elem Z

append :: List elem n → List elem m → List elem (Add n m)
append Nil l = l
append (Cons h t) l = Cons h (append t l)
```

To see how the first clause of *append* type-checks: by the type signature of *append*, the returned value should have type *List elem* (*Add* Z m); l actually has type *List elem* m, which is the same, since *Add* Z m reduces to m by the definition of *Add*. For the second clause, the type of the returned value should be *List elem* (*Add* (S n) m), and *Cons* h (*append* t l) has type *List elem* (S (*Add* n m)), which is the same after the second clause of *Add* is applied, in reverse.

2.4 Compilation phases

In this section we briefly describe the transformation steps that take place in the compiler. These compilation steps are fairly typical of a compiler for a call-by-value functional

language. The overall structure of the compiler is as follows:

$$\lambda \xrightarrow{\text{typecheck}} \lambda_{\rightarrow} \xrightarrow[\textit{convert}]{\text{CPS}} \lambda_{\mathcal{K}} \xrightarrow[\textit{convert}]{\textit{deBruijn}} \lambda_{\mathcal{K}}^{b} \xrightarrow[\textit{convert}]{\textit{closure}} \lambda_{\mathcal{C}} \xrightarrow{\textit{hoist}} \lambda_{\mathcal{H}} \xrightarrow[\textit{code}]{\textit{generate}} \text{TAL}$$

The source language ($\lambda_{\rightarrow}$) and each intermediate language ($\lambda_{\mathcal{K}}$, $\lambda_{\mathcal{K}}^{b}$, etc.) has its own syntax and type system, so each is encoded as a separate GADT. The language $\lambda_{\rightarrow}$ is similar to L_S but has more features, including parametric polymorphism (cf. Chapter 3). The final target language is a typed assembly language (TAL), which has the general characteristics of an assembly language for a RISC computer, but which has a static type system and carries type information. The first phase infers types for all subterms of the source program, and all the subsequent ones are then careful to preserve them. In general, the way a transformation affects the types is captured by a function on types. For example, the effect of the CPS conversion on the types of System F is captured by a function (namely $\mathcal{K}_{\text{type}}[\![-]\!]$, cf. Figure 4.4), which maps function types ($\tau_1 \rightarrow \tau_2$) to continuation types (written $\tau \rightarrow 0$) and leaves the other types unchanged.

We briefly describe each transformation below and illustrate them by showing the compilation of a simple program.

2.4.1 Type checking

The type checking phase takes a simple abstract data type *AST*, then it infers and checks its type t, and returns a GADT of type *Exp* t which does not just represent the syntax but also a proof that the expression is properly typed, in the form of a type derivation. In order for the CPS phase to more closely match the natural presentation, we make it work on a *higher-order abstract syntax* (HOAS) representation of the code, so the type checking phase also converts the first order abstract syntax (where variables are represented by their names) to a HOAS (where variables are represented by meta variables) at the same time.

The conversion to HOAS is implemented using Template Haskell (Sheard and Jones 2002), a compile-time meta-programming facility bundled with GHC – that is, it allows us to construct a piece of Haskell code under program control. This piece of code gets

type-checked by GHC, and since the program representation we construct is strongly typed, we get a source-level type checker for free.

Constructing HOAS terms by meta-programming gives us an efficient representation, in contrast to a direct implementation which would lead to residual redexes, i.e. recursive calls to the conversion (or parsing) function hidden inside closures for functional arguments, like those for λ or let. To illustrate the problem, consider a parser that directly produces a higher-order representation; such a parser would be of essentially this form:

```
parse ... = case ... of
                ... → Lam (λx → parse ... x ...)
                ...
```

The problem is that the body of the function being parsed may indeed refer to the newly bound variable (x), so the variable has to be passed as argument in the recursive call to *parse*. The resulting syntax tree contains a call to *parse* under every *Lam* node, with dramatic consequences on the compiler's performance. Using Template Haskell, we avoid this problem by constructing a "fresh" Haskell expression which contains no reference to the functions that produce the abstract syntax tree.

2.4.2 CPS conversion

The first transformation rewrites the program in *continuation-passing style* (CPS), in which all intermediate computational results are given a name, and the control structure of the program is made explicit. An important difference is that in CPS, a function does not return a value to the caller, but instead communicates its result by calling a *continuation*, which is a function that represents the "rest of the computation", that is, the context of the computation that will consume the value produced. Additionally a special form halt is used to indicate the final "answer" produced by the program.

An example is shown in Figure 2.2. After the conversion, the function *c2f* takes a continuation k as an additional parameter; the body of the function introduces variables v_0 and v_1 to hold intermediate results of the computation, and finally invokes the continuation k. The call to *c2f* passes a continuation, which in this case merely applies the primitive form halt to the value produced by *c2f*.

$$\begin{array}{l}\mathsf{let}\ a = 1.8\\ \quad b = 32\\ \quad c = 24\\ \quad c2f = \lambda x\ .\ a \times x\ +\ b\\ \mathsf{in}\ c2f\ c\end{array} \overset{\text{CPS}}{\Longrightarrow} \begin{array}{l}\mathsf{let}\ a = 1.8\\ \quad b = 32\\ \quad c = 24\\ \quad c2f = \lambda\langle x, k\rangle\ .\\ \qquad\qquad \mathsf{let}\ v_0 = a \times x\\ \qquad\qquad\quad v_1 = v_0 + b\\ \qquad\qquad \mathsf{in}\ k\ v_1\\ \mathsf{in}\ c2f\ \langle c, \lambda v\ .\ \mathsf{halt}\ v\rangle\end{array}$$

Figure 2.2: Example of CPS conversion.

$$\begin{array}{l}\mathsf{let}\ a = 1.8\\ \quad b = 32\\ \quad c = 24\\ \quad c2f = \langle\lambda\langle\langle x, k\rangle, env\rangle\ .\ \mathsf{let}\ a = env.0\\ \qquad\qquad\qquad\qquad\qquad\quad b = env.1\\ \qquad\qquad\qquad\qquad\qquad\quad v_0 = a \times x\\ \qquad\qquad\qquad\qquad\qquad\quad v_1 = v_0 + b\\ \qquad\qquad\qquad\qquad\qquad\quad \langle k_f, k_{env}\rangle = k\\ \qquad\qquad\qquad\qquad\qquad \mathsf{in}\ k_f\ \langle v_1, k_{env}\rangle,\\ \qquad\qquad \langle a, b\rangle\rangle\\ \quad \langle c2f_f, c2f_{env}\rangle = c2f\\ \mathsf{in}\ c2f_f\ \langle\langle c, \langle\lambda\langle v, env\rangle\ .\ \mathsf{halt}\ v, \langle\rangle\rangle\rangle, c2f_{env}\rangle\end{array}$$

Figure 2.3: Closure-converted program.

For an input $\lambda_{\rightarrow}$ expression of type τ, represented internally as a value of type $\mathit{Exp}\ t$, the output of the CPS conversion should be of type $\mathit{ExpK}\ (K\ t)$, where K is a type family that describes the way types are modified by this phase. In particular, input types of the form $\tau_1 \rightarrow \tau_2$ are mapped to $\langle\tau_1', \tau_2' \rightarrow 0\rangle \rightarrow 0$, where $\tau \rightarrow 0$ is the type of a continuation which consumes a value of type τ. The type of the CPS conversion, in simplified form:

$$\mathit{cps} :: \mathit{Exp}\ t \rightarrow \mathit{ExpK}\ (K\ t)$$

expresses and enforces directly that the function preserves types.

2.4.3 Closure conversion

A closure is a data structure that consists of a function, paired with a tuple holding a copy of the function's free variables, called the *environment*. The function is made to receive the environment as an extra parameter. Functions inside closures are *closed*, i.e. they do not have any free variables, as they access their free variables by extracting them

$$
\begin{array}{l}
\mathsf{let}\ a = 1.8 \\
\quad b = 32 \\
\quad c = 24 \\
\quad c2f = \mathsf{pack}\ [\langle \mathsf{int}, \mathsf{int} \rangle, \\
\qquad\qquad \langle \lambda \langle \langle x, k \rangle, env \rangle\ .\ \mathsf{let}\ a = env.0 \\
\qquad\qquad\qquad\qquad\qquad b = env.1 \\
\qquad\qquad\qquad\qquad\qquad v_0 = a \times x \\
\qquad\qquad\qquad\qquad\qquad v_1 = v_0 + b \\
\qquad\qquad\qquad\qquad\qquad (\beta, \langle k_f, k_{env} \rangle) = \mathsf{unpack}\ k \\
\qquad\qquad\qquad\qquad\quad \mathsf{in}\ k_f\ \langle v_1, k_{env} \rangle, \\
\qquad\qquad \langle a, b \rangle \rangle] \\
\qquad\quad \mathsf{as}\ \tau_{c2f} \\
\quad (\beta, \langle c2f_f, c2f_{env} \rangle) = \mathsf{unpack}\ c2f \\
\mathsf{in}\ c2f_f\ \langle \langle c, \mathsf{pack}\ [\langle \rangle, \langle \lambda \langle v, env \rangle\ .\ \mathsf{halt}\ v, \langle \rangle \rangle]\ \mathsf{as}\ \tau_{halt} \rangle, c2f_{env} \rangle \\
\text{where } \tau_{halt} = \exists \alpha . \langle \langle \mathsf{int}, \alpha \rangle \rightarrow 0, \alpha \rangle \\
\qquad\quad \tau_{c2f} = \exists \beta . \langle \langle \langle \mathsf{int}, \tau_{halt} \rangle, \beta \rangle \rightarrow 0, \beta \rangle
\end{array}
$$

Figure 2.4: Closure-converted program, with existential types.

from the environment parameter. At the call site, the closure is taken apart into its function and environment components and the call is made by passing the environment as an additional argument to the function. Closure conversion makes the manipulation of closures explicit.

For example, the above example in CPS will be transformed by the closure conversion as shown in Figure 2.3. The function *c2f* takes as an extra parameter *env*, which contains the value of its free variables, namely a and b. In the body of *c2f*, access to those variables is done through projections of the variable *env*. When *c2f* is called, it is passed the environment (extracted from the closure) as an extra parameter.

From the point of view of type preservation, closure conversion must take into account that two functions of identical types would be translated into closures of different types, if the types of their free variables are different. For example, consider two functions f_1 and f_2, both of type $\tau \rightarrow 0$, whose environments are characterized by the types env_1 and env_2, so that the types of the corresponding closures would be as follows:

$$f_1' : \langle \langle \tau, env_1 \rangle \rightarrow 0, env_1 \rangle$$
$$f_2' : \langle \langle \tau, env_2 \rangle \rightarrow 0, env_2 \rangle$$

The commonly accepted solution is to use existential types to abstract away from the

type of the environment. In this way, the two closures can be of the following type:

$$\exists\beta.\langle\langle\tau,\beta\rangle \rightarrow 0, \beta\rangle$$

Syntactically, the form pack is used to construct a so-called "existential package" of a specified type, and thereby hides a part of the type of the expression. For example, we would obtain closures of suitable type in this way:

$$\mathsf{pack}\ [env_1, f_1'] \ \mathsf{as}\ \exists\beta.\langle\langle\tau,\beta\rangle \rightarrow 0, \beta\rangle$$
$$\mathsf{pack}\ [env_2, f_2'] \ \mathsf{as}\ \exists\beta.\langle\langle\tau,\beta\rangle \rightarrow 0, \beta\rangle$$

The form unpack opens up a package, bringing in scope a type variable that stands for the abstracted type (the variable β in this example.)

Our example program modified to represent closures as existential packages is shown in Figure 2.4. In our example, existential types are not necessary since every call site applies a unique function, but they are needed in general if the source program uses higher-order functions. Higher-order functions are functions which receive functions as parameters or produce functions as return values. Consequently, for a given function application $e_1\ e_2$, the function e_1 does not necessarily correspond to a unique function in the program. For example, the following program fragment defines a higher-order function g, which takes a function of type $\tau \rightarrow 0$ as parameter and applies it to some value v of type τ:

$$\mathsf{let}\ g :: (\tau \rightarrow 0) \rightarrow 0$$
$$g = \lambda f\ .\ f\ v$$
$$\mathsf{in\ if0}\ w\ (g\ f_1)\ (g\ f_2)$$

The function g is then applied to two function f_1 and f_2, assuming their type is as given above. The parameter f will successively assume the values of f_1 and f_2 as a result of the calls to g. To be able to give a type to the closure-converted form of the function g, the closure-converted form of f_1 and f_2 must have identical types, hence the need for existential types.

$$
\begin{array}{l}
\mathsf{let}\ \ell_0 = \lambda\langle\langle x, k\rangle, env\rangle\ .\ \mathsf{let}\ a = env.0 \\
\qquad\qquad\qquad\qquad\qquad\quad b = env.1 \\
\qquad\qquad\qquad\qquad\qquad\quad v_0 = a \times x \\
\qquad\qquad\qquad\qquad\qquad\quad v_1 = v_0 + b \\
\qquad\qquad\qquad\qquad\qquad\quad (\beta, \langle k_f, k_{env}\rangle) = \mathsf{unpack}\ k \\
\qquad\qquad\qquad\qquad\qquad\ \mathsf{in}\ k_f\ \langle v_1, k_{env}\rangle \\
\quad \ell_1 = \lambda\langle v, env\rangle\ .\ \mathsf{halt}\ v \\
\quad a = 1.8 \\
\quad b = 32 \\
\quad c = 24 \\
\quad c2f = \mathsf{pack}\ [\langle \mathsf{int}, \mathsf{int}\rangle, \langle \ell_0, \langle a, b\rangle\rangle]\ \mathsf{as}\ \tau_{c2f} \\
\quad (\beta, \langle c2f_f, c2f_{env}\rangle) = \mathsf{unpack}\ c2f \\
\mathsf{in}\ c2f_f\ \langle\langle c, \mathsf{pack}\ [\langle\rangle, \langle \ell_1, \langle\rangle\rangle]\ \mathsf{as}\ \tau_{halt}\rangle, c2f_{env}\rangle \\
\text{where}\ \ \tau_{halt} = \exists\alpha.\langle\langle \mathsf{int}, \alpha\rangle \to 0, \alpha\rangle \\
\qquad\quad\ \ \tau_{c2f} = \exists\beta.\langle\langle\langle \mathsf{int}, \tau_{halt}\rangle, \beta\rangle \to 0, \beta\rangle
\end{array}
$$

Figure 2.5: Example program after the function hoisting transformation.

2.4.4 Hoisting

After closure conversion, λ-abstractions are closed and can be moved freely in the program. A function hoisting phase moves all λ-abstractions to the top level. The result of the function hoisting transform on the example program is shown in Figure 2.5. This phase rearranges the bindings so that all functions appear at the top level, but does not otherwise affect the types.

2.4.5 Code generation

The final stage of compilation generates code for a hypothetical typed assembly language. Our target language models the machine language of a typical RISC computer. The instruction set has the usual data movement, arithmetic, and control transfer operations, as well as an abstract instruction to allocate a tuple on the heap.

The code generated for our sample program is shown in Figure 2.6. The code still carries type annotations (not shown in the example), so it can be type-checked independently of the source program. The type system constrains the contents of registers at every point in the program. Code blocks are polymorphic, and type applications, as well packing and unpacking of existentials are explicit (not shown in the example).

ℓ_0 :	ld $\mathbf{r}_2$ $\mathbf{r}_0[0]$	$\mathbf{r}_2 : x$
	ld $\mathbf{r}_3$ $\mathbf{r}_0[1]$	$\mathbf{r}_3 : k$
	ld $\mathbf{r}_4$ $\mathbf{r}_0[0]$	$\mathbf{r}_4 : env.0$
	ld $\mathbf{r}_5$ $\mathbf{r}_0[1]$	$\mathbf{r}_5 : env.1$
	mul $\mathbf{r}_6$ $\mathbf{r}_4$ $\mathbf{r}_2$	$\mathbf{r}_6 : v_0$
	add $\mathbf{r}_6$ $\mathbf{r}_6$ $\mathbf{r}_5$	$\mathbf{r}_6 : v_1$
	ld $\mathbf{r}_7$ $\mathbf{r}_3[0]$	$\mathbf{r}_7 : k_f$
	ld $\mathbf{r}_8$ $\mathbf{r}_3[1]$	$\mathbf{r}_8 : k_{env}$
	mktuple $\mathbf{r}_0$ $\langle \mathbf{r}_6, \mathbf{r}_8 \rangle$	
	jmp $\mathbf{r}_7$	
ℓ_1 :	halt	
start :	mov $\mathbf{r}_0$ 1.8	$\mathbf{r}_0 : a$
	mov $\mathbf{r}_1$ 32	$\mathbf{r}_1 : b$
	mov $\mathbf{r}_2$ 24	$\mathbf{r}_2 : c$
	mktuple $\mathbf{r}_3$ $\langle \mathbf{r}_0, \mathbf{r}_1 \rangle$	$\mathbf{r}_3 : \langle a, b \rangle$
	mktuple $\mathbf{r}_4$ $\langle \ell_0, \mathbf{r}_3 \rangle$	$\mathbf{r}_4 : c2f$(closure for ℓ_0)
	mktuple $\mathbf{r}_5$ $\langle \rangle$	$\mathbf{r}_5 : \langle \rangle$
	mktuple $\mathbf{r}_6$ $\langle \ell_1, \mathbf{r}_5 \rangle$	$\mathbf{r}_6$: closure for ℓ_1
	mktuple $\mathbf{r}_7$ $\langle \mathbf{r}_2, \mathbf{r}_6 \rangle$	$\mathbf{r}_7 : \langle c, \text{closure for } \ell_1 \rangle$
	ld $\mathbf{r}_8$ $\mathbf{r}_4[1]$	
	mktuple $\mathbf{r}_0$ $\langle \mathbf{r}_7, \mathbf{r}_8 \rangle$	
	ld $\mathbf{r}_9$ $\mathbf{r}_4[0]$	$\mathbf{r}_9 : c2f_f$
	jmp $\mathbf{r}_9$	

Figure 2.6: Assembly language program. The right column describes the contents of the registers in terms of the variable names used in the example from Figure 2.5.

Chapter 3

Encoding of System F

This chapter describes the strongly typed program representation of the variant of System F that constitutes our source language. Given its relative simplicity, System F is a remarkably expressive language, which makes it a good choice for the internal representation in a compiler. Our source language supports term-level recursion and is representative of the internal languages used by compilers for strongly typed functional languages. For example, previous versions of GHC used a variant of System F as its main intermediate language. We have chosen System F as our source language primarily because of is simplicity, and because many language features can be compiled into System F, so the results we obtain here are also valid for a large class of languages.

System F is a formal system that extends the simply typed λ-calculus (cf. Section 2.2) with parametric polymorphism. Parametric polymorphism is a feature that allows the same program entity to be interpreted under different types. Consider the higher-order function *double*, which recieves a function f as an argument and returns a function which applies f twice to its argument:

$$double = \lambda f\ .\ \lambda x\ .\ f\ (f\ x)$$

Suppose we want to apply the function *double* on functions of different types. In a simply typed (or "monomorphic") language, we would need to define different versions of *double* for each type, for instance:

$$double_{\mathsf{int}} :: (\mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to \mathsf{int}$$
$$double_{\mathsf{int}} = \lambda(f : \mathsf{int} \to \mathsf{int})\ .\ \lambda(x : \mathsf{int})\ .\ f\ (f\ x)$$

$$double_{\mathsf{bool}} :: (\mathsf{bool} \to \mathsf{bool}) \to \mathsf{bool} \to \mathsf{bool}$$
$$double_{\mathsf{bool}} = \lambda(f : \mathsf{bool} \to \mathsf{bool})\ .\ \lambda(x : \mathsf{bool})\ .\ f\ (f\ x)$$

With parametric polymorphism, we can define a generic version of *double* that is not committed to a particular type for the argument to f. This is done using a *type abstraction*, as follows:

$$double = \Lambda\alpha.\ \lambda(f : \alpha \to \alpha)\ .\ \lambda(x : \alpha)\ .\ f\ (f\ x)$$

The construct Λ binds a variable that stands for a type (in the same way that λ binds a variable that stands for a value.) Such a polymorphic term is given a universally quantified type:

$$double :: \forall\alpha.\ (\alpha \to \alpha) \to \alpha \to \alpha$$

Finally, we can instantiate a polymorphic term (e) with a particular type (τ) using a *type application* (written $e[\tau]$). For example, we obtain a version of *double* applicable to functions on integers as follows:

$$double_{\mathsf{int}} :: (\mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to \mathsf{int}$$
$$double_{\mathsf{int}} = double[\mathsf{int}]$$

Formally this is captured by the following typing rules for type abstraction and type application:

$$\frac{\Delta, \alpha; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\alpha.\ e : \forall\alpha.\ \tau} \qquad \frac{\Delta; \Gamma \vdash e : \forall\alpha.\ \tau_1}{\Delta; \Gamma \vdash e[\tau_2] : \tau_1[\tau_2/\alpha]}$$

In the typing judgment $\Delta; \Gamma \vdash e : \tau$, Δ is a component of the static context that lists the type variables in scope (and Γ assigns types to term variables in scope, as in Section 2.2). In particular, this judgment implies that all type variables appearing in τ must be listed in Δ. The type context Δ is of the form $\alpha_0, \ldots, \alpha_{n-1}$, and we will write $\bullet$ to denote the empty context (i.e. when $n = 0$).

The source language of the compiler, $\lambda_\to$, is shown in Figure 3.1. The term language is similar to the one from Section 2.2, except that it replaces the λ-abstraction with a more general construct by which recursive functions can be defined ($\mathsf{letrec}\ f\ x = e_1\ \mathsf{in}\ e_2$,

$$
\begin{array}{lll}
(\textit{type context}) & \Delta ::= \alpha_0, \ldots, \alpha_{n-1} & \\
(\textit{value context}) & \Gamma\ ::= x_0 : \tau_0;\ \ldots;\ x_{n-1} : \tau_{n-1} & \\
(\textit{types}) & \tau\ ::= \tau_1 \to \tau_2 \mid \forall \alpha.\ \tau \mid \alpha \mid \langle \tau_1, \tau_2 \rangle \mid \mathsf{int} & \\
(\textit{exps}) & e\ ::= \mathsf{letrec}\ f\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 & \\
 & \quad \mid x \mid e_1\ e_2 \mid \Lambda\alpha.\ e \mid e[\tau] \mid \langle e_1, e_2 \rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e & \\
 & \quad \mid n \mid e_1\ p\ e_2 \mid \mathsf{if0}\ e_1\ e_2\ e_3 & \\
(\textit{primops}) & p\ ::= + \mid - \mid \times &
\end{array}
$$

Figure 3.1: Syntax of $\lambda_{\to}$.

f being in scope in both e_1 and e_2). As a matter of notation, we will sometimes write $\lambda x.e$ as a shorthand for $\mathsf{letrec}\ f\ x = e\ \mathsf{in}\ f$. The source language also includes product constructions and projections.

The remainder of this chapter describes the encoding of this language. The key issue is the choice of representation to use for each binding. In a language like System F there are three distinct classes of binders to consider:

1. at the term level, those that bind values (such as letrec and let);
2. at the term level, those that introduce type variables (Λ);
3. those that bind types at the type level (the $\forall$ quantifier).

We first discuss the encoding of type-level binders, which is essentially the same throughout the compiler. We then turn to the encoding of type abstraction at the term level, which takes a different form whether value abstraction is higher-order (as in the case of the CPS conversion) or first-order (as in the subsequent phases).

The code listing of the encoding of the source language used in the compiler is shown in Appendix A (in the file Src.hs, page 156; the type families discussed in Section 3.4 are defined in the file Tp.hs, page 148).

3.1 Types

Of course, encoding System F in a GADT implies that introduction and elimination of type variables take place at Haskell's type level. While HOAS would be our preferred choice to represent type-level bindings, GHC does not provide λ-expressions at the level

$\lambda_{\rightarrow}$ type	Haskell type
$\tau_1 \rightarrow \tau_2$	$t_1 \rightarrow t_2$
$\forall\alpha.\ \tau$	*All t*
α	*Var i*
$\langle\tau_1, \tau_2\rangle$	(t_1, t_2)
int	*Int*

Figure 3.2: Encoding of $\lambda_{\rightarrow}$ types.

of types, which constrains our representation of System F types to be first-order: bound type variables are represented with type-level de Bruijn indices.

The encoding of types we use is summarized in Figure 3.2. A universal type $\forall\alpha.\ \tau$ is represented as the Haskell type *All t*, where the type constructor *All* implicitly binds a type variable in t. A type variable is then represented using the type constructor *Var*, with a parameter that encodes its de Bruijn index as a Peano number. The type constructors *All* and *Var* are defined as follows:

data *All t*

data *Var i*

Indeed, as these types are only intended to appear as type parameters to GADTs used for identifying object-level types, their definition do not introduce any data constructor.

To illustrate the encoding of object-level types, the type of the usual *swap* function for pairs:

$$\forall\alpha\ \beta.\ \langle\alpha, \beta\rangle \rightarrow \langle\beta, \alpha\rangle$$

is represented as the Haskell type:

$$All\ (All\ ((Var\ (S\ Z), Var\ Z) \rightarrow (Var\ Z, Var\ (S\ Z))))$$

3.2 Higher-order term encoding

Consider again the typing rules for type abstraction and type application:

$$\frac{\Delta, \alpha; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\alpha.\ e : \forall\alpha.\ \tau} \qquad \frac{\Delta; \Gamma \vdash e : \forall\alpha.\ \tau_1}{\Delta; \Gamma \vdash e[\tau_2] : \tau_1[\tau_2/\alpha]}$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data } \mathit{Exp}\ t \textbf{ where}} \\
\quad \mathit{TpAbs} & :: (\forall t.\ \mathit{Exp}\ (\mathit{Subst}\ s\ t\ Z)) & \rightarrow \mathit{Exp}\ (\mathit{All}\ s) \\
\quad \mathit{TpApp} & :: \mathit{Exp}\ (\mathit{All}\ s) & \rightarrow \mathit{Exp}\ (\mathit{Subst}\ s\ t\ Z) \\
 & & \\
\quad \mathit{Let} & :: \mathit{Exp}\ t_1 \rightarrow (\mathit{Exp}\ t_1 \rightarrow \mathit{Exp}\ t_2) & \rightarrow \mathit{Exp}\ t_2 \\
\quad \mathit{Letrec} & :: (\mathit{Exp}\ (t_1 \rightarrow t_2) \rightarrow \mathit{Exp}\ t_1 \rightarrow \mathit{Exp}\ t_2) \rightarrow & \\
 & \quad (\mathit{Exp}\ (t_1 \rightarrow t_2) \rightarrow \mathit{Exp}\ t) & \rightarrow \mathit{Exp}\ t \\
\quad \mathit{App} & :: \mathit{Exp}\ (t_1 \rightarrow t_2) \rightarrow \mathit{Exp}\ t_1 & \rightarrow \mathit{Exp}\ t_2 \\
 & & \\
\quad \mathit{Pair} & :: \mathit{Exp}\ t_1 \rightarrow \mathit{Exp}\ t_2 & \rightarrow \mathit{Exp}\ (t_1, t_2) \\
\quad \mathit{Fst} & :: \mathit{Exp}\ (t_1, t_2) & \rightarrow \mathit{Exp}\ t_1 \\
\quad \mathit{Snd} & :: \mathit{Exp}\ (t_1, t_2) & \rightarrow \mathit{Exp}\ t_2 \\
 & & \\
\quad \mathit{Num} & :: \mathit{Int} & \rightarrow \mathit{Exp}\ \mathit{Int} \\
\quad \mathit{Prim} & :: \mathit{Op} \rightarrow \mathit{Exp}\ \mathit{Int} \rightarrow \mathit{Exp}\ \mathit{Int} & \rightarrow \mathit{Exp}\ \mathit{Int} \\
\quad \mathit{If0} & :: \mathit{Exp}\ \mathit{Int} \rightarrow \mathit{Exp}\ t \rightarrow \mathit{Exp}\ t & \rightarrow \mathit{Exp}\ t
\end{array}
$$

Figure 3.3: Higher-order encoding of $\lambda_{\rightarrow}$.

If we translate $\lambda_{\rightarrow}$ types in de Bruijn, and thus eliminate all type variable names, these two typing rules become:

$$
\frac{\Delta + 1; \mathsf{shiftEnv}\ \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda e : \forall \tau}
\qquad
\frac{\Delta; \Gamma \vdash e : \forall \tau_1}{\Delta; \Gamma \vdash e[\tau_2] : \tau_1[\tau_2/0]}
$$

where 0 is the smallest de Bruijn index. Here, Δ only tracks the *number* of type variables in scope. In the hypothesis of the judgment for type abstraction, the type context is extended to $\Delta + 1$ to account for the new type variable. The operator $\mathsf{shiftEnv}$ increments all indices corresponding to free variables in the types listed in a type environment; it is applied to Γ in order to avoid capture of the free variables by the new $\forall$ binder. In the judgment for type application, the form $\tau[\tau'/i]$ yields the type τ where the index i has been eliminated, and τ' has been substituted in place of it. Substitution $(-[-/-])$ and the $\mathsf{shiftEnv}$ operator are formally defined in Section 3.4.

We can extend the higher-order encoding from Section 2.2.2 with type abstraction and application (as well as introduction and projections of products) as shown in Figure 3.3. The concrete representation of $\lambda_{\rightarrow}$ that we use is actually different (as will be clarified in Section 3.2.1 below), but this simplified version helps single out the issues of polymorphism.

Term-level type variables As the term encoding is higher-order, the static context $\Delta;\Gamma$ is implicit. A type abstraction $\Lambda\tau$ is represented as a polymorphic term that, when instantiated at a given type τ_2, assumes type $\tau_1[\tau_2/0]$:

$$TpAbs :: (\forall t.\ Exp\ (Subst\ s\ t\ Z)) \rightarrow Exp\ (All\ s)$$

where *Subst* is a type family that implements substitution (defined in Section 3.4 below.) This representation of term-level type variables is higher-order in the sense that an object-level type variable is represented by a Haskell type variable (bound by an implicit type abstraction.)

To illustrate the encoding used, the *swap* function:

$$swap = \Lambda\alpha.\ \Lambda\beta.\ \lambda(x : \langle\alpha, \beta\rangle) : \langle x.1, x.0\rangle.$$

is encoded as:

$$swap :: Exp\ (All\ (All\ ((Var\ (S\ Z), Var\ Z) \rightarrow (Var\ Z, Var\ (S\ Z)))))$$
$$swap = TpAbs\ (TpAbs\ (Letrec\ (\lambda f\ x \rightarrow Pair\ (Snd\ x)\ (Fst\ x))\ (\lambda f \rightarrow f)))$$

3.2.1 Washburn and Weirich's encoding of HOAS

The higher-order representation shown above suffers from the fact that the function space of Haskell is larger than the syntactic classes we want to encode. Consider the following Haskell term:

```
Letrec (λf x → case x of
                  Num n → Num (n + 1)
                  a → a)
       (λf → f)
```

This term performs case analysis in the host language, so clearly it does not represent an expression. Such terms, that do not correspond to expressions of the object language, are called *exotic* terms.

The concrete encoding of $\lambda_{\rightarrow}$ that we use is shown in Figure 3.4. It is essentially a type-indexed version of the parametric representation of HOAS developed by Washburn

$$
\begin{array}{l}
\textbf{type}\ \mathit{Exp}\ \alpha\ t = \mathit{Rec}\ \mathit{ExpF}\ \alpha\ t \\[1ex]
\textbf{data}\ \mathit{ExpF}\ \alpha\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{TpAbs} & :: (\forall t_2.\ \alpha\ (\mathit{Subst}\ t_1\ t_2\ Z)) & \rightarrow \mathit{ExpF}\ (\alpha\ (\mathit{All}\ t_1)) \\
\mathit{TpApp} & :: \alpha\ (\mathit{All}\ t_1) & \rightarrow \mathit{ExpF}\ (\alpha\ (\mathit{Subst}\ t_1\ t_2\ Z)) \\[1ex]
\mathit{Let} & :: \alpha\ t_1 \rightarrow (\alpha\ t_1 \rightarrow \alpha\ t_2) & \rightarrow \mathit{ExpF}\ (\alpha\ t_2) \\
\mathit{Letrec} & :: (\alpha\ (t_1 \rightarrow t_2) \rightarrow \alpha\ t_1 \rightarrow \alpha\ t_2) \rightarrow & \\
 & \quad (\alpha\ (t_1 \rightarrow t_2) \rightarrow \alpha\ t) & \rightarrow \mathit{ExpF}\ (\alpha\ t) \\
\mathit{App} & :: \alpha\ (t_1 \rightarrow t_2) \rightarrow \alpha\ t_1 & \rightarrow \mathit{ExpF}\ (\alpha\ t_2) \\[1ex]
\mathit{Pair} & :: \alpha\ t_1 \rightarrow \alpha\ t_2 & \rightarrow \mathit{ExpF}\ (\alpha\ (t_1, t_2)) \\
\mathit{Fst} & :: \alpha\ (t_1, t_2) & \rightarrow \mathit{ExpF}\ (\alpha\ t_1) \\
\mathit{Snd} & :: \alpha\ (t_1, t_2) & \rightarrow \mathit{ExpF}\ (\alpha\ t_2) \\[1ex]
\mathit{Num} & :: \mathit{Int} & \rightarrow \mathit{ExpF}\ (\alpha\ \mathit{Int}) \\
\mathit{Prim} & :: \mathit{Op} \rightarrow \alpha\ \mathit{Int} \rightarrow \alpha\ \mathit{Int} & \rightarrow \mathit{ExpF}\ (\alpha\ \mathit{Int}) \\
\mathit{If0} & :: \alpha\ \mathit{Int} \rightarrow \alpha\ t \rightarrow \alpha\ t & \rightarrow \mathit{ExpF}\ (\alpha\ t)
\end{array}
\end{array}
$$

Figure 3.4: Parametric higher-order encoding of $\lambda_{\rightarrow}$ (cf. source file Src.hs, page 156).

and Weirich (2003). To illustrate the difference with the simplified encoding, consider the constructor for let:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Exp}\ t\ \textbf{where} \\
\quad \mathit{Let} :: \mathit{Exp}\ t_1 \rightarrow (\mathit{Exp}\ t_1 \rightarrow \mathit{Exp}\ t_2) \rightarrow \mathit{Exp}\ t_2 \\
\quad \ldots
\end{array}
$$

In the concrete representation, it takes the form:

$$
\begin{array}{l}
\textbf{data}\ \mathit{ExpF}\ \alpha\ \textbf{where} \\
\quad \mathit{Let} :: \alpha\ t_1 \rightarrow (\alpha\ t_1 \rightarrow \alpha\ t_2) \rightarrow \mathit{ExpF}\ (\alpha\ t_2) \\
\quad \ldots \\
\textbf{type}\ \mathit{Exp}\ \alpha\ t = \mathit{Rec}\ \mathit{ExpF}\ \alpha\ t
\end{array}
$$

An expression of object type τ is represented as a Haskell term of type:

$$\forall \alpha.\ \mathit{Exp}\ \alpha\ t$$

where the parametricity in α rules out exotic terms. For example, the body of a let expression is represented by a function of type $\alpha\ t_1 \rightarrow \alpha\ t_2$, and since the type for which α stands is not known, the function cannot apply case analysis on its argument.

$$
\begin{array}{llll}
\textbf{data}\ \mathit{Exp}\ \mathit{ctx}\ t\ \textbf{where} & & & \\
\quad \mathit{TpAbs} & :: \mathit{Exp}\ (S\ i, \mathit{ShiftEnv}\ ts)\ s & & \rightarrow \mathit{Exp}\ (i, ts)\ (\mathit{All}\ s) \\
\quad \mathit{TpApp} & :: \mathit{Exp}\ (i, ts)\ (\mathit{All}\ s) & & \rightarrow \mathit{Exp}\ (i, ts)\ (\mathit{Subst}\ s\ t\ Z) \\
\quad \mathit{Var} & :: \mathit{Index}\ ts\ t & & \rightarrow \mathit{Exp}\ (i, ts)\ t \\
\quad \mathit{Let} & :: \mathit{Exp}\ (i, ts)\ t_1 \rightarrow \mathit{Exp}\ (i, (t_1, ts))\ t_2 & & \rightarrow \mathit{Exp}\ (i, ts)\ t_2 \\
\quad \mathit{Letrec} & :: \mathit{Exp}\ (i, (t_1 \rightarrow t_2, (t_1, ts)))\ t_2 \rightarrow & & \\
 & \quad \mathit{Exp}\ (i, (t_1 \rightarrow t_2, ts))\ t & & \rightarrow \mathit{Exp}\ (i, ts)\ t \\
\quad \mathit{App} & :: \mathit{Exp}\ (i, ts)\ (s \rightarrow t) \rightarrow \mathit{Exp}\ (i, ts)\ s & & \rightarrow \mathit{Exp}\ (i, ts)\ t \\
\quad \mathit{Pair} & :: \mathit{Exp}\ (i, ts)\ t_1 \rightarrow \mathit{Exp}\ (i, ts)\ t_2 & & \rightarrow \mathit{Exp}\ (i, ts)\ (t_1, t_2) \\
\quad \mathit{Fst} & :: \mathit{Exp}\ (i, ts)\ (t_1, t_2) & & \rightarrow \mathit{Exp}\ (i, ts)\ t_1 \\
\quad \mathit{Snd} & :: \mathit{Exp}\ (i, ts)\ (t_1, t_2) & & \rightarrow \mathit{Exp}\ (i, ts)\ t_2 \\
\quad \mathit{Num} & :: \mathit{Int} & & \rightarrow \mathit{Exp}\ (i, ts)\ \mathit{Int} \\
\quad \mathit{Prim} & :: \mathit{Op} \rightarrow \mathit{Exp}\ (i, ts)\ \mathit{Int} \rightarrow \mathit{Exp}\ (i, ts)\ \mathit{Int} & & \rightarrow \mathit{Exp}\ (i, ts)\ \mathit{Int} \\
\quad \mathit{If0} & :: \mathit{Exp}\ (i, ts)\ \mathit{Int} \rightarrow \mathit{Exp}\ (i, ts)\ t \rightarrow \mathit{Exp}\ (i, ts)\ t & & \rightarrow \mathit{Exp}\ (i, ts)\ t
\end{array}
$$

Figure 3.5: First-order encoding of $\lambda_{\rightarrow}$.

The type constructor *Rec* plays the role of a fixed-point type operator. We will discuss programming techniques for manipulating this representation of HOAS in Section 4.3, where the definition of an iterator justifies the use of this type-level fixed-point operator. The type *Rec* is defined as follows:

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Rec}\ \alpha\ \beta\ t\ \textbf{where} & & \\
\quad \mathit{Roll} & :: (\alpha\ (\mathit{Rec}\ \alpha\ \beta\ t)) & \rightarrow \mathit{Rec}\ \alpha\ \beta\ t \\
\quad \mathit{Place} & :: (\beta\ t) & \rightarrow \mathit{Rec}\ \alpha\ \beta\ t
\end{array}
$$

The data constructor *Roll* plays the role of the usual *roll* operator of iso-recursive types, and the *Place* constructor is an artifact used internally by the iterator. Intuitively, the type parameter α (of kind $\star \rightarrow \star$, where $\star$ is the kind for Haskell types which classify values) stands for the type of the nodes in the recursive structure (it is instantiated with *ExpF* in the above example), the type parameter β (of kind $\star \rightarrow \star$) makes the representation parametric and abstracts the type of the result of a computation applied to the recursive structure, and t (of kind $\star$) is the type parameter we use for indexing this structure with object-level types.

$$
\begin{aligned}
(\tau_1 \to \tau_2)[\tau/i] &= \tau_1[\tau/i] \to \tau_2[\tau/i] \\
(\forall \tau_0)[\tau/i] &= \forall(\tau_0[\tau/i+1]) \\
j[\tau/i] &= \begin{cases} j-1 & \text{if } j > i; \\ U_0^i(\tau) & \text{if } j = i; \\ j & \text{if } j < i. \end{cases} \\
\langle \tau_1, \tau_2 \rangle[\tau/i] &= \langle \tau_1[\tau/i], \tau_2[\tau/i] \rangle \\
\mathsf{int}[\tau/i] &= \mathsf{int}
\end{aligned}
\qquad
\begin{aligned}
U_k^i(\tau_1 \to \tau_2) &= U_k^i(\tau_1) \to U_k^i(\tau_2) \\
U_k^i(\forall \tau) &= \forall(U_{k+1}^i(\tau)) \\
U_k^i(j) &= \begin{cases} j+i & \text{if } j > k; \\ j & \text{if } j \leq k. \end{cases} \\
U_k^i(\langle \tau_1, \tau_2 \rangle) &= \langle U_k^i(\tau_1), U_k^i(\tau_2) \rangle \\
U_k^i(\mathsf{int}) &= \mathsf{int}
\end{aligned}
$$

Figure 3.6: Substitution over $\lambda_{\to}$ types in de Bruijn.

3.3 First-order term encoding

We extend the first-order encoding from Section 2.2.3 with polymorphism as shown in Figure 3.5. The static context now takes the form (i, ts), where i encodes the *length* of Δ (as a Peano number), and ts encodes Γ as in the simply typed case.[1]

The introduction of a type variable (by the constructor *TpAbs*) is reflected in the type context $(S\ i)$. The type variable implicitly bound by the *All* constructor can be referred to as *Var Z* in the type s; in order to avoid capture of this new type variable in the types forming to the value context ts, the type family *ShiftEnv* must be applied. The effect of this type family is to increase all indices by one.

3.4 Substitution

The capture-avoiding substitution is formally defined in Figure 3.6. It is a conventional substitution over de Bruijn terms (as in, e.g. Kamareddine (2001)). When substituting τ in place of the index i, the free variables of τ must be incremented in order to avoid capture; this is accomplished by the "update" function $U_k^i(\tau)$ (sometimes also called "shift") whose effect is to adjust all indices no smaller than k (those are the free variables) by incrementing them by i.

The substitution and update functions can be encoded directly as Haskell type fam-

[1] We have chosen to aggregate the value and type contexts (ts and i respectively) into a single type parameter (i, ts) to *Exp*. Alternatively, we could define *Exp* with an extra type parameter, and write *Exp i ts t* instead of *Exp* (i, ts) t. We will follow the same convention in subsequent chapters, where the static context will sometimes have more than two components, as in Chapter 7. One advantage of this formulation is that it obviates the need to spell out the components of the static context in the type signature of functions that preserve the static context, e.g. *Exp ctx* $t_1 \to$ *Exp ctx* t_2.

$$
\begin{array}{ll}
\textbf{type instance } \mathit{Subst}\ (t_1 \to t_2)\ t\ i & = (\mathit{Subst}\ t_1\ t\ i) \to (\mathit{Subst}\ t_2\ t\ i) \\
\textbf{type instance } \mathit{Subst}\ (\mathit{All}\ s)\ t\ i & = \mathit{All}\ (\mathit{Subst}\ s\ t\ (S\ i)) \\
\textbf{type instance } \mathit{Subst}\ (\mathit{Var}\ j)\ t\ i & = \mathit{CMP}\ i\ j\ (\mathit{Var}\ (\mathit{Pred}\ j)) \\
& \qquad\qquad\quad (U\ Z\ i\ t) \\
& \qquad\qquad\quad (\mathit{Var}\ j) \\
\textbf{type instance } \mathit{Subst}\ (t_1, t_2)\ t\ i & = (\mathit{Subst}\ t_1\ t\ i, \mathit{Subst}\ t_2\ t\ i) \\
\textbf{type instance } \mathit{Subst}\ \mathit{Int}\ t\ i & = \mathit{Int} \\
\\
\textbf{type instance } U\ k\ i\ (t_1 \to t_2) & = (U\ k\ i\ t_1) \to (U\ k\ i\ t_2) \\
\textbf{type instance } U\ k\ i\ (\mathit{All}\ t) & = \mathit{All}\ (U\ (S\ k)\ i\ t) \\
\textbf{type instance } U\ k\ i\ (\mathit{Var}\ j) & = \mathit{Var}\ (\mathit{CMP}\ j\ k\ j\ j\ (\mathit{Add}\ j\ i)) \\
\textbf{type instance } U\ k\ i\ (t_1, t_2) & = (U\ k\ i\ t_1, U\ k\ i\ t_2) \\
\textbf{type instance } U\ k\ i\ \mathit{Int} & = \mathit{Int} \\
\\
\textbf{type instance } \mathit{Pred}\ (S\ i) & = i \\
\\
\textbf{type instance } \mathit{CMP}\ Z\ Z\ \mathit{lt}\ \mathit{eq}\ \mathit{gt} & = \mathit{eq} \\
\textbf{type instance } \mathit{CMP}\ Z\ (S\ t)\ \mathit{lt}\ \mathit{eq}\ \mathit{gt} & = \mathit{lt} \\
\textbf{type instance } \mathit{CMP}\ (S\ t)\ Z\ \mathit{lt}\ \mathit{eq}\ \mathit{gt} & = \mathit{gt} \\
\textbf{type instance } \mathit{CMP}\ (S\ s)\ (S\ t)\ \mathit{lt}\ \mathit{eq}\ \mathit{gt} & = \mathit{CMP}\ s\ t\ \mathit{lt}\ \mathit{eq}\ \mathit{gt}
\end{array}
$$

Figure 3.7: Type instance declarations for substitution.

ilies. As their definition involve arithmetic over indices, we also need to define type functions accordingly. The complete list of type functions, with their meaning, is as follows:

$$
\begin{array}{lll}
\textbf{type family } \mathit{Subst}\ t_1\ t_2\ i & — & \tau_1[\tau_2/i] \\
\textbf{type family } U\ k\ i\ t & — & U_k^i(\tau) \\
\textbf{type family } \mathit{Pred}\ i & — & i-1 \\
\textbf{type family } \mathit{Add}\ i\ j & — & i+j \\
\textbf{type family } \mathit{CMP}\ i\ j\ t_1\ t_2\ t_3 & — & \begin{cases} \tau_1 & \text{if } i < j; \\ \tau_2 & \text{if } i = j; \\ \tau_3 & \text{if } i > j. \end{cases}
\end{array}
$$

The instance declarations for these type families are shown in Figure 3.7. We also define an auxiliary type family to update all the types in a given context:

$$
\begin{array}{ll}
\textbf{type family } \mathit{Uenv}\ k\ i\ \mathit{ts} & \\
\textbf{type instance } \mathit{Uenv}\ k\ i\ () & = () \\
\textbf{type instance } \mathit{Uenv}\ k\ i\ (t, \mathit{ts}) & = (U\ k\ i\ t, \mathit{Uenv}\ k\ i\ \mathit{ts})
\end{array}
$$

Finally, "shift" functions (for individual types and entire contexts) perform a unit increment on indices corresponding to free variables:

type *Shift* $t = U\ Z\ (S\ Z)\ t$

type *ShiftEnv* $ts = Uenv\ Z\ (S\ Z)\ ts$

Summary

In this chapter we have presented a strongly typed encoding of System F using GADTs and type families. The techniques presented here are used in the encoding in all of our intermediate languages, as each of them incorporates a notion of polymorphism. We also apply similar techniques in the encoding of existential types introduced by closure conversion (Section 6.2).

Our encoding is based on a first-order representation of type-level expressions, where type variables are identified using de Bruijn indices. A type family (*Subst*) implements a notion of substitution, used to eliminate a type variable (as in the typing rule for type application).

We have presented two variants of the encoding, one in which the representation of terms is higher-order, and one in which it is first-order. In the next chapter, we will see a CPS transformation performed over the higher-order encoding from Section 3.2.1. Chapter 5 will show a translation from the higher-order encoding to a first-order one, and all the remaining code transformations will be performed over first-order representations.

Chapter 4

CPS conversion

This chapter presents the implementation of a CPS conversion over the variant of System F described in the previous chapter.

The conversion to continuation-passing style (CPS) makes the control structure of the program explicit, and names all the intermediate results of the computation. For example, the following simple program:

$$\begin{array}{l} \mathsf{let}\ f = \lambda x\ .\ 4 \times x + 3 \\ \mathsf{in}\ f\ 7 \end{array}$$

would be translated to the following program in CPS:

$$\begin{array}{l} \mathsf{let}\ f = \lambda\langle x, k\rangle\ .\ \mathsf{let}\ v_1 = 4 \times x \\ \qquad\qquad\qquad\qquad\quad v_2 = v_1 + 3 \\ \qquad\qquad\qquad\quad\ \mathsf{in}\ k\ v_2 \\ \mathsf{in}\ f\ \langle 7, \lambda x\ .\ \mathsf{halt}\ x\rangle \end{array}$$

The function f is made to take an extra parameter (k), called its *continuation*. Functions in CPS do not return a value to the caller, but instead apply their continuation, which is a function that abstracts the context that will consume the value produced by the function. The final result of the program is returned using the special construct halt.

Syntactically, the formal definition of the CPS language makes a distinction between expressions and values. (We see how this can be made to work with our higher-program representation in Section 4.1.) This distinction is maintained in the subsequent intermediate languages used in the compiler.

$$
\begin{array}{lll}
(\textit{type context}) & \Delta & ::= \alpha_0, \ldots, \alpha_{n-1} \\
(\textit{value context}) & \Gamma & ::= x_0 \!:\! \tau_0;\ \ldots;\ x_{n-1} \!:\! \tau_{n-1} \\
(\textit{types}) & \tau & ::= \forall \vec{\alpha}.\ \tau \to 0 \mid \alpha \mid \langle \tau_1, \tau_2 \rangle \mid \mathsf{int} \\
(\textit{values}) & v & ::= x \mid n \\
(\textit{exps}) & e & ::= \mathsf{letrec}\ f[\vec{\alpha}]\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{let}\ x = v\ \mathsf{in}\ e \\
& & \quad \mid \mathsf{let}\ x = \langle v_1, v_2 \rangle\ \mathsf{in}\ e \mid \mathsf{let}\ x = \mathsf{fst}\ v\ \mathsf{in}\ e \\
& & \quad \mid \mathsf{let}\ x = \mathsf{snd}\ v\ \mathsf{in}\ e \mid \mathsf{let}\ x = v_1\ p\ v_2\ \mathsf{in}\ e \mid v_1[\vec{\tau}]\ v_2 \\
& & \quad \mid \mathsf{if0}\ v\ e_1\ e_2 \mid \mathsf{halt}\ v
\end{array}
$$

Figure 4.1: Syntax of $\lambda_{\mathcal{K}}$.

The CPS conversion presented here is performed over the higher-order representation of System F developed in Section 3.2. A higher-order representation is not a common choice in compilers, but in the case of CPS conversion, it leads to a particularly concise and elegant formulation, and lends itself well to verification of type preservation.

The amount of type annotations in this implementation is notably low, especially in the simply-typed fragment. It is essentially limited to annotating the type of the CPS conversion function, as the code itself requires no further annotations. Unfortunately, in its current state the treatment of polymorphism requires that we annotate the constructors of type abstraction and type application with type representatives, in order to apply a lemma that captures the effect of the translation on type substitutions (cf. Section 4.4).

The code listing of the encoding of the CPS language and the CPS conversion is reported in Appendix A (in the files LK.hs, page 158, and CPS.hs, page 161).

4.1 Target language

The source language of our CPS translation is $\lambda_{\rightarrow}$, defined in the previous chapter. The target language, $\lambda_{\mathcal{K}}$, is shown in Figure 4.1. Function types ($t_1 \to t_2$) and universal types ($\forall \alpha.\ \tau$) are replaced by a single polymorphic continuation type ($\forall \alpha_0, \ldots, \alpha_{n-1}.\ \tau \to 0$), which binds a number of type variables at once. We write $\forall \vec{\alpha}.\ \tau \to 0$ as a shorthand for $\forall \alpha_0, \ldots, \alpha_{n-1}.\ \tau \to 0$. We omit the $\forall$ quantifier for monomorphic continuation types (i.e. when $n = 0$), and simply write $\tau \to 0$. Note that 0 is *not* a type, only $\tau \to 0$ is a type. We use the symbol 0 to suggest the "void" type, that is, a type which is not inhabited by any value. This notation emphasizes the fact that a continuation does not return a value

Object type	Haskell type
$\forall\vec{\alpha}.\ \tau \to 0$	$\mathit{Cont}\ k\ t$
α	$\mathit{Var}\ i$
$\langle\tau_1, \tau_2\rangle$	(t_1, t_2)
int	Int

Figure 4.2: Encoding of the types of $\lambda_{\mathcal{K}}$.

(unlike a function in direct style).

The CPS language makes a syntactic distinction between values, which represent the results of computations, and expressions, which represent the computations proper. Values are just variables and integer literals. Expressions are formed by a series of let (or letrec) bindings ending with either an unconditional control transfer (i.e. a function application) or the special form halt; they can also contain a conditional branching to either of two sub-expressions (if0). The letrec construct introduces a function which can be polymorphic in a number of type variables, and can also be recursive. The let forms bind a variable to either a value (v), a pair constructed from two values, the projection of the first or second component of a pair, or the result of an elementary operation on integers.

The static semantics of $\lambda_{\mathcal{K}}$ is defined by two typing judgments, for values and expressions:

$\Delta; \Gamma \vdash_{\mathcal{K}} v : \tau$	value v has type τ in context $\Delta; \Gamma$
$\Delta; \Gamma \vdash_{\mathcal{K}} e$	expression e is well-typed in context $\Delta; \Gamma$

Note that the typing judgment for expressions does not mention a type, as an expression in CPS does not return a value.

Strongly typed representation

The encoding of $\lambda_{\mathcal{K}}$ types is shown in Figure 4.2. In particular, a polymorphic continuation type $\forall\vec{\alpha}.\ \tau \to 0$ is represented as the type $\mathit{Cont}\ k\ t$, where k encodes the *number* of abstracted type variables as a Peano number (i.e. it encodes $|\vec{\alpha}|$.) For example, the type $\forall\alpha.\ \langle\alpha, \mathit{Int}\rangle \to 0$ is encoded as $\mathit{Cont}\ (S\ Z)\ (\mathit{Var}\ Z, \mathit{Int})$.

The representation of $\lambda_{\mathcal{K}}$ uses parametric higher-order abstract syntax, as in Sec-

type $\mathit{ExpK}\ \alpha = \mathit{Rec}\ \mathit{ExpKF}\ \alpha\ \mathit{Void}$
type $\mathit{ValK}\ \alpha\ t = \mathit{Rec}\ \mathit{ExpKF}\ \alpha\ (V\ t)$

data $V\ t$
data Void

data $\mathit{ExpKF}\ \alpha$ **where**

```
  -- values
  Knum        :: Int                                          → ExpKF (α (V Int))

  -- expressions
  Kletrec     :: (α (V (Cont Z t)) → α (V t) → α Void) →
                 (α (V (Cont Z t)) → α Void)                 → ExpKF (α Void)
  KletPolyFun :: (∀t₂. α (V (Subst t₁ t₂ Z)) → α Void) →
                 (α (V (Cont (S Z) t₁)) → α Void)            → ExpKF (α Void)

  Klet        :: α (V t) → (α (V t) → α Void)                → ExpKF (α Void)
  KletPair    :: α (V t₁) → α (V t₂) →
                 (α (V (t₁, t₂)) → α Void)                   → ExpKF (α Void)
  KletFst     :: α (V (t₁, t₂)) → (α (V t₁) → α Void)        → ExpKF (α Void)
  KletSnd     :: α (V (t₁, t₂)) → (α (V t₂) → α Void)        → ExpKF (α Void)
  KletPrim    :: PrimOp → α (V Int) → α (V Int) →
                 (α (V Int) → α Void)                        → ExpKF (α Void)

  Kapp        :: α (V (Cont Z t)) → α (V t)                  → ExpKF (α Void)
  KpolyApp    :: α (V (Cont (S Z) t₁)) →
                 α (V (Subst t₁ t₂ Z))                       → ExpKF (α Void)

  Kif0        :: α (V Int) → α Void → α Void                 → ExpKF (α Void)

  Khalt       :: α (V Int)                                   → ExpKF (α Void)
```

Figure 4.3: Concrete representation of $\lambda_{\mathcal{K}}$ values and expressions (cf. source file LK.hs, page 158).

tion 3.2. A value v satisfying $\Delta; \Gamma \vdash_{\mathcal{K}} v : \tau$ is represented by a term of type:

$$\forall \alpha.\ \mathit{ValK}\ \alpha\ t \triangleq \forall \alpha.\ \mathit{Rec}\ \mathit{ExpKF}\ \alpha\ (V\ t)$$

and an expression e satisfying $\Delta; \Gamma \vdash_{\mathcal{K}} e$ is represented by a term of type

$$\forall \alpha.\ \mathit{ExpK}\ \alpha \triangleq \forall \alpha.\ \mathit{Rec}\ \mathit{ExpKF}\ \alpha\ \mathit{Void}$$

Note that the type constructor *ExpK* does not take an object-level type as parameter (as *ValK* does), as an expression in CPS does not return a value. The definition of these two types is shown in Figure 4.3, and we clarify a few points about this representation below.

Syntactic classes Ideally, we would like to define *ValK* and *ExpK* as two mutually recursive types. However, the fixed point operator (*Rec*) can only be applied to a single type, so instead we use the same type for the two syntactic categories. (Alternatively, it might be possible to extend the recursion scheme to the case of two or more types, but we have not attempted that.) The distinction between expressions and values is actually not lost: we take advantage of the GADTs to recover this distinction by encoding the corresponding syntactic constraints as type constraints: values have source type $V\ t$ whereas expressions have source type *Void*, so types statically enforce that constructors for values cannot appear where an expression is expected and vice versa.

Polymorphism Whereas on paper it is simpler to have a single letrec operator that abstracts type and term variables and provides recursion, and a single construct that performs both function and (multiple) type applications, it simplifies subsequent transformations somewhat to use more specialized constructs. In particular, we use the following data constructors:

- *Kletrec*, which introduces a monomorphic function which can be recursive (i.e. it covers the case letrec $f[\vec{\alpha}]\ x = e_1$ in e_2, when $\vec{\alpha} = \bullet$),
- *KletPolyFun*, which introduces a polymorphic, non-recursive function which abstracts a single type variable (i.e. it covers the case letrec $f[\vec{\alpha}]\ x = e_1$ in e_2, when $\vec{\alpha} = \tau_1$ and f does not appear free in e_1),

- *Kapp*, which encodes a monomorphic function application (i.e. it covers the case $v_1[\vec{\tau}]\ v_2$, when $\vec{\tau} = \bullet$),
- *KpolyApp*, which encodes a polymorphic function application that applies a single type argument (i.e. it covers the case $v_1[\vec{\tau}]\ v_2$, when $\vec{\tau} = \tau_1$).

This particular choice was made considering that the CPS conversion only ever abstracts a single type variable at once, and the polymorphic functions it introduces are not recursive (although they can introduce recursive functions in their body.) This way, we avoid the complications of multiple type abstraction and applications in the syntax (which would also complicate the subsequent conversion to de Bruijn indices, cf. Chapter 5).

4.2 Translation

The CPS conversion of types, programs (i.e. closed terms) and open terms is shown in Figure 4.4.

The type translation ($\mathcal{K}_{\text{type}}[\![-]\!]$) maps function types (and universal types) to continuations, and leaves the other types unchanged. The type translation is taken from the type-preserving CPS conversion of Morrisett et al. (1999). The type family that encodes $\mathcal{K}_{\text{type}}[\![-]\!]$ is defined as follows:

type family *Ktype t*
type instance *Ktype* $(t_1 \rightarrow t_2)$ = *Cont Z* (*Ktype* t_1, *Cont Z* (*Ktype* t_2))
type instance *Ktype* (*All t*) = *Cont* (*S Z*) (*Cont Z* (*Ktype t*))
type instance *Ktype* (*Var i*) = *Var i*
type instance *Ktype* (t_1, t_2) = (*Ktype* t_1, *Ktype* t_2)
type instance *Ktype Int* = *Int*

Type preservation The usual type preservation theorem states that CPS conversion takes well-typed $\lambda_{\rightarrow}$ programs (i.e. closed expressions) to well-typed $\lambda_{\mathcal{K}}$ programs:

Theorem 4.1 *(CPS type preservation) If* $\bullet;\bullet \vdash e : \tau$ *then* $\bullet;\bullet \vdash_{K} \mathcal{K}_{prog}[\![e]\!]$.

Note that the input type τ does not appear in the typing judgment for the target program. (In the CPS-converted program, the form halt will be applied to a value of type $\mathcal{K}_{\text{type}}[\![\tau]\!]$.)

Theorem 4.1 is reflected in the type of the function which implements $\mathcal{K}_{\text{prog}}[\![-]\!]$:

types:

$$
\begin{aligned}
\mathcal{K}_{\text{type}}[\![\tau_1 \to \tau_2]\!] &= \langle \mathcal{K}_{\text{type}}[\![\tau_1]\!], \mathcal{K}_{\text{type}}[\![\tau_2]\!] \to 0 \rangle \to 0 \\
\mathcal{K}_{\text{type}}[\![\forall \alpha.\ \tau]\!] &= \forall \alpha.\ ((\mathcal{K}_{\text{type}}[\![\tau]\!]) \to 0) \to 0 \\
\mathcal{K}_{\text{type}}[\![\alpha]\!] &= \alpha \\
\mathcal{K}_{\text{type}}[\![\langle \tau_1, \tau_2 \rangle]\!] &= \langle \mathcal{K}_{\text{type}}[\![\tau_1]\!], \mathcal{K}_{\text{type}}[\![\tau_2]\!] \rangle \\
\mathcal{K}_{\text{type}}[\![\mathsf{int}]\!] &= \mathsf{int}
\end{aligned}
$$

programs:

$$
\mathcal{K}_{\text{prog}}[\![e]\!] = \mathcal{K}[\![e]\!]\ (\lambda x\ .\ \mathsf{halt}\ x)
$$

expressions:

$$
\begin{aligned}
\mathcal{K}[\![\mathsf{letrec}\ f\ x = e_1\ \mathsf{in}\ e_2]\!]\ \kappa &= \mathsf{letrec}\ f\ \langle x, k \rangle = \mathcal{K}[\![e_1]\!]\ (\lambda v.\ k\ v) \\
&\ \mathsf{in}\ \mathcal{K}[\![e_2]\!]\ \kappa \\
\mathcal{K}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\ \kappa &= \mathcal{K}[\![e_1]\!]\ (\lambda v\ .\ \mathsf{let}\ x = v\ \mathsf{in}\ \mathcal{K}[\![e_2]\!]\ \kappa) \\
\mathcal{K}[\![x]\!]\ \kappa &= \kappa\ x \\
\mathcal{K}[\![e_1\ e_2]\!]\ \kappa &= \mathcal{K}[\![e_1]\!]\ (\lambda v_1\ .\ \mathcal{K}[\![e_2]\!]\ \lambda v_2\ .\ v_1\ \langle v_2, \kappa \rangle) \\
\mathcal{K}[\![\Lambda \alpha.\ e]\!]\ \kappa &= \kappa\ (\lambda[\alpha]\ c.\ \mathcal{K}[\![e]\!]\ c) \\
\mathcal{K}[\![e[\tau]]\!]\ \kappa &= \mathcal{K}[\![e]\!]\ (\lambda v\ .\ v[\mathcal{K}_{\text{type}}[\![\tau]\!]]\ (\lambda x\ .\ \kappa\ x)) \\
\mathcal{K}[\![\langle e_1, e_2 \rangle]\!]\ \kappa &= \mathcal{K}[\![e_1]\!]\ (\lambda v_1\ .\ \mathcal{K}[\![e_2]\!]\ (\lambda v_2\ .\ \mathsf{let}\ x = (v_1, v_2)\ \mathsf{in}\ \kappa\ x)) \\
\mathcal{K}[\![e.i]\!]\ \kappa &= \mathcal{K}[\![e]\!]\ (\lambda v\ .\ \mathsf{let}\ x = v.i\ \mathsf{in}\ \kappa\ x) \\
e\mathcal{K}[\![n]\!]\ \kappa &= \kappa\ n \\
\mathcal{K}[\![e_1\ p\ e_2]\!]\ \kappa &= \mathcal{K}[\![e_1]\!]\ (\lambda v_1\ .\ \mathcal{K}[\![e_2]\!]\ (\lambda v_2\ .\ \mathsf{let}\ x = v_1\ p\ v_2\ \mathsf{in}\ \kappa\ x)) \\
\mathcal{K}[\![\mathsf{if0}\ e_1\ e_2\ e_3]\!]\ \kappa &= \mathcal{K}[\![e_1]\!]\ (\lambda v\ .\ \mathsf{let}\ c = \lambda x\ .\ \kappa\ x \\
&\qquad\qquad \mathsf{in}\ \mathsf{if0}\ v\ (\mathcal{K}[\![e_2]\!]\ (\lambda v_1\ .\ c\ v_1)) \\
&\qquad\qquad\qquad (\mathcal{K}[\![e_3]\!]\ (\lambda v_2\ .\ c\ v_2)))
\end{aligned}
$$

Figure 4.4: Call-by-value CPS conversion over $\lambda_{\to}$.

$cpsProg :: (\forall\alpha.\ Exp\ \alpha\ t) \rightarrow (\forall\alpha.\ ExpK\ \alpha)$

The proof of this theorem relies of on a lemma which states that $\mathcal{K}[\![-]\!]$ – takes well-typed expressions to well-typed expressions, provided that the supplied continuation has the expected type:[1]

Lemma 4.1 *($\lambda_{\rightarrow} - \lambda_{\mathcal{K}}$ type correspondence) If $\Gamma \vdash e : \tau$ and*

$$\Delta; \mathcal{K}_{type}[\![\Gamma]\!] \vdash_K \lambda x\ .\ \kappa\ x : (\mathcal{K}_{type}[\![\tau]\!] \rightarrow 0) \rightarrow 0$$

then

$$\Delta; \mathcal{K}_{type}[\![\Gamma]\!] \vdash_K \mathcal{K}[\![e]\!]\ \kappa.$$

This lemma is reflected in the signature of the functions that implement $\mathcal{K}[\![-]\!]$ – which, in simplified form, is as follows:

$cpsE :: \forall\beta\ t.\ (\forall\alpha.\ Exp\ \alpha\ t) \rightarrow (ValK\ \beta\ (Ktype\ t) \rightarrow ExpK\ \beta) \rightarrow ExpK\ \beta$

The implementation of *cpsProg* and *cpsE* constitutes a proof of Theorem 4.1 and Lemma 4.1 (which is mechanically verified when the compiler's code is type-checked.) Note that, since we use HOAS and the context $\Delta; \Gamma$ is implicit in our encoding, we get preservation of the type environment "for free".

4.3 Implementation

In this section we discuss the techniques employed to manipulate our higher-order program representation, and show how the mildly optimizing CPS transform of Danvy and Filinski's are implemented in our setting.

4.3.1 Fegaras and Sheard's iterator

There are inherent difficulties with programming with higher-order abstract syntax. For example, suppose we try to implement an evaluator over the simply-typed λ-calculus, that is, a function taking expressions to values:

[1] In this lemma the continuation κ lies at the meta level and must be lifted into an object-level function so it can be the subject of a typing judgment.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Rec}\ \alpha\ \beta\ t\ \textbf{where} \\
\quad \mathit{Roll}\ \ :: (\alpha\ (\mathit{Rec}\ \alpha\ \beta\ t)) \to \mathit{Rec}\ \alpha\ \beta\ t \\
\quad \mathit{Place} :: (\beta\ t) \qquad\qquad\quad\ \to \mathit{Rec}\ \alpha\ \beta\ t \\
\\
\mathit{iter} :: (\forall t.\ \mathit{ExpF}\ (\beta\ t) \to \beta\ t) \to (\forall t.\ (\forall \alpha.\ \mathit{Exp}\ \alpha\ t\ ) \to \beta\ t) \\
\mathit{iter}\ f\ x = \mathit{cata}\ f\ x \\
\\
\mathit{cata} :: (\forall t.\ \mathit{ExpF}\ (\alpha\ t) \to \alpha\ t) \to (\forall t.\ \mathit{Exp}\ \alpha\ t\ \ \to \alpha\ t) \\
\mathit{cata}\ f\ (\mathit{Roll}\ x) = f\ ((\mathit{xmapExpF}\ (\mathit{cata}\ f)\ \mathit{Place})\ x) \\
\mathit{cata}\ f\ (\mathit{Place}\ x) = x \\
\\
\mathit{xmapExpF} ::\ \ (\forall t.\ \alpha\ t \to \beta\ t) \\
\qquad\qquad\quad \to (\forall t.\ \beta\ t \to \alpha\ t) \\
\qquad\qquad\quad \to (\forall t.\ (\mathit{ExpF}\ (\alpha\ t) \to \mathit{ExpF}\ (\beta\ t))) \\
\mathit{xmapExpF}\ f\ g\ x = \\
\quad \textbf{case}\ x\ \textbf{of} \\
\qquad \mathit{TpAbs}\ e \qquad\quad \to \mathit{TpAbs}\ (f\ e) \\
\qquad \mathit{TpApp}\ e \qquad\quad \to \mathit{TpApp}\ (f\ e) \\
\\
\qquad \mathit{Let}\ e_1\ e_2 \qquad\ \ \to \mathit{Let}\ (f\ e_1)\ (f\ .\ e_2\ .\ g) \\
\qquad \mathit{Letrec}\ e_1\ e_2 \quad \to \mathit{Letrec}\ (\lambda\ a\ b \to f\ (e_1\ (g\ a)\ (g\ b)))\ (f\ .\ e_2\ .\ g) \\
\qquad \mathit{App}\ e_1\ e_2 \qquad \to \mathit{App}\ (f\ e_1)\ (f\ e_2) \\
\\
\qquad \mathit{Pair}\ e_1\ e_2 \qquad \to \mathit{Pair}\ (f\ e_1)\ (f\ e_2) \\
\qquad \mathit{Fst}\ e \qquad\qquad\ \to \mathit{Fst}\ (f\ e) \\
\qquad \mathit{Snd}\ e \qquad\qquad \to \mathit{Snd}\ (f\ e) \\
\\
\qquad \mathit{Num}\ i \qquad\qquad \to \mathit{Num}\ i \\
\qquad \mathit{Prim}\ op\ e_1\ e_2 \to \mathit{Prim}\ op\ (f\ e_1)\ (f\ e_1) \\
\qquad \mathit{If0}\ e_1\ e_2\ e_3 \quad\ \to \mathit{If0}\ (f\ e_1)\ (f\ e_2)\ (f\ e_3)
\end{array}
$$

Figure 4.5: Fegaras and Sheard's iterator for the encoding of $\lambda_{\to}$ in Figure 3.4.

$eval :: Exp \rightarrow Val$

with expressions and values defined as:

data *Exp* **where**

$Lambda :: (Exp \rightarrow Exp) \rightarrow Exp \qquad - \lambda x.\ e$

$App :: Exp \rightarrow Exp \rightarrow Exp \qquad - e_1\ e_2$

data *Val* **where**

$Fun :: (Val \rightarrow Val) \rightarrow Val$

To convert a λ-expression (*Lambda*) to a function (*Fun*), we must construct a function on values ($Val \rightarrow Val$) out of a function on expressions ($Exp \rightarrow Exp$). This of course implies converting a value back into an expression. To define the evaluator, we would thus need to also define its inverse function:

$uneval :: Val \rightarrow Exp$

Fegaras and Sheard (1996) developed a technique by which a function (f) over HOAS can be defined without having to define its inverse, by somehow replacing calls to the inverse function by placeholders, and eventually having them cancel out with calls to f. This is achieved by defining an *iterator*, which in essence maps functions that performs an operation on a single element of a data structure, into functions over entire data structures. (The best known example of an iterator is the function *fold* on lists.) Washburn and Weirich (2003) showed how to apply this technique with their parametric representation of HOAS, as used here. The type of the iterator over the parametric encoding of $\lambda_\rightarrow$ is as follows:

$iter :: \forall\beta.\ (\forall t.\ ExpF\ (\beta\ t) \rightarrow \beta\ t) \rightarrow (\forall t.\ (\forall\alpha.\ Exp\ \alpha\ t\ ) \rightarrow \beta\ t)$

Intuitively, the type β stands for "the result of the computation" over the source term (indexed by object-level type). Here, we obtain *cpsE* by applying *iter* with $\beta\ t$ instantiated at the type $CPS\ \alpha\ t$, defined as follows:

type $CPS\ \alpha\ t = (ValK\ \alpha\ (Ktype\ t) \rightarrow ExpK\ \alpha) \rightarrow ExpK\ \alpha$

$cpsE :: \forall t\ \beta.\ (\forall\alpha.\ Exp\ \alpha\ t) \rightarrow CPS\ \beta\ t$

$cpsE = iter\ cpsAux$

The function passed to the iterator visits a single node in the syntax tree, and has type:

$cpsAux :: \forall t\ \alpha.\ ExpF\ (CPS\ \alpha\ t) \rightarrow CPS\ \alpha\ t$

The iterator is a recursive function that applies this function to every node in the abstract syntax tree. The full description of how Fegaras and Sheard's iterator works is outside the scope of this text, but the definition of *iter* for the encoding of $\lambda_{\rightarrow}$ from Figure 3.4 is reproduced in Figure 4.5 for reference. The figure also gives the definition of *Rec*, as it is formulated specifically to accommodate the iterator.

Note that the the data constructors of the type *ExpF* actually take extra arguments, as explained in Section 4.4.1, which affects the definition of *xMapExpF*; see the file SRC.hs, page 156, in Appendix A, for the actual definition which handles those extra arguments. Also note that the version of *xMapExpF* shown in the figure would actually not be accepted by the type checker, because of the implicit instantiation of the type parameter t_2 in the constructors *TpAbs* and *TpApp*, as the type checker cannot verify that the type parameter is instantiated with the same type in the input and output terms. This is not a concern in the actual code, since types are reified at the term level, so that the type parameters are instantiated properly.

4.3.2 Danvy and Filinski's CPS transform

Our compiler actually implements the one-pass CPS conversion of Danvy and Filinski (1992), where so-called *administrative redexes* are reduced on-the-fly. An administrative redex is a redex – i.e. a term of the form $(\lambda x\ .\ e_1)\ e_2$ – which has been introduced as an artifact of the conversion, and does not correspond to a redex in the source program. Such redexes are undesirable as they make the target program unnecessarily large. Danvy and Filinski showed how such redexes can be reduced along the conversion, by defining an auxiliary transformation function, which expects an object-level continuation (i.e. a continuation in the target language) rather than a continuation in the meta language (such as the argument κ to $\mathcal{K}[\![-]\!]\ -$). As shown by Washburn and Weirich (2003), this conversion can be conveniently implemented by adding an extra component to the result of *cpsAux*, that expects an object-level continuation (*cpsObj*) instead of a meta-level one (*cpsMeta*):

$$
\begin{aligned}
&\textbf{data}\ \mathit{CPS}\ \alpha\ t\ \textbf{where}\\
&\quad \mathit{CPS} :: \ ((\mathit{ValK}\ \alpha\ (\mathit{Ktype}\ t) \to \mathit{ExpK}\ \alpha) \to \mathit{ExpK}\ \alpha) && (\mathit{cpsMeta})\\
&\qquad \to ((\mathit{ValK}\ \alpha\ (\mathit{Cont}\ Z\ (\mathit{Ktype}\ t))) \to \mathit{ExpK}\ \alpha) && (\mathit{cpsObj})\\
&\qquad \to \mathit{CPS}\ \alpha\ t
\end{aligned}
$$

By pairing up two functions in the result of *cpsAux*, we are in effect defining two mutually recursive functions. The code of the CPS translation defines the two recursive functions simultaneously. For example, the code that handles the conversion of arithmetic operations is implemented as follows:

$$
\begin{aligned}
&\mathit{cpsAux}\ (\mathit{Prim}\ op\ a\ b) =\\
&\quad \mathit{CPS}\ (\lambda k \to \mathit{cpsMeta}\ a\ (\lambda v_1 \to \mathit{cpsMeta}\ b\ (\lambda v_2 \to\\
&\qquad\quad \mathit{KletPrim}\ op\ v_1\ v_2\ k)))\\
&\qquad (\lambda c \to \mathit{cpsMeta}\ a\ (\lambda v_1 \to \mathit{cpsMeta}\ b\ (\lambda v_2 \to\\
&\qquad\quad \mathit{KletPrim}\ op\ v_1\ v_2\ (\lambda k \to \mathit{Kapp}\ c\ k))))
\end{aligned}
$$

where *cpsMeta* and *cpsObj* are two projection functions used to access the two functions contained in a structure of type $\mathit{CPS}\ \alpha\ t$, defined as follows:

$$
\begin{aligned}
&\mathit{cpsMeta}\ e = \textbf{case}\ e\ \textbf{of}\ \mathit{CPS}\ \mathit{meta}\ _ \to \mathit{meta}\\
&\mathit{cpsObj}\ e = \textbf{case}\ e\ \textbf{of}\ \mathit{CPS}\ _\ \ \mathit{obj} \to \mathit{obj}
\end{aligned}
$$

4.4 Polymorphism

While Theorem 4.1 and Lemma 4.1 cover the theory of type preservation for simple types, polymorphism introduces issues of its own. The technical difficulty is to convince the type checker that we obtain a well-typed term when converting type applications (and abstractions), as it involves reconstructing a term whose type is defined by a substitution. The proof that the constructed term is indeed well-typed relies on the fact that our notion of substitution commutes with the type translation:

Lemma 4.2 *($\mathcal{K}_{type}[\![-]\!]$–subst commute) For any $\lambda_{\to}$ types τ_1, τ_2 and index i,*

$$\mathcal{K}_{type}[\![\tau_1[\tau_2/i]]\!] = (\mathcal{K}_{type}[\![\tau_1]\!])[\mathcal{K}_{type}[\![\tau_2]\!]/i].$$

$$
\begin{array}{l}
\textbf{data}\ \mathit{TypeRep}\ t\ \textbf{where} \\
\quad \mathit{Rarw} :: \mathit{TypeRep}\ t_1 \to \mathit{TypeRep}\ t_2 \to \mathit{TypeRep}\ (t_1 \to t_2) \\
\quad \mathit{Rall} :: \mathit{TypeRep}\ t \to \mathit{TypeRep}\ (\mathit{All}\ t) \\
\quad \mathit{Rvar} :: \mathit{NatRep}\ n \to \mathit{TypeRep}\ (\mathit{Var}\ n) \\
\quad \mathit{Rpair} :: \mathit{TypeRep}\ t_1 \to \mathit{TypeRep}\ t_2 \to \mathit{TypeRep}\ (t_1, t_2) \\
\quad \mathit{Rint} :: \mathit{TypeRep}\ \mathit{Int}
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ \mathit{NatRep}\ n\ \textbf{where} \\
\quad \mathit{Nz} :: \mathit{NatRep}\ Z \\
\quad \mathit{Ns} :: \mathit{NatRep}\ n \to \mathit{NatRep}\ (S\ n)
\end{array}
$$

Figure 4.6: Singleton types representing object-level types.

This means that we actually need to make a coercion like:

$$
\begin{array}{l}
\quad \mathit{ValK}\ (\mathit{Ktype}\ (\mathit{Subst}\ t_1\ t_2\ Z)) \\
\to \mathit{ValK}\ (\mathit{Subst}\ (\mathit{Ktype}\ t_1)\ (\mathit{Ktype}\ t_2)\ Z)
\end{array}
$$

If the types for which t_1 and t_2 stand were known at compile-time, the type-checker could normalize the two types (i.e. apply the definition of the type families) and verify that are indeed equal. But t_1 and t_2 are not known – they can be the representation of any source types of $\lambda_{\to}$. Currently, there is no way of doing such coercion purely at the type level. We therefore need to implement the lemma at the term level, as a function that produces a witness that the coercion is valid for given types t_1 and t_2:

$$
\begin{array}{l}
\mathit{substCpsCommute} :: \mathit{TypeRep}\ t_1 \to \mathit{TypeRep}\ t_2 \\
\qquad \to \mathit{Equiv}\ (\mathit{Ktype}\ (\mathit{Subst}\ t_1\ t_2\ Z)) \\
\qquad\qquad (\mathit{Subst}\ (\mathit{Ktype}\ t_1)\ (\mathit{Ktype}\ t_2)\ Z)
\end{array}
$$

The first two parameters are *singleton types* used to reify object-level types at the term level: a value of type $\mathit{TypeRep}\ t$ is the run-time representation of the object τ, where the type t represents τ. For a given type t, the type $\mathit{TypeRep}\ t$ is inhabited by a single (terminating) term, which is why they are called singleton types. The definition of *TypeRep* is shown in Figure 4.6 (which also defines a singleton type *NatRep* that reifies natural numbers).

The type *Equiv* is used to witness the equality of two types at run-time, and is defined as follows:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Equiv}\ s\ t\ \textbf{where} \\
\quad \mathit{Equiv} :: s \sim t \Rightarrow \mathit{Equiv}\ s\ t
\end{array}
$$

It uses another feature introduced in GHC along with type families, namely *type equality coercions* (Sulzmann et al. 2007). The context $(s \sim t)$ means that the types s and t, although possibly syntactically different, are equivalent after applying a process of normalization (which in particular eliminates applications of type functions.)

There are various ways in which the function *substCpsCommute* can be implemented. It can be written by direct case analysis over the type representatives, and follow the structure of a direct inductive proof of the lemma. For instance, the case for pairs would look as follows:

```
substCpsCommute (Rpair t1^a t1^b) t2 =
  case substCpsCommute t1^a t2 of
    Equiv →
      case substCpsCommute t1^b t2 of
        Equiv → Equiv
```

Another way, which is the one currently employed in the compiler, is to construct the run-time representation of the two types (t_1 and t_2), and then verify that they are equal:

```
substCpsCommute t1 t2 =
  case typesEqual (substT (kType t1) (kType t2) Nz)
                  (kType (substT t1 t2 Nz)) of
    Just Equiv → Equiv
```

where *substT* and *kType* reify the corresponding type families at the term level, and *typesEqual* performs comparison on type representatives:

```
kType :: TypeRep t → TypeRep (Ktype t)
substT :: TypeRep t1 → TypeRep t2 → NatRep i → TypeRep (Subst t1 t2 i)
typesEqual :: TypeRep t1 → TypeRep t2 → Maybe (Equiv t1 t2)
```

The function *typesEqual* is a simple recursive function that compares two type representatives and constructs a proof that they are equal, or returns *Nothing* if the types differ. Of course, the implementation of our lemmas assume that *typesEqual* always succeeds and returns a proof, but if it fails and returns *Nothing*, it means that the "lemma" does not hold for the types in question, and the result will be a run-time error.

Note that, although Haskell's lazy evaluation strategy may suggest otherwise, proof objects are actually checked at run time. The case analysis (**case**) forces the evaluation

of the proof object (of type $Equiv\ t_1\ t_2$); the effect of this case analysis is to check that the result is indeed a proof duely constructed with the *Equiv* data construcotor.

4.4.1 Lemma application

Of course, in order to be able to apply the lemma in its current form, we need to annotate the syntax tree with type representatives. In particular, the data constructors for type abstraction and type application actually need to bear representatives for each object-level type involved, and the polymorphic argument to the constructor for type abstraction needs to be parameterized by a type representative (we show how it is used below, in Section 4.4.2):

$$\begin{aligned} &TpAbs :: TypeRep\ t_1 \rightarrow \\ &\qquad (\forall t_2.\ TypeRep\ t_2 \rightarrow \alpha\ (Subst\ t_1\ t_2\ Z)) \rightarrow ExpF\ (\alpha\ (All\ t_1)) \end{aligned}$$

$$\begin{aligned} &TpApp :: TypeRep\ t_1 \rightarrow TypeRep\ t_2 \rightarrow \\ &\qquad \alpha\ (All\ t_1) \qquad\qquad \rightarrow ExpF\ (\alpha\ (Subst\ t_1\ t_2\ Z)) \end{aligned}$$

Note that these type constructors are enhancements to the type *ExpF* from Figure 4.3 (see the actual code in the source file LK.hs, page 158). With these type representatives stored in the abstract syntax tree, *cpsAux* can then call the lemma to get the required type assumption as needed:

$$\begin{aligned} &cpsAux\ (TpApp\ t_1\ t_2\ e)\ = \\ &\quad \mathbf{case}\ substCpsCommute\ t_1\ t_2\ \mathbf{of} \\ &\qquad Equiv \rightarrow CPS\ (\lambda k \rightarrow cpsMeta\ a\ (\lambda x \rightarrow Kletrec\ (\lambda_{_}\ v \rightarrow k\ v) \\ &\qquad\qquad\qquad\qquad\qquad\qquad (\lambda k' \rightarrow KpolyApp\ x\ k'))) \\ &\qquad\qquad\qquad (\lambda c \rightarrow cpsMeta\ a\ (\lambda x \rightarrow KpolyApp\ x\ c)) \end{aligned}$$

where we have omitted the type representatives in the target program for brevity.

Annotating the abstract syntax tree with type representatives for the constructors *TpAbs* and *TpApp* is sufficient to type-check the CPS-conversion. However, more type representatives are actually needed to instantiate other lemmas in the subsequent phases. These type representatives must be propagated from the abstract syntax tree of the source program through the various program transformations. In particular, the code transformations over first-order representations need to apply lemmas about the type

context. To be able to re-construct the type context while traversing an expression, we will need a type representative for every construct that binds a term variable (such as let). See the source file Src.hs, page 156 in Appendix A.

4.4.2 Type abstraction

A consequence of the higher-order encoding of type abstraction is that the function $\mathcal{K}_{\text{type}}[\![-]\!]$ must be invertible. Consider the data constructors for type abstraction in $\lambda_{\rightarrow}$ and $\lambda_{\mathcal{K}}$:

$$
\begin{aligned}
\mathit{TpAbs} &:: \mathit{TypeRep}\ t_1 \rightarrow \\
&\quad (\forall t_2.\ \mathit{TypeRep}\ t_2 \rightarrow \alpha\ (\mathit{Subst}\ t_1\ t_2\ Z)) \quad \rightarrow \mathit{ExpF}\ (\alpha\ (\mathit{All}\ t_1))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{KletPolyFun} &:: \mathit{TypeRep}\ t_1 \rightarrow \\
&\quad (\forall t_2.\ \mathit{TypeRep}\ t_2 \rightarrow \\
&\qquad \alpha\ (V\ (\mathit{Subst}\ t_1\ t_2\ Z)) \rightarrow \alpha\ \mathit{Void}) \rightarrow \\
&\quad (\alpha\ (V\ (\mathit{Cont}\ (S\ Z)\ t_1)) \rightarrow \alpha\ \mathit{Void}) \qquad \rightarrow \mathit{ExpKF}\ (\alpha\ \mathit{Void})
\end{aligned}
$$

We need to convert the functional argument of *TpAbs* (say f) to that of *KletPolyFun* (say f'). The function f' receives a representative of a type in CPS form, and constructs a representative of the originating type in direct style so as to be able to apply f, and finally converts the resulting term back in CPS. To achieve this, we must define the inverse of $\mathcal{K}_{\text{type}}[\![-]\!]$ as a type family:

$$
\begin{aligned}
&\textbf{type family}\ \mathit{UnKtype}\ t \\
&\textbf{type instance}\ \mathit{UnKtype}\ (\mathit{Cont}\ Z\ (t_1, \mathit{Cont}\ Z\ t_2)) = \quad (\mathit{UnKtype}\ t_1) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \rightarrow (\mathit{UnKtype}\ t_2) \\
&\textbf{type instance}\ \mathit{UnKtype}\ (\mathit{Cont}\ (S\ Z)\ (\mathit{Cont}\ Z\ t)) = \mathit{All}\ (\mathit{UnKtype}\ t) \\
&\textbf{type instance}\ \mathit{UnKtype}\ (\mathit{Var}\ i) \qquad\qquad\qquad\quad = \mathit{Var}\ i \\
&\textbf{type instance}\ \mathit{UnKtype}\ (t_1, t_2) \qquad\qquad\qquad\quad = (\mathit{UnKtype}\ t_1, \mathit{UnKtype}\ t_2) \\
&\textbf{type instance}\ \mathit{UnKtype}\ \mathit{Int} \qquad\qquad\qquad\qquad = \mathit{Int}
\end{aligned}
$$

We also need to reify the type family *UnKtype* at the term level so that we can construct the type representative:

$$\mathit{unKtype} :: \mathit{TypeRep}\ t \rightarrow \mathit{TypeRep}\ (\mathit{UnKtype}\ t)$$

Note that this function on types is actually partial, as not all the types of $\lambda_{\mathcal{K}}$ are the image of $\lambda_{\rightarrow}$ types under $\mathcal{K}_{\text{type}}[\![-]\!]$.

As in the case of type application, the type safety of the conversion of a type abstraction requires the application of the commutativity lemma, *substCpsCommute*. In addition, it requires the application of a lemma which states that the type families *Ktype* and *UnKtype* are inverses:

$$\mathit{lemmaKtypeInverse} :: \mathit{TypeRep}\ t \rightarrow \mathit{Equiv}\ (\mathit{Ktype}\ (\mathit{UnKtype}\ t))\ t$$

This lemma is required to convince the type checker that the supplied continuation is of a type compatible with the converted expression, as the type of the converted expression is obtained by mapping a type in CPS into direct style, and back to CPS.

4.5 Discussion

Our CPS conversion was originally presented at the *PLPV* symposium (Guillemette and Monnier 2006), but was restricted to the simply typed case. As type families were not available at the time, the implementation had to rely on GADTs to encode functions on types. That is, since we could not directly write:

$$\mathit{cpsE} :: \mathit{Exp}\ t\ \rightarrow (\mathit{ValK}\ (\mathit{Ktype}\ t) \rightarrow \mathit{ExpK}) \rightarrow \mathit{ExpK}$$

we had to encode it indirectly, as follows:

$$\mathit{cpsE} :: \mathit{Exp}\ t\ \rightarrow (\exists t'.\ (\mathit{CpsG}\ t\ t', (\mathit{ValK}\ t' \rightarrow \mathit{ExpK}) \rightarrow \mathit{ExpK}))$$

where $\mathit{CpsG}\ t\ t'$ is a GADT that encodes a proof that $\mathcal{K}_{\text{type}}[\![\tau]\!] = \tau'$, where t encodes τ and t' encodes τ'. This scheme has important drawbacks. In particular, the packing and unpacking of existentials clutters the compiler's code and imposed severe run-time overhead. The comparison of the two implementations was the basis of the article presented at the *TFP* symposium (Guillemette and Monnier 2008a).

The fact that the proof of Lemma 4.2 is implemented at the term level is unsatisfactory, as it incurs run-time overhead and forces us to include type representatives in the syntax trees. Also the type checker cannot guarantee that there are no missing cases or infinite loops in the proof of the lemmas. Ideally such lemmas should be verified statically,

something that apparently cannot be done with type families and type equality coercions alone. We have proposed a language extension for the static support of such invariants, allowing the programmer to specify the invariants and provide proofs that type family instances satisfy them (Guillemette and Monnier 2008a; Schrijvers et al. 2008).

Summary

We have seen the implementation of the CPS conversion, which had an important impact on the general structure of the program, as every intermediate computational result is now explicitly bound to some variable (introduced by the let construct), and expressions (including function bodies) do not produce values, but instead communicate the result of their computations by applying a continuation.

This transformation is performed over HOAS, which gives a concise and elegant implementation. It allows us to use function application in the host language to relate variables in the source and target program, and thus dispenses us from having to track variables explicitly, as we will need to do in the closure conversion and hoisting phases (cf. Chapter 6 and 7).

For the code of the CPS conversion to type-check, we had to implement a lemma about type families for the type translation and substitution. We will need similar lemmas in the other code transformations as well.

HOAS was convenient for CPS conversion, but it is not very suitable for closure conversion, so we will switch to a first-order representation in Chapter 5, before we resume actual transformation of the object program.

Chapter 5

Conversion to de Bruijn indices

This chapter documents the conversion from HOAS to de Bruijn indices used in our compiler. We first clarify the reasons which led us to switch from a higher-order representation to a first-order one, and then present the conversion.

The fact that HOAS does not represent variables explicitly has the unfortunate consequence that variables cannot be identified: given two variables a and b, we cannot (directly) determine whether the two variables are actually the same. This ability is needed to perform closure conversion, as it should become clear in Section 6.1. To recover this ability, one needs to somehow "inject" identity into variables, for example by annotating binders with some sort of names or indices. This approach tends to negate the advantages of HOAS in terms of conciseness and elegance, as α-equivalence and substitution ought to come "for free". One would argue that such an "augmented" representation makes HOAS degenerate into something actually more complex than de Bruijn indices – why not simply use de Bruijn indices, then?

Even if closure conversion could be performed over a higher-order encoding and produce higher-order terms as output, such a representation would again be problematic for the next transformation, which hoists functions to the top-level. The function hoisting phase actually relies on the fact that functions are closed. De Bruijn contexts can express this property directly, but HOAS cannot, as value contexts are implicit. It is not clear how such a transformation can be implemented in a language like Haskell. Chlipala's closure conversion over (a variant of) HOAS (2008) manages to do this, but relies on an explicit well-formedness predicate which relates a term to its value context; such a predicate

requires dependent types, so it is not an option in Haskell.

In the face of these arguments in favor of a first-order encoding, we settled for de Bruijn indices for the task of closure conversion and hoisting, although we could probably have managed with HOAS.

The code listing of the conversion to a first-order encoding, and the first-order encoding itself, is reported in Appendix A (in the files LKb.hs, page 166, and ToB.hs, page 170)

Overview

In essence, the conversion to de Bruijn form introduces indices in place of variable occurrences, which are represented by variables of the implementation language in the higher-order encoding. There are two kinds of term-level variables in $\lambda_{\mathcal{K}}$: those that abstract values, and those that abstract types.

For variables abstracting values, we construct an index by comparing the value context at the place where a variable is bound (say, Γ) and the value context at the place of variable occurrence (say, Γ'); the difference in "length" between these two contexts (that is, the number of intervening binders) indicates which de Bruijn index to use.

The treatment of term-level type variables is tricky, as the de Bruijn representation of an object-level type depends on the local type context (Δ). For the purpose of the translation, we will temporarily represent object-level types as reverse de Bruijn indices. Reverse de Bruijn indices reflect the number of traversed binders between the top-level and the place where the variable is bound, so they are *not* sensitive to the local type context, unlike normal de Bruijn indices. This way we avoid tricky interaction between the higher-order term representation and the first-order representation of types (although it would probably be possible to do the conversion in one step.)

In overview, the conversion is performed in two steps:

1. The first step converts the higher-order terms into first-order terms, where the object-level types are represented using reverse de Bruijn indices, but term variables are represented as normal de Bruijn indices.

2. The second step produces an equivalent first-order representation where object-level types are represented back in "normal" de Bruijn indices.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{ValKr}\ \mathit{ctx}\ t\ \textbf{where} & & \\
\quad \mathit{KRvar} :: \mathit{Index}\ \mathit{ts}\ t & \rightarrow \mathit{ValKr}\ (i, \mathit{ts})\ t & \\
\quad \mathit{KRnum} :: \mathit{Int} & \rightarrow \mathit{ValKr}\ (i, \mathit{ts})\ \mathit{Int} &
\end{array}
$$

$$
\begin{array}{lll}
\textbf{data}\ \mathit{ExpKr}\ \mathit{ctx}\ \textbf{where} & & \\
\quad \mathit{KRletrec} & :: \mathit{ExpKr}\ (i, (t, (\mathit{Cont}\ Z\ t, \mathit{ts}))) \rightarrow & \\
 & \quad \mathit{ExpKr}\ (i, (\mathit{Cont}\ Z\ t, \mathit{ts})) & \rightarrow \mathit{ExpKr}\ (i, \mathit{ts}) \\
\quad \mathit{KRletPolyFun} & :: \mathit{ExpKr}\ (S\ i, (t, \mathit{ShiftEnvR}\ i\ \mathit{ts})) \rightarrow & \\
 & \quad \mathit{ExpKr}\ (i, (\mathit{Cont}\ (S\ Z)\ t, \mathit{ts})) & \rightarrow \mathit{ExpKr}\ (i, \mathit{ts}) \\
\quad \mathit{KRlet} & :: \mathit{ValKr}\ (i, \mathit{ts})\ t \rightarrow \mathit{ExpKr}\ (i, (t, \mathit{ts})) & \rightarrow \mathit{ExpKr}\ (i, \mathit{ts}) \\
\quad \mathit{KRletPair} & :: \mathit{ValKr}\ (i, \mathit{ts})\ t_1 \rightarrow \mathit{ValKr}\ (i, \mathit{ts})\ t_2 \rightarrow & \\
 & \quad \mathit{ExpKr}\ (i, ((t_1, t_2), \mathit{ts})) & \rightarrow \mathit{ExpKr}\ (i, \mathit{ts}) \\
\quad \mathit{KRletFst} & :: \mathit{ValKr}\ (i, \mathit{ts})\ (t_1, t_2) \rightarrow & \\
 & \quad \mathit{ExpKr}\ (i, (t_1, \mathit{ts})) & \rightarrow \mathit{ExpKr}\ (i, \mathit{ts}) \\
\quad \mathit{KRletSnd} & :: \mathit{ValKr}\ (i, \mathit{ts})\ (t_1, t_2) \rightarrow & \\
 & \quad \mathit{ExpKr}\ (i, (t_2, \mathit{ts})) & \rightarrow \mathit{ExpKr}\ (i, \mathit{ts}) \\
\quad \mathit{KRletPrim} & :: \mathit{PrimOp} \rightarrow & \\
 & \quad \mathit{ValKr}\ (i, \mathit{ts})\ \mathit{Int} \rightarrow \mathit{ValKr}\ \mathit{Int} \rightarrow & \\
 & \quad \mathit{ExpKr}\ (i, (\mathit{Int}, \mathit{ts})) & \rightarrow \mathit{ExpKr}\ (i, \mathit{ts}) \\
\quad \mathit{KRapp} & :: \mathit{ValKr}\ (i, \mathit{ts})\ (\mathit{Cont}\ Z\ t) \rightarrow \mathit{ValKr}\ (i, \mathit{ts})\ t & \rightarrow \mathit{ExpKr}\ (i, \mathit{ts}) \\
\quad \mathit{KRpolyApp} & :: \mathit{ValKr}\ (i, \mathit{ts})\ (\mathit{Cont}\ (S\ Z)\ t_1) \rightarrow & \\
 & \quad \mathit{ValKr}\ (i, \mathit{ts})\ (\mathit{SubstR}\ t_1\ t_2\ i) & \rightarrow \mathit{ExpKr}\ (i, \mathit{ts}) \\
\quad \mathit{KRif0} & :: \mathit{ValKr}\ (i, \mathit{ts})\ \mathit{Int} \rightarrow & \\
 & \quad \mathit{ExpKr}\ (i, \mathit{ts}) \rightarrow \mathit{ExpKr}\ (i, \mathit{ts}) & \rightarrow \mathit{ExpKr}\ (i, \mathit{ts}) \\
\quad \mathit{KRhalt} & :: \mathit{ValKr}\ (i, \mathit{ts})\ \mathit{Int} & \rightarrow \mathit{ExpKr}\ (i, \mathit{ts})
\end{array}
$$

Figure 5.1: Representation of $\lambda_{\mathcal{K}}$ with reverse de Bruijn indices for types and de Bruijn indices for terms (cf. source file LKb.hs, page 166).

In the rest of this chapter, we explain the program representation that uses reverse de Bruijn indices to encode the types (Section 5.1), then show how terms in this representation are constructed (Section 5.2), and finally show how the object-level types are translated back to normal de Bruijn indices (Section 5.3).

5.1 Type-level reverse de Bruijn indices

Contrary to normal de Bruijn indices, reverse de Bruijn indices reflect the number of traversed binders between the top-level and the place where the variable is bound. That is, the type variable bound by a quantifier appearing at the top-level will be represented with

the index 0; the next variable bound in the scope of this variable will be represented with the index 1, and so on. For example, the type of the *swap* function, $\forall\alpha\ \beta.\ \langle\alpha,\beta\rangle \rightarrow \langle\beta,\alpha\rangle$, which is represented in de Bruijn as the Haskell type:

$$\mathit{All}\ (\mathit{All}\ ((\mathit{Var}\ (S\ Z), \mathit{Var}\ Z) \rightarrow (\mathit{Var}\ Z, \mathit{Var}\ (S\ Z))))$$

is represented in reverse de Bruijn as the Haskell type:

$$\mathit{All}\ (\mathit{All}\ ((\mathit{VarR}\ Z, \mathit{VarR}\ (S\ Z)) \rightarrow (\mathit{VarR}\ (S\ Z), \mathit{VarR}\ Z)))$$

were *VarR* is the type constructor we use to introduce reverse de Bruijn indices. Just like *All* and *Var* (cf. Section 3.1), *VarR* is a type constructor with no data constructor:

data *VarR i*

The strongly typed representation of $\lambda_{\mathcal{K}}$ which uses this encoding of object-level types is shown in Figure 5.1. It is very similar to the first-order order encoding of System F from Section 3.3. The only difference is in the representation of object-level types, which requires a different notion of substitution and shifting, consistent with reverse de Bruijn indices. This is reflected in the types of the constructors for type abstraction and type application:

$$\begin{aligned}
\mathit{KRletPolyFun} &:: \mathit{ExpKr}\ (S\ i, (t, \mathit{ShiftEnvR}\ i\ \mathit{ts})) \rightarrow \\
&\quad\ \mathit{ExpKr}\ (i, (\mathit{Cont}\ (S\ Z)\ t, \mathit{ts})) \qquad \rightarrow \mathit{ExpKr}\ (i, \mathit{ts}) \\
\mathit{KRpolyApp} &:: \mathit{ValKr}\ (i, \mathit{ts})\ (\mathit{Cont}\ (S\ Z)\ t_1) \rightarrow \\
&\quad\ \mathit{ValKr}\ (i, \mathit{ts})\ (\mathit{SubstR}\ t_1\ t_2\ i) \qquad \rightarrow \mathit{ExpKr}\ (i, \mathit{ts})
\end{aligned}$$

These types refer to a different type families (*SubstR* and *ShiftEnvR*) which operate over the reverse de Bruijn type encoding. As in the encoding from Section 3.3, the type parameter i reflects the number of type variables in scope.

The type family *SubstR* is defined as follows:

type family *SubstR s t i*
type instance *SubstR* (*Cont k s*) *t i* = *Cont k* (*SubstR s* (*Ur k i t*) *i*)
type instance *SubstR* (*VarR n*) *t i* = *CMP n i* (*VarR n*) *t* (*VarR* (*Pred n*))
type instance *SubstR* (s_1, s_2) *t i* = (*SubstR* s_1 *t i*, *SubstR* s_2 *t i*)
type instance *SubstR Int t i* = *Int*

The type $\mathit{SubstR}\ s\ t\ i$ stands for the type s where variable corresponding to the reverse de Bruijn index i has been replaced with the type t (where both s and t are represented with reverse de Bruijn indices.)

We also define a type family Ur to implement an update function on reverse de Bruijn types, as follows:

$$
\begin{array}{ll}
\textbf{type family}\ \mathit{Ur}\ k\ i\ t & \\
\textbf{type instance}\ \mathit{Ur}\ k\ i\ (\mathit{Cont}\ j\ t) & = \mathit{Cont}\ j\ (\mathit{Ur}\ k\ i\ t) \\
\textbf{type instance}\ \mathit{Ur}\ k\ i\ (\mathit{VarR}\ n) & = \mathit{VarR}\ (\mathit{CMP}\ n\ k\ n\ (\mathit{Add}\ i\ n)\ (\mathit{Add}\ i\ n)) \\
\textbf{type instance}\ \mathit{Ur}\ k\ i\ (t_1, t_2) & = (\mathit{Ur}\ k\ i\ t_1, \mathit{Ur}\ k\ i\ t_2) \\
\textbf{type instance}\ \mathit{Ur}\ k\ i\ \mathit{Int} & = \mathit{Int}
\end{array}
$$

The type $\mathit{Ur}\ k\ i\ t$ stands for the type t where every index not smaller than k has been incremented by i. Note that, contrary to update on de Bruijn indices, which affects the *free* variables, update on reverse de Bruijn indices affects the *bound* variables.

We also define an auxiliary type family to update all the types in a given context:

$$
\begin{array}{ll}
\textbf{type family}\ \mathit{UenvR}\ k\ i\ ts & \\
\textbf{type instance}\ \mathit{UenvR}\ k\ i\ () & = () \\
\textbf{type instance}\ \mathit{UenvR}\ k\ i\ (t, ts) & = (\mathit{Ur}\ k\ i\ t, \mathit{UenvR}\ k\ i\ ts)
\end{array}
$$

as well as "shift" functions (for individual types and entire contexts), which perform a unit increment on indices corresponding to bound variables:

$$
\begin{array}{l}
\textbf{type}\ \mathit{ShiftR}\ i\ t = \mathit{Ur}\ i\ (S\ Z)\ t \\
\textbf{type}\ \mathit{ShiftEnvR}\ i\ ts = \mathit{UenvR}\ i\ (S\ Z)\ ts
\end{array}
$$

5.2 Construction of first-order terms

In this section we discuss the implementation of the conversion of the higher-order program representation (as defined in Figure 4.3) to the first-order representation defined in the previous section. Its type is:

$$\mathit{toR} :: (\forall \alpha.\ \mathit{ExpK}\ \alpha) \rightarrow \mathit{ExpKr}\ (Z, ())$$

The conversion is implemented using an iterator, similar to that for $\lambda_{\rightarrow}$, but suited to $\lambda_{\mathcal{K}}$. The iterator follows the same pattern as that for $\lambda_{\rightarrow}$; its definition is reproduced along

with the higher-order representation of $\lambda_{\mathcal{K}}$ (see the source file LK.hs, page 158.) Referring to Figure 4.5, the iterator for $\lambda_{\mathcal{K}}$ defines versions of *iter* and *cata* (called *iterK* and *cataK*) for $\lambda_{\mathcal{K}}$ that are essentially the same as those for $\lambda_{\rightarrow}$, but which refer to a version of *xmapExpF* (called *xmapExpKF*) consistent with the data constructors of *ExpKF*[1].

The type of the iterator for $\lambda_{\mathcal{K}}$ is as follows:

$$iterK :: (\forall t.\ ExpKF\ (\beta\ t) \rightarrow \beta\ t) \rightarrow (\forall t.\ (\forall \alpha.\ RecT\ ExpKF\ \alpha\ t) \rightarrow \beta\ t)$$

The core function of the conversion is the one passed to *iterK*, which is of the following type:

$$toRaux :: ExpKF\ (ToR\ \alpha\ t) \rightarrow ToR\ \alpha\ t$$

where the type *ToR* is used to represent the result of the conversion to the first-order representation, and is defined as follows:

$$\begin{array}{l}
\textbf{data}\ ToR\ \alpha\ t\ \textbf{where} \\
\quad ToRv :: (\forall i, ts.\ (NatRep\ i, EnvRep\ ts) \rightarrow \\
\qquad\qquad ValKr\ (i, ts)\ (Rtype\ i\ t)) \qquad \rightarrow ToR\ \alpha\ (V\ t) \\
\quad ToRe :: (\forall i, ts.\ (NatRep\ i, EnvRep\ ts) \rightarrow \\
\qquad\qquad ExpKr\ (i, ts)) \qquad\qquad \rightarrow ToR\ \alpha\ Void
\end{array}$$

The conversion of a value (of type $ValK\ \alpha\ t$) is represented by a term of type $ToR\ \alpha\ (V\ t)$ and introduced by the constructor *ToRv*. Similarly, the conversion of an expression (of type $ExpK\ \alpha$) is represented by a term of type $ToR\ \alpha\ Void$ and introduced by the constructor *ToRe*. In the type of these two constructors, the type parameters i and ts are used to describe the target context in which the expression or value will appear. Notably, the types listed in ts are already converted to reverse de Bruijn form.

The type family *Rtype* converts object-level types from de Bruijn indices to reverse de Bruijn indices, when the type appears with a number of type variables in scope indicated by a parameter i. The type family is defined as follows:

[1] Washburn and Weirich (2003) showed how to automatically derive functions like *xmapExpF*/*xmapExpKF* from the definition of *ExpF*/*ExpKF* by means of polytypic programming.

$$
\begin{aligned}
&\textbf{type family } \mathit{Rtype}\ i\ t \\
&\textbf{type instance } \mathit{Rtype}\ i\ (\mathit{Cont}\ Z\ t) &&= \mathit{Cont}\ Z\ (\mathit{Rtype}\ i\ t) \\
&\textbf{type instance } \mathit{Rtype}\ i\ (\mathit{Cont}\ (S\ Z)\ t) &&= \mathit{Cont}\ (S\ Z)\ (\mathit{Rtype}\ (S\ i)\ t) \\
&\textbf{type instance } \mathit{Rtype}\ i\ (\mathit{Var}\ n) &&= \mathit{VarR}\ (\mathit{Subtract}\ i\ (S\ n)) \\
&\textbf{type instance } \mathit{Rtype}\ i\ (t_1, t_2) &&= (\mathit{Rtype}\ i\ t_1, \mathit{Rtype}\ i\ t_2) \\
&\textbf{type instance } \mathit{Rtype}\ i\ \mathit{Int} &&= \mathit{Int}
\end{aligned}
$$

Note that, since we replace a de Bruijn index with a reverse de Bruijn index while doing the conversion to first-order terms, we must supply a instance of *Rtype* for *Rvar*.

The key part of the implementation is the conversion of variables. Consider the constructor for let in the higher-order encoding (from Figure 4.3):

$$\mathit{Klet} :: \alpha\ (V\ t) \rightarrow (\alpha\ (V\ t) \rightarrow \alpha\ \mathit{Void}) \rightarrow \mathit{ExpKF}\ (\alpha\ \mathit{Void})$$

The term in negative position, of type $\alpha\ (V\ t)$, must be instantiated to a value of type $\mathit{ToR}\ \alpha\ (V\ t)$. The term in question will be a closure which will compare (term-level representatives of) the value context where the variable is bound (say *ts*) and that at the place where the variable occurs (say ts').

A subtle point to consider is that new type variables may have been introduced between the binder and the variable occurrence. In this case, the type environment ts' will have to be shifted accordingly. The number of type variables introduced is witnessed by the difference between the type contexts at the binder and variable occurrence (say i and i').

The function which constructs such variables has the following type:

$$\mathit{toRvar} :: \mathit{TypeRep}\ s \rightarrow \mathit{NatRep}\ i \rightarrow \mathit{EnvRep}\ ts \rightarrow \mathit{ToR}\ \alpha\ (V\ s)$$

Now, the part that "does the work" inspects the two contexts *ts* and *ts*' and forms an index accordingly:

$$\mathit{mkIndex} :: \mathit{EnvRep}\ (t, ts) \rightarrow \mathit{EnvRep}\ ts' \rightarrow \mathit{Index}\ ts'\ t$$

The function *mkIndex* traverses both contexts and constructs an index which reflects the length of the segment of the context ts' exceeding the original context (t, ts). For *mkIndex* to succeed, *ts*' must actually be an extension of the type context (t, ts), in the

sense that new binders may have been introduced between the initial context and that in which the variable appears.

Although it is indeed expected to always be the case, the types we use do not statically guarantee it, so we have to perform a dynamic test that compares the term-level representative of the value contexts.

Polymorphism Consider again the constructor for type abstraction in the higher-order encoding:

$$\begin{aligned} \mathit{KletPolyFun} :: \; & (\forall t_2.\, \alpha\ (V\ (\mathit{Subst}\, t_1\ t_2\ Z)) \rightarrow \alpha\ \mathit{Void}) \rightarrow \\ & (\alpha\ (V\ (\mathit{Cont}\ (S\ Z)\ t_1)) \rightarrow \alpha\ \mathit{Void}) \qquad \rightarrow \mathit{ExpKF}\ (\alpha\ \mathit{Void}) \end{aligned}$$

This time, the argument in negative position is instantiated with a term of type:

$$\mathit{ToR}\ \alpha\ (V\ (\mathit{Subst}\, t_1\ (\mathit{VarR}\ i)\ Z)$$

where i reflects the number of type variables in scope where the type abstraction appears. Since substitution is applied on types in de Bruijn form, the type family *Subst* must be extended with a case for *VarR*:

$$\textbf{type instance}\ \mathit{Subst}\ (\mathit{VarR}\ n)\ t\ i = \mathit{VarR}\ n$$

It turns out that the conversion of a type abstraction does not require the application of a lemma. This can be explained by the fact that the threading of the types is implicit (as it takes place in the host language). However, the dynamic test operated by *toRvar* (i.e. the test which verifies that the type context where the variable appears is an extension of the type context where the variable is bound, taking into account any intervening type abstraction) plays a role similar to that of a lemma.

In contrast, the conversion of a type application, where the argument whose object type is defined by a substitution which appears in positive position, does require the application of a lemma. The lemma states that the conversion to reverse de Bruijn indices commutes with substitution:

$$\begin{aligned} \mathit{substRtypeCommute} :: \; & \mathit{NatRep}\ i \rightarrow \mathit{TypeRep}\ t_1 \rightarrow \mathit{TypeRep}\ t_2 \\ & \rightarrow \mathit{Equiv}\ (\mathit{Rtype}\ (\mathit{Subst}\, t_1\ t_2\ i)\ ) \\ & \qquad (\mathit{SubstR}\ (\mathit{Rtype}\ S\ i\ t_1)\ (\mathit{Rtype}\ i\ t_2)\ i) \end{aligned}$$

```
data ValKb ctx t where
  KBvar  :: Index ts t → ValKb (i, ts) t
  KBnum  :: Int        → ValKb (i, ts) Int

data ExpKb ctx where
  KBletrec     :: ExpKb (i, (t, (Cont Z t, ts))) →
                  ExpKb (i, (Cont Z t, ts))                  → ExpKb (i, ts)
  KBletPolyFun :: ExpKb (S i, (t, ShiftEnv ts)) →
                  ExpKb (i, (Cont (S Z) t, ts))              → ExpKb (i, ts)

  KBlet        :: ValKb (i, ts) t → ExpKb (i, (t, ts))       → ExpKb (i, ts)
  KBletPair    :: ValKb (i, ts) t1 → ValKb (i, ts) t2 →
                  ExpKb (i, ((t1, t2), ts))                  → ExpKb (i, ts)
  KBletFst     :: ValKb (i, ts) (t1, t2) →
                  ExpKb (i, (t1, ts))                        → ExpKb (i, ts)
  KBletSnd     :: ValKb (i, ts) (t1, t2) →
                  ExpKb (i, (t2, ts))                        → ExpKb (i, ts)
  KBletPrim    :: PrimOp →
                  ValKb (i, ts) Int → ValKb Int →
                  ExpKb (i, (Int, ts))                       → ExpKb (i, ts)

  KBapp        :: ValKb (i, ts) (Cont Z t) → ValKb (i, ts) t → ExpKb (i, ts)
  KBpolyApp    :: ValKb (i, ts) (Cont (S Z) t1) →
                  ValKb (i, ts) (Subst t1 t2 Z)              → ExpKb (i, ts)

  KBif0        :: ValKb (i, ts) Int →
                  ExpKb (i, ts) → ExpKb (i, ts)              → ExpKb (i, ts)

  KBhalt       :: ValKb (i, ts) Int                          → ExpKb (i, ts)
```

Figure 5.2: Representation of $\lambda_{\mathcal{K}}$ with de Bruijn indices for types and terms (cf. source file LKb.hs, page 166).

5.3 Reverting to type-level de Bruijn indices

The final stage reverts the representation of object-level types to normal de Bruijn indices. This transformation mainly affects the types, and the target program representation is almost identical to the one in Section 5.1, except that it uses the substitution and update functions for de Bruijn indices. The target representation is shown in Figure 5.2. The only difference lies in the types of the constructors for type abstraction and application:

$$
\begin{aligned}
&\mathit{KBletPolyFun} :: \mathit{ExpKb}\ (S\ i, (t, \mathit{ShiftEnv}\ ts)) \rightarrow \\
&\qquad \mathit{ExpKb}\ (i, (\mathit{Cont}\ (S\ Z)\ t, ts)) \quad \rightarrow \mathit{ExpKb}\ (i, ts) \\
&\mathit{KBpolyApp} :: \mathit{ValKb}\ (i, ts)\ (\mathit{Cont}\ (S\ Z)\ t_1) \rightarrow \\
&\qquad \mathit{ValKb}\ (i, ts)\ (\mathit{Subst}\ t_1\ t_2\ Z) \quad \rightarrow \mathit{ExpKb}\ (i, ts)
\end{aligned}
$$

Note that these type families (*Subst* and *ShifEnv*) are the ones used in the original higher-order encoding of $\lambda_{\mathcal{K}}$ (cf. Section 4.1).

The functions that performs the translation for values and expressions have the following types:

$$
\begin{aligned}
&\mathit{toBv} :: \mathit{NatRep}\ i \rightarrow \mathit{EnvRep}\ ts \rightarrow \mathit{ValKr}\ (i, ts)\ t \\
&\qquad \rightarrow \mathit{ValKb}\ (i, \mathit{Benv}\ i\ ts)\ (\mathit{Btype}\ i\ t) \\
&\mathit{toBe} :: \mathit{NatRep}\ i \rightarrow \mathit{EnvRep}\ ts \rightarrow \mathit{ExpKr}\ (i, ts) \\
&\qquad \rightarrow \mathit{ExpKb}\ (i, \mathit{Benv}\ i\ ts)
\end{aligned}
$$

The type parameter *ts* represents the value context, containing types expressed in reverse de Bruijn. The type family *Btype* converts the representation of an object-level type from reverse de Bruijn to normal de Bruijn indices, with respect to a parameter i which reflects the number of type variables in scope at the point where the type appears in the program. The type family *Benv* does the same for an entire value context, applying *Btype* pointwise to every types in the context.

The type family *Btype*, which is the inverse of the function *Rtype* from the previous section, is defined as follows:

$$
\begin{aligned}
&\textbf{type family}\ \mathit{Btype}\ i\ t \\
&\textbf{type instance}\ \mathit{Btype}\ i\ (\mathit{Cont}\ Z\ t) = \mathit{Cont}\ Z\ (\mathit{Btype}\ i\ t) \\
&\textbf{type instance}\ \mathit{Btype}\ i\ (\mathit{Cont}\ (S\ Z)\ t) = \mathit{Cont}\ (S\ Z)\ (\mathit{Btype}\ (S\ i)\ t) \\
&\textbf{type instance}\ \mathit{Btype}\ i\ (\mathit{Var}\ n) = \mathit{Var}\ n \\
&\textbf{type instance}\ \mathit{Btype}\ i\ (\mathit{VarR}\ n) = \mathit{Var}\ (\mathit{Subtract}\ i\ (S\ n)) \\
&\textbf{type instance}\ \mathit{Btype}\ i\ (t_1, t_2) = (\mathit{Btype}\ i\ t_1, \mathit{Btype}\ i\ t_2) \\
&\textbf{type instance}\ \mathit{Btype}\ i\ \mathit{Int} = \mathit{Int}
\end{aligned}
$$

and the type family *Benv* is defined as follows:

$$
\begin{aligned}
&\textbf{type family}\ \mathit{Btype}\ i\ ts \\
&\textbf{type instance}\ \mathit{Benv}\ i\ () = () \\
&\textbf{type instance}\ \mathit{Benv}\ i\ (t, ts) = (\mathit{Btype}\ i\ t, \mathit{Benv}\ i\ ts)
\end{aligned}
$$

Finally, the conversion to first-order terms (*toR*) and the reversal to type-level de Bruijn indices (*toBe*) are composed into one:

$$toB :: (\forall\alpha.\ ExpK\ \alpha) \to ExpKb\ (Z, ())$$

Summary

We have seen a conversion from a higher-order representation (of the CPS language, $\lambda_{\mathcal{K}}$) to a first-order one. This conversion has no real effect on the code, as these are merely different ways of representing the same program. This conversion turns out to be a relatively complex one to implement. In retrospect, implementing the front-end over a first-order encoding would indeed simplify the compiler's code.

If we wanted to stick to HOAS for closure conversion, this could probably be done by combining closure conversion and hoisting in a single phase, as done by Chlipala (2008). As functions are constructed directly at the top level, he avoids the problems of a higher-order program representation where functions are closed but have free variables in scope.

Unfortunately, some important invariants of our implementation escape static verification. The static context of type and term variables, which is implicit in the higher-order encoding, gets constructed as binders are traversed. Index formation involves explicitly *comparing* segments of static contexts; if there are intervening binders for type variables, then the types in the original context must be "shifted" to reflect these new binders. These manipulations take place at run time and amount to testing rather than verification.

Also, we do not prove that the type-level conversion to reverse de Bruijn indices and the conversion back to de Bruijn indices indeed cancel out. This invariant cannot be reflected because each conversion is operated over an entire expression, whose type does not specify an object-level type since it is in CPS. If the object language were in direct style, the relationship between the object-level types would be explicit.

The original motivation for this conversion was to facilitate closure conversion, as it requires explicit type contexts (to prove that functions are closed), and some way of identifying variables, which our higher-order representation lacked. Closure conversion and the subsequent code transformations will be performed over first-order representations.

Chapter 6

Closure conversion

Closure conversion is the core transformation in a compiler for a higher-order functional language, and its implementation is considerably more involved than that of CPS conversion, or the other phases in our compiler.

The effect of closure conversion is to transform all functions in a program so that they are closed (i.e. do not have free variables), by arranging for the functions to receive a copy of their free variables as an additional parameter. Once function are closed, they can be moved around in the program, and the hoisting transformation (cf. Chapter 7) will take advantage of this fact to simplify the structure of the program, turning its nested structure into a linear one.

Closure conversion addresses the problem of accessing free variables at run time. Consider the following program:

$$\begin{array}{l} \mathsf{let}\ f = \lambda x\ .\ \lambda y\ .\ x + y \\ \quad f_1 = f\ 1 \\ \quad f_2 = f\ 2 \\ \mathsf{in}\ f_1\ 2 + f_2\ 4 \end{array}$$

The program introduces a function f which receives a parameter x, and returns a function which add x to its argument y. Two functions f_1 and f_2 are created by applying f with different values, in effect constructing instances of the function $\lambda y\ .\ x + y$, in which x is bound to different values at run time (1 and 2, respectively). At the time f_1 or f_2 is called, the function f has already returned, so the variable x is not in scope, and its value

cannot be accessed as an ordinary formal parameter. To remedy this, we arrange for the function $\lambda y\ .\ x + y$ to take x as an extra parameter:

$$
\begin{aligned}
&\mathsf{let}\ f = \lambda x\ .\ \langle \lambda \langle y, x\rangle\ .\ x + y, x\rangle \\
&\quad f_1 = f\ 1 \\
&\quad f_2 = f\ 2 \\
&\mathsf{in}\ (\mathsf{fst}\ f_1)\ \langle \mathsf{snd}\ f_1, 2\rangle + (\mathsf{fst}\ f_2)\ \langle \mathsf{snd}\ f_2, 4\rangle
\end{aligned}
$$

The function $\lambda y\ .\ x + y$ is replaced by a tuple consisting of a *closed* function, $\lambda\langle y, x\rangle\ .\ x + y$, as well as a copy of the free variable x. To apply the function, it must first be extracted from the tuple, and the stored variable (x) must be passed to it as an extra argument. Of course, in general, there may be more than one free variable, and those are aggregated into a tuple, which we call the *environment*. Note that we have not shown the effect of closure conversion on the outer abstraction, ($\lambda x \ldots$), but the corresponding function would in fact be made to take an extra argument as well (to be instantiated with an empty tuple since the function has no free variables.)

The difficult part in implementing closure conversion is to produce the code that creates the closure. It requires the analysis of free variables, used for forming the closure environment. It must also arrange for the free variables to be accessed through the extra parameter to the function. In the above example, the free variable (x) is simply referred to by its name, so the body of the function remains unchanged, but the actual implementation uses de Bruijn indices, and the code for constructing a closure requires delicate index manipulations.

Closure conversion of System F is a complex matter to implement. Since closure conversion affects the way variables are represented at run time, its implementation is directly concerned with the representation of variables, recursion, and polymorphism. To make the presentation more digestible, we first review the basic workings of closure conversion in a simplified case, before proceeding to the technical presentation of the actual implementation.

Our article presented at the Haskell Workshop (Guillemette and Monnier 2007) showed in detail an implementation of closure conversion for a simply typed functional language. Our article presented at *ICFP* (Guillemette and Monnier 2008b) showed how we extended the implementation to handle polymorphism (and term-level recursion), but there was

$$
\begin{array}{rcl}
\mathcal{C}[\![x]\!] & = & x \\
\mathcal{C}[\![\lambda x\ .\ e]\!] & = & \langle \lambda \langle x, x_{env} \rangle\ .\ e_{body}, e_{env} \rangle \\
 & & \text{where } y_1, \ldots, y_n = FV(e) \\
 & & \qquad e_{body} = \mathsf{let}\ y_0 = x_{env}.1 \\
 & & \qquad\qquad \vdots \\
 & & \qquad\qquad y_{n-1} = x_{env}.n \\
 & & \qquad\quad \mathsf{in}\ \mathcal{C}[\![e]\!] \\
 & & \qquad e_{env} = \langle y_0, \ldots, y_{n-1} \rangle \\
\mathcal{C}[\![e_1\ e_2]\!] & = & \mathsf{let}\ \langle x_f, x_{env} \rangle = \mathcal{C}[\![e_1]\!] \\
 & & \mathsf{in}\ x_f\ \langle \mathcal{C}[\![e_2]\!], x_{env} \rangle
\end{array}
$$

Figure 6.1: Closure conversion of the simply typed (direct style) λ-calculus.

little room for elaboration given the scope of the paper. This chapter synthesizes the two presentations, and discusses the implementation in full detail.

The code listing of the encoding of the closure converted language and the closure conversion is reported in Appendix A (in the files LC.hs, page 177, and CC.hs, page 179).

6.1 Closure conversion and de Bruijn indices

The purpose of this section is to explain how closure conversion is made to work with de Bruijn indices. To simplify the presentation, we ignore the issues of type preservation, polymorphism and recursion for the moment; we will show the closure conversion of the simply typed λ-calculus in direct style and will return to our actual language in the subsequent sections. The essential translation rules of closure conversion are shown in Figure 6.1. In closure-converting the body of a λ-abstraction, one must arrange for (free) variable references to be turned into references to the corresponding variables stored in the environment. In the definition of $\mathcal{C}[\![-]\!]$, this is simply achieved by instantiating a number of let-bindings with the *same names* as the original variables, each variable being bound to the corresponding value in the environment. (For instance, in the example from Section 2.4.3, the function *c2f* accesses the free variables a and b through the *local* bindings of the same names, suitably instantiated to values from the environment.) Here, we wish to apply this technique to our concrete representation with de Bruijn indices; but indeed, given that there are no variable names, we have to work a little harder.

Essentially, since we cannot rely on names, we have to carry around a map that

$$
\begin{array}{rcl}
\mathcal{C}_b[\![i]\!]m & = & \mathit{lookup}\ m\ i \\
\mathcal{C}_b[\![\lambda\ e]\!]m & = & \langle \lambda\ e_{body}, e_{env} \rangle \\
 & & \text{where } (m', [j_0, \ldots, j_{n-1}]) = \mathit{mkMap}\ (\mathit{tail}\ (\mathit{fvs}\ e))\ 0 \\
 & & \qquad e_{body} = \mathsf{let}\ i_0.0 \qquad \textit{(original argument)} \\
 & & \qquad\qquad\qquad i_1.1 \qquad \textit{(environment)} \\
 & & \qquad\qquad\quad \mathsf{in}\ \mathcal{C}_b[\![e]\!](i_1 : \mathit{map}\ (\lambda j\ .\ i_0.j)\ m') \\
 & & \qquad e_{env} = \langle m\ j_0, \ldots, m\ j_{n-1} \rangle \\
\mathcal{C}_b[\![e_1\ e_2]\!]m & = & \mathsf{let}\ \mathcal{C}_b[\![e_1]\!]m \\
 & & \quad i_0.0 \qquad (x_f) \\
 & & \quad i_1.1 \qquad (x_{env}) \\
 & & \mathsf{in}\ i_1\ \langle \mathcal{C}_b[\![e_2]\!]m, i_0 \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{mkMap}\ []\ j & = & ([], []) \\
\mathit{mkMap}\ (\mathit{False} : \mathit{bs})\ j & = & ((\bot : m), js) \\
\mathit{mkMap}\ (\mathit{True} : \mathit{bs})\ j & = & ((n : m), js \mathbin{+\!\!+} [j]) \\
\quad \text{where } (m, js) = \mathit{mkMap}\ \mathit{bs}\ (j + 1) & &
\end{array}
$$

$$
\begin{array}{l}
\mathit{fvs}\ e = [b_0, b_1, \ldots \mid b_i = True \text{ if } i_i \text{ appears in } e; \\
\qquad\qquad\qquad\qquad False \text{ otherwise}]
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{shift}\ i_n & = & i_{n+1} \\
\mathsf{shift}\ i_n.k & = & i_{n+1}.k
\end{array}
$$

Figure 6.2: Closure conversion with de Bruijn indices.

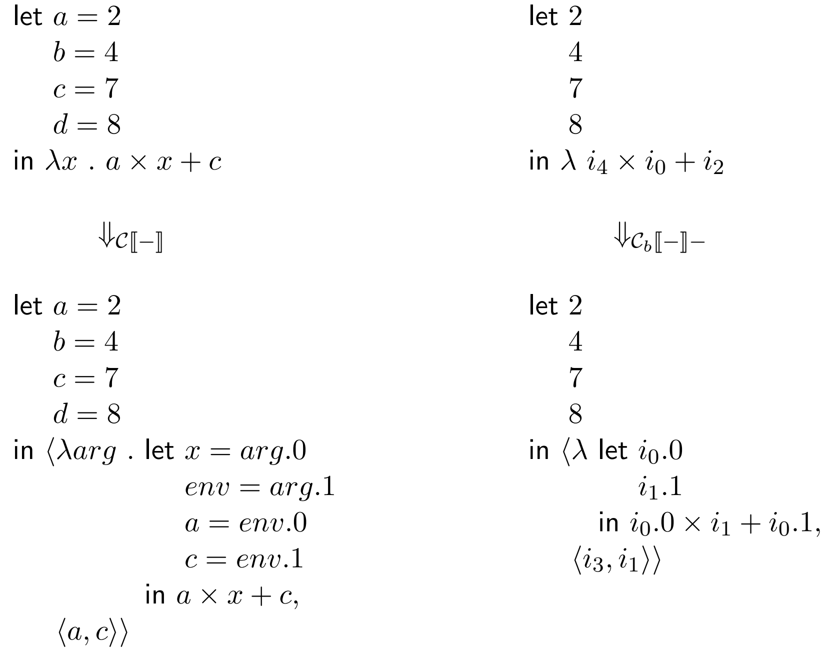

Figure 6.3: Example of closure conversion with variable names (left) and de Bruijn indices (right).

gives the local binding in the converted program for each variable in scope in the source program. We denote $\mathcal{C}_b[\![e]\!]m$ the closure-converted form of source program e given local bindings map m; the function $\mathcal{C}_b[\![-]\!]-$ is defined in Figure 6.2. It refers to auxiliary functions *mkMap* and *fvs* that are used to construct the map m when forming closures. Since we are using de Bruijn indices, we omit all variables names, so we write $\lambda x \,.\, e$ as $\lambda\ e$ and $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ as $\mathsf{let}\ e_1\ \mathsf{in}\ e_2$; we write de Bruijn indices as i_0, i_i, and so on, and use the meta-variables i and j to stand for de Bruijn indices.

The local variables map m, for a source term with n variables in scope, is of the form $[e_0, \ldots, e_{n-1}]$, where e_k gives the local binding in the target program for source variable i_k. In general, e_k will be either a de Bruijn index (when i_k is a local variable of the function being converted) or a projection of the environment (when i_k is a free variable.)

To illustrate, consider the source program shown at the top of Figure 6.3; the final result of the conversion is shown at the bottom. We now go through the steps involved in closure-converting this function.

The first step computes the free variables. Rather than producing a set, the *fvs* function produces a "bit-map", indicating whether each index in scope appears in the

term. Taking the free variables of the function body, we have:

$$fvs\ (i_4 \times i_0 + i_2) = [True, False, True, False, True]$$

which reads, from left to right: i_0 appears in the term, i_1 does not, i_2 appears, and so on.

Next is the construction of the environment and the corresponding local variables map, which is handled by *mkMap*. We have:

$$\begin{aligned} & (m', [j_0, \ldots, j_{n-1}]) \\ = {} & mkMap\ (tail\ (fvs\ (i_4 \times i_0 + i_2)))\ 0 \\ = {} & mkMap\ (tail\ [True, False, True, False, True])\ 0 \\ = {} & mkMap\ [False, True, False, True]\ 0 \\ = {} & ([\bot, 1, \bot, 0], [i_3, i_1]) \end{aligned}$$

The first component, m', maps variables in scope in the function's body (except the function's original argument, i_0) to corresponding projections of the environment. From this m', $\mathcal{C}_b[\![-]\!]-$ constructs a map in which to interpret the function's body:

$$(i_1 : map\ (\lambda j\ .\ i_0.j)\ m') = [i_1, \bot, i_0.1, \bot, i_0.0]$$

which reads, form left to right:

1. the source variable i_0 is mapped to local variable i_1,
2. the source variable i_1 is not mapped to any local variable, as indicated by $\bot$ (since the variable is in scope but does not appear in the source term, this is indeed what we want),
3. the source variable i_2 is mapped to $i_0.1$, the first projection of the environment,

and so on. The second component produced by *mkMap*, namely $[j_0, \ldots, j_{n-1}]$, simply enumerates the source variables that appear in the function's body. Finally, the function's body can be converted:

$$\mathcal{C}_b[\![i_4 \times i_0 + i_2]\!][i_1, \bot, i_0.1, \bot, i_0.0] = i_0.0 \times i_1 + i_0.1$$

$$
\begin{array}{lll}
(\textit{type context}) & \Delta & ::= \alpha_0, \ldots, \alpha_{n-1} \\
(\textit{value context}) & \Gamma & ::= x_0 : \tau_0,\ \ldots,\ x_{n-1} : \tau_{n-1} \\
(\textit{types}) & \tau & ::= \forall \vec{\alpha}.\ \tau \to 0 \mid \exists \alpha.\ \tau \mid \alpha \mid \langle \tau_0, \ldots, \tau_{n-1} \rangle \mid \mathsf{int} \\
(\textit{values}) & v & ::= \mathsf{fix}\ f[\vec{\alpha}]\ x.\ e \mid x \mid \mathsf{pack}\ [\tau_1, v]\ \mathsf{as}\ \tau_2 \mid v[\tau] \mid n \\
(\textit{exps}) & e & ::= \mathsf{let}\ x = v\ \mathsf{in}\ e \mid \mathsf{let}\ [\alpha, x] = \mathsf{unpack}\ v\ \mathsf{in}\ e \\
& & \quad \mid \mathsf{let}\ x = \langle v_0, \ldots, v_{n-1} \rangle\ \mathsf{in}\ e \mid \mathsf{let}\ x = v.i\ \mathsf{in}\ e \\
& & \quad \mid \mathsf{let}\ x = v_1\ p\ v_2\ \mathsf{in}\ e \mid v_1\ v_2 \mid \mathsf{if0}\ v\ e_1\ e_2 \mid \mathsf{halt}\ v
\end{array}
$$

Figure 6.4: Syntax of $\lambda_{\mathcal{C}}$.

Object type	Haskell type
$\forall \vec{\alpha}.\ \tau \to 0$	*Cont k t*
$\exists \alpha.\ \tau$	*Exists t*
α	*Var i*
$\langle \tau_0, \ldots, \tau_{n-1} \rangle$	$\mathit{Tup}\ (t_0, (\ldots, (t_{n-1}, ())\ldots))$
int	*Int*

Figure 6.5: Encoding of the types of $\lambda_{\mathcal{C}}$.

What we have shown here is a mostly conventional formulation of closure conversion, only slightly contrived to facilitate typing. In the rest of this chapter, we will show the implementation on our actual source language, and assign types to $\mathcal{C}_b[\![-]\!]-$, *fvs* and *mkMap*.

6.2 Target language

The actual target language of our closure conversion, $\lambda_{\mathcal{C}}$, is shown in Figure 6.4. It extends $\lambda_{\mathcal{K}}$ with existential types, used to represent closures. Another important difference with $\lambda_{\mathcal{K}}$ is that its static semantics forces functions introduced by fix to be closed (and moves function introduction to the syntactic class of values.) It also decouples type application from function application, moving type applications to the syntactic class of values as well. Finally, it replaces pairs with n-tuples, used for the representation of closure environments.

The static semantics of $\lambda_{\mathcal{C}}$ is defined by two typing judgments:

$\Delta; \Gamma \vdash_{\mathcal{C}} v : \tau$	value v has type τ in context $\Delta; \Gamma$
$\Delta; \Gamma \vdash_{\mathcal{C}} e$	expression e is well-typed in context $\Delta; \Gamma$

```
data ValC ctx t where
  Cfix    :: ExpC (k, (t, (Cont k t, ()))) → ValC (i, ts) (Cont k t)
  Cvar    :: Index ts t                    → ValC (i, ts) t
  Cpack   :: ValC (i, ts) (Subst t1 t2 Z)  → ValC (i, ts) (Exists t1)
  CtpApp  :: ValC (i, ts) (Cont (S k) t1)  → ValC (i, ts) (Cont k (Subst t1 t2 k))
  Cnum    :: Int                           → ValC (i, ts) Int

data ExpC ctx where
  Clet     :: ValC (i, ts) t → ExpC (i, (t, ts))            → ExpC (i, ts)
  Cunpack  :: ValC (i, ts) (Exists t) →
              ExpC (S i, (t, ShiftEnv ts))                   → ExpC (i, ts)

  CletTup  :: MapT (ValC (i, ts)) ts1 →
              ExpC (i, (Tup ts1, ts))                        → ExpC (i, ts)
  CletProj :: ValC (i, ts) (Tup ts1) → Index ts1 t →
              ExpC (i, (t, ts))                              → ExpC (i, ts)
  CletPrim :: PrimOp →
              ValC (i, ts) Int → ValC Int →
              ExpC (i, (Int, ts))                            → ExpC (i, ts)

  Capp     :: ValC (i, ts) (Cont Z t) → ValC (i, ts) t → ExpC (i, ts)

  Cif0     :: ValC (i, ts) Int →
              ExpC (i, ts) → ExpC (i, ts)                    → ExpC (i, ts)

  Chalt    :: ValC (i, ts) Int                               → ExpC (i, ts)

data MapT c ts where
  M0 :: MapT c ()
  Ms :: c t → MapT c ts → MapT c (t, ts)
```

Figure 6.6: Strongly typed representation of $\lambda_{\mathcal{C}}$ (cf. source file LC.hs, page 177).

Strongly typed representation

The strongly typed representation of $\lambda_{\mathcal{C}}$ is constructed in the same way as the first-order representation of $\lambda_{\mathcal{K}}$ with de Bruijn indices at the type level (cf. Section 5.3). The encoding of object-level types is shown in Figure 6.5. In particular, it introduces a new type constructor for existential types (*Exists*) which implicitly binds a type variable. It also introduces a constructor for the representation of tuple types (*Tup*); note that argument to *Tup* is of the same form as a de Bruijn value context (cf. Section 2.2.3).

The encoding of the abstract syntax of $\lambda_{\mathcal{C}}$ is shown in Figure 6.6. A value v satisfying $\Delta; \Gamma \vdash_{\mathcal{C}} v : \tau$ is represented by a term of type:

$$\mathit{ValC}\ (i, \mathit{ts})\ t$$

and an expression e satisfying $\Delta; \Gamma \vdash_{\mathcal{C}} e$ is represented by a term of type:

$$\mathit{ExpC}\ (i, \mathit{ts})$$

where i encodes the length of Δ, ts encodes Γ, and t encodes τ.

The fix operator of $\lambda_{\mathcal{C}}$ binds a number of type variables (in addition to the function's argument and the binder for the recursive call.) The typing rule for fix forces the function to be closed, i.e. exempt of free term or type variables:

$$\frac{\vec{\alpha}; x : \tau, f : \forall \vec{\alpha}.\ \tau \to 0 \vdash_{\mathcal{C}} e}{\Delta; \Gamma \vdash_{\mathcal{C}} \mathsf{fix}\ f[\vec{\alpha}]\ x.\ e : \forall \vec{\alpha}.\ \tau \to 0}$$

The function is closed in the sense that the function's body (e) must be well-typed in an environment where only the type variables abstracted by the function ($\vec{\alpha}$), the function itself (f), and the function's argument (x) are in scope. The encoding of $\lambda_{\mathcal{C}}$ includes a single constructor (*Cfix*) that directly encodes the typing rule for fix. (This is in contrast to the encoding of $\lambda_{\mathcal{K}}$, where type abstraction and term-level recursion were treated separately.) Its type reflects the closedness conditions: the body's term variable context finishes with "()", meaning that it cannot have free term variables, and since t appears in a context where k variables are in scope, t cannot involve type variables other than those bound by the fix.

n-tuples The constructor *CletTup* introduces a tuple made of an arbitrary number of values. The values are aggregated using the auxiliary type *MapT* (also shown in Figure 6.6). A tuple of values $\langle v_0, v_1, \ldots, v_{n-1}\rangle$ of type $\langle \tau_0, \tau_1, \ldots, \tau_{n-1}\rangle$ is introduced with a term of the form:

$$\mathit{Ms}\ u_0\ (\mathit{Ms}\ u_1\ (\ldots\ (\mathit{Ms}\ u_{n-1}\ \mathit{M0})))$$

which is of type:

$$\mathit{MapT}\ (\mathit{ValC}\ (i, \mathit{ts}))\ (t_0, (\ldots, (t_{n-1}, ())\ldots))$$

where each type t_j is the Haskell representation of the type τ_j, and each term u_j is the Haskell representation of the value v_j and has type $\mathit{ValC}\,(i, \mathit{ts})\ t_j$. Note that the first type parameter of the type constructor *MapT* must be a type function (in our case, a type constructor), i.e. a Haskell type of kind $\star \to \star$. We will further discuss the type *MapT* and use it for other purposes than representing syntax in Section 6.5.

The constructor *CletProj* projects a particular component of a tuple value and binds it to a variable. As the representation of tuple types follows the same form as that of value contexts (*ts*), we re-use the *Index* type used to represent de Bruijn indices (cf. Section 2.2.3) to identify the component to be extracted.

Existential types The language introduces existential types, which are used to abstract the type of the environment when forming closures. The usual typing rules for existential types are as follows:

$$\frac{\Delta;\Gamma \vdash_C v : \tau_2[\tau_1/\alpha]}{\Delta;\Gamma \vdash_C \mathsf{pack}\ [\tau_1, v]\ \mathsf{as}\ \exists\alpha.\ \tau_2 : \exists\alpha.\ \tau_2} \qquad \frac{\Delta;\Gamma \vdash_C v : \exists\alpha.\ \tau \quad \alpha,\Delta; x:\tau,\Gamma \vdash_C e}{\Delta;\Gamma \vdash_C \mathsf{let}\ [\alpha, x] = \mathsf{unpack}\ v\ \mathsf{in}\ e}$$

The data constructors for pack and unpack encode these typing rules in much the same way that we did for universal types. Note that in the type of *Cunpack*, since a new type variable is in scope in the body of the expression (e), the types appearing in its context (Γ) must be adjusted, hence the application of *ShiftEnv*.

types:

$$\begin{array}{rcl}
\mathcal{C}_{\text{type}}[\![\forall\vec{\alpha}.\ \tau \to 0]\!] & = & \exists\beta.\ \langle \forall\vec{\alpha}.\ \langle \mathcal{C}_{\text{type}}[\![\tau]\!], \beta\rangle \to 0, \beta\rangle \\
\mathcal{C}_{\text{type}}[\![\alpha]\!] & = & \alpha \\
\mathcal{C}_{\text{type}}[\![\langle\tau_1,\tau_2\rangle]\!] & = & \langle \mathcal{C}_{\text{type}}[\![\tau_1]\!], \mathcal{C}_{\text{type}}[\![\tau_2]\!]\rangle \\
\mathcal{C}_{\text{type}}[\![\mathsf{int}]\!] & = & \mathsf{int}
\end{array}$$

values:

$$\begin{array}{rcl}
\mathcal{C}_{\text{val}}[\![x]\!]m & = & \mathit{lookup}\ m\ x \\
\mathcal{C}_{\text{val}}[\![n]\!]m & = & n
\end{array}$$

expressions:

$$\begin{array}{rcl}
\mathcal{C}_{\text{exp}}[\![\mathsf{letrec}\ f[\vec{\alpha}]\ x = (e_1)^{\tau}\ \mathsf{in}\ e_2]\!]m & = & \mathsf{let}\ x_{closure} = \mathsf{pack}\ [\tau_{env}, \langle v_{code}[\vec{\beta}], v_{env}\rangle] \\
& & \qquad\qquad \mathsf{as}\ \mathcal{C}_{\text{type}}[\![\tau]\!] \\
& & \mathsf{in}\ \mathcal{C}_{\text{exp}}[\![e_2]\!](f \Rightarrow x_{closure}; m) \\
& & \text{where} \\
& & \quad \vec{\beta} = ftvs\ e_1 - \vec{\alpha} \\
& & \quad y_0^{\tau_0}, ..., y_{n-1}^{\tau_{n-1}} = fvs\ e_1 - \{f, x\} \\
& & \quad \tau_{env} = \langle \mathcal{C}_{\text{type}}[\![\tau_0]\!], \ldots, \mathcal{C}_{\text{type}}[\![\tau_{n-1}]\!]\rangle \\
& & \quad v_{code} = \mathsf{fix}\ f[\vec{\beta}, \vec{\alpha}]\ x. \\
& & \qquad\qquad \mathsf{let}\ x' = x.0 \\
& & \qquad\qquad\quad env = x.1 \\
& & \qquad\qquad\quad f' = \mathsf{pack}\ [\tau_{env}, \langle f, env\rangle] \\
& & \qquad\qquad\qquad \mathsf{as}\ \mathcal{C}_{\text{type}}[\![\tau]\!] \\
& & \qquad\qquad \mathsf{in}\ \mathcal{C}_{\text{exp}}[\![e_1]\!](x \Rightarrow x', f \Rightarrow f', \\
& & \qquad\qquad\qquad y_0 \Rightarrow env.0, ..., \\
& & \qquad\qquad\qquad y_{n-1} \Rightarrow env.(n-1)) \\
& & \quad v_{env} = \langle \mathit{lookup}\ m\ y_0, ..., \mathit{lookup}\ m\ y_{n-1}\rangle \\
\mathcal{C}_{\text{exp}}[\![v_1[\tau_1, ..., \tau_n]\ v_2]\!]m & = & \mathsf{let}\ [\alpha, x] = \mathsf{unpack}\ \mathcal{C}_{\text{val}}[\![v_1]\!]m \\
& & \quad x_f = x.0 \\
& & \quad x_{env} = x.1 \\
& & \mathsf{in}\ x_f[\mathcal{C}_{\text{type}}[\![\tau_0]\!], ..., \mathcal{C}_{\text{type}}[\![\tau_{n-1}]\!]]\ \langle \mathcal{C}_{\text{val}}[\![v_2]\!]m, x_{env}\rangle \\
\mathcal{C}_{\text{exp}}[\![\mathsf{let}\ x = v\ \mathsf{in}\ e]\!]m & = & \mathsf{let}\ x' = \mathcal{C}_{\text{val}}[\![v]\!]m\ \mathsf{in}\ \mathcal{C}_{\text{exp}}[\![e]\!](x \Rightarrow x'; m) \\
\mathcal{C}_{\text{exp}}[\![\mathsf{let}\ x = \langle v_1, v_2\rangle\ \mathsf{in}\ e]\!]m & = & \mathsf{let}\ x' = \langle \mathcal{C}_{\text{val}}[\![v_1]\!]m, \mathcal{C}_{\text{val}}[\![v_2]\!]m\rangle \\
& & \mathsf{in}\ \mathcal{C}_{\text{exp}}[\![e]\!](x \Rightarrow x'; m) \\
\mathcal{C}_{\text{exp}}[\![\mathsf{let}\ x = \mathsf{fst}\ v\ \mathsf{in}\ e]\!]m & = & \mathsf{let}\ x' = (\mathcal{C}_{\text{val}}[\![v]\!]m).0\ \mathsf{in}\ \mathcal{C}_{\text{exp}}[\![e]\!](x \Rightarrow x'; m) \\
\mathcal{C}_{\text{exp}}[\![\mathsf{let}\ x = \mathsf{snd}\ v\ \mathsf{in}\ e]\!]m & = & \mathsf{let}\ x' = (\mathcal{C}_{\text{val}}[\![v]\!]m).1\ \mathsf{in}\ \mathcal{C}_{\text{exp}}[\![e]\!](x \Rightarrow x'; m) \\
\mathcal{C}_{\text{exp}}[\![\mathsf{let}\ x = v_1\ p\ v_2\ \mathsf{in}\ e]\!]m & = & \mathsf{let}\ x' = \mathcal{C}_{\text{val}}[\![v_1]\!]m\ p\ \mathcal{C}_{\text{val}}[\![v_2]\!]m \\
& & \mathsf{in}\ \mathcal{C}_{\text{exp}}[\![e]\!](x \Rightarrow x'; m) \\
\mathcal{C}_{\text{exp}}[\![\mathsf{if0}\ v\ e_1\ e_2]\!]m & = & \mathsf{if0}\ (\mathcal{C}_{\text{val}}[\![v]\!]m)\ (\mathcal{C}_{\text{exp}}[\![e_1]\!]m)\ (\mathcal{C}_{\text{exp}}[\![e_2]\!]m) \\
\mathcal{C}_{\text{exp}}[\![\mathsf{halt}\ v]\!]m & = & \mathsf{halt}\ (\mathcal{C}_{\text{val}}[\![v]\!]m)
\end{array}$$

$$\mathit{lookup}\ (y \Rightarrow z; m)\ x = \begin{cases} z & \text{if } x = y; \\ \mathit{lookup}\ m\ x & \text{otherwise.} \end{cases}$$

Figure 6.7: Closure conversion over $\lambda_{\mathcal{K}}$.

6.3 Translation

The closure conversion of values ($\mathcal{C}_{\text{val}}[\![-]\!]m$) and expressions ($\mathcal{C}_{\text{exp}}[\![-]\!]m$), and the effect of closure conversion on types ($\mathcal{C}_{\text{type}}[\![-]\!]$), are shown in shown in Figure 6.7.

The type translation for continuations introduces an existential variable β that abstracts the type of the closure environment and pairs up the function (which is made to receive the environment as an extra argument) with the environment. The type translation is defined as a type family as follows:

$$
\begin{aligned}
&\textbf{type family } \mathit{Ctype}\ \ t \\
&\textbf{type instance } \mathit{Ctype}\ (\mathit{Cont}\ i\ t) = \mathit{CLOSURE}\ i\ t \\
&\textbf{type instance } \mathit{Ctype}\ (\mathit{Var}\ n) \quad = \mathit{Var}\ n \\
&\textbf{type instance } \mathit{Ctype}\ (t_1, t_2) \quad\ = \mathit{PAIR}\ (\mathit{Ctype}\ t_1)\ (\mathit{Ctype}\ t_2) \\
&\textbf{type instance } \mathit{Ctype}\ \mathit{Int} \qquad\ = \mathit{Int}
\end{aligned}
$$

where *PAIR* and *CLOSURE* are type synonyms used to abbreviate the pair (i.e. tuples of size 2) and closure types:

$$
\begin{aligned}
&\textbf{type } \mathit{PAIR}\ t_1\ t_2 = \mathit{Tup}\ (t_1, (t_2, ())) \\
&\textbf{type } \mathit{CLOSURE}\ k\ t = \mathit{Exists}\ (\mathit{PAIR}\ (\mathit{Cont}\ k\ (\mathit{PAIR}\ (U\ \ (S\ \ Z)\ k\ t)\ (\mathit{Var}\ k))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Var}\ Z))
\end{aligned}
$$

Note that, in the type of a closure, the update function must be applied in order to prevent free type variables from being captured by the existential quantifier. The types of the functions implementing the translations on values ($\mathcal{C}_{\text{val}}[\![-]\!]m$) and expressions ($\mathcal{C}_{\text{exp}}[\![-]\!]m$) are as follows:

$$
\begin{aligned}
\mathit{ccV} ::\ & \mathit{ValKb}\ (i, \mathit{ts})\ t \\
& \rightarrow (\forall \mathit{ts}'.\ \mathit{MapT}\ (\mathit{ValC}\ (i, \mathit{ts}'))\ (\mathit{Cenv}\ \mathit{ts}) \rightarrow \mathit{ValC}\ (i, \mathit{ts}')\ (\mathit{Ctype}\ t))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{ccE} ::\ & \mathit{ExpKb}\ (i, \mathit{ts}) \\
& \rightarrow (\forall \mathit{ts}'.\ \mathit{MapT}\ (\mathit{ValC}\ (i, \mathit{ts}'))\ (\mathit{Cenv}\ \mathit{ts}) \rightarrow \mathit{ExpC}\ (i, \mathit{ts}'))
\end{aligned}
$$

Informally, these types mean that the conversion takes a $\lambda_{\mathcal{K}}$ value (or expression) in context $\Delta;\Gamma$, to a $\lambda_{\mathcal{C}}$ value of the converted type (or an expression) in any context $\Delta;\Gamma'$, provided that the supplied map (m) takes every term variable in Γ to a value of the converted type in $\Delta;\Gamma'$. Formally this is captured by the following lemmas:

Lemma 6.1 *(CC type correspondence for values) If $\Delta; \Gamma \vdash_{\mathcal{K}} v : \tau$ and the map m satisfies*

$$\forall x \in dom(\Gamma).\ \Delta; \Gamma' \vdash_{\mathcal{C}} lookup\ m\ x : \mathcal{C}_{type}[\![\Gamma\ x]\!]$$

then

$$\Delta; \Gamma' \vdash_{\mathcal{C}} \mathcal{C}_{val}[\![v]\!]m : \mathcal{C}_{type}[\![\tau]\!].$$

Lemma 6.2 *(CC type correspondence for expressions) If $\Delta; \Gamma \vdash_{\mathcal{K}} e$ and the map m satisfies*

$$\forall x \in dom(\Gamma).\ \Delta; \Gamma' \vdash_{\mathcal{C}} lookup\ m\ x : \mathcal{C}_{type}[\![\Gamma\ x]\!]$$

then

$$\Delta; \Gamma' \vdash_{\mathcal{C}} \mathcal{C}_{val}[\![e]\!]m.$$

The details of how the map is represented and how it is constructed when closures are formed are explained in Section 6.5 below.

The translation shown in Figure 6.7 actually abuses the syntax of $\lambda_{\mathcal{C}}$, in the sense that the map m takes $\lambda_{\mathcal{K}}$ variables to projections of the environment ($y_i \Rightarrow env.i$), but projections are not $\lambda_{\mathcal{C}}$ values, and must actually be introduced by the let construct (let $x =$ $env.i$ in $\ldots$). It is intended that the body of a closure-converted function will actually bind the individual components of the environment to distinct variables using the let construct, and refer to these variables instead of projecting the environment. In the code, we found it easier to first map variables to projections of the environment, and then replace these explicit projections by variables bound by let; the concrete representation of $\lambda_{\mathcal{C}}$ contains a constructor for projections as values, but this constructor is only used internally by the closure conversion, and does not appear in the final result. In the code (cf. Appendix A, the file CC.hs, page 179), the function *openEnv* is used to replace projections as values to variable references.

6.4 Polymorphism

By the definition of $\mathcal{C}_{\text{exp}}[\![-]\!]m$, the function stored inside a closure is closed with respect to type variables: it is made to take an extra set of type variables $\vec{\beta}$ that are the original function's free type variables. When forming the closure, the closed function is passed the

free type variables, so as to get a closure of the expected type. The way this "forwarding" of type variables preserves types is captured by this simple lemma:

Lemma 6.3 *(forwarding) If $\vec{\beta} \subseteq \Delta$ and*

$$\Delta; \Gamma \vdash_C v : \forall \vec{\beta}, \vec{\alpha}.\ \tau \to 0$$

then

$$\Delta; \Gamma \vdash_C v[\vec{\beta}] : \forall \vec{\alpha}.\ \tau \to 0.$$

In our implementation, *all* type variables in scope are captured when forming a closure, rather than just those that actually appear free in the function (that is, we take $\vec{\beta} = \Delta$.) It would require additional data structures and type families to perform free type variable analysis, and afterward selectively abstract and apply those variables (and it is far from obvious that it could be done in a convincing way.) In contrast, capturing all the type variables can be done directly. Then, their application ($v[\vec{\beta}]$) is constructed by a function that implements the forwarding lemma:

$$
\begin{aligned}
\mathit{tpAppMulti} ::\ & \mathit{NatRep}\ i \to \mathit{NatRep}\ k \to \mathit{TypeRep}\ t \\
& \to \mathit{ValC}\ (i, \mathit{ts})\ (\mathit{Cont}\ (\mathit{Add}\ i\ k)\ t) \\
& \to \mathit{ValC}\ (i, \mathit{ts})\ (\mathit{Cont}\ k\ t)
\end{aligned}
$$

The proof of Lemma 6.3 is easy, and relies on the simple fact that, for any type τ and type variable α, $\tau[\alpha/\alpha] = \tau$. But this argument is not so easily demonstrated when types are expressed using de Bruijn indices. To illustrate the problem, consider the application of the lemma with these particular types:

$$
\begin{aligned}
\Delta = \vec{\beta} &= \beta_1, \beta_2 \\
\vec{\alpha} &= \alpha_1, \alpha_2 \\
\tau &= \langle \beta_1, \beta_2, \alpha_1, \alpha_2 \rangle
\end{aligned}
$$

so that the following judgments holds about v and its type instantiations:

$$
\begin{aligned}
\beta_1, \beta_2; \Gamma &\vdash_C v : \forall \beta_1, \beta_2, \alpha_1, \alpha_2.\ \langle \beta_1, \beta_2, \alpha_1, \alpha_2 \rangle \to 0 \\
\beta_1, \beta_2; \Gamma &\vdash_C v[\beta_1] : \forall \beta_2, \alpha_1, \alpha_2.\ \langle \beta_1, \beta_2, \alpha_1, \alpha_2 \rangle \to 0 \\
\beta_1, \beta_2; \Gamma &\vdash_C v[\beta_1, \beta_2] : \forall \alpha_1, \alpha_2.\ \langle \beta_1, \beta_2, \alpha_1, \alpha_2 \rangle \to 0
\end{aligned}
$$

If we translate these judgments to de Bruijn indices, they would look as follows:

$$2; \Gamma \vdash_C v : \forall^4.\ \langle t_3, t_2, t_1, t_0 \rangle \to 0$$
$$2; \Gamma \vdash_C v[t_1] : \forall^3.\ \langle t_4, t_2, t_1, t_0 \rangle \to 0$$
$$2; \Gamma \vdash_C v[t_1, t_0] : \forall^2.\ \langle t_3, t_2, t_1, t_0 \rangle \to 0$$

We have eliminated the type variable names ($\beta_1, \beta_2, \alpha_1, \alpha_2$), and replaced them with de Bruijn indices, written t_0, t_1, etc. Here, t_i is the type-level de Bruijn index that stands for the ith type variable in scope; that is, t_0 stands for the type variable bound by the closest $\forall$ binder, t_1 stands for one bound by the next closest $\forall$ binder, and so on.

Although the last judgment can be mapped directly onto the lemma, the middle one cannot, because the type variable which is bound outside of the $\forall$ quantifier temporarily assumes a different index (t_4). To capture the effect of the type applications of a number of type variables in scope, we define a type family as follows:

type family *MultiApp j c*
type instance *MultiApp* (*S j*) (*Cont* (*S k*) *t*) =
    MultiApp j (*Cont k* (*Subst t* (*Var j*) *k*))
type instance *MultiApp Z t* = *t*

The type *MultiApp j c* stands for the type c applied with a number of type variables identified by j (by repeatedly applying the type t_j and reducing j to $j - 1$). The type family satisfies the invariant that:

MultiApp i (*Cont* (*Add i k*) *t*) = *Cont k t*

It is then straightforward to implement Lemma 6.3 in terms of this type family. Note that the implementation of *tpAppMulti* is the only place in the compiler where a type family is needed locally, to prove a lemma.

Type application When translating the application of a polymorphic function, it takes a few manipulations to show that the function is of a type compatible with its supplied argument. As in the case of CPS conversion, we need to apply a lemma stating that substitution commute with the type translation:

Lemma 6.4 *($\mathcal{C}_{type}[\![-]\!]$–subst commute) For any $\lambda_{\mathcal{K}}$ types τ_1, τ_2 and index i,*

$$\mathcal{C}_{type}[\![\tau_1[\tau_2/i]]\!] = (\mathcal{C}_{type}[\![\tau_1]\!])[\mathcal{C}_{type}[\![\tau_2]\!]/i].$$

As the type translation explicitly shifts indices, we also need similar lemmas showing that $\mathcal{C}_{\text{type}}[\![-]\!]$, $U^i_k(-)$, and substitution commute pairwise (see source code, page 194). These lemmas are encoded as term-level functions, as in Section 4.4.

6.5 Auxiliary functions

In this section we give the type of the auxiliary functions *fvs* and *mkMap*. We first define the notion of type-preserving maps, which is used to aggregate values in the representation of $\lambda_{\mathcal{C}}$, and also to represent the result of the *fvs* function as well as the local variables map constructed by *mkMap* and passed to *ccV*/*ccE*.

Type-preserving maps Conceptually, a type-preserving map associates each component in a type environment with a value of the corresponding type. For a type environment $ts = (t_0, (t_1, \ldots (t_{n-1}, ())))$, a type-preserving map, of type *MapT c ts*, maps each component t_i of the environment to a value of type $c\ t_i$. The parameter c abstracts the type of the values stored in the map. For example, a type-safe evaluator over de Bruijn expressions might be given the type:

$eval :: MapT\ Value\ ts \rightarrow ExpS\ ts\ t \rightarrow Value\ t$

where the evaluation environment (*MapT Value ts*) maps each component t_i of the type environment *ts* (which correspond to a variable in scope) to a value of the corresponding type (of type $Value\ t_i$). When using *MapT* to represent the map m passed to *ccV*/*ccE*, c is instantiated to the type of $\lambda_{\mathcal{C}}$ values, so that source variables are mapped to the values in the target program used to access those variables.

A type-preserving map is represented as a list whose i^{th} component stores the value associated with the component t_i of the environment *ts*:

data *MapT c ts* **where**
  $M0 :: MapT\ c\ ()$
  $Ms :: c\ t \rightarrow MapT\ c\ ts \rightarrow MapT\ c\ (t, ts)$

The type *MapT* supports the usual functions over associative lists:

$$lookupT :: MapT\ c\ ts \to Index\ ts\ t \to c\ t$$
$$updateT :: MapT\ c\ ts \to Index\ ts\ t \to c\ t \to MapT\ c\ ts$$

6.5.1 Free variables

We defined the functions *fvsV* and *fvsE* which, given a $\lambda_{\mathcal{K}}$ value or expression, indicates whether each index in scope appears free in it. Its implementation produces its result in the type *MapT*:

$$fvsV :: ValKb\ (i, ts)\ t \to MapT\ BoolT\ ts$$
$$fvsE :: ExpKb\ (i, ts) \to MapT\ BoolT\ ts$$

where *BoolT* is a wrapper for the type *Bool* that has an extra type argument t that is simply ignored:

data *BoolT t* = *BoolT Bool*

In practice, it is necessary for *fvsV* and *fvsE* to actually examine the type context *ts*, and we have in fact:

$$fvsV :: EnvRep\ ts \to ValKb\ (i, ts)\ t \to MapT\ BoolT\ ts$$
$$fvsE :: EnvRep\ ts \to ExpKb\ (i, ts) \to MapT\ BoolT\ ts$$

where *EnvRep ts* reifies the type context *ts* as a Haskell value. Note that the parameter of type *EnvRep ts* is necessary because our maps are represented as lists; it could be avoided if we used a functional representation, such as this one:

type $MapT\ ts\ t = Index\ ts\ t \to c\ t$

but traversing the entire map would require generating all the indices of *ts*.

Our implementation make use of a small number of combinators. For example, the clause for variables is as follows:

$$fvsV\ ts\ (KBvar\ i) = updateT\ (falseMap\ ts)\ i\ (BoolT\ True)$$

where *falseMap* constructs a maps that takes every index to *False*:

$$falseMap :: EnvRep\ ts \to MapT\ BoolT\ ts$$

For cases which analyze constructs having multiple sub-expressions, we combine the result using a variant of the usual *zipWith* function on lists:

$$zipWithT :: (\forall t.\ c_1\ t \to c_2\ t \to c_3\ t) \to MapT\ c_1\ ts \to MapT\ c_2\ ts \to MapT\ c_3\ ts$$

which constructs a list by applying a binary operator point-wise on every element of two lists.

6.5.2 Construction of the variables map

The function *mkMap* in essence consumes the list of free variables and produces two results:

1. a local variables map, mapping each index in scope to a projection of the environment, and
2. a list of indices to be packed in the environment.

There is of course a direct connection between the two: the local variables map assumes a target context formed out of the environment being constructed. We can readily express this in types as follows:

$$\begin{aligned} mkMap :: MapT\ BoolT\ ts \to \exists env.\ (&MapT\ (Index\ env)\ (Cenv\ ts), \\ &MapT\ (Cenv\ ts)\ env) \end{aligned}$$

While this type captures the essence of what *mkMap* does, the index-mangling it performs creates slight complications. For one, the local variables map (m) and the environment ($j_0, \ldots, j_{n-1}$) grow in opposite directions as the recursion proceeds (cf. the case $mkMap\ (True : bs)\ j$). It takes a little extra machinery to track the way indices are appended to the environment. In terms of de Bruijn contexts, this means adding a binding "outside" a term, thus leaving intact an existing context where $i_0, \ldots, i_{n-1}$ are in scope while bringing into scope and extra index i_n. We handle such context extensions with the a type family that performs concatenation of contexts:

$$\begin{aligned} &\textbf{type family}\ Cat\ ts_0\ ts \\ &\textbf{type instance}\ Cat\ ()\ ts \quad\ \ = ts \\ &\textbf{type instance}\ Cat\ (s, ts_0)\ ts = (s, Cat\ ts_0\ ts) \end{aligned}$$

The actual construction of the variables map requires more bookkeeping that the type signature of *mkMap* exposes, and most of the work is accomplished by an auxiliary function of the following type:

$$
\begin{aligned}
\mathit{mkMapAux} :: \;& \mathit{NatRep}\ i \to \mathit{EnvRep}\ ts \to \mathit{EnvRep}\ env_0 \\
& \to \mathit{MapT}\ \mathit{BoolT}\ ts \\
& \to \mathit{MapT}\ (\mathit{Index}\ ts_0)\ ts \\
& \to (\exists env\ env'.\ (\mathit{Cat}\ env_0\ env \sim env') \Rightarrow \\
& \qquad (\mathit{EnvRep}\ env, \\
& \qquad\ \mathit{MapT}\ (\mathit{Index}\ (\mathit{Cenv}\ env')\ )\ (\mathit{Cenv}\ ts), \\
& \qquad\ \mathit{MapT}\ (\mathit{ValKb}\ (i, ts_0))\ env))
\end{aligned}
$$

where ts_0 is the de Bruijn context of the source term, *ts* is that part of the context which remains to be processed, env_0 is the part of the environment that has already been constructed, *env* is the segment of the environment which is constructed while processing the part of the context corresponding to *ts*, and env' is the completed environment, i.e. the concatenation of env_0 and *env*.

In the implementation of *mkMapAux*, we need to generate a new index in an existing context:

$$\mathit{newIndex} :: \mathit{EnvRep}\ ts \to \mathit{Index}\ (\mathit{Cat}\ ts\ (t, ()))\ t$$

We also need to interpret an existing index in a context that has been extended:

$$\mathit{weakenIndex} :: \mathit{EnvRep}\ ts \to \mathit{Index}\ ts_0\ t \to \mathit{Index}\ (\mathit{Cat}\ ts_0\ ts)\ t$$

This corresponds to "weakening" a typing judgment about the variable, as the added elements in the context represent extra assumptions.

Summary

The closure conversion phase turned functions with free variables into closed ones, by making the functions receive the value of their free variables through an extra parameter.

Closure conversion manipulates variable bindings extensively, and the mechanism used to control variable access in closure conversion is the most complex and delicate part of our compiler. We rely on an explicit variable map to relate variables in the source and

target programs, and a couple of auxiliary functions to identify which variables to put in the environment (*fvsV*/*fvsE*) and to construct a map accordingly (*mkMap*).

As in the case of CPS conversion, we needed to implement a lemma which states that substitution commutes with the type translation. But for the CPS conversion, the program representation was higher-order, so the type context of an expression was implicit and we thus did not need to prove anything about it. For the closure conversion, the type context was explicit in the representation, and that context was actually constructed explicitly when constructing the function inside a closure. In consequence, we also needed to apply a number of commutativity lemmas about the type families we applied on type contexts.

After closure conversion, the functions are closed but are still arbitrarily nested in the program, and the code tranformation presented in the next chapter will give the program a linear structure by moving all the functions to the top level.

Chapter 7

Hoisting

The hoisting transformation moves all the functions, which are closed as a result of closure conversion, to the top level. The resulting program is thus "linearized", i.e. it assumes a flat structures where functions are never nested, but instead refer to each other through top-level variables.

The hoisting transformation is a simple "code motion" phase: functions are simply moved around and replaced by variable references. Type preservation for this phase ought to be particularly obvious: a function appearing somewhere in the program is simply replaced by a variable which has the *same type* as the original function. Note that the hoisting transformation is the only one in our compiler which does not affect the object-level types (and does not introduce a type family.)

It in not uncommon in compilers to combine closure conversion and hoisting in a single phase. We preferred to implement them separately, to better single out the issues of the already intricate closure conversion, although there is no indication that a single phase would not work well in our setting.

The code listing of the encoding of the linearized language and the function hoisting transformation is reported in Appendix A (in the files LH.hs, page 198, and Hoist.hs, page 200).

$$
\begin{array}{lll}
(\textit{types}) & \tau & ::= \forall\alpha_0,\ldots,\alpha_{n-1}.\ \tau \to 0 \mid \exists\alpha.\ \tau \mid \alpha \mid \langle\tau_0,\ldots,\tau_{n-1}\rangle \\
 & & \quad \mid \mathsf{int} \\
(\textit{type context}) & \Delta & ::= \alpha_0,\ldots,\alpha_{n-1} \\
(\textit{value context}) & \Gamma & ::= x_0 : \tau_0,\ldots,x_{n-1} : \tau_{n-1} \\
(\textit{programs}) & p & ::= \mathsf{letrec}\ x_0 = c_0,\ldots,x_{n-1} = c_{n-1}\ \mathsf{in}\ e \\
(\textit{code blocks}) & c & ::= \mathsf{code}[\alpha_0,\ldots,\alpha_{n-1}](x : \tau).\ e \\
(\textit{values}) & v & ::= x \mid v[\tau] \mid \mathsf{pack}\ [v,\tau_1]\ \mathsf{as}\ \tau_2 \mid n \\
(\textit{exps}) & e & ::= \mathsf{let}\ x = v\ \mathsf{in}\ e \mid \mathsf{let}\ [\alpha,x] = \mathsf{unpack}\ v\ \mathsf{in}\ e \\
 & & \quad \mid \mathsf{let}\ x = \langle v_0,\ldots,v_{n-1}\rangle\ \mathsf{in}\ e \mid \mathsf{let}\ x = v.i\ \mathsf{in}\ e \\
 & & \quad \mid \mathsf{let}\ x = v_1\ p\ v_2\ \mathsf{in}\ e \mid v_1\ v_2 \mid \mathsf{if0}\ v\ e_1\ e_2 \mid \mathsf{halt}\ v \\
(\textit{primops}) & p & ::= + \mid - \mid \times
\end{array}
$$

Figure 7.1: Syntax of $\lambda_{\mathcal{H}}$.

7.1 Target language

The target language ($\lambda_{\mathcal{H}}$, shown in Figure 7.1) extends $\lambda_{\mathcal{C}}$ with a syntactic category of *programs*, containing the letrec construct, and eliminates the fix construct. The letrec construct ($\mathsf{letrec}\ x_0 = c_0,\ldots,x_{n-1} = c_{n-1}\ \mathsf{in}\ e_{main}$) introduces a number of variables bound to *code blocks*. A code block ($\mathsf{code}[\alpha_0,\ldots,\alpha_{n-1}](x : \tau).\ e$) is a top-level function that abstracts a number of type variables ($\alpha_0,\ldots,\alpha_{n-1}$) and one term variable (x) in its body (e). The scope of the variables introduced by letrec (namely $x_0,\ldots,x_{n-1}$) spans the body of all the code blocks plus the program body (e_{main}).

The static semantics is defined by three typing judgments:

$\vdash_{\mathcal{H}} p$	program p is well-typed
$\Delta;\Gamma \vdash_{\mathcal{H}} v : \tau$	value v has type τ in context $\Delta;\Gamma$
$\Delta;\Gamma \vdash_{\mathcal{H}} e$	expression e is well-typed in context $\Delta;\Gamma$

The context Γ gives the type of term variables, bound by either the global letrec construct or the let expressions. The context Δ lists the type variables in scope.

Strongly typed representation

The encoding of $\lambda_{\mathcal{H}}$ types is the same as for $\lambda_{\mathcal{C}}$ (cf. Section 6.2). The term representation is shown in Figure 7.2.

One notable difference with the representation of $\lambda_{\mathcal{C}}$ is that expressions and values have distinct contexts for globally bound variables and locally bound ones. A value v

```
data ProgramH where
  Hletrec :: MapT (CodeBlockH fs) fs → ExpH (fs, Z, ()) → ProgramH

data CodeBlockH fs t where
  Hblock :: TypeRep (Cont k t) → ExpH (fs, k, (t, ())) → CodeBlockH fs (Cont k t)

data ValH ctx t where
  Hvar    :: Index ts t                     → ValH (fs, i, ts) t
  Hlam    :: Index fs t                     → ValH (fs, i, ts) t

  HtpApp :: ValH (fs, i, ts) (Cont (S k) t1) → ValH (fs, i, ts) (Cont k (Subst t1 t2 k))
  Hpack   :: ValH (fs, i, ts) (Subst t1 t2 Z) → ValH (fs, i, ts) (Exists t1)
  Hnum    :: Int                            → ValH (fs, i, ts) Int

data ExpH ctx where
  Hlet      :: ValH (fs, i, ts) t → ExpH (i, (t, ts))          → ExpH (fs, i, ts)
  Hunpack   :: ValH (fs, i, ts) (Exists t) →
               ExpH (fs, S i, (t, ShiftEnv ts))                → ExpH (fs, i, ts)

  HletTup   :: MapT (ValH (fs, i, ts)) t →
               ExpH (fs, i, (Tup t, ts))                       → ExpH (fs, i, ts)
  HletProj  :: ValH (fs, i, ts) (Tup t1) → Index t1 t2 →
               ExpH (fs, i, (t2, ts))                          → ExpH (fs, i, ts)
  HletPrim  :: PrimOp →
               ValH (fs, i, ts) Int → ValH Int →
               ExpH (fs, i, (Int, ts))                         → ExpH (fs, i, ts)

  Happ      :: ValH (fs, i, ts) (Cont Z t) → ValH (fs, i, ts) t → ExpH (fs, i, ts)

  Hif0      :: ValH (fs, i, ts) Int →
               ExpH (fs, i, ts) → ExpH (fs, i, ts)             → ExpH (fs, i, ts)

  Hhalt     :: ValH (fs, i, ts) Int                            → ExpH (fs, i, ts)
```

Figure 7.2: Strongly typed representation of $\lambda_{\mathcal{H}}$ (cf. source file LH.hs, page 198).

satisfying $\Delta; \Gamma \vdash_{\mathcal{H}} v : \tau$ is represented by a term of type:

$$\mathit{ValH}\ (\mathit{fs}, i, \mathit{ts})\ t$$

and an expression e satisfying $\Delta; \Gamma \vdash_{\mathcal{H}} e$ is represented by a term of type

$$\mathit{ExpH}\ (\mathit{fs}, i, \mathit{ts})$$

where i encodes the length of Δ, *fs* encodes the part of Γ corresponding to global variables (i.e. those bound by letrec), *ts* encodes the part of Γ corresponding to local variables (i.e. those bound by the other let forms), and t encodes τ.[1]

Logically, the scope of the types variables which i accounts for is *ts*, as the types appearing in *fs* are closed. This is the reason why *ShiftEnv* must be applied to *ts* but not to *fs* in the type of *Hunpack*.

In the representation of values, two distinct constructors (*Hvar* and *Hlam*) introduce local and global variables (respectively), and the type associated with the variable in question is drawn from either the context *ts* or *fs* accordingly.

A program satisfying $\vdash_{\mathcal{H}} p$ is represented by a term of type *ProgramH*. This type has a single constructor (*Hletrec*) which aggregates a number of bindings using the type *MapT* (cf. Section 6.5), along with the program's main expression. An individual code block (of type *CodeBlockH fs* (*Cont k t*)) contains a type representative describing the object-level type of the continuation as well as the expression itself. Specifically, the type parameter *fs* reflects the type of every term bound by the top-level letrec; the parameter *fs* characterizes the type of the collection of code blocks, and also appears in the type of every individual code block so that code blocks can refer to each other.

Note that the type *CodeBlockH* is introduced so that we can collect the mutually recursive functions definitions with a direct application of *MapT*. Alternatively, we could define a specialized type that collects the functions and the required type representative (i.e. one that combines the effect of *CodeBlockH* and *MapT*), but the chosen solution appears simpler and more elegant.

[1] Note that we aggregate the components of the static context into a single type parameter instead of adding extra type parameters to *ValH*/*ExpH* following the convention from Chapter 3; see footnote on page 41.

values:

$$\begin{aligned}
&\mathit{collectV}\ m\ i\ x &&= (\mathit{lookup}\ m\ x, [])\\
&\mathit{collectV}\ m\ i\ (\mathsf{fix}\ f[\vec{\alpha}]\ x.\ e) &&= (x_i, [x_i = \mathsf{code}[\vec{\alpha}](x').\ e', b_{i+1}, \ldots, b_j])\\
&\quad \text{where } (e', [b_{i+1}, \ldots, b_j]) = \mathit{collectE}\ (m\{f \Rightarrow x_i, x \Rightarrow x'\})\ i\ e
\end{aligned}$$

expressions:

$$\begin{aligned}
&\mathit{collectE}\ m\ i\ (\mathsf{let}\ x = v\ \mathsf{in}\ e) = (\mathsf{let}\ x' = v'\ \mathsf{in}\ e', [b_i, \ldots, b_j, b_{j+1}, \ldots, b_k])\\
&\quad \text{where } (v', [b_i, \ldots, b_j]) = \mathit{collectV}\ m\ i\ v\\
&\qquad\quad (e', [b_{j+1}, \ldots, b_k]) = \mathit{collectE}\ (m\{x \Rightarrow x'\})\ (j+1)\ e\\
&\ldots
\end{aligned}$$

programs:

$$\begin{aligned}
&\mathit{hoist}\ e = \mathsf{letrec}\ b_0, \ldots, b_{n-1}\ \mathsf{in}\ e'\\
&\quad \text{where } (e', [e_0, \ldots, e_{n-1}]) = \mathit{collectE}\ \bullet\ 0\ e
\end{aligned}$$

Figure 7.3: Hoisting transformation (transforms $\lambda_{\mathcal{C}}$ into $\lambda_{\mathcal{H}}$).

7.2 Translation

The hoisting transformation is shown in Figure 7.3. The transformation proceeds by collecting every function into a bundle (whose type reflects the type of every function in it) and then assembles the program.

The auxiliary functions *collectV* and *collectE*, as the names imply, collect the functions contained in $\lambda_{\mathcal{C}}$ values and expressions. They are defined by equations of the form:

$$\mathit{collectV}\ m\ i\ v = (v', [b_i, \ldots, b_j])$$

The function receives a source value v (or expression), and returns a value v' (or expression) where all fix values have been replaced by variables, along with a set of bindings to be placed in the top-level letrec. The set of bindings ($[b_i, \ldots, b_j]$) is of the form:

$$\begin{aligned}
x_i &= \mathsf{code}[\vec{\alpha_i}](x).\ e_i,\\
&\vdots\\
x_j &= \mathsf{code}[\vec{\alpha_j}](x).\ e_j
\end{aligned}$$

The parameter i is used to control the assignment of fresh variables to code blocks: i is

the smallest number such that x_i has not already been assigned. In the implementation, these bindings are identified using indices: x_i will be represented by the ith index bound in the global context (*fs*).

The parameter m is a map from source variables to variables in the target program. In particular, it maps variables used to make recursive calls in the body of a fix value to variable bound by letrec, and maps all other variables to local variables. Note that the parameter m would not be needed if the language did not have term-level recursion: the hoisting transformation does not affect variables except those that are used for recursive calls (and we did not have it in our original presentation of closure conversion (Guillemette and Monnier 2007)).

7.3 Implementation

The types of the functions that implement *collectV* and *collectE* are as follows:

$$\begin{aligned}
\mathit{collectV} ::\ & \mathit{MapT}\ (\mathit{ValH}\ (\mathit{fs}_0, i, \mathit{ts}))\ \mathit{ts} \rightarrow \mathit{EnvRep}\ \mathit{fs}_0 \\
& \rightarrow \mathit{ValC}\ (i, \mathit{ts})\ t \\
& \rightarrow \exists \mathit{fs}.\ (\mathit{ValH}\ (\mathit{Cat}\ \mathit{fs}_0\ \mathit{fs}, i, \mathit{ts})\ t, \\
& \qquad \mathit{MapT}\ (\mathit{CodeBlockH}\ (\mathit{Cat}\ \mathit{fs}_0\ \mathit{fs})\ )\ \mathit{fs})
\end{aligned}$$

$$\begin{aligned}
\mathit{collectE} ::\ & \mathit{MapT}\ (\mathit{ValH}\ (\mathit{fs}_0, i, \mathit{ts}))\ \mathit{ts} \rightarrow \mathit{EnvRep}\ \mathit{fs}_0 \\
& \rightarrow \mathit{ExpC}\ (i, \mathit{ts}) \\
& \rightarrow \exists \mathit{fs}.\ (\mathit{ExpH}\ (\mathit{Cat}\ \mathit{fs}_0\ \mathit{fs}, i, \mathit{ts}), \\
& \qquad \mathit{MapT}\ (\mathit{CodeBlockH}\ (\mathit{Cat}\ \mathit{fs}_0\ \mathit{fs})\ )\ \mathit{fs})
\end{aligned}$$

The type parameter fs_0 characterizes the functions already collected and turned into code blocks; the existentially quantified type parameter *fs* characterizes those that are produced by traversing the current value or expression. Thus the second parameter to the function (of type $\mathit{EnvRep}\ \mathit{fs}_0$) serves the purpose of the parameter i in the "formal" definition of *collectV* / *collectE*.

The implementation of the two functions is straightforward, but it involves much low-level manipulation of de Bruijn indices. We often need to combine sets of code blocks, and this involves weakening the expressions contained in one of the code blocks to account

for the bindings contained in the other. This employs a “weakening” function on values and expressions:

$$weakenVal :: EnvRep\ fs \rightarrow ValH\ (fs_0, i, ts)\ t \rightarrow ValH\ (Cat\ fs_0\ fs, i, ts)\ t$$
$$weakenExp :: EnvRep\ fs \rightarrow ExpH\ (fs_0, i, ts) \rightarrow ExpH\ (Cat\ fs_0\ fs, i, ts)$$

These functions traverse the source term and eventually perform weakening on all de Bruijn indices which stand for global variables (that is, those introduced by *Hlam*) appearing in them.

Value abstraction When traversing a binder which abstracts a value (e.g. *Hlet*), the local context (ts) is extended, so the elements of the map must be shifted to account for the new binder. That is, we must take the map of type:

$$MapT\ (ValH\ (fs_0, i, ts))\ ts$$

to a map of type:

$$MapT\ (ValH\ (fs_0, i, (t, ts)))\ (t, ts)$$

This involves “shifting” the individual values in the map:

$$shiftVal :: ValH\ (fs, i, ts)\ t \rightarrow ValH\ (fs, i, (t', ts))\ t$$

which means incrementing the indices introduced by *Hvar*. Note that, as the map takes variables to variables, the function *shiftVal* only needs to handle the cases of *Hlam* and *Hvar*, and can safely omit the cases for other constructors of *ValH*.

Type abstraction When traversing a binder for a type variable (in the case of *Hunpack*), we must take a map of type:

$$MapT\ (ValH\ (fs, i, ts_0))\ ts$$

to a map of type:

$$MapT\ (ValH\ (fs, S\ i, ShiftEnv\ ts_0))\ (t, ShiftEnv\ ts)$$

This involves re-interpreting the values in the map at a different type, where an extra variable is in scope:

$$tpShiftValH :: ValH\ (fs, i, ts)\ t \rightarrow ValH\ (fs, i, ShiftEnv\ ts)\ (Shift\ s)$$

This function in turn must re-interpret indices introduced by *Hvar*:

$$tpShiftIndex :: Index\ ts\ t \rightarrow Index\ (ShiftEnv\ ts)\ (Shift\ t)$$

The functions *tpShiftValH* and *tpShiftIndex* do not actually modify values or indices, but merely assign different types to them.

Summary

The hoisting phase simply flattens the structure of the closure-converted code, and is by far the most conceptually simple of the code transformations in our compiler. But due to its existentially quantified return type, its implementation is less compact than that of the CPS or closure conversion (excluding its auxiliary functions). This transformation does not affect the types, so it does not need to introduce a type family or any associated lemmas.

The function hoisting phase is actually the last tranform that is specific to functional languages. After the function hoisting phase, the code has lost much of its functional flavour, and all of it higher-order features, as a program then takes the form of a collection of (mutually recursive) closed functions in CPS, ripe for generating assembly code.

Chapter 8

Code generation

This chapter presents the final stage of compilation which generates code in a typed assembly language. The typed assembly language models the machine language of a reduced instruction set computer (RISC).

Indeed, any compiler for a real machine must perform some form of code generation. Unlike CPS or closure conversion, the code generation phase is not specific to the compilation of functional languages. We have implemented this phase to demonstrate that the techniques we have developed can indeed be applied to all the code transformations in a compiler. The implementation did not reveal notable technical difficulties; for the most part, it is an application of the implementation techniques from the previous chapters (mainly Chapter 6 and 7).

In a typical compiler, the target language is untyped; all the type information about the program is discarded at some point before assembly language is produced. In contrast, the code generated by our compiler follows a type discipline and carries type information, and can thus be seen as a form *proof-carrying code*. Proof-carrying code (PCC) (Necula and Lee 1996) is a general technique for safely executing code of untrusted source, by locally checking a proof of safety which is distributed along with the code. The code produced by our compiler contains enough type information so that it can be easily type-checked. The verifications guarantees that the program will not “go wrong” at run time, by doing an illegal instruction or supplying inappropriate values as operands.

The code generation phase in a typical compiler performs instruction selection and register allocation, and can also perform optimizations on the generated assembly code.

In our implementation, we make the simplifying assumption that an infinite supply of registers is available. Thus, our register allocation does not have to deal with the case where too few registers are available to store the variables which are active at the same time (which is normally done by "spilling" some of the variables, i.e. storing their values on the heap temporarily.) The code generation phase presented here is thus admittedly fairly simple compared to that found in a typical optimizing compiler for a real machine.

The main conceptual difference between an assembly language and the functional calculi used to this point is that program meaning is sensitive to the actual registers and code labels that appear in the program. In a functional program, changing the name of a bound variable will not affect the meaning of the program. For example, the terms $\lambda\ x.\ x$ and $\lambda\ y.\ y$ are equivalent – one can be substituted for the other in any context without altering the program's meaning. Whereas every variable in a functional program has a well-defined scope and type, registers behave like global variables which assume different types in different regions of the program.

The type system of our target assembly language tracks the type of the registers at every point of the program. It also associates types to the code labels: when control is transferred (by a "jump" instruction), the type system guarantees that the registers contain values of the types expected by the target code block. The program representation is different from those in the previous chapters, in that it does not use local scopes; instead the same register names are re-used (see the type family *Update* defined in Section 8.1.)

The code generation presented here is largely based on the one from Morrisett et al. (1999). A notable difference is that they use an abstract machine instruction for tuple allocation (and a separate instruction for initialization), whereas we use a coarser abstraction which performs tuple creation in one step. Their compilation scheme actually includes an explicit allocation phase, in between function hoisting and code generation, which turns tuple creations into a sequence of operations that allocates a tuple in memory and initializes its components. We decided to use an atomic tuple creation construct in our typed assembly language so that we would not need to implement this allocation phase (as the insight gained by doing so would likely not justify the extra complexity.) Also, a minor difference with their presentation is that we use explicit parameters to control the assignment of fresh identifiers (for code labels and registers) in order to better

(*types*)	τ ::= $\forall \alpha_0, \ldots, \alpha_{m-1}.\ (r_0 : \tau_0,\ \ldots,\ r_{n-1} : \tau_{n-1}) \to 0 \mid \exists \alpha.\ \tau$
	$\mid \alpha \mid \langle \tau_0, \ldots, \tau_{n-1} \rangle \mid \mathsf{int}$
(*code seg. types*)	$\Psi ::= \ell_0 : \tau_0;\ \ldots;\ \ell_{n-1} : \tau_{n-1}$
(*type context*)	$\Delta ::= \alpha_0, \ldots, \alpha_{n-1}$
(*register file type*)	$\Gamma ::= r_0 : \tau_0;\ \ldots;\ r_{n-1} : \tau_{n-1}$
(*programs*)	$p ::= \ell_0 \to c_0;\ \ldots;\ \ell_{n-1} \to c_{n-1};\ \mathit{start} \to I$
(*code blocks*)	$c ::= \mathsf{code}[\alpha_0, \ldots, \alpha_{m-1}](r_0 : \tau_0,\ \ldots,\ r_{n-1} : \tau_{n-1}).\ I$
(*values*)	$v ::= r \mid \ell \mid n \mid v[\tau] \mid \mathsf{pack}\ [v, \tau_1]\ \mathsf{as}\ \tau_2$
(*instructions*)	$\iota ::= \mathsf{add}\ r_d\ r_s\ v \mid \mathsf{sub}\ r_d\ r_s\ v \mid \mathsf{mul}\ r_d\ r_s\ v \mid \mathsf{bnz}\ r\ v$
	$\mid \mathsf{mov}\ r_d\ r_s\ v \mid \mathsf{unpack}\ [\alpha, r_d]\ v$
	$\mid \mathsf{mktuple}\ r_d\ \langle v_0, \ldots, v_{n-1} \rangle \mid \mathsf{ld}\ r_d\ r_s[i]$
(*instr. seq.*)	$I ::= \iota; I \mid \mathsf{jmp}\ v \mid \mathsf{halt}$

Figure 8.1: Syntax of TAL.

reflect our implementation.

The code listing of the encoding of the typed assembly language and the code generation is reported in Appendix A (in the files TAL.hs, page 208, and CG.hs, page 210).

8.1 Typed assembly language

The syntax of TAL is shown in Figure 8.1. A TAL program consists of a set of code blocks identified by labels, along with an instruction sequence identified with the designated label *start*, which corresponds to the entry point of the program. Each code block assumes a number of type variables to be instantiated and the contents of the first n registers to have specific types. A code block then consists of sequence of instructions terminated by either an unconditional jump or the halt instruction, which terminates the program.

Instruction operands are registers and values. A value specifies a register, code label or integer literal. A value can also be a type application or the construction of an existential package.

The instruction set contains the usual instructions for arithmetic, a conditional jump, a move instruction, as well as an unpack pseudo-instruction (which opens up an existential package and loads its content in a register). There is also an abstract instruction to construct a tuple on the heap. The values of the tuple components are specified as individual operands. A “load” instruction loads a specified component of a tuple on the heap into a register. This instruction set is sufficient to execute our programs since our

source language is pure (i.e. has no side-effects) and assuming an unlimited amount of heap memory. A realistic implementation would need a garbage collector, which would require additional instructions for update and deallocation of tuples on the heap.

The static semantics of TAL is defined by three typing judgments:

$\vdash_{\mathcal{T}} p$	program p is well-typed
$\Psi; \Delta; \Gamma \vdash_{\mathcal{T}} v : \tau$	value v has type τ in context $\Psi; \Delta; \Gamma$
$\Psi; \Delta; \Gamma \vdash_{\mathcal{T}} I$	instruction sequence I is well-typed in context $\Psi; \Delta; \Gamma$

The judgments on values and instruction sequences refer to a context Ψ that lists the types of the code labels in the program. The context Δ lists the type variables in scope, and Γ lists the types of the first n registers at the point in the program where the instruction sequence appears.

8.2 Translation

Code generation is formally specified in Figure 8.2 and 8.3. The type translation ($\mathcal{T}_{type}[\![-]\!]$) leaves the types mostly unchanged, except that it maps a polymorphic function to a polymorphic code block which receives its argument through the register $\mathbf{r}_0$.

The parameter γ of the value translation ($\mathcal{T}^{\gamma}_{val}[\![-]\!]$) is used for variable access. Specifically, γ maps $\lambda_{\mathcal{H}}$ global variables (bound by letrec) to code labels (ℓ_i), and local variables to registers.

The core part of the code generation is the expression translation, shown in Figure 8.3. It is defined by equations of the form:

$$\mathcal{T}^{\gamma,\Delta,\Gamma,i}_{exp}[\![e]\!] = \langle C, I \rangle$$

From an expression e, it produces an instruction sequence I, along with a set of labeled code blocks C. This set of code blocks is actually needed for the case of if0, for which the code generation must create an extra code block. The parameter γ is passed to the value translation as needed, and gets extended as new variable bindings are encountered (when generating code for the let constructs.) The parameter Δ lists the type variables in scope. The parameter Γ lists the types of the first n registers in the context where I is to appear. Both Δ and and Γ are used when a new code block is created (in the case

programs:

$$
\begin{array}{rcl}
\mathcal{T}_{prog}[\![\textsf{letrec } x_0 = c_0, \ldots, x_{n-1} = c_{n-1} \textsf{ in } e]\!] & = & \ell_0 \rightarrow \textsf{code}[\vec{\alpha_0}](r_0 : \mathcal{T}_{type}[\![\tau_0]\!]).\ I_0 \\
 & & \vdots \\
 & & \ell_{n-1} \rightarrow \textsf{code}[\vec{\alpha_{n-1}}](r_{n-1} : \mathcal{T}_{type}[\![\tau_{n-1}]\!]).\ I_{n-1} \\
 & & C_0; \ldots; C_{n-1}; C_e; \\
 & & start \rightarrow I_e \\
 & & \text{where } \langle C_i, I_i \rangle = \mathcal{T}_{exp}^{\gamma_0, \vec{\alpha_i}, \{r_0 : \mathcal{T}_{type}[\![\tau_i]\!]\}, \ell_{j[i]}}[\![e_i]\!] \\
 & & \qquad \langle C_e, I_e \rangle = \mathcal{T}_{exp}^{\gamma_0, \cdot, \cdot, \ell_{j[n]}}[\![e]\!] \\
 & & \qquad \textsf{code}[\vec{\alpha_i}](x : \tau_i).\ e_i = c_i \\
 & & \qquad \gamma_0 = x_0 \rightarrow \ell_0, \ldots, x_{n-1} \rightarrow \ell_{n-1} \\
 & & \qquad j[i] = n + |C_0| + \cdots + |C_{i-1}|
\end{array}
$$

types:

$$
\begin{array}{rcl}
\mathcal{T}_{type}[\![\forall \alpha_0, \ldots, \alpha_{n-1}.\ \tau \rightarrow 0]\!] & = & \forall \alpha_0, \ldots, \alpha_{n-1}.\ (r_0 : \mathcal{T}_{type}[\![\tau]\!]) \rightarrow 0 \\
\mathcal{T}_{type}[\![\exists \alpha.\ \tau]\!] & = & \exists \alpha.\ \mathcal{T}_{type}[\![\tau]\!] \\
\mathcal{T}_{type}[\![\alpha]\!] & = & \alpha \\
\mathcal{T}_{type}[\![\langle \tau_0, \ldots, \tau_{n-1} \rangle]\!] & = & \langle \mathcal{T}_{type}[\![\tau_0]\!], \ldots, \mathcal{T}_{type}[\![\tau_{n-1}]\!] \rangle \\
\mathcal{T}_{type}[\![\textsf{int}]\!] & = & \textsf{int}
\end{array}
$$

values:

$$
\begin{array}{rcl}
\mathcal{T}_{val}^{\gamma}[\![x]\!] & = & \gamma(x) \\
\mathcal{T}_{val}^{\gamma}[\![n]\!] & = & n \\
\mathcal{T}_{val}^{\gamma}[\![v[\tau]]\!] & = & \mathcal{T}_{val}^{\gamma}[\![v]\!][\mathcal{T}_{type}[\![\tau]\!]] \\
\mathcal{T}_{val}^{\gamma}[\![\textsf{pack } [v, \tau_1] \textsf{ as } \tau_2]\!] & = & \textsf{pack } [\mathcal{T}_{val}^{\gamma}[\![v]\!], \mathcal{T}_{type}[\![\tau_1]\!]] \textsf{ as } \mathcal{T}_{type}[\![\tau_2]\!]
\end{array}
$$

Figure 8.2: Translation of programs, types, and values to TAL(cf. source file TAL.hs, page 208).

$$
\begin{array}{rcl}
\mathcal{T}_{exp}^{\gamma,\Delta,\Gamma,i}[\![\mathsf{let}\ x:\tau = v\ \mathsf{in}\ e]\!] & = & \langle C, \mathsf{mov}\ r\ \mathcal{T}_{val}^{\gamma}[\![v]\!]\ ; I\rangle \\
& & \text{where } \langle C, I\rangle = \mathcal{T}_{exp}^{\gamma\{x\to r\},\Delta,\Gamma\{r:\mathcal{T}_{type}[\![\tau]\!]\},i}[\![e]\!] \\
& & \qquad r \notin dom\ \Gamma \\
\mathcal{T}_{exp}^{\gamma,\Delta,\Gamma,i}[\![\mathsf{let}\ [\alpha,x] = \mathsf{unpack}\ v^{\exists\alpha.\ \tau}\ \mathsf{in}\ e]\!] & = & \langle C, (\mathsf{unpack}\ [\alpha, r]\ \mathcal{T}_{val}^{\gamma}[\![v]\!]; I)\\
& & \text{where } \langle C, I\rangle = \mathcal{T}_{exp}^{\gamma\{x\to r\},\Delta\{\alpha\},\Gamma\{r:\mathcal{T}_{type}[\![\tau]\!]\},\ell}[\![e]\!] \\
& & \qquad \alpha \notin \Delta, r \notin dom\ \Gamma \\
\mathcal{T}_{exp}^{\gamma,\Delta,\Gamma,i}[\![\mathsf{let}\ x = \langle v_0,\ldots,v_{n-1}\rangle^{\tau}\ \mathsf{in}\ e]\!] & = & \langle C, (\mathsf{mktuple}\ r\ \langle \mathcal{T}_{val}^{\gamma}[\![v_0]\!],\ldots,\mathcal{T}_{val}^{\gamma}[\![v_{n-1}]\!]\rangle; \\
& & \quad I)\rangle \\
& & \text{where } \langle C, I\rangle = \mathcal{T}_{exp}^{\gamma\{x\to r\},\Delta,\Gamma\{r:\mathcal{T}_{type}[\![\tau]\!]\},i}[\![e]\!] \\
& & \qquad r \notin dom\ \Gamma \\
\mathcal{T}_{exp}^{\gamma,\Delta,\Gamma,i}[\![\mathsf{let}\ x:\tau = v.i\ \mathsf{in}\ ]\!] & = & \langle C, (\mathsf{mov}\ r\ \mathcal{T}_{val}^{\gamma}[\![v]\!]\ ; \\
& & \quad \mathsf{ld}\ r\ r[i]; \\
& & \quad I)\rangle \\
& & \text{where } \langle C, I\rangle = \mathcal{T}_{exp}^{\gamma\{x\to r\},\Delta,\Gamma\{r:\mathcal{T}_{type}[\![\tau]\!]\},i}[\![e]\!] \\
& & \qquad r \notin dom\ \Gamma \\
\mathcal{T}_{exp}^{\gamma,\Delta,\Gamma,i}[\![\mathsf{let}\ x = v_1 p v_2\ \mathsf{in}\ e]\!] & = & \langle C, (\mathsf{mov}\ r\ \mathcal{T}_{val}^{\gamma}[\![v_1]\!]\ ; \\
& & \quad arith_p\ r\ r\ \mathcal{T}_{val}^{\gamma}[\![v_2]\!]; \\
& & \quad I)\rangle \\
& & \text{where } \langle C, I\rangle = \mathcal{T}_{exp}^{\gamma\{x\to r\},\Delta,\Gamma\{r:\mathsf{int}\},i}[\![e]\!] \\
& & \qquad arith_{+} = \mathsf{add} \\
& & \qquad arith_{-} = \mathsf{sub} \\
& & \qquad arith_{\times} = \mathsf{mul} \\
& & \qquad r \notin dom\ \Gamma \\
\mathcal{T}_{exp}^{\gamma,\Delta,\Gamma,i}[\![v_1\ v_2]\!] & = & \langle C, (\mathsf{mov}\ r\ \mathcal{T}_{val}^{\gamma}[\![v_1]\!]\ ; \\
& & \quad \mathsf{mov}\ r_0\ \mathcal{T}_{val}^{\gamma}[\![v_2]\!]\ ; \\
& & \quad \mathsf{jmp}\ r)\rangle \\
& & \text{where } r \neq r_0, r \notin dom\ \Gamma \\
\mathcal{T}_{exp}^{\gamma,\Delta,\Gamma,i}[\![\mathsf{if0}\ v\ e_1\ e_2]\!] & = & \langle C_1 C_2\{\ell \to c\}, (\mathsf{mov}\ r\ \mathcal{T}_{val}^{\gamma}[\![v]\!]\ ; \\
& & \qquad\qquad \mathsf{bnz}\ r\ \ell'[\Delta]; I')\rangle \\
& & \text{where } \langle C_1, I_1\rangle = \mathcal{T}_{exp}^{\gamma,\Delta,\Gamma,i}[\![e_1]\!] \\
& & \qquad \langle C_2, I_2\rangle = \mathcal{T}_{exp}^{\gamma,\Delta,\Gamma,\ell+|C_1|}[\![e_2]\!] \\
& & \qquad c = \mathsf{code}[\Delta](\Gamma).\ I_2 \\
& & \qquad \ell' = \ell + |C_1| + |C_2| \\
& & \qquad r \notin dom\ \Gamma \\
\mathcal{T}_{exp}^{\gamma,\Delta,\Gamma,i}[\![\mathsf{halt}\ v]\!] & = & \langle \emptyset, \mathsf{mov}\ r_0\ \mathcal{T}_{val}^{\gamma}[\![v]\!]\ ; \\
& & \quad \mathsf{halt}\rangle
\end{array}
$$

Figure 8.3: Translation of expressions to TAL.

of if0). Finally, the parameter ℓ is used to control "freshness" of code labels: it identifies the smallest unassigned code label. Incidentally, Γ is also used for the same purpose, i.e. controlling the freshness of registers.

Finally, the program translation $\mathcal{T}_{prog}[\![-]\!]$ assembles code blocks for the individual entries in the global letrec, as well as the additional code blocks generated. The generated code blocks corresponding to n code blocks in the $\lambda_{\mathcal{H}}$ program are labeled $\ell_0, \ell_1, \ldots, \ell_{n-1}$. The assignment of labels to extra code blocks is controlled by the variable $j[i]$. Specifically, the assignment of labels to extra code blocks for the ith binding of the letrec begin with label $\ell_{j[i]}$; the value of $j[i]$ depends on the number of code blocks generated for the previous bindings.

8.3 Type preservation

The high-level theorem of type preservation for code generation states that $\mathcal{T}_{prog}[\![-]\!]$ takes well-typed $\lambda_{\mathcal{H}}$ programs to well-typed TAL programs:

Theorem 8.1 *For any $\lambda_{\mathcal{H}}$ program p, if $\vdash_{\mathcal{H}} p$, then $\vdash_{\mathcal{T}} \mathcal{T}_{prog}[\![p]\!]$.*

The proof of the above theorem relies on the two auxiliary lemmas establishing that the value and expression translations preserve types. In the case of values, the lemma states that $\mathcal{T}^{\gamma}_{val}[\![-]\!]$ takes well-typed values to well-types values, provided that the variable map (γ) takes variables to registers or code labels of the corresponding types:

Lemma 8.1 *For any $\lambda_{\mathcal{H}}$ value v, if*

1. $\Delta; \Gamma \vdash_{\mathcal{H}} v : \tau$, *and*

2. $\forall x \in dom\ \Gamma$. *if* $\Gamma(x) = \tau'$, *then either*

   – $\Gamma'(\gamma(x)) = \mathcal{T}_{type}[\![\tau']\!]$ *or*

   – $\Psi(\gamma(x)) = \mathcal{T}_{type}[\![\tau']\!]$

then $\Psi; \Delta; \Gamma' \vdash_{\mathcal{T}} \mathcal{T}^{\gamma}_{val}[\![v]\!] : \mathcal{T}_{type}[\![\tau]\!]$.

In the case of expressions, the lemma similarly states that a well-typed expression is produced, and that all the extra code blocks are well-typed and associated to contiguous labels.

$$
\begin{array}{l}
\textbf{data}\ \mathit{ProgramT}\ \textbf{where} \\
\quad \mathit{Tletrec} :: \mathit{MapT}\ (\mathit{CodeBlockT}\ \mathit{cs})\ \mathit{cs} \rightarrow \mathit{Instr}\ \mathit{cs}\ (\mathit{Code}\ Z\ ()) \rightarrow \mathit{ProgramT}
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ \mathit{CodeBlockT}\ \mathit{cst}\ \textbf{where} \\
\quad \mathit{Tblock} :: \mathit{TypeRep}\ (\mathit{Code}\ k\ \mathit{rs}) \rightarrow \\
\qquad\qquad \mathit{Instr}\ \mathit{cs}\ (\mathit{Code}\ k\ \mathit{rs}) \quad \rightarrow \mathit{CodeBlockH}\ \mathit{cs}\ (\mathit{Code}\ k\ \mathit{rs})
\end{array}
$$

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{data}\ \mathit{ValT}\ g\ t\ \textbf{where}} \\
\quad \mathit{Treg} & :: \mathit{Index}\ \mathit{rs}\ t & \rightarrow & \mathit{ValT}\ (\mathit{cs}, i, \mathit{rs})\ t \\
\quad \mathit{Tlabel} & :: \mathit{Index}\ \mathit{cs}\ t & \rightarrow & \mathit{ValT}\ (\mathit{cs}, i, \mathit{rs})\ t \\
\quad \mathit{TtpApp} & :: \mathit{ValT}\ (\mathit{cs}, i, \mathit{rs})\ (\mathit{Code}\ (S\ k)\ t_1) & \rightarrow & \mathit{ValT}\ (\mathit{cs}, i, \mathit{rs}) \\
 & & & \quad (\mathit{Code}\ k\ (\mathit{SubstEnv}\ t_1\ t_2\ k)) \\
\quad \mathit{Tpack} & :: \mathit{ValT}\ (\mathit{cs}, i, \mathit{rs})\ (\mathit{Subst}\ t_1\ t_2\ Z) & \rightarrow & \mathit{ValT}\ (\mathit{cs}, i, \mathit{rs})\ (\mathit{Exists}\ t_1) \\
\quad \mathit{Tnum} & :: \mathit{Int} & \rightarrow & \mathit{ValT}\ (\mathit{cs}, i, \mathit{rs})\ \mathit{Int}
\end{array}
$$

Figure 8.4: Encoding of TAL programs, code blocks and values.

Lemma 8.2 *If a $\lambda_{\mathcal{H}}$ expression e satisfies*

1. $\Delta; \Gamma \vdash_{\mathcal{H}} e$,
2. $\forall x \in dom\ \Gamma.$ *if* $\Gamma(x) = \tau'$, *then either*
   - $\Gamma'(\gamma(x)) = \mathcal{T}_{type}[\![\tau']\!]$ *or*
   - $\Psi(\gamma(x)) = \mathcal{T}_{type}[\![\tau']\!]$,

and $\langle C, I\rangle = \mathcal{T}_{exp}^{\gamma,\Delta,\Gamma,i}[\![e]\!]$, *then*

1. $\Psi; \Delta; \Gamma \vdash_{\mathcal{T}} I$,
2. $dom\ C = \ell_i, \ldots, \ell_{i+|C|}$
3. $\forall(\ell_j \rightarrow \mathsf{code}[\Delta'](\Gamma').\ I') \in C.\ \Psi; \Delta'; \Gamma' \vdash_{\mathcal{T}} I'$.

8.4 Implementation

This section shows the details of the concrete representation of TAL as well as the types of the main functions that implement code generation. The encoding of TAL is somewhat more elaborate than those of the intermediate languages. The extra complexity stems from the fact that registers have different types in different parts of the program.

The types of TAL are encoded in essentially the same way as those of $\lambda_{\mathcal{H}}$, except for the case of polymorphic code blocks. Specifically, the type of a code block,

$$\forall \alpha_0, \ldots, \alpha_{m-1}.\ (r_0 : \tau_0,\ \ldots,\ r_{n-1} : \tau_{n-1}) \rightarrow 0$$

is represented using the Haskell type:

$$\mathit{Code}\ n\ (t_0, (t_1, \ldots (t_{n-1}, ()) \ldots))$$

A value v satisfying $\Psi; \Delta; \Gamma \vdash_{\mathcal{T}} v : \tau$ is represented as a term of type:

$$\mathit{ValT}\ (cs, i, rs)\ t$$

and an instruction sequence satisfying $\Psi; \Delta; \Gamma \vdash_{\mathcal{T}} I$ is represented as a term of type:

$$\mathit{InstrT}\ cs\ (\mathit{Code}\ i\ rs)$$

where cs encodes Ψ, i encodes Δ, rs encodes Γ and t encodes τ.

In concrete terms, the type parameter cs reflects the type of all the code blocks in the program. It has the form $(t_0, (t_1, \ldots (t_{n-1}, ()) \ldots))$, where t_i gives the type of the code block associated to label ℓ_i. The parameter rs gives the types of the registers in the context where a value or expressions appears; it is of the same form as cs, with t_i giving the type associated to register $\mathbf{r}_i$.

A program satisfying $\vdash_{\mathcal{T}} p$ is represented as a term of type *ProgramT*. The encoding of TAL programs and values is shown in Figure 8.4. The representation of programs follows the same scheme as in the case of $\lambda_{\mathcal{H}}$ (cf. Section 7.1). A program consists of a set of code blocks, plus an instruction sequence corresponding to the program entry point. An individual code block (of type *CodeBlockT*) consists of a type representative of describing the type of the code block, and an instruction sequence. The code blocks are aggregated using the *MapT* type.

Instructions We do not use distinct types for instructions (ι) and instruction sequences (I). Instead, we use a single type for encoding instructions sequences, analogous to the

representation of expressions in CPS. The encoding of instruction sequences is shown in Figure 8.5. Note that every constructor except *JMP* and *HALT* has an instruction sequence as its last argument, which represents the remaining instructions in the sequence, and thus plays the role of a continuation.

Registers used as source operands are represented using typed indices (*Index rs t*, the same structure used for de Bruijn indices). Registers used as targets are represented as natural numbers with singleton types (*NatRep d*); typed indices are not used because the type of the corresponding register before the instruction is irrelevant. Updates to the register file are captured by a type family defined as follows:

type family *Update rs i t*
type instance *Update* (s, ts) Z $t = (t, ts)$
type instance *Update* $()$ Z $t = (t, ())$
type instance *Update* (s, ts) $(S\ n)$ $t = (s,$ *Update ts n t*$)$

The type environment *Update rs d t* stands for the type environment *rs* where there d^{th} element has been set to t.

In the constructors for conditional and unconditional code transfers (*BNZ* and *JMP*), the argument of type *Sub rs rs′* is a witness that the target of the jump is of a type that is compatible with the current context. More precisely, it means that the first n registers listed in *rs* agree in types with *rs′*, or in other words that *rs′* is a prefix of *rs*. The type *Sub* is a GADT defined as follows:

data *Sub rs rs′* **where**
  S0 :: *Sub rs* $()$
  Sx :: *Sub rs rs′* $\rightarrow$ *Sub* (s, rs) (s, rs')

8.4.1 Type translation

As usual the type translation ($\mathcal{T}_{type}[\![-]\!]$) and its generalization to type environments are implemented as type families:

```
data Instr cs t where
  ARITH ::    PrimOp                                    -- p
           → NatRep d                                   -- r_d
           → Index rs Int                               -- r_s
           → ValT (cs, i, rs) Int                       -- v
           → Instr cs (Code i (Update rs d Int))
           → Instr cs (Code i rs)
  BNZ ::      Sub rs t
           → Index rs Int                               -- r
           → ValT (cs, i, rs) (Code Z t)                -- v
           → Instr cs (Code i rs)
           → Instr cs (Code i rs)
  MV ::       NatRep d                                  -- r_d
           → ValT (cs, i, rs) t
           → Instr cs (Code i (Update rs d t))
           → Instr cs (Code i rs)
  UNPACK ::   NatRep d                                  -- r_d
           → ValT (cs, i, rs) (Exists s)                -- v
           → Instr cs (Code (S i) (Update (ShiftEnv rs) d s))
           → Instr cs (Code i rs)
  MKTUP ::    NatRep d                                  -- r_d
           → MapT (ValT (cs, i, rs) ) t                 -- ⟨v_0, ..., v_{n-1}⟩
           → Instr cs (Code i (Update rs d (Tup t)))
           → Instr cs (Code i rs)
  LD ::       NatRep d                                  -- r_d
           → Index rs (Tup tup)                         -- r_s
           → Index tup t
           → Instr cs (Code i (Update rs d t))
           → Instr cs (Code i rs)
  JMP ::      Sub rs t
           → ValT (cs, i, rs) (Code Z t)                -- v
           → Instr cs (Code i rs)
  HALT ::     Instr cs (Code i (Int, rs))
```

Figure 8.5: Encoding of TAL instruction sequences (cf. source file TAL.hs, page 208).

$$\begin{array}{ll}
\textbf{type family } \mathit{Ttype}\ t & \\
\textbf{type instance } \mathit{Ttype}\ (\mathit{Cont}\ k\ t) & = \mathit{Code}\ k\ (\mathit{Ttype}\ t, ()) \\
\textbf{type instance } \mathit{Ttype}\ (\mathit{Exists}\ t) & = \mathit{Exists}\ (\mathit{Ttype}\ t) \\
\textbf{type instance } \mathit{Ttype}\ (\mathit{Var}\ v) & = \mathit{Var}\ v \\
\textbf{type instance } \mathit{Ttype}\ (\mathit{Tup}\ t) & = \mathit{Tup}\ (\mathit{Tenv}\ t) \\
\textbf{type instance } \mathit{Ttype}\ \mathit{Int} & = \mathit{Int}
\end{array}$$

$$\begin{array}{ll}
\textbf{type family } \mathit{Tenv}\ ts & \\
\textbf{type instance } \mathit{Tenv}\ () & = () \\
\textbf{type instance } \mathit{Tenv}\ (s, ts) & = (\mathit{Ttype}\ s, \mathit{Tenv}\ ts)
\end{array}$$

8.4.2 Term translation

Theorem 8.1, along with Lemma 8.1 and Lemma 8.2, are reflected in the types of the functions that implement the translation of $\lambda_{\mathcal{H}}$ programs, values and expressions.

Programs The top-level function that performs code generation expresses Theorem 8.1:

$$\mathit{cgProg} :: \mathit{ProgramH} \rightarrow \mathit{ProgramT}$$

The function *cgProg* implements $\mathcal{T}_{prog}[\![-]\!]$, so internally it assembles the code blocks for the individual bindings of the $\lambda_{\mathcal{H}}$ program. This is accomplished by a local recursive function that traverses the list of code blocks; this function's type is as follows:

$$\begin{array}{lll}
\mathit{cgBindings} :: & \forall fs, cs. & \mathit{MapT}\ (\mathit{CodeBlockH}\ fs_0\ )\ fs \\
& \rightarrow & \exists cs.\ (\mathit{EnvRep}\ cs, \\
& & \quad \mathit{MapT}\ (\mathit{CodeBlockT}\ (\mathit{Cat}\ (\mathit{Tenv}\ fs_0)\ cs))\ (\mathit{Tenv}\ fs), \\
& & \quad \mathit{MapT}\ (\mathit{CodeBlockT}\ (\mathit{Cat}\ (\mathit{Tenv}\ fs_0)\ cs))\ cs)
\end{array}$$

The function receives the set of code blocks in the $\lambda_{\mathcal{H}}$ program, and returns the list of code blocks corresponding to these original bindings, along with the list of extra code blocks.

In this type signature, fs_0 represents the type of *all* the global bindings in the original programs and fs represents the fragment (suffix) of fs_0 which is yet to be processed, and cs represents the type of extra code blocks generated so far.

Values Similarly, the type of the function that implements the value translation ($\mathcal{T}^{\gamma}_{val}[\![-]\!]$) reflects Lemma 8.1:

$$
\begin{aligned}
cgVal :: \ & MapT\ (Index\ rs)\ (Tenv\ ts) \\
& \rightarrow MapT\ (Index\ cs\ )\ (Tenv\ fs) \\
& \rightarrow ValH\ (fs, i, ts)\ t \\
& \rightarrow ValT\ (cs, i, rs)\ (Ttype\ t)
\end{aligned}
$$

The first two parameters together encode the variable map (γ): the first one maps local variables in the source programs to registers in the target program, and the second one maps global variables to code labels.

As expected, the translation of existential packages ($\mathsf{pack}\ [e, \tau_1]\ \mathsf{as}\ \tau_2$) and type applications ($e[\tau]$) require the invocation of a commutativity lemma:

Lemma 8.3 *For any source types τ_1, τ_2 and index i,*

$$\mathcal{T}_{type}[\![\tau_1[\tau_2/i]]\!] = \mathcal{T}_{type}[\![\tau_1]\!][\mathcal{T}_{type}[\![\tau_2]\!]/i].$$

The lemma is implemented in the same way as the lemmas from Section 4.4, as a term-level function. Its type is:

$$
\begin{aligned}
& substTtypeCommute :: \\
& \quad TypeRep\ t_1 \rightarrow TypeRep\ t_2 \rightarrow NatRep\ i \\
& \rightarrow Equiv\ (Ttype\ (Subst\ t_1\ t_2\ i))\ (Subst\ (Ttype\ t_1)\ (Ttype\ t_2)\ i)
\end{aligned}
$$

Expressions Finally, the type of the function that implements the translation of $\lambda_{\mathcal{H}}$ expressions ($\mathcal{T}^{\gamma,\Delta,\Gamma,i}_{exp}[\![-]\!]$) reflects Lemma 8.2:

$$
\begin{aligned}
cgExp :: \ & NatRep\ i \rightarrow EnvRep\ ts \rightarrow EnvRep\ cs_0 \rightarrow EnvRep\ rs \\
& \rightarrow MapT\ (Index\ cs_0)\ (Tenv\ fs) \\
& \rightarrow MapT\ (Index\ rs)\ (Tenv\ ts) \\
& \rightarrow ExpH\ (fs, i, ts) \\
& \rightarrow \exists cs.\ (EnvRep\ cs, \\
& \qquad MapT\ (CodeBlockT\ (Cat\ cs_0\ cs))\ cs, \\
& \qquad Instr\ (Cat\ cs_0\ cs)\ (Code\ i\ rs))
\end{aligned}
$$

The function receives the variable maps (same as for *cgVal*), along with the source expression, and returns the converted expression and the set of extra code blocks generated.

The type parameter cs_0 gives the type of the code blocks generated so far, and the type variable rs gives the type of the first n registers in the context where the translated expression will appear. The existentially quantified type variable cs characterizes the set of extra code blocks generated while traversing this expression.

A term-level representative of the type i is needed for the construction of the extra code blocks. The representatives of rs and cs_0 are used to generate fresh labels and register identifiers. The representative of ts is needed in the translation of `unpack`, to instantiate a lemma stating that the "shifting" of the context commutes with the type translation:

$$\mathit{shiftTenvCommute} :: \mathit{EnvRep}\ ts \to \mathit{Equiv}\ (\mathit{Tenv}\ (\mathit{ShiftEnv}\ ts))\ (\mathit{ShiftEnv}\ (\mathit{Tenv}\ ts))$$

Summary

This chapter presented a simple code generation phase which is the last step in the compilation pipeline. This phase mapped the "linearized" language from the previous chapter into a more rudimentary language, which does not use local variables, and where operations are arranged into linear sequences of instructions (unlike the $\lambda_{\mathcal{H}}$ expressions which are nested to some degree because of the `if0` construct.)

The implementation of code generation mostly applies the techniques already developed in the implementation of the previous two phases. A notable exception is the use of the *Update* type family to assign different types to a given register in different regions of the program. Compared to the closure conversion or hoisting phase, the implementation of code generation may be the more "natural". In particular, the fact that the notion of binding in $\lambda_{\mathcal{H}}$ and TAL is fundamentally different makes the variable map (γ) more legitimate, as it is necessary even if one does not want to prove type preservation. This is in contrast to closure conversion or hoisting, where the variable map comes as a consequence of our particular choice of program representation.

Chapter 9

Benchmarks

In this chapter we report and briefly analyze compilation times achieved by our compiler. As stated in the introduction, one of the potential benefits of our general approach to compilation is an improvement in compilation time, as a result of eliminating all dynamic checks that take place in a conventional compiler. Indeed, this benefit is not fully realized in the current implementation. In particular, some type information must still be manipulated as data, and the lemmas about the various type families defined in the implementation must still be "executed" at run-time. Beside type issues, a notable source of inefficiency is the unary representation of de Bruijn indices.

We have not attempted to optimize the compiler. Figures are provided to give an idea of the behavior of the initial version. We identify the most expensive phases and operations in our compiler, and estimate the overhead imposed by the run-time manipulation of type information.

The chapter focuses on compilation times, but does not report or analyze performance of the compiled programs. Our contribution is the type-safe implementation of a number of code transformations, but the code transformations themselves are fairly typical of a compiler for a call-by-value functional language. We have not implemented any code optimizations that would motivate us to look into the performance of the compiled code.

Sample programs

We will use two simple programs as benchmarks. One is a small program that makes use of parametric polymorphism. The other is a simple program that consists of a number of

mutually recursive functions, which can be made arbitrarily large by varying the number of functions in the program. We have chosen this set of sample programs so that all the features of the source language are exercised, and compilation will take long enough to yield meaningful profile information about the compiler. They also allow us to see you compilations times are affected by program size.

Sum types The first benchmark program exercises parametric polymorphism, which is used to construct an encoding of sum types. A sum type $(\tau_1 + \tau_2)$ is represented as the type:

$$\forall\alpha.\ (\tau_1 \to \alpha) \to (\tau_2 \to \alpha) \to \alpha$$

and the constructors are represented as functions:

$$\begin{aligned}
&\mathit{inLeft} :: \forall t_1, t_2.\ t_1 \to (\forall\alpha.\ (t_1 \to \alpha) \to (t_2 \to \alpha) \to \alpha) \\
&\mathit{inLeft} = \lambda x.\ \lambda f.\ \lambda g.\ f\ x \\
&\mathit{inRight} :: \forall t_1, t_2.\ t_2 \to (\forall\alpha.\ (t_1 \to \alpha) \to (t_2 \to \alpha) \to \alpha) \\
&\mathit{inRight} = \lambda x.\ \lambda f.\ \lambda g.\ g\ x
\end{aligned}$$

Analysis of a value in a sum type is simply achieved by a function application. The sample program simply constructs a value and analyses it:

$$\begin{aligned}
&\mathsf{let}\ s = \mathit{inLeft}[\mathsf{int}, \langle\mathsf{int}, \mathsf{int}\rangle]\ 4 \\
&\mathsf{in}\ s\ (\lambda x.\ x)\ (\lambda x.\ \mathsf{fst}\ x + \mathsf{snd}\ x)
\end{aligned}$$

TAK The second benchmark program is used to measure compilation times for artificially large programs. It is based on the *TAK* function which was devised by Takeuchi; see (Knuth 1997). The *TAK* function is defined as follows:

$$\begin{aligned}
\mathit{TAK} = \lambda x\ y\ z.\ \mathsf{if}\ & y < z \\
\mathsf{then}\ & \mathit{TAK}\ (\mathit{TAK}\ (x-1)\ y\ z) \\
& \qquad (\mathit{TAK}\ (y-1)\ z\ x) \\
& \qquad (\mathit{TAK}\ (z-1)\ x\ y) \\
\mathsf{else}\ & z
\end{aligned}$$

Our benchmark program defines n mutually recursive instances of the *TAK* function, called $\mathit{TAK}_0, \ldots \mathit{TAK}_{n-1}$, and each recursive call to *TAK* will invoke one of the n instances

chosen at random; for a given value of n, we call the sample program TAK^n. For example, TAK^2 looks as follows:

$$
\begin{array}{l}
\textsf{letrec } TAK_0 = \lambda x\ y\ z.\ \textsf{if } y < z \\
\qquad\qquad\qquad \textsf{then } TAK_1\ (TAK_1\ (x-1)\ y\ z) \\
\qquad\qquad\qquad\qquad\qquad (TAK_0\ (y-1)\ z\ x) \\
\qquad\qquad\qquad\qquad\qquad (TAK_1\ (z-1)\ x\ y) \\
\qquad\qquad\qquad \textsf{else } z \\
\qquad TAK_1 = \lambda x\ y\ z.\ \textsf{if } y < z \\
\qquad\qquad\qquad \textsf{then } TAK_0\ (TAK_1\ (x-1)\ y\ z) \\
\qquad\qquad\qquad\qquad\qquad (TAK_0\ (y-1)\ z\ x) \\
\qquad\qquad\qquad\qquad\qquad (TAK_0\ (z-1)\ x\ y) \\
\qquad\qquad\qquad \textsf{else } z \\
\textsf{in } TAK_0\ 8\ 7\ 2
\end{array}
$$

Note that there is no mutual recursion in our source language. To define these mutually recursive functions, we use a function which receives an extra integer argument i, and applies the corresponding instance TAK_i, as follows:

$$
\begin{array}{ll}
\textsf{letrec } TAK\ i = \textsf{if } i = 0 & \\
\qquad\qquad \textsf{then } \lambda x\ y\ z.\ \textsf{if } y < z & -\ TAK_0 \\
\qquad\qquad\qquad\qquad \textsf{then } TAK\ 1\ (TAK\ 1\ (x-1)\ y\ z) & \\
\qquad\qquad\qquad\qquad\qquad\qquad (TAK\ 0\ (y-1)\ z\ x) & \\
\qquad\qquad\qquad\qquad\qquad\qquad (TAK\ 1\ (z-1)\ x\ y) & \\
\qquad\qquad\qquad\qquad \textsf{else } z & \\
\qquad\qquad \textsf{else } \lambda x\ y\ z.\ \textsf{if } y < z & -\ TAK_1 \\
\qquad\qquad\qquad\qquad \textsf{then } TAK\ 0\ (TAK\ 1\ (x-1)\ y\ z) & \\
\qquad\qquad\qquad\qquad\qquad\qquad (TAK\ 0\ (y-1)\ z\ x) & \\
\qquad\qquad\qquad\qquad\qquad\qquad (TAK\ 0\ (z-1)\ x\ y) & \\
\qquad\qquad\qquad\qquad \textsf{else } z & \\
\textsf{in } TAK\ 0\ 8\ 7\ 2 &
\end{array}
$$

	Safe	Unsafe
Sum types	0.38	0.25
TAK^1	0.30	0.25
TAK^2	0.36	0.26
TAK^4	0.55	0.30
TAK^8	1.12	0.36
TAK^{16}	3.11	0.54
TAK^{32}	10.54	1.02
TAK^{64}	40.40	2.55
TAK^{128}	174.52	8.94

Figure 9.1: Compilation times (in seconds).

Compilation times

The compilation times[1] for the benchmark programs are reported in Figure 9.1 These times do not include any time spent in the compiler front-end, as the source programs are directly encoded using the constructors of the type *Exp* of Chapter 3. However, to give more realistic timing, the module in which the source programs are "hard-coded" is *not* pre-compiled, but rather interpreted by GHC's interactive environment; this is meant to approximate the time that a front-end using Template Haskell would consume.

Compilation times are reported for two versions of the program, marked as "safe" and "unsafe" in the table. The "safe" version is the one which executes all the lemmas at run-time. The "unsafe" version skips the execution of some of the lemmas, identified as the most time-consuming ones (using GHC's profiling facility.) It turns out that skipping these lemmas saves 34% on very small programs, and the "optimized" version of TAK^{128} runs almost 20 times faster than the safe one.

The lemmas in question are the following four:

1. *shiftCenvCommute*, stating that *ShiftEnv* and the type translation for closure conversion (*Cenv*) commute;

2. *shiftTenvCommute*, stating that *ShiftEnv* and the type translation for code generation (*Tenv*) commute;

3. *catCenvCommute*, stating that *Cenv* and concatenation (of type contexts) commute

[1]The programs are compiled with GHC version 6.8.3. The test system is a 2.4GHz Core 2 Duo (MacBook Pro running Mac OS X 10.5.6).

Phase	Execution time (%)
CPS conversion	0
conversion to de Bruijn	7.1
closure conversion	14.3
function hoisting	7.1
code generation	42.9
pretty printing	28.8

Figure 9.2: Breakdown of compilation time for TAK^{64}, unsafe version.

4. *catAssoc*, stating that concatenation (of type contexts) is associative (which is required by hoisting and code generation)

The way that a lemma execution is skipped is by returning a generic proof object (using an unsafe coercion) instead of constructing a proper one. For example, the unsafe version of *catAssoc* is defined as follows:

$$\begin{aligned} \mathit{catAssocUnsafe} :: \; & \mathit{EnvRep}\ ts_0 \rightarrow \mathit{EnvRep}\ ts_1 \rightarrow \mathit{EnvRep}\ ts_2 \\ & \rightarrow \mathit{Equiv}\ (\mathit{Cat}\ ts_0\ (\mathit{Cat}\ ts_1\ ts_2)) \\ & \qquad\qquad (\mathit{Cat}\ (\mathit{Cat}\ ts_0\ ts_1)\ ts_2) \\ \mathit{catAssocUnsafe}\ _\ _\ _ & = \mathit{unsafeCoerce}\ \mathit{Equiv} \end{aligned}$$

Here, *unsafeCoerce* is a library function which can coerce among arbitrary types:

$$\mathit{unsafeCoerce} :: \forall \alpha, \beta.\ \alpha \rightarrow \beta$$

It is "safe" to apply *unsafeCoerce* only when the run-time representation of α and β are the same; improper uses will result in memory corruption and unpredictable behavior.

Compilation phases

The relative cost of the individual compilation phases for the unsafe version of TAK^{64} is reported in Figure 9.2.

CPS conversion takes up so little time that it does not account for a measurable fraction of the overall compilation time. Note that the input program is rather small in comparison to the input of the other phases, as continuations and closures multiply the size of the program a few times. Conversion to de Bruijn indices performs many comparisons on contexts, which make it somewhat expensive.

Code generation is the most expensive phase, accounting for 42.9% of the compiler's execution time. It turns out that 50% of the time of code generation is spent on weakening (i.e. interpreting instructions, values in indices in extended contexts.) Weakening is conceptually a no-op (i.e. an identity function), but its implementation must traverse all instructions, values and indices until the base case of indices (the constructor $I0$) is reached. If we replace all weakening operations involved in code generation by an unsafe coercion, we measure an overall speedup of 17% for TAK^{64} (from 2.55s to 2.12s), and 30% for TAK^{128} (from 8.94s to 6.32s.)

For what it is, the pretty-printer uses up a rather large fraction of the time. It is implemented using pretty-printing combinators, so using a lower-level formatting facility would yield an easy performance improvement.

Summary

We have not tried to optimize the compiler, so it still contains obvious sources of inefficiency, and the benchmarks presented here give a general idea of the performance penalties they incur. The main sources of inefficiency are the manipulation of types as data, including the execution of the lemmas about type families, and the unary representation of indices.

The execution of the lemmas turns out to be very costly, and their cost increases in proportion with larger object programs, to the point that it accounts for the vast majority of the compilation time for large enough programs. The Haskell extension we proposed (Guillemette and Monnier 2008a; Schrijvers et al. 2008) for the static verification of invariants on type families would eliminate this overhead.

There is a super-linear component in compilation times (cf. Figure 9.1), which was to be expected as a consequence of the unary representation of indices and the linear representation of static environments. The transformations on first-order representations manipulate indices and contexts extensively. In particular, a frequently used operation is the concatenation of contexts, which takes linear time. A representation of static contexts based on a data structure that supports concatenation in constant time would bring a substantial performance improvement; difference lists are such a data structure, see e.g.

(Shapiro and Sterling 1994). Also, index weakening takes linear time, due to our unary representation of indices. Note that index weakening is an operation at the term-level, so language support for static verification of invariants on type families would not help here. As for the representation of static contexts, a representation of indices which supports weakening in constant time would bring a substantial performance improvement.

Chapter 10

Conclusion

In this final chapter we make general observations on our experience developing this compiler.

We have formally established important properties of our compiler following an approach to formal verification based on types as a formal method. Type preservation in a compiler is by nature a prime subject for such type-based verification. When undertaking the construction of a compiler for System F, we had a precise idea *a priori* of the type discipline that each code transformation was intended to follow, as laid out in the work on compilation to typed assembly language of Morrisett et al. (1999). Yet, as one would expect, the implementation effort raised many subtle issues.

Identifying the right program representations was perhaps the most delicate part of our work, and we comment on the choices we have made. As in any form of type-based verification, we had to make judicious use of the features of the implementation language, and cope with its constraints and limitations. We comment on these more generic issues, and our use of GADTs and type families, as well as other features that we used or considered using in past versions of our compiler. We also make a broader review of binder representations used elsewhere, and contrast our type-preserving compiler with other systems.

10.1 Representation of bindings

The representation of binders is an issue that deserves close attention in any application that manipulates programs or similar structures, such as compilers and theorem provers. When working on System F we considered many options for representing variables, both at the level of types and terms.

Type-level type variables. For type-level type variables (i.e. those introduced by $\forall$, $\exists$, etc.), a de Bruijn encoding of types combined with type families for type-level operations (such as substitution) provides a fairly reasonable representation. De Bruijn indices do require delicate manipulations, but these are mostly concealed in the definition of the type families (*Subst* and U), and leave the definition of the term representation relatively simple and direct. A de Bruijn encoding of types is also a common choice in compilers using a typed intermediate language.

Given the fact that we do not need to analyze the types bound at the type level, HOAS would be an attractive representation. This would allow us to encode a universal type ($\forall\alpha.\tau$) directly as an anonymous type-level function ($All\ (\lambda a.\ t)$). Then, the type given to a type application ($e[\tau]$) could simply be expressed as a type application in the host language ($t_1\ t_2$), instead of using an explicit type family for substitution ($Subst\ t_1\ t_2\ Z$). Unfortunately, GHC does not support type-level λ-expressions, so this is not an option. Note that type families (or, in general, type synonyms) are not λ-expressions, but named functions that can only be defined at the top level of the program. We could imagine a scheme where a type synonym is introduced for each universal type in the object program by meta-programming, but this approach is rather unattractive, as it compromises the clean distinction of binding times (compile *vs.* run time) in the compiler, and limits static safety.

Term variables. For term variables (i.e. those introduced by λ, let, etc.), we started using HOAS, which is rather uncommon and is poorly supported in most languages, but served us well for the CPS transform. It is arguably more elegant than de Bruijn indices, and requires fewer type annotations since the typing environment is treated implicitly.

For closure conversion, on the other hand, HOAS cannot be used because of its inability

to identify variables (i.e. determine whether or not two given variables are actually the same) or express that a term is closed (since contexts are implicit). As discussed in Chapter 5, a hybrid representation could be used to instantiate variables with some concrete data and thus recover the ability to identify variables. Also, by combining the closure conversion and the hoisting phases, we could probably manage without explicit contexts, as in Chlipala's compiler (2008).

The more common representation of term variables in compilers is as names, usually represented as small integers or as pointers, so that would be our favorite choice, but lifting small integers or pointers to the type level to reason about them at the type level is rarely supported, and GHC is no exception. Even using less efficient representations of names, which lend themselves to singleton types (e.g. Peano numbers), still suffers from the extra complexity of having to reason about freshness.

So we ended up using de Bruijn indices. As demonstrated, they do work, but they require delicate index updates at various places and accompanying lemmas, for example when moving code into or out of a scope, or when inserting or removing variable bindings, which phases like closure conversion and hoisting do all the time. The complexity we have in our current code is bearable, but we had to fine-tune it to get there: e.g., some apparently minor changes to the type of the constructors for type abstraction (such as *Cfix*, page 82) can lead to significant complications in the compiler's code. Another problem with de Bruijn indices is that most people find them mind-boggling to debug, although this is more true in untyped settings, where errors are caught too late.

When put in context, the choice of HOAS for CPS conversion is hard to justify from an engineering standpoint, as it forces us to convert to and from first-order representations. The overall implementation of our compiler would be simplified by using de Bruijn indices throughout all compilation phases, as it would allow us to eliminate both the conversion to HOAS in the front-end of the compiler, and the conversion to de Bruijn indices in between CPS and closure conversion.

Term-level type variables. If the term encoding is in HOAS, then the best option for term-level type variables (i.e. those bound by Λ, `unpack`, etc.) is to use HOAS as well, so that is what we have done.

If the term encoding is first-order, then HOAS may still be a very good choice, (if the host language's monomorphism restriction does not get in the way), but in our case, for the same reason we could not use HOAS for term variables, we could not use HOAS for term-level type variables during closure conversion: we need to express the fact that the functions we output are also closed with respect to types.

Compilers tend to avoid de Bruijn indices in favor of names for term-level type variables, again in order to avoid the issues linked with shifting indices when moving code into or out of a scope. But we again decided to use de Bruijn indices for the same reason as for term variables: names are difficult to represent efficiently as types in GHC, and reasoning about freshness would introduce a lot of complexity and force us to restructure the code significantly.

In other words, essentially the same arguments that led us to choose de Bruijn indices for the term variables, led us to use de Bruijn for term-level type variables. And again, although we believe this choice to be the best there is right now, it is not satisfactory.

10.2 Implementation language

In this section, we comment on our use of GADTs and type families, as well as other features that we used or considered using in past versions of our compiler.

Some of the issues discussed here are relevant to type-based program verification in general, and are not specific to type-preserving compilation, in the sense that any form of advanced type-based program verification which makes use of the same features would likely meet the same limitations. Note in particular that our proposal to extend Haskell with support for enforcing invariant of type families (discussed below) would benefit a potentially large class of applications, as many applications that implements complex type-level notions using type families would need such a mechanism.

Type families Before type families were made available in GHC, we used GADTs to encode witnesses of type preservation (Guillemette and Monnier 2006, 2007). Essentially, every time a term was produced, it was accompanied by a witness that the term was of the expected type. The drawbacks of this scheme are run-time overhead, a substantial

amount of code bloat (for manipulating the existential packages), and the fact that our "proofs" were encoded in an unsound logic. Type families essentially solved these problems. We further compare the schemes that use only GADTs or GADTs plus type families in (Guillemette and Monnier 2008a).

The representation of System F also benefited from type families. In the past we actually worked on a representation which relied on GADTs to encode witnesses of type applications (essentially encoding the type families from Section 3.4 as GADTs). Type families obviously make this representation much more direct.

Lemmas over type families As seen in Section 4.4, we need to prove properties of the type families we define for the transformations over System F to type-check. In our current implementation, such lemmas are implemented as term-level functions that produce a proof witness that the lemma holds at particular types. This is unsatisfactory in a number of ways: it incurs run-time overhead, it forces us to carry type-representatives as part of the syntax trees, and it encodes the lemma in an unsound logic (due to non-termination at the term level).

To address this limitation, we are investigating an extension to Haskell to directly support lemmas over type families: to have the type-checker verify that all instances satisfy the lemmas declared by the programmer (Guillemette and Monnier 2008a; Schrijvers et al. 2008). This way we could get the type equality coercion we need without having to encode it explicitly in the syntax tree.

Type Classes Having started this work from an existing untyped compiler using algebraic data types for its term representation, it was only natural to use GADTs. This said, there is no indication that the same could not be done with multi-parameter type classes, but GADTs are probably a more natural representation for abstract syntax trees in a functional language.

Early on, we tried to use type classes to encode type-level functions as well as various proof objects. This was meant to help us by letting the type checker infer more of the type annotations and hence leave us with a cleaner code more focused on the actual algorithm than on the type preservation proof. Unfortunately we bumped into serious difficulties due to the fact that the then current version of GHC was not yet able to properly handle

tight interactions been GADTs and type classes. More specifically the internal language of GHC had limitations that prevented some "exotic" uses of functional dependencies. Those limitations can appear without GADTs, but in our use of GADTs, we bumped into them all the time. In the mean time, type families appeared and provided an alternative way to let the type system and type inference do more of the work.

The shift to F_C (Sulzmann et al. 2007) as the internal language in GHC potentially improves the interaction between GADTs and type classes. Yet, as we discussed elsewhere (Guillemette and Monnier 2008a), using type classes to prove type preservation necessitates extra annotations (in the form of class constraints) on the syntax tree, which must be propagated from phase to phase and would compromise modularity.

10.3 Related work

In this section we discuss representations of bindings used elsewhere, and compare our work to other projects in type-preserving and certified compilation.

10.3.1 Representations of bindings

The representation of term-level variables for a compiler like ours is still a problem in search of a satisfactory solution.

In our experience, HOAS was satisfactory for CPS conversion but not directly applicable for closure conversion or hoisting. An alternative developed by Pientka (2008) might offer a solution, as it extends HOAS with explicit contexts which could be used to express closedness and is able to identify variables and would therefore allow us to implement closure conversion and other transformations; see also (Pientka and Dunfield 2008). As it is dependently typed, it should lend itself naturally to verification of type preservation.

Chlipala (2008) proposed a variant of HOAS called *parametric higher-order abstract syntax* (PHOAS) meant to combine the advantages of first-order and higher-order representations. It is a generalized weak HOAS (Despeyroux et al. 1995), which is a form of higher-order abstract syntax with an explicit introduction form (i.e. data constructor) for variables. Weak HOAS is typically used in systems like Coq where negative occurrences

are forbidden, so that HOAS cannot be used directly. PHOAS generalizes HOAS in the sense that the type of a syntax tree is parameterized by the types associated with variables, in a way similar to the parametric representation of HOAS used in the present thesis, due to Washburn and Weirich (2003). The equivalent of a first-order representation can be obtained by instantiating the type parameter to an integer type, so as to represent variables as numbers. If the type parameter is left abstract, the representation behaves like HOAS.

It is not clear that our implementation would benefit from using PHOAS. It might be an elegant way to use first-order and higher-order representations in different parts of the compiler, and would eliminate the need for the conversion to de Bruijn indices. However, as the encoding is higher-order, contexts are implicit, and in cases like closure conversion where explicitly reasoning about the context is required, a separate judgment that relates the expression and the context is needed. This can be done naturally in a language with full dependent types, but it is impractical in a language like Haskell. For example, Chlipala's closure conversion over PHOAS makes use of a well-formedness judgment, *Wf fvs* e, to ensure that an expression (e) only refers to the variables in a given set (*fvs*). To apply this technique in Haskell, *Wf* would need to be a GADT, and e would be a type parameter, so we would have to reify expressions at the level of types.

Name-based representations

Nominal logic (Pitts 2001) is a theory of bindings based on the notion of permutation (or swapping) of names which gives a systematic treatment of α-equivalence, fresh name generation and substitution. This approach has been incorporated in some logic and functional programming languages, namely αProlog (Cheney and Urban 2004) and FreshML (Shinwell et al. 2003), and also implemented in Haskell in the form of a library (Cheney 2005). Urban (2008) has also implemented it in Isabelle and used it to mechanize standard proofs such as Barendregt's substitution lemma for the simply typed λ-calculus, and developed a solution to one of the problems of the POPLmark challenge (Aydemir et al. 2005).

Locally nameless representations of binders use de Bruijn indices for bound variables and names for free variables. The intent is to combine the advantages of the two tech-

niques: de Bruijn indices enable trivial α-equivalence testing, and names eliminate the need of shifting indices in operations like substitution. McBride and McKinna (2004) discuss an implementation of these techniques in Haskell, as used in the implementation of Epigram.

It would be worth investigating to what extent these techniques could help alleviate the difficulties we face with our current program representations as, to our knowledge, neither nominal approaches nor locally nameless representations have been extensively used in the context of type-preserving (or certified) compilation so far, although they are popular in metatheory, e.g. (Aydemir et al. 2008).

10.3.2 Typed intermediate languages

There has been a lot of work on typed intermediate languages, originally motivated by the optimizations opportunities offered by the extra type information. The TIL compiler (Tarditi et al. 1996) makes use of type information to compile polymorphic code to efficient executable code, and relies on dynamic type dispatch to eliminates the overhead of boxing and unboxing all data representations. The FLINT compiler (Shao and Appel 1995; Shao 1997b) applies similar techniques but supports a larger set of language features including the ML module system, and performs a more precise analysis to reduce the overhead of run-time type analysis.

Type-directed compilers such as TIL and FLINT at some point discard the type information and generate untyped code. Morrisett et al. (1999) showed how the type information can be preserved all the way down to typed assembly language in a compiler for System F.

Proof-Carrying Code (PCC) (Necula 1997) is a general framework for safely executing code of untrusted source, by locally checking a proof of safety which is supplied with the program. PCC is more general than type assembly language as the safety proofs are encoded in a first-order logic, although this logic usually contains specific extensions tied to a particular type system. The PCC framework does not prescribe a particular way of generating the proofs, and compiling to typed assembly language can be seen as a systematic way of generating PCC. Foundational PCC (Appel 2003; Hamid et al. 2002) takes an approach where the proofs are encoded in a more fundamental logic, with the

advantage that the verifier for this logic can be smaller and is less likely to contain bugs (and is easier to prove correct.) In counterpart, the proofs will typically be larger, as they will need to construct language-specific abstractions on top of the bare logic, and need to include artifacts such as a proof of type soundness.

Shao et al. (2002) show a low-level typed intermediate language for use in the later stages of a compiler. It incorporates a powerful proof language at the type-level, based on the calculus of inductive constructions (Paulin-Mohring 1993), which can be used to state and prove some properties of program expressions. They show how the proofs can be preserved as the code undergoes CPS and closure conversion.

10.3.3 Certified compilation

The construction of certified compilers, that is, compilers with formal proofs of correctness, has been extensively researched over the past decades (Dave 2003).

Moore's work (1989) is an early example of a compiler with a complete proof of correctness. Its source language is a "high-level" assembly language, with features such as local variables and recursive subroutines. This source language is thus fairly low level compared to ours. The proof is formalized in the logic of the Boyer-Moore theorem prover.

A modern example is Leroy's compiler (Leroy 2006; Blazy et al. 2006) for a C-like language, written in the Coq proof assistant, which has a completely formalized correctness proof. It includes a number of optimization phases. Again, the source language is fairly low-level compared to ours, as it is first-order and does not have nested variable scopes. The correctness proof is also fairly large, consisting of about 17000 lines of Coq code.

Certified compilers from (variants of) Java to bytecode, as well as bytecode verifiers have been developed in Isabelle/HOL by a number of authors (Strecker 2002; Klein and Nipkow 2004; Klein and Strecker 2004). The source language is higher-order in the sense that it incorporates a notion of dynamic dispatch, but dynamic dispatch is also present in the target language, so the translation is conceptually simpler than when compiling down to assembly language, as done here.

More closely related to our work, Chlipala's certified compiler (2007) translates a higher-order functional language to typed assembly language. He uses first-order program representations similar to ours, and his compiler contains code transformations similar to

ours, including a CPS and closure conversion. His compiler is restricted to the simply-typed case, and does not treat recursion. His proof technique, which is based on denotational semantics, makes his proof rather different from ours. The main development is roughly 4500 lines of code, but relies on a Coq library for the formalization of programming languages which is 3520 lines of code, and 2717 lines of support code written in OCaml. In his later work (Chlipala 2008), he used PHOAS to develop a certified CPS conversion over System F, which notably uses higher-order encoding of types (using PHOAS as well.)

Tian (2006) showed a certified CPS translation over the simply typed λ-calculus, which uses HOAS and is implemented in Twelf. The translation is performed in one pass and reduces administrative redexes on-the-fly, like the one we presented in Chapter 4.

Hannan and Pfenning (1992) show a conversion from higher-order abstract syntax to de Bruijn indices, with a proof of correctness expressed in LF. Their conversion algorithm is similar to the one we showed in Chapter 5, although it is restricted to the simply typed λ-calculus.

Our compiler is the only one which handles the translation of System F all the way down to typed assembly language. By focusing on preservation of static semantics rather than full compiler correctness, we were able to handle a larger set of language features. Our work is the only one which verifies a closure conversion for a polymorphic language, which is the more challenging and crucial part of the compiler.

The proof of static semantic preservation is tightly integrated within the compiler's code, in the form of type annotations. In contrast, certified compilers either come with a correctness proof separate from the compiler's code, or the compiler's code is extracted from the proof, as in the case of the compilers of Leroy and Chlipala. Our compiler is implemented in 3486 lines of Haskell code, so it is fairly compact compared to the certified compilers mentioned above.

The implementation language we use is relatively simple, as our implementation essentially relies on GADTs and type families. In particular the proof language of Coq is substantially more conceptually complex than ours, since it supports full dependent types. By using Haskell rather than a language with full dependent types, we narrow down the "semantic gap" between the source and implementation language. GADTs and type families can actually be encoded in a variant of System F with type equality coer-

cions (Sulzmann et al. 2007). We are thus getting closer to being able to bootstrap our compiler, i.e. to have the same source and implementation language, so as to be able to compile our own compiler.

Typed program transformations

Individual code transformations have been statically verified for type preservation. To our knowledge, our project is the first to formally verify type preservation for an entire compiler.

Chiyan Chen et al. (2003) also show a CPS transformation where the type preservation property is encoded in the host language's type system. It is implemented in DML (Xi and Scott 1999), using GADTs and a type-level functions. Their term representation is first-order using de Bruijn indices, and their implementation is somewhat more verbose than ours, as it requires an explicit variable map, while we manage without one since we used HOAS. Linger and Sheard (2004) show a CPS transform over a GADT-based representation with de Bruijn indices.

A detailed proof of type preservation for an earlier formulation of closure conversion over System F was formulated by Minamide et al. (1996). Pottier and Gauthier (2004) have shown defunctionalization over a superset of System F to be type-preserving.

Pašalić (2004) constructed a statically verified type-safe interpreter with staging for a language with binding structures that include pattern matching. The representation he uses is based on de Bruijn indices and relies on type equality proofs in Haskell.

10.4 Future work

Further work on extending Haskell with support for statically verified invariants of type families would greatly benefit the implementation of our compiler, as it would allow us to eliminate most (if not all) of the remaining run-time checks. This would allow us to eliminate redundant parts of the code, as the run-time checks are currently implemented using term-level functions which duplicate the definition of the type families (cf. Section 4.4).

Perhaps the most attractive line of research at this point is the extension of the

source language with more features. Features similar to those currently supported, such as existential types, recursive types, and sum types, should come as a fairly natural extension of our current implementation. Supporting these particular features would allow us to encode algebraic datatypes, and support a language such as Core ML. A more ambitious goal is to accept an input language comparable to GHC's internal System F-like language, so as to be able to bootstrap, as we discuss in the next section.

10.4.1 Bootstrap

We have verified important properties of our compiler using the type system of Haskell. The validity of this formal development depends on the type soundness of Haskell, and the correct implementation of the type checker in GHC, which verifies our proof. The correct execution of our compiler also depends on the correctness of the translation implemented in GHC. To gain better assurance about our results, we would thus need to mechanize the metatheory of Haskell (i.e. mechanically prove type soundness), and certify the compiler implementation, including the type checker and code transformations. For this latter aspect, we would certainly be interested in applying the techniques developed in this thesis. Ideally, we would like the source language to be the same as the implementation language, so that we could compile our own compiler.

The features of the implementation language that we use (essentially, GADTs and type families) can be encoded in a variant of System F called System F_C (Sulzmann et al. 2007), which is currently the core internal representation used in GHC. System F_C extends System F with type equality coercions, which are witnesses of type equality. Coercions are type-level entities, which are abstracted and applied much like the types of System F. A term-level construct (coerce) is used to coerce a value using available evidence; this operation is a no-op at run time, coercions being purely static entities. A System F_C program introduces a number of type constructors, and corresponding data constructors (guarded by type equality constraints, used for supporting GADTs). A program also introduces axioms at the top level, from which coercions can be derived (these axioms are used for supporting type families.)

Supporting System F_C in a compiler like ours implies the development of a type-preserving translation to some typed assembly language with a built-in notion of type

equality. This could likely be worked out as an extension of the type-preserving translation of System F of Morrisett et al. (1999). As coercions in System F_C behave like types in System F, we can expect their interaction with CPS and closure conversion to be fairly limited. As System F_C maintains a clear distinction between types and terms, we are not facing the challenges of compiling a language with dependent types in a type-preserving way; see e.g. (Barthe and Uustalu 2002). Nevertheless, extending our compiler for supporting the extra features of System F_C – type and data constructors, axioms, etc. – would require a substantial amount of work, but as these features are mostly orthogonal to the code transformations (which mostly affect the representation of term-level functions), this can likely be done while preserving the general structure of the compiler.

10.5 Summary

In this dissertation, we have shown that it is possible to impose a strong type discipline on a compiler implementation for a language as expressive as System F.

Our approach is an attractive compromise between full compiler certification, where the correctness proof can easily overwhelm the compiler's actual code, and traditional compilers employing typed intermediate languages, which manipulate types intensively but do not exploit the type system of the implementation language.

Current limitations in the implementation language prevented us from having a purely static solution, as our implementation still needs to manipulate types at run-time and perform dynamic checks. This complication could probably have been avoided had we chosen a more powerful implementation language, such as Coq or Agda. In any case, applications like ours motivate the design of language extensions, and we can reasonably expect there will be such extension in the future that will solve our current issues.

The main obstacle to having a code as compact and readable as that of a conventional compiler may be the lack of a suitable representation of binders. We had recourse to de Bruijn indices for most of the transformations, while a conventional compiler would typically use names. In a language like Haskell, it seems promising to research programming

techniques which will make manipulation of first-order representations more abstract, so as to reduce the explicit mangling of indices and contexts to a minimum, while preserving the control and precision of first-order representations.

We have developed an extensive case study in the application of type systems to prove important properties of a compiler. As type systems and their implementation in compilers progresses, applications like our type-preserving compiler will be better supported, and further motivated by the greater expressive power of the languages of the future.

Bibliography

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

Andrew W. Appel. Foundational proof-carrying code. *Foundations of Intrusion Tolerant Systems*, 0:25, 2003.

Lennart Augustsson. Cayenne—a language with dependent types. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 239–250, New York, NY, USA, 1998.

B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, 2005.

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 3–15, New York, NY, USA, 2008.

Gilles Barthe and Tarmo Uustalu. Cps translating inductive and coinductive types. *SIGPLAN Not.*, 37(3):131–142, 2002. ISSN 0362-1340.

Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a c compiler front-end. In *International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475, aug 2006.

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

James Cheney. Scrap your nameplate: (functional pearl). In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 180–191, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-064-7.

James Cheney and Christian Urban. Alpha-prolog: A logic programming language with names, binding and alpha-equivalence. In *Proceefdings of the 20th International Conference on Logic Programming (ICLP)*, 2004.

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

Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 143–156, New York, NY, USA, 2008.

Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Software Engineering Notes*, 28(6):2–2, 2003.

N. G. de Bruijn. Lambda calculus notation with nameless dummies. *Indagationes mathematicae*, 34:381–392, 1972.

Joelle Despeyroux, Amy Felty, and Andre Hirschowitz. Higher-order abstract syntax in coq. In Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, editors, *TLCA*, volume 902 of *Lecture Notes in Computer Science*. Springer, 1995. ISBN 3-540-59048-X.

Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Conf. Record 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'96, St. Petersburg Beach, FL, USA, 21–24 Jan. 1996*, pages 284–294. ACM Press, New York, 1996.

David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 28–38, New York, NY, USA, 1986. ACM.

Louis-Julien Guillemette and Stefan Monnier. Type-safe code transformations in Haskell. In *Programming Languages meets Program Verification*, volume 174(7) of *Electronic Notes in Theoretical Computer Science*, pages 23–39, August 2006.

Louis-Julien Guillemette and Stefan Monnier. A type-preserving closure conversion in Haskell. In *Haskell Workshop*. ACM Press, September 2007.

Louis-Julien Guillemette and Stefan Monnier. One vote for type families in Haskell! In *The 9th symposium on Trends in Functional Programming*, 2008a.

Louis-Julien Guillemette and Stefan Monnier. A type-preserving compiler in Haskell. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 75–86, New York, NY, USA, 2008b. ACM.

Nadeem Abdul Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *Annual Symposium on Logic in Computer Science*, pages 89–100, Copenhagen, Denmark, July 2002.

John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418. IEEE Computer Society Press, 1992.

Robert Harper and Mark Lillibridge. Explicit polymorphism and cps conversion. In *In Twentieth ACM Symposium on Principles of Programming Languages*, pages 206–219. ACM Press, 1993.

Nevin Heintze and Jon G. Riecke. The slam calculus: programming with secrecy and integrity. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 365–377, New York, NY, USA, 1998. ACM.

Ralf Hinze. Fun with phantom types. In *The Fun of Programming*, Cornerstones in Computing, pages 245–262. Palgrave Macmillan, 2003.

Stefan Kahrs. Red-black trees with types. *Journal of Functional Programming*, 11(4):425–432, 2001.

Fairouz Kamareddine. Reviewing the classical and the de bruijn notation for λ-calculus and pure type systems. *Journal of Logic and Computation*, 11, 2001.

Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Sydney, March 2004.

Gerwin Klein and Martin Strecker. Verified Bytecode Verification and type-certifying Compilation. *Journal of Logic and Algebraic Programming*, 58(1–2):27–60, 2004.

Donald E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)*. Addison-Wesley Professional, November 1997. ISBN 0201896842.

Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Symposium on Principles of Programming Languages*, pages 42–54, New York, NY, USA, January 2006. ACM Press.

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

Nathan Linger and Tim Sheard. Programming with static invariants in Omega. Unpublished, 2004.

Conor McBride and James McKinna. Functional pearl: I am not a number–I am a free variable. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 1–9, New York, NY, USA, 2004. ACM.

Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *POPL'96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 271–283, New York, NY, USA, 1996.

E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

J Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5:461–492, 1989.

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997.

George C. Necula and Peter Lee. Proof-carrying code. Technical Report CMU-CS-96-165, Carnegie Mellon University, 1996.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

Peter Ørbæk and Jens Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7 (6):557–591, 1997.

Emir Pasalic. *The Role of Type Equality in Meta-Programming*. PhD thesis, Oregon Health and Sciences University, The OGI School of Science and Engineering, 2004.

Christine Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In *TLCA '93: Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 328–345, London, UK, 1993. Springer-Verlag.

Simon Peyton-Jones et al. The Haskell Prime Report. Working Draft, 2007.

F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 199–208, New York, NY, USA, 1988. ACM Press.

Frank Pfenning and Carsten Schurmann. System description: Twelf — a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16*, pages 202–206. Springer-Verlag LNAI, 1999.

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

Brigitte Pientka and Joshua Dunfield. Programming with proofs and explicit contexts. In *Symposium on Principles and Practice of Declarative Programming*, 2008.

Andrew M. Pitts. Nominal logic: A first order theory of names and binding. *Lecture Notes in Computer Science*, 2215:219+, 2001.

François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization. *SIGPLAN Not.*, 39 (1):89–98, 2004. ISSN 0362-1340.

Tom Schrijvers, Martin Sulzmann, Simon Peyton Jones, and Manuel M. T. Chakravarty. Towards open type functions for Haskell. Presented at IFL 2007, 2007.

Tom Schrijvers, Louis-Julien Guillemette, and Stefan Monnier. Type invariants for Haskell. In *PLPV '09: Proceedings of the 3rd workshop on Programming languages meets program verification*, pages 39–48, New York, NY, USA, 2008. ACM.

Zhong Shao. Flexible representation analysis. In *International Conference on Functional Programming*, pages 85–98. ACM Press, June 1997a.

Zhong Shao. An overview of the FLINT/ML compiler. In *International Workshop on Types in Compilation*, June 1997b.

Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Symposium on Programming Languages Design and Implementation*, pages 116–129, La Jolla, CA, June 1995. ACM Press.

Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. In *Symposium on Principles of Programming Languages*, pages 217–232, January 2002.

Leon Shapiro and Ehud Y. Sterling. *The Art of PROLOG: Advanced Programming Techniques*. The MIT Press, 1994.

Tim Sheard. Languages of the future. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 116–119, New York, NY, USA, 2004. ACM Press.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-605-6.

M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 263–274. ACM Press, August 2003.

Martin Strecker. Formal verification of a Java compiler in Isabelle. In *CADE-18: Proceedings of the 18th International Conference on Automated Deduction*, pages 63–77, London, UK, 2002. Springer-Verlag. ISBN 3-540-43931-5.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *Types in Language Design and Implementation*, January 2007.

David Tarditi, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Symposium on Programming Languages Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996. ACM Press.

Ye Henry Tian. Mechanically verifying correctness of cps compilation. In *CATS '06: Proceedings of the 12th Computing: The Australasian Theroy Symposium*, pages 41–51, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

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

Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 249–262, Uppsala, Sweden, August 2003. ACM SIGPLAN.

Hongwei Xi and Dana Scott. Dependent types in practical programming. In *In Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227. ACM Press, 1999.

Appendix A

Source code

This appendix gives the full code listing of the compiler, consisting of 3486 lines of Haskell code. The source files are listed below, with references to relevant sections in the text.

Source file	Page	Description	Reference
Tp.hs	148	Global type definitions	
Src.hs	156	Source language ($\lambda_{\rightarrow}$)	Section 3.2
LK.hs	158	Higher-order encoding of the CPS language ($\lambda_{\mathcal{K}}$)	Section 4.1
CPS.hs	161	CPS conversion	Section 4.2, 4.3, 4.4
LKb.hs	166	First-order encoding of the CPS language ($\lambda^{b}_{\mathcal{K}}$)	Section 5.1, 5.3
ToB.hs	170	Conversion to de Bruijn indices	Section 5.2, 5.3
LC.hs	177	Closure-converted language ($\lambda_{\mathcal{C}}$)	Section 6.2
CC.hs	179	Closure conversion	Section 6.3, 6.4, 6.5
LH.hs	198	Linearized language ($\lambda_{\mathcal{H}}$)	Section 7.1
Hoist.hs	200	Function hoisting phase	Section 7.2
TAL.hs	208	Typed assembly language (TAL)	Section 8.1
CG.hs	210	Code generation	Section 8.2
Main.hs	221	Compiler driver	

Tp.hs

```
1  {-# OPTIONS -fglasgow-exts  -XUndecidableInstances #-} {-
2
3    Global type definitions
4
5  -}
6
7  module Tp (
8    RecT(..), Name, EnvRep, TypeRep(..), rPair,
9    typesEqual, envEqual, nat_eq,
10   Equiv(..), PrimOp(..), Cont, Code,
11   Closed, Exists,
12   S, Z,
13   V, Void, Tup, NatRep(..),
14   Shift, ShiftEnv, shift_tr, shift_env,
15
16   Index(..), MapT(..), lookupT, updateT, mapT, i0, i1, i2, i3, i4,
17
18   All, Var, VarI,
19   Subst, SubstEnv, U, Pred, CMP, Add, Uenv,
20
21   cmpT, cmpV, addT, uT, uEnv, predT, substT, substEnv,
22
23   tr_index_shift, nat_to_int,
24
25   Cat, catT, newIndex, cat_nil, weaken_index, mapAppend,
26   lemma_cat_assoc
27 ) where
28
29
30 -------------------------------------------------------------------------------
31 -- Syntax and object-level types
32
33 -- fixed point operator
34 data RecT a b t = Roll (a (RecT a b t))
35                 | Place (b t)
36
37 -- arithmetic operators
38 data PrimOp where
39   Add  :: PrimOp
40   Sub  :: PrimOp
41   Mult :: PrimOp
42
43 -- used to distinguish values from expressions (in CPS)
44 data V a
45 data Void
46
47 -- type constructors used to encode object-level types
48 data All s
49 data Cont i t
50 data Exists t
51 data Var i     -- a de Bruijn index
```

```
52  data VarI i    -- a reverse de Bruijn index
53  data Tup t
54  data Closed t
55  data Code k t
56
57  -- natural numbers
58  data Z
59  data S i
60
61  -- identifiers
62  type Name = String
63
64  ------------------------------------------------------------------------------
65  -- Singleton types
66
67  data TypeRep t where
68     Rint    :: TypeRep Int
69     Rarw    :: TypeRep t1 -> TypeRep t2 -> TypeRep (t1 -> t2)
70     Rcont   :: NatRep i -> TypeRep t -> TypeRep (Cont i t)
71     Rtup    :: EnvRep t -> TypeRep (Tup t)
72     Rall    :: TypeRep u -> TypeRep (All u)
73     Rvar    :: NatRep i -> TypeRep (Var i)
74     RvarI   :: NatRep i -> TypeRep (VarI i)
75     Rexists :: TypeRep u -> TypeRep (Exists u)
76     Rpair   :: TypeRep a -> TypeRep b -> TypeRep (a, b)
77     Rclosed :: TypeRep t -> TypeRep (Closed t)
78     Rcode   :: NatRep k -> EnvRep t -> TypeRep (Code k t)
79
80  data NatRep n where
81   Nz :: NatRep Z
82   Ns :: NatRep s -> NatRep (S s)
83
84  type EnvRep ts = MapT TypeRep ts
85
86  rPair r1 r2 = Rtup (Ms r1 (Ms r2 M0))
87
88  ------------------------------------------------------------------------------
89  -- Proof objects
90
91  data Equiv s t where
92    Equiv :: (s ~ t) => Equiv s t
93
94
95  ------------------------------------------------------------------------------
96  -- Type families
97
98  type family Subst t a i
99  type instance Subst (All t)       a i = All (Subst t a (S i))
100 type instance Subst (Var n)       a i = CMP n i (Var n)
101                                                 (U Z i a)
102                                                 (Var (Pred n))
103 type instance Subst (VarI n)      a i = VarI n
```

```
104  type instance Subst (s -> t)      a i = (Subst s a i) -> (Subst t a i)
105  type instance Subst (Cont k t)    a i = Cont k (Subst t a (Add i k))
106
107  type instance Subst (Exists t)    a i = Exists (Subst t a (S i))
108  type instance Subst (Tup t)       a i = Tup (SubstEnv t a i)
109
110  type instance Subst (s, t)        a i = (Subst s a i, Subst t a i)
111
112  type instance Subst Int           a i = Int
113  type instance Subst (Closed t)    a i = Closed t
114  type instance Subst (Code k t)    a i = Code k (SubstEnv t a (Add i k))
115
116  type family U k i a
117  type instance U k i (All a)     = All (U (S k) i a)
118  type instance U k i (Cont k' t) = Cont k' (U (Add k k') i t)
119  type instance U k i (Exists t)  = Exists (U (S k) i t)
120  type instance U k i (Var n)     = Var (CMP n k
121                                            n
122                                            (Add i n)
123                                            (Add i n))
124  type instance U k i (VarI n)    = VarI n
125  type instance U k i (s -> t)    = (U k i s) -> (U k i t)
126  type instance U k i (s, t)      = (U k i s, U k i t)
127  type instance U k i Int         = Int
128  type instance U k i (Tup t)     = Tup (Uenv k i t)
129  type instance U k i (Closed t)  = Closed t
130  type instance U k i (Code k' t) = Code k' (Uenv (Add k k') i t)
131
132  type family Shift t
133  type instance Shift t = U Z (S Z) t
134
135  type ShiftEnv ts = Uenv Z (S Z) ts
136
137  type family Uenv k i ts
138  type instance Uenv k i () = ()
139  type instance Uenv k i (s, ts) = (U k i s, Uenv k i ts)
140
141
142  type family CMP a b lt eq gt
143  -- base cases
144  type instance CMP Z Z lt eq gt     = eq
145  type instance CMP Z (S t) lt eq gt = lt
146  type instance CMP (S t) Z lt eq gt = gt
147  -- congruence
148  type instance CMP (S s) (S t) lt eq gt = CMP s t lt eq gt
149
150  type family Add i j
151  type instance Add Z i = i
152  type instance Add (S i) i' = S (Add i i')
153
154  type family Pred n
155  type instance Pred (S i) = i
```

```
156
157  type family SubstEnv t a i
158  type instance SubstEnv (s, ts)    a i = (Subst s a i, SubstEnv ts a i)
159  type instance SubstEnv ()         a i = ()
160
161
162
163  ------------------------------------------------------------------------------
164  -- Utility functions
165
166  -- comparison of type representatives
167  typesEqual :: TypeRep t1 -> TypeRep t2 -> Maybe (Equiv t1 t2)
168  typesEqual Rint Rint = Just Equiv
169  typesEqual (Rarw (t1_r::TypeRep t1) (t2_r::TypeRep t2))
170        (Rarw (t1'_r::TypeRep t1') (t2'_r::TypeRep t2')) =
171      case (typesEqual t1_r t1'_r, typesEqual t2_r t2'_r) of
172        (Just Equiv, Just Equiv) -> Just Equiv
173        _ -> Nothing
174  typesEqual (Rpair (t1_r::TypeRep t1) (t2_r::TypeRep t2))
175        (Rpair (t1'_r::TypeRep t1') (t2'_r::TypeRep t2')) =
176      case (typesEqual t1_r t1'_r, typesEqual t2_r t2'_r) of
177        (Just Equiv, Just Equiv) -> Just Equiv
178        _ -> Nothing
179  typesEqual (Rcont i a) (Rcont j b) =
180     case (typesEqual a b, nat_eq i j) of
181       (Just Equiv, Just Equiv) -> Just Equiv
182       _ -> Nothing
183  typesEqual (Rcode i a) (Rcode j b) =
184     case (envEqual a b, nat_eq i j) of
185       (Just Equiv, Just Equiv) -> Just Equiv
186       _ -> Nothing
187  typesEqual (Rvar v1) (Rvar v2) =
188    case nat_eq v1 v2 of
189      Just Equiv -> Just Equiv
190      _ -> Nothing
191  typesEqual (RvarI v1) (RvarI v2) =
192    case nat_eq v1 v2 of
193      Just Equiv -> Just Equiv
194      _ -> Nothing
195  typesEqual (Rall s) (Rall t) =
196    case typesEqual s t of
197      Just Equiv -> Just Equiv
198      _ -> Nothing
199  typesEqual (Rexists s) (Rexists t) =
200    case typesEqual s t of
201      Just Equiv -> Just Equiv
202      _ -> Nothing
203  typesEqual (Rtup s) (Rtup t) =
204    case envEqual s t of
205      Just Equiv -> Just Equiv
206      _ -> Nothing
207  typesEqual a b = Nothing
```

```
208
209 envEqual :: EnvRep ts -> EnvRep ts' -> Maybe (Equiv ts ts')
210 envEqual M0 M0 = Just Equiv
211 envEqual (Ms s tb) (Ms s' tb') =
212   case typesEqual s s' of
213     Nothing -> Nothing
214     Just Equiv ->
215       case envEqual tb tb' of
216         Just Equiv -> Just Equiv
217         Nothing -> Nothing
218
219 nat_eq :: NatRep i -> NatRep j -> Maybe (Equiv i j)
220 nat_eq (Ns i) (Ns j) =
221   case nat_eq i j of
222     Just Equiv -> Just Equiv
223     Nothing -> Nothing
224 nat_eq Nz Nz = Just Equiv
225 nat_eq (Ns _) Nz = Nothing
226 nat_eq Nz (Ns _) = Nothing
227
228 shift_tr :: TypeRep s -> TypeRep (Shift s)
229 shift_tr t_r = uT Nz (Ns Nz) t_r
230
231 shift_env :: EnvRep ts -> EnvRep (ShiftEnv ts)
232 shift_env M0 = M0
233 shift_env (Ms t_r ts_r) = Ms (shift_tr t_r) (shift_env ts_r)
234
235 nat_to_int :: NatRep i -> Int
236 nat_to_int Nz = 0
237 nat_to_int (Ns i) = 1 + nat_to_int i
238
239
240 --------------------------------------------------------------------------------
241 -- Indices and maps
242
243 -- typed indices
244 data Index ts t where
245   I0 :: Index (t, ts) t
246   Ix :: Index ts t -> Index (t0, ts) t
247
248 i0 = I0
249 i1 = Ix i0
250 i2 = Ix i1
251 i3 = Ix i2
252 i4 = Ix i3
253
254 data MapT ts c where
255   M0 :: MapT c ()
256   Ms :: c s -> MapT c ts -> MapT c (s, ts)
257
258 lookupT :: MapT c ts -> Index ts t -> c t
259 lookupT (Ms e _) I0 = e
```

```
260 lookupT (Ms _ m) (Ix i) = lookupT m i
261
262 updateT :: MapT c ts -> Index ts s -> c s -> MapT c ts
263 updateT (Ms _ t) I0 c'     = Ms c' t
264 updateT (Ms c t) (Ix i) c' = Ms c (updateT t i c')
265
266 mapT :: (forall t . c t -> d t) -> MapT c ts -> MapT d ts
267 mapT f M0 = M0
268 mapT f (Ms e tb) = Ms (f e) (mapT f tb)
269
270
271 -------------------------------------------------------------------------------
272 -- Type families reified as term-level functions
273
274 substT :: TypeRep s -> TypeRep t ->  NatRep i -> TypeRep (Subst s t i)
275 substT (Rall t) a i = Rall (substT t a (Ns i))
276 substT (Rexists t) a i = Rexists (substT t a (Ns i))
277 substT (Rvar n) a i = cmpT n i (Rvar n) (uT Nz i a) (Rvar (predT n))
278 substT (RvarI n) a i = RvarI n
279 substT (Rarw s t) a i  = Rarw (substT s a i) (substT t a i)
280 substT Rint a i        = Rint
281 substT (Rpair s t) a i  = Rpair (substT s a i) (substT t a i)
282 substT (Rcont k t) a i = Rcont k (substT t a (addT i k))
283 substT (Rclosed t) a i = Rclosed t
284 substT (Rtup t) a i = Rtup (substEnv t a i)
285 substT (Rcode k t) a i = Rcode k (substEnv t a (addT i k))
286
287 uT :: NatRep k -> NatRep i -> TypeRep a -> TypeRep (U k i a)
288 uT k i (Rall s) = Rall (uT (Ns k) i s)
289 uT k i (Rexists s) = Rexists (uT (Ns k) i s)
290 uT k i (Rvar n) = Rvar   (cmpV n k n (addT i n) (addT i n))
291 uT k i (RvarI n) = RvarI n
292 uT k i (Rarw s t)  = Rarw (uT k i s) (uT k i t)
293 uT k i (Rpair s t)  = Rpair (uT k i s) (uT k i t)
294 uT k i Rint  = Rint
295 uT k i (Rcont i_r t_r) = Rcont i_r (uT (addT k i_r) i t_r)
296 uT k i (Rtup t)    = Rtup (uEnv k i t)
297 uT k i (Rcode i_r t)   = Rcode i_r (uEnv (addT k i_r) i t)
298 uT k i (Rclosed t) = Rclosed t
299
300 uEnv :: NatRep k -> NatRep i -> EnvRep a -> EnvRep (Uenv k i a)
301 uEnv k i M0 = M0
302 uEnv k i (Ms t tup) = Ms (uT k i t) (uEnv k i tup)
303
304 cmpT :: NatRep a -> NatRep b -> TypeRep lt -> TypeRep eq -> TypeRep gt
305          -> TypeRep (CMP a b lt eq gt)
306 cmpT Nz Nz _ eq _ = eq
307 cmpT Nz (Ns _) lt _ _ = lt
308 cmpT (Ns _) Nz _ _ gt = gt
309 cmpT (Ns i) (Ns i') lt eq gt = cmpT i i' lt eq gt
310
311 cmpV :: NatRep a -> NatRep b -> NatRep lt -> NatRep eq -> NatRep gt
```

```
312          -> NatRep (CMP a b lt eq gt)
313 cmpV Nz Nz _ eq _ = eq
314 cmpV Nz (Ns _) lt _ _ = lt
315 cmpV (Ns _) Nz _ _ gt = gt
316 cmpV (Ns i) (Ns i') lt eq gt = cmpV i i' lt eq gt
317
318 addT :: NatRep a -> NatRep b -> NatRep (Add a b)
319 addT Nz i      = i
320 addT (Ns i) i' = Ns (addT i i')
321
322 predT :: NatRep i -> NatRep (Pred i)
323 predT (Ns i) = i
324
325 substEnv :: EnvRep s -> TypeRep t ->  NatRep i -> EnvRep (SubstEnv s t i)
326 substEnv M0 _ _ = M0
327 substEnv (Ms t tup) a i = Ms (substT t a i) (substEnv tup a i)
328
329 tr_index_shift :: Index ts t -> Index (ShiftEnv ts) (Shift t)
330 tr_index_shift I0 = I0
331 tr_index_shift (Ix i) = Ix (tr_index_shift i)
332
333
334 ----------------------------------------------------------------------------
335 -- Concatenation of type environments
336
337 type family Cat ts0 ts
338 type instance Cat ()      ts' = ts'
339 type instance Cat (s, ts) ts' = (s, Cat ts ts')
340
341 catT :: EnvRep ts0 -> EnvRep ts -> EnvRep (Cat ts0 ts)
342 catT M0             ts_r = ts_r
343 catT (Ms s_r ts0_r) ts_r = Ms s_r (catT ts0_r ts_r)
344
345 weaken_index :: EnvRep ts -> Index ts0 t -> Index (Cat ts0 ts) t
346 weaken_index _ I0 = I0
347 weaken_index ts_r (Ix i) = Ix (weaken_index ts_r i)
348
349 cat_nil :: EnvRep ts -> Equiv (Cat ts ()) ts
350 cat_nil M0 = Equiv
351 cat_nil (Ms _ ts_r) = case cat_nil ts_r of Equiv -> Equiv
352
353 newIndex :: EnvRep ts -> Index (Cat ts (s, ())) s
354 newIndex M0 = I0
355 newIndex (Ms _ ts_r) = Ix (newIndex ts_r)
356
357 mapAppend :: MapT c ts -> c t -> MapT c (Cat ts (t, ()))
358 mapAppend M0 v = Ms v M0
359 mapAppend (Ms v0 m) v = Ms v0 (mapAppend m v)
360
361 lemma_cat_assoc :: EnvRep hs0 -> EnvRep hs1 -> EnvRep hs2
362               -> Equiv (Cat hs0 (Cat hs1 hs2))
363                        (Cat (Cat hs0 hs1) hs2)
```

```
364 lemma_cat_assoc hs0_r hs1_r hs2_r =
365   case envEqual (catT hs0_r (catT hs1_r hs2_r))
366                  (catT (catT hs0_r hs1_r) hs2_r) of
367     Just Equiv -> Equiv
368
```

Src.hs

```
1  {-# OPTIONS -fglasgow-exts #-} {-
2
3     Source language
4
5  -}
6
7  module Src (
8    ExpF(..), Exp, iter0,
9    let_, letrec_, fun, app, tp_abs, tp_app,
10   num, if0, add, sub, mult, times, minus
11 ) where
12
13 import Tp
14
15 --------------------------------------------------------------------------------
16 -- Source language
17
18 {- NOTE: The TypeRep arguments to the constructors other than Tp_abs
19       and Tp_app are not required for the CPS conversion, but only for
20       the subsequent code transformations.
21 -}
22
23 data ExpF a where
24   Tp_abs  :: TypeRep s ->
25              (forall t. TypeRep t -> a (Subst s t Z)) -> ExpF (a (All s))
26   Tp_app  :: TypeRep s -> TypeRep t -> a (All s)      -> ExpF (a (Subst s t Z))
27
28   Let     :: Name -> TypeRep t1 ->
29              a t1 -> (a t1 -> a t2)                   -> ExpF (a t2)
30
31   Letrec  :: Name -> Name -> TypeRep s -> TypeRep t ->
32              (a (s -> t) -> a s -> a t) ->
33              (a (s -> t) -> a u)                      -> ExpF (a u)
34
35   App     :: TypeRep t1 -> TypeRep t2 ->
36              a (t1 -> t2) -> a t1                     -> ExpF (a t2)
37
38   Pair    :: TypeRep t1 -> TypeRep t2 -> a t1 -> a t2 -> ExpF (a (t1, t2))
39   Fst     :: TypeRep t1 -> a (t1, t2)                 -> ExpF (a t1)
40   Snd     :: TypeRep t2 -> a (t1, t2)                 -> ExpF (a t2)
41
42   Num     :: Int                                      -> ExpF (a Int)
43   Prim    :: PrimOp -> a Int -> a Int                 -> ExpF (a Int)
44   If0     :: a Int -> a t -> a t                      -> ExpF (a t)
45
46 type Exp a t = RecT ExpF a t
47
48
49 --------------------------------------------------------------------------------
50 -- HOAS boilerplate code
51
```

```
52  xmapExpF  ::   (forall t. a t -> b t)
53              -> (forall t. b t -> a t)
54              -> (forall t. (ExpF (a t) -> ExpF (b t)))
55  xmapExpF (f::forall t. a t -> b t) g x =
56    case x of
57      Tp_abs s_r x -> Tp_abs s_r (\t_r ->  f (x t_r))
58      Tp_app s_r t_r x -> Tp_app s_r t_r (f x)
59
60      Let n s_r v x -> Let n s_r (f v) (f . x . g)
61      Letrec s_r t_r f_n x_n x e -> Letrec s_r t_r f_n x_n
62                                     (\ a b -> f (x (g a) (g b)))
63                                     (f . e . g)
64      App s_r t_r x y   -> App s_r t_r (f x) (f y)
65
66      Pair s_r t_r a b -> Pair s_r t_r (f a) (f b)
67      Fst s_r a        -> Fst s_r (f a)
68      Snd s_r a        -> Snd s_r (f a)
69
70      Num i        -> Num i
71      Prim op x y  -> Prim op (f x) (f y)
72      If0 x y z    -> If0 (f x) (f y) (f z)
73
74
75  cata  ::    (forall t. (ExpF (a t) -> a t))
76           -> (forall t. Exp a t -> a t)
77  cata f (Roll x) =
78     f ((xmapExpF (cata f) Place) x)
79  cata f (Place x) = x
80
81  iter0 ::  (forall t. ExpF (b t) -> b t)
82         -> (forall t. ((forall a. Exp a t) -> b t))
83  iter0 proto x = cata proto x
84
85  -- constructors
86  let_ n s_r v x = Roll (Let n s_r v x)
87  letrec_ f_n x_n s_r t_r f e = Roll (Letrec f_n x_n s_r t_r f e)
88  fun f_n x_n s_r t_r f = Roll (Letrec f_n x_n s_r t_r (\_ x -> f x) (\f -> f))
89  app s_r t_r x y = Roll (App s_r t_r x y)
90  num a = Roll (Num a)
91  prim op a b = Roll (Prim op a b)
92  if0 x y z = Roll (If0 x y z)
93  tp_abs ::    TypeRep s
94            -> (forall t . TypeRep t -> Exp a (Subst s t Z))
95            -> Exp a (All s)
96  tp_abs s_r e = Roll (Tp_abs s_r e)
97  tp_app :: TypeRep s -> TypeRep t -> Exp a (All s) -> Exp a (Subst s t Z)
98  tp_app s_r t_r e = Roll (Tp_app s_r t_r e)
99  add a b = prim Add a b
100 sub a b = prim Sub a b
101 mult a b = prim Mult a b
102 minus = prim Sub
103 times = prim Mult
```

LK.hs

```
1  {-# OPTIONS -XGADTs -XRankNTypes -XScopedTypeVariables -XPatternSignatures #-}
2  {-
3
4     CPS language
5
6  -}
7
8  module LK (
9    ExpKF(..), ExpK, ValK,
10   numK, letK, letrecK, appK, if0K, haltK, cataK, iter0K,
11   let_pairK, let_fstK, let_sndK, let_primK,
12   let_poly_funK, poly_appK
13 ) where
14
15 import Tp
16
17
18 --------------------------------------------------------------------------------
19 -- CPS language
20
21 data ExpKF a where
22   -- values
23   KVnum     :: Int                                  -> ExpKF (a (V Int))
24
25   -- expressions
26   Kletrec  :: Name -> Name -> TypeRep s
27                 -> (a (V (Cont Z s)) -> a (V s) -> a Void)
28                 -> (a (V (Cont Z s)) -> a Void)
29                 -> ExpKF (a Void)
30   Klet_poly_fun :: TypeRep s
31                 -> (forall t. TypeRep t -> a (V (Subst s t Z)) -> a Void)
32                 -> (a (V (Cont (S Z) s)) -> a Void)
33                 -> ExpKF (a Void)
34
35   Klet      :: Name -> TypeRep t
36                 -> a (V t) -> (a (V t) -> a Void)    -> ExpKF (a Void)
37   Klet_pair :: Name -> TypeRep t1 -> TypeRep t2 ->
38                        a (V t1) -> a (V t2) -> (a (V (t1, t2)) -> a Void)
39                                                      -> ExpKF (a Void)
40   Klet_fst  :: Name -> TypeRep t1 ->
41                  a (V (t1, t2)) -> (a (V t1) -> a Void)
42                                                      -> ExpKF (a Void)
43   Klet_snd  :: Name -> TypeRep t2 ->
44                a (V (t1, t2)) -> (a (V t2) -> a Void)
45                                                      -> ExpKF (a Void)
46   Klet_prim :: Name -> PrimOp -> a (V Int) -> a (V Int) -> (a (V Int) -> a Void)
47                                                      -> ExpKF (a Void)
48
49   Kapp      :: TypeRep s -> a (V (Cont Z s)) -> a (V s) -> ExpKF (a Void)
50   Kpoly_app :: TypeRep s -> TypeRep t
51                 -> a (V (Cont (S Z) s))
```

```
52                  -> a (V (Subst s t Z))
53                  -> ExpKF (a Void)
54    Kif0      :: a (V Int) -> a Void -> a Void         -> ExpKF (a Void)
55
56    Khalt     :: a (V t)                               -> ExpKF (a Void)
57
58
59  type ValK a t = RecT ExpKF a (V t)
60  type ExpK a = RecT ExpKF a Void
61
62
63  --------------------------------------------------------------------------------
64  -- HOAS boilerplate
65
66  xmapExpKF ::
67          (forall t. a t -> b t)
68       -> (forall t. b t -> a t)
69       -> (forall t. (ExpKF (a t) -> ExpKF (b t)))
70  xmapExpKF (f::forall t. a t -> b t) g x =
71    case x of
72      Klet n s_r v x           -> Klet n s_r (f v) (f . x . g)
73      Kletrec f_n x_n s_r x e  -> Kletrec f_n x_n s_r
74                                          (\ a b -> f (x (g a) (g b)))
75                                          (f . e . g)
76      Kpoly_app s_r t_r v w    -> Kpoly_app s_r t_r (f v) (f w)
77      Klet_poly_fun s_r v e    -> Klet_poly_fun s_r (\a b -> f (v a (g b)))
78                                                 (f . e . g)
79      Kapp s_r v w             -> Kapp s_r (f v) (f w)
80
81      Klet_pair n s_r t_r v1 v2 x -> Klet_pair n s_r t_r (f v1) (f v2) (f . x . g)
82      Klet_fst n s_r v x       -> Klet_fst n s_r (f v) (f . x . g)
83      Klet_snd n s_r v x       -> Klet_snd n s_r (f v) (f . x . g)
84
85      KVnum i                  -> KVnum i
86      Klet_prim n op v w x     -> Klet_prim n op (f v) (f w) (f . x . g)
87      Kif0 v x y               -> Kif0 (f v) (f x) (f y)
88
89      Khalt v                  -> Khalt (f v)
90
91
92  cataK  ::    (forall t. (ExpKF (a t) -> a t))
93            -> (forall t. RecT ExpKF a t -> a t)
94  cataK f (Roll x) =
95     f ((xmapExpKF (cataK f) Place) x)
96  cataK f (Place x) = x
97
98  iterOK ::  (forall t. ExpKF (b t) -> b t)
99          -> (forall t. ((forall a. RecT ExpKF a t) -> b t))
100 iterOK proto x = cataK proto x
101
102
103 numK a = Roll (KVnum a)
```

```
104 appK s_r v1 v2 = Roll (Kapp s_r v1 v2)
105 letK n s_r v e = Roll (Klet n s_r v e)
106 let_fstK n s_r v e = Roll (Klet_fst n s_r v e)
107 let_sndK n s_r v e = Roll (Klet_snd n s_r v e)
108 let_pairK n s_r t_r v1 v2 e = Roll (Klet_pair n s_r t_r v1 v2 e)
109 letrecK f_n x_n s_r f body =  Roll (Kletrec f_n x_n s_r f body)
110
111 let_primK n op v1 v2 e = Roll (Klet_prim n op v1 v2 e)
112 if0K v e1 e2 = Roll (Kif0 v e1 e2)
113 haltK v = Roll (Khalt v)
114
115 poly_appK s_r t_r f x = Roll (Kpoly_app s_r t_r f  x)
116
117 let_poly_funK ::
118      TypeRep s
119   -> (forall t. TypeRep t -> ValK a (Subst s t Z) -> ExpK a)
120   -> (ValK a (Cont (S Z) s) -> ExpK a)
121   -> ExpK a
122 let_poly_funK s_r e x = Roll (Klet_poly_fun s_r e x)
123
```

CPS.hs

```
1  {-# OPTIONS -fglasgow-exts -XUndecidableInstances #-} {-
2
3     CPS conversion
4
5  -}
6
7  module CPS (
8    cps
9  ) where
10
11 import Tp
12 import Src
13 import LK
14
15
16 --------------------------------------------------------------------------------
17 -- CPS conversion [TYPES]
18
19 -- Ktype is the type translation for CPS conversion
20 type family Ktype t
21 type instance Ktype (s -> t) = Cont Z (Ktype s, Cont Z (Ktype t))
22 type instance Ktype (All u)  = Cont (S Z) (Cont Z (Ktype u))
23 type instance Ktype (Var i)  = Var i
24 type instance Ktype (VarI i) = VarI i
25 type instance Ktype Int      = Int
26 type instance Ktype (a, b)   = (Ktype a, Ktype b)
27
28 -- UnKtype is the inverse of Ktype
29 type family UnKtype t
30 type instance UnKtype (Cont Z (s, Cont Z t))  = (UnKtype s) -> (UnKtype t)
31 type instance UnKtype (Cont (S Z) (Cont Z u)) = All (UnKtype u)
32 type instance UnKtype (Var i)                 = Var i
33 type instance UnKtype (VarI i)                = VarI i
34 type instance UnKtype Int                     = Int
35 type instance UnKtype (s, t)                  = (UnKtype s, UnKtype t)
36
37 -- Kenv applies Ktype to every type in a type environment
38 type family Kenv ts
39 type instance Kenv ()      = ()
40 type instance Kenv (s, ts) = (Ktype s, Kenv ts)
41
42
43 ---------- type families reified as term-level functions
44
45 kType :: TypeRep t -> TypeRep (Ktype t)
46 kType (Rarw s t)  = Rcont Nz (Rpair (kType s) (Rcont Nz (kType t)))
47 kType (Rall s_r)  = Rcont (Ns Nz) (Rcont Nz (kType s_r))
48 kType (Rvar i_r)  = Rvar i_r
49 kType (RvarI i_r) = RvarI i_r
50 kType Rint        = Rint
51 kType (Rpair a b) = Rpair (kType a) (kType b)
```

```
52
53  unKtype :: TypeRep t -> TypeRep (UnKtype t)
54  unKtype (Rcont Nz (Rpair s_r (Rcont Nz (t_r)))) = Rarw (unKtype s_r)
55                                                         (unKtype t_r)
56  unKtype (Rcont (Ns Nz) (Rcont Nz s_r)) = Rall (unKtype s_r)
57  unKtype (Rvar i_r)  = Rvar i_r
58  unKtype (RvarI i_r) = RvarI i_r
59  unKtype Rint        = Rint
60  unKtype (Rpair a b) = Rpair (unKtype a) (unKtype b)
61
62  kEnv :: EnvRep ts -> EnvRep (Kenv ts)
63  kEnv M0 = M0
64  kEnv (Ms s_r ts_r) = Ms (kType s_r) (kEnv ts_r)
65
66
67  --------------------------------------------------------------------------------
68  -- CPS conversion [TERMS]
69
70  data CPS a t where
71    -- CPS form of terms
72    CPS ::     ((ValK a (Ktype t) -> ExpK a) -> ExpK a)  -- cps-meta
73            -> ((ValK a (Cont Z (Ktype t))) -> ExpK a)   -- cps-obj
74            -> CPS a t
75
76  meta :: CPS a t -> (ValK a (Ktype t) -> ExpK a)  -> ExpK a
77  meta e = case e of CPS e_meta _ -> e_meta
78
79  obj :: CPS a t -> (ValK a (Cont Z (Ktype t)))  -> ExpK a
80  obj e = case e of CPS _ e_obj -> e_obj
81
82  value :: TypeRep (Ktype t) -> ValK a (Ktype t) -> CPS a t
83  value t_r v = CPS (\k -> k v)           -- cps-meta
84                    (\c -> appK t_r c v)  -- cps-obj
85
86  cpsAux :: forall a t. ExpF (CPS a t) -> CPS a t
87
88  -- values
89  cpsAux (Num n)   = value Rint (numK n)
90
91  -- expressions
92  cpsAux (Letrec f_n x_n s_r t_r f body) =
93    CPS (\k -> letrecK f_n "arg"
94                        (Rpair (kType s_r) (Rcont Nz (kType t_r)))
95                        (\f' -> \xk ->
96                           let_fstK x_n (kType s_r) xk (\x ->
97                           let_sndK "k" (Rcont Nz (kType t_r)) xk (\c ->
98                           meta (f (value (kType (Rarw s_r t_r)) f')
99                                   (value (kType s_r) x))
100                               (\n -> appK (kType t_r) c n))))
101                       (\a -> meta (body (value (kType (Rarw s_r t_r)) a)) k))
102       (\c -> letrecK f_n "arg"
103                       (Rpair (kType s_r) (Rcont Nz (kType t_r)))
```

```
104                      (\f' -> \xk ->
105                         let_fstK x_n (kType s_r) xk (\x ->
106                         let_sndK "k" (Rcont Nz (kType t_r)) xk (\c ->
107                         meta (f (value (kType (Rarw s_r t_r)) f')
108                                 (value (kType s_r) x))
109                              (\n -> appK (kType t_r) c n))))
110                      (\a -> obj (body (value (kType (Rarw s_r t_r)) a)) c))
111
112  cpsAux (Let n (s_r::TypeRep s) v x) =
113    CPS (\k ->
114          (meta v (\v1 ->
115                     letK n (kType s_r) v1
116                              (\r ->
117                                 meta (x (value (kType s_r) r)) k))))
118        (\c ->
119          (meta v (\v1 ->
120                     letK n (kType s_r) v1
121                              (\r ->
122                                 obj (x (value (kType s_r) r)) c))))
123
124  cpsAux (App s_r t_r e1 e2) =
125    let appexp c = meta e1 (\y1 ->
126                     meta e2 (\y2 ->
127                       let_pairK "p" (kType s_r) (Rcont Nz (kType t_r))
128                                     y2 c $ (\p ->
129                       appK (Rpair (kType s_r) (Rcont Nz (kType t_r)))
130                            y1 p)))
131    in CPS (\k -> letrecK "k" "a" (kType t_r)
132                     (\_ a -> k a)
133                     (\c -> appexp c))
134           (\c -> appexp c)
135
136  cpsAux (Prim op a b) =
137    CPS (\k -> (meta a (\v1 -> meta b (\v2 ->
138           let_primK "r" op v1 v2 k))))
139        (\c -> (meta a (\v1 -> meta b (\v2 ->
140           let_primK "r" op v1 v2 (\k -> appK Rint c k)))))
141
142  cpsAux (Pair s_r t_r a b) =
143    CPS (\k -> (meta a (\v1 -> meta b (\v2 ->
144           let_pairK "p" (kType s_r) (kType t_r) v1 v2 k))))
145        (\c -> (meta a (\v1 -> meta b (\v2 ->
146           let_pairK "p" (kType s_r) (kType t_r) v1 v2
147                     (\k -> appK (kType (Rpair s_r t_r)) c k)))))
148
149  cpsAux (Fst s_r a) =
150    CPS (\k -> (meta a (\v1 ->
151           let_fstK "r" (kType s_r) v1 k)))
152        (\c -> (meta a (\v1 ->
153           let_fstK "r" (kType s_r) v1 (\k -> appK (kType s_r) c k))))
154
155  cpsAux (Snd s_r a) =
```

```
156    CPS (\k -> (meta a (\v1 ->
157           let_sndK "r" (kType s_r) v1 k)))
158        (\c -> (meta a (\v1 ->
159           let_sndK "r" (kType s_r) v1 (\k -> appK (kType s_r) c k))))
160
161  cpsAux (If0 a e1 e2) =
162    CPS (\k -> meta a (\v1 -> if0K v1 (meta e1 k) (meta e2 k)))
163        (\c -> meta a (\v1 -> if0K v1 (obj e1 c) (obj e2 c)))
164
165  cpsAux (Tp_abs (u_r::TypeRep u) e) =
166    CPS (\k ->
167            let_poly_funK
168                (Rcont Nz (kType u_r))
169                (\(t_r::TypeRep t') ->
170                    \c -> obj (e (unKtype t_r))
171                              (case lemma_ktype_inverse t_r of
172                                 Equiv ->
173                                    case lemma_ktype_subst u_r (unKtype t_r) of
174                                      Equiv -> c))
175                k)
176        (\c ->
177            let_poly_funK
178                (Rcont Nz (kType u_r))
179                (\(t_r::TypeRep t') ->
180                    \c -> obj (e (unKtype t_r))
181                              (case lemma_ktype_inverse t_r of
182                                 Equiv ->
183                                    case lemma_ktype_subst u_r (unKtype t_r) of
184                                      Equiv -> c))
185                (\v -> appK (kType (Rall u_r)) c v))
186
187
188  cpsAux (Tp_app (s_r::TypeRep s) (t_r :: TypeRep t') e1 ) =
189    case lemma_ktype_subst s_r t_r of
190     Equiv ->
191        CPS (\k ->
192               meta e1 (\x ->
193                          letrecK "" "" (substT (kType s_r) (kType t_r) Nz)
194                                 (\_ x -> k x) (\k' ->
195                          poly_appK (Rcont Nz (kType s_r))  (kType t_r)
196                                    x k')))
197            (\c ->
198               meta e1 (\(x::ValK a (Ktype (All s))) ->
199                              poly_appK (Rcont Nz (kType s_r)) (kType t_r)
200                                         x c))
201
202
203  cps :: (forall a. Exp a t) -> (forall a. ExpK a)
204  cps x =
205    let cpsE :: forall t b . (forall a . Exp a t) -> CPS b t
206        cpsE = iter0 cpsAux
207    in meta ((iter0 cpsAux) x) haltK
```

```
208
209
210 ------------------------------------------------------------------------------
211 -- Lemmas
212
213 lemma_ktype_subst :: TypeRep s -> TypeRep t
214              -> Equiv (Ktype (Subst s t Z))
215                       (Subst (Ktype s) (Ktype t) Z)
216 lemma_ktype_subst s_r t_r =
217   case typesEqual (kType (substT s_r t_r Nz))
218                   (substT (kType s_r) (kType t_r) Nz)
219     of Just Equiv -> Equiv
220
221 lemma_ktype_inverse :: TypeRep cps_t -> Equiv (Ktype (UnKtype cps_t)) cps_t
222 lemma_ktype_inverse cps_t_r =
223    case typesEqual (kType (unKtype cps_t_r)) cps_t_r of
224      Just Equiv -> Equiv
225
```

LKb.hs

```
1  {-# OPTIONS -XGADTs -XTypeFamilies -XUndecidableInstances #-} {-
2
3     First-order representation of the CPS language
4
5  -}
6
7  module LKb (
8    ValKb(..), ExpKb(..), SubstR, ShiftR, ShiftEnvR, Ur, UenvI,
9    shiftR, shiftEnvR, substR, uEnvI
10 ) where
11
12 import Tp
13
14 {- NOTE:
15
16   The implementation uses a single representation (the types ValKb/ExpKb) to
17   represent both
18     (1) the first-order encoding with *reverse* de Bruijn types
19         (the types ValKr/ExpKr from Section 5.1)
20     (2) the first-order encoding with *normal* de Bruijn types
21         (the types ValKb/ExpKb from Section 5.3)
22
23   - for *reverse* de Bruijn indices, we use the constructors:
24        KBlet_poly_funR
25        KBpoly_appR
26   - for *normal* de Bruijn indices, we use the constructors:
27        KBlet_poly_fun
28        KBpoly_app
29   - all other constructors are used for both representations
30
31 -}
32
33 data ValKb tsfs t where
34   Kvar      :: Index ts t                          -> ValKb (i,ts) t
35   Knum      :: Int                                 -> ValKb g Int
36
37 data ExpKb ts where
38   KBletrec  :: Name -> Name -> TypeRep t
39                   -> ExpKb (i, (t, (Cont Z t, ts)))
40                   -> ExpKb (i, (Cont Z t, ts))
41                   -> ExpKb (i, ts)
42   KBlet_poly_fun :: Name -> Name -> TypeRep t
43                   -> ExpKb (S i, (t, ShiftEnv ts))
44                   -> ExpKb (i,   (Cont (S Z) t, ts))
45                   -> ExpKb (i,   ts)
46
47   KBlet      :: Name -> TypeRep s -> ValKb (i, ts) s
48                   -> ExpKb (i, (s, ts))              -> ExpKb (i, ts)
49   KBlet_pair :: Name -> TypeRep s -> TypeRep t
50                   -> ValKb (i, ts) s -> ValKb (i, ts) t
51                   -> ExpKb (i, ((s, t), ts))
```

```
52                    -> ExpKb (i, ts)
53    KBlet_fst  :: Name -> TypeRep t1 -> ValKb (i, ts) (t1, t2)
54                                 -> ExpKb (i, (t1, ts))    -> ExpKb (i, ts)
55    KBlet_snd  :: Name -> TypeRep t2 -> ValKb (i, ts) (t1, t2)
56                                -> ExpKb (i, (t2, ts))    -> ExpKb (i, ts)
57    KBlet_prim :: Name -> PrimOp
58                    -> ValKb (i, ts) Int -> ValKb (i, ts) Int
59                    -> ExpKb (i, (Int, ts))                   -> ExpKb (i, ts)
60
61    KBapp      :: TypeRep s -> ValKb ts (Cont Z s) -> ValKb ts s -> ExpKb ts
62    KBpoly_app :: TypeRep s -> TypeRep t
63                   -> ValKb g (Cont (S Z) s)
64                   -> ValKb g (Subst s t Z)
65                   -> ExpKb g
66
67    KBif0      :: ValKb (i, ts) Int -> ExpKb (i, ts) -> ExpKb (i, ts)
68                                                         -> ExpKb (i, ts)
69    KBhalt     :: ValKb g t                              -> ExpKb g
70
71    -- used only temporarily during conversion to de Bruijn indices
72    KBlet_poly_funR :: Name -> Name -> TypeRep t
73                    -> ExpKb (S i, (t, ShiftEnvR i ts))
74                    -> ExpKb (i,   (Cont (S Z) t, ts))
75                    -> ExpKb (i,   ts)
76
77    KBpoly_appR :: TypeRep s -> TypeRep t
78                   -> ValKb (i, ts) (Cont (S Z) s)
79                   -> ValKb (i, ts) (SubstR s t i)
80                   -> ExpKb (i, ts)
81
82  ----------------------------------------------------------------------------
83  -- Type families for substitution over reverse de Bruijn indices
84
85  type family SubstR s t i
86  type instance SubstR (Cont Z s) t i = Cont Z (SubstR s t i)
87  type instance SubstR (Cont (S Z) s) t i = Cont (S Z) (SubstR s t i)
88  type instance SubstR (VarI n) t i = CMP n i (VarI n)
89                                              t
90                                              (VarI (Pred n))
91  type instance SubstR (s1, s2) t i = (SubstR s1 t i, SubstR s2 t i)
92  type instance SubstR Int t i = Int
93
94
95  type family Ur i k t
96  type instance Ur i k (All t)  = All (Ur i k t)
97  type instance Ur i k (Cont j t)  = Cont j (Ur i k t)
98  type instance Ur i k (VarI n) = VarI (CMP n k n
99                                                (Add i n)
100                                               (Add i n))
101 type instance Ur i k (s1, s2) = (Ur i k s1, Ur i k s2)
102 type instance Ur i k Int = Int
103
```

```
104 type family UenvI i k ts
105 type instance UenvI i k () = ()
106 type instance UenvI i k (s, ts) = (Ur i k s, UenvI i k ts)
107
108
109 -- add one to all indices >= i in t
110 type ShiftR i t = Ur (S Z) i t
111
112 type family ShiftEnvR i t
113 type instance ShiftEnvR i () = ()
114 type instance ShiftEnvR i (s, ts) = (ShiftR i s, ShiftEnvR i ts)
115
116 ------------ type families reified as term-level functions
117
118 uR :: NatRep i -> NatRep k -> TypeRep t -> TypeRep (Ur i k t)
119 uR i_r k_r (Rcont j_r t_r) = Rcont j_r (uR i_r k_r t_r)
120 uR i_r k_r (RvarI n_r) = RvarI (cmpV n_r k_r n_r
121                                              (addT i_r n_r)
122                                              (addT i_r n_r))
123 uR i_r k_r (Rpair s_r t_r) = Rpair (uR i_r k_r s_r)
124                                    (uR i_r k_r t_r)
125 uR i_r k_r Rint = Rint
126
127
128 shiftR :: NatRep i -> TypeRep t -> TypeRep (ShiftR i t)
129 shiftR i_r t_r = uR (Ns Nz) i_r t_r
130
131 shiftEnvR :: NatRep i -> EnvRep t -> EnvRep (ShiftEnvR i t)
132 shiftEnvR i_r M0 = M0
133 shiftEnvR i_r (Ms t_r ts_r) = Ms (shiftR i_r t_r) (shiftEnvR i_r ts_r)
134
135
136 substR :: TypeRep s -> TypeRep t -> NatRep i -> TypeRep (SubstR s t i)
137 substR (Rcont Nz s_r) t_r i_r = Rcont Nz (substR s_r t_r i_r)
138 substR (Rcont (Ns Nz) s_r) t_r i_r = Rcont (Ns Nz) (substR s_r t_r i_r)
139 substR (RvarI n_r) t_r i_r = cmpT n_r i_r
140                                   (RvarI n_r) t_r (RvarI (predT n_r))
141 substR (Rpair s1_r s2_r) t_r i_r = Rpair (substR s1_r t_r i_r)
142                                          (substR s2_r t_r i_r)
143 substR Rint t_r i_r = Rint
144
145
146 uiT :: NatRep i -> NatRep k -> TypeRep t -> TypeRep (Ur i k t)
147 uiT i_r k_r (Rcont j_r t_r) = Rcont j_r (uiT i_r k_r t_r)
148 uiT i_r k_r (Rall t_r) = Rall (uiT i_r k_r t_r)
149 uiT i_r k_r (RvarI n_r) = RvarI (cmpV n_r k_r n_r
150                                              (addT i_r n_r)
151                                              (addT i_r n_r))
152 uiT i_r k_r (Rpair s_r t_r) = Rpair (uiT i_r k_r s_r)
153                                     (uiT i_r k_r t_r)
154 uiT i_r k_r Rint = Rint
155
```

```
156
157 uEnvI :: NatRep i -> NatRep k -> EnvRep t -> EnvRep (UenvI i k t)
158 uEnvI _ _ M0 = M0
159 uEnvI i_r k_r (Ms t_r ts_r) = Ms (uiT i_r k_r t_r) (uEnvI i_r k_r ts_r)
160
```

ToB.hs

```
1  {-# OPTIONS -fglasgow-exts -XUndecidableInstances  #-}{-
2
3     Conversion to de Bruijn indices
4
5  -}
6
7  module ToB (
8     toB
9  ) where
10
11 import Tp
12 import LK
13 import LKb
14
15
16 --------------------------------------------------------------------------------
17 -- Conversion of types to *reverse* de Bruijn
18
19 type family Rtype i t
20 type instance Rtype i (Cont Z t)     = Cont Z (Rtype i t)
21 type instance Rtype i (Cont (S Z) t) = Cont (S Z) (Rtype (S i) t)
22 type instance Rtype i (Var n)        = VarI (Subtract i (S n))
23 type instance Rtype i (VarI n)       = VarI n
24 type instance Rtype i (s, t)         = (Rtype i s, Rtype i t)
25 type instance Rtype i Int            = Int
26
27 type family Renv i ts
28 type instance Renv i () = ()
29 type instance Renv i (s, ts) = (Rtype i s, Renv i ts)
30
31 type family Subtract a b
32 type instance Subtract n Z = n
33 type instance Subtract (S a) (S n) = Subtract a n
34
35
36 --------------------------------------------------------------------------------
37 -- Conversion of types to *normal* de Bruijn
38
39 type family Btype i t
40 type instance Btype i (Cont Z t)= Cont Z (Btype i t)
41 type instance Btype i (Cont (S Z) t) = Cont (S Z) (Btype (S i) t)
42 type instance Btype i Int       = Int
43 type instance Btype i (Var n)   = Var n
44 type instance Btype i (s, t)    = (Btype i s, Btype i t)
45 type instance Btype i (VarI n)  = Var (Subtract i (S n))
46
47 type family Benv i ts
48 type instance Benv i () = ()
49 type instance Benv i (s, ts) = (Btype i s, Benv i ts)
50
51
```

```
52  ---------- type families reified as term-level functions
53
54  bType :: NatRep i -> TypeRep t -> TypeRep (Btype i t)
55  bType i_r (Rcont Nz t_r) = Rcont Nz (bType i_r t_r)
56  bType i_r (Rcont (Ns Nz) t_r) = Rcont (Ns Nz) (bType (Ns i_r) t_r)
57  bType i_r Rint = Rint
58  bType i_r (Rvar n_r) = Rvar n_r
59  bType i_r (Rpair a b) = Rpair (bType i_r a) (bType i_r b)
60  bType i_r (RvarI n)  = Rvar (subtractN i_r (Ns n))
61
62  bEnv :: NatRep i -> EnvRep ts -> EnvRep (Benv i ts)
63  bEnv i_r M0 = M0
64  bEnv i_r (Ms t_r ts_r) = Ms (bType i_r t_r) (bEnv i_r ts_r)
65
66  rType :: NatRep i -> TypeRep t -> TypeRep (Rtype i t)
67  rType i_r (Rcont Nz t_r) = Rcont Nz (rType i_r t_r)
68  rType i_r (Rcont (Ns Nz) t_r) = Rcont (Ns Nz) (rType (Ns i_r) t_r)
69  rType i_r (Rpair a b) = Rpair (rType i_r a) (rType i_r b)
70  rType i_r (Rvar n_r) = RvarI (subtractN i_r (Ns n_r))
71  rType i_r (RvarI n)  = RvarI n
72  rType _ Rint = Rint
73
74  rEnv :: NatRep i -> EnvRep ts -> EnvRep (Renv i ts)
75  rEnv i_r M0 = M0
76  rEnv i_r (Ms t_r ts_r) = Ms (rType i_r t_r) (rEnv i_r ts_r)
77
78  subtractN :: NatRep a -> NatRep b -> NatRep (Subtract a b)
79  subtractN i_r Nz = i_r
80  subtractN (Ns i_r) (Ns j_r) = subtractN i_r j_r
81
82
83  --------------------------------------------------------------------------------
84  -- Main conversion function
85
86  toB :: (forall a. ExpK a) -> ExpKb (Z, ())
87  toB x = toBe Nz M0 (toR x)
88
89
90  --------------------------------------------------------------------------------
91  -- Conversion to deBruijn terms
92
93  data ToR a t where
94   ToRv   :: (forall i ts.
95                  (NatRep i, EnvRep ts)
96               -> ValKb (i, ts) (Rtype i t))
97               -> ToR a (V t)
98   ToRe   :: (forall i ts.
99                  (NatRep i, EnvRep ts)
100              -> ExpKb (i, ts))
101              -> ToR a Void
102
103 unBv :: ToR a (V t) -> (forall i ts . (NatRep i, EnvRep ts)
```

```
104             -> ValKb (i, ts) (Rtype i t))
105 unBv (ToRv x) = x
106
107 unBe :: ToR a Void -> (forall i ts . (NatRep i, EnvRep ts)
108             -> ExpKb (i, ts))
109 unBe (ToRe x) = x
110
111
112 toRaux :: ExpKF (ToR a t) -> ToR a t
113
114 toRaux (KVnum n) = ToRv (\_ -> Knum n)
115
116 toRaux (Klet_poly_fun (s_r::TypeRep s) f e) =
117   ToRe $ \(i_r::NatRep i, ts_r::EnvRep ts) ->
118     let f' = f (RvarI i_r)
119                (toRvar (substT s_r (RvarI i_r) Nz) (i_r, ts_r))
120         f'' :: ExpKb (S i, (Rtype (S i) s, ShiftEnvR i ts))
121         f'' = unBe f' (Ns i_r, Ms (rType (Ns i_r) s_r) (shiftEnvR i_r ts_r))
122
123         e' = e (toRvar (Rcont (Ns Nz) s_r) (i_r, ts_r))
124         e''  = unBe e' (i_r, (Ms (Rcont (Ns Nz) (rType (Ns i_r) s_r)) ts_r))
125     in KBlet_poly_funR "f" "x"
126                        (rType (Ns i_r) s_r)
127                        f''
128                        e''
129
130 toRaux (Kpoly_app
131           (s_r::TypeRep s) (t_r::TypeRep t)
132           f
133           x) =
134   ToRe $ \((i_r::NatRep i), (ts_r::EnvRep ts)) ->
135     let f' :: ValKb (i, ts) (Cont (S Z) (Rtype (S i) s))
136         f' = unBv f (i_r, ts_r)
137         x' :: ValKb (i, ts) (Rtype i (Subst s t Z))
138         x' = unBv x (i_r, ts_r)
139         x'' :: ValKb (i, ts)
140                      (SubstR (Rtype (S i) s) (Rtype i t) i)
141         x'' = case lemma_rtype_subst i_r s_r t_r of Equiv -> x'
142     in KBpoly_appR (rType (Ns i_r) s_r)
143                    (rType i_r t_r)
144                    f'
145                    x''
146
147 toRaux (Klet n s_r v x) = ToRe $ \(i_r, ts_r) ->
148   let v' = unBv v (i_r, ts_r)
149       e' = unBe (x (toRvar s_r (i_r, ts_r))) (i_r, (Ms (rType i_r s_r) ts_r))
150   in KBlet n (rType i_r s_r) v' e'
151
152 toRaux (Kletrec f_n x_n
153           (s_r::TypeRep s)
154           e1
155           e2) =
```

```
156  ToRe $ \(i_r, ts_r) ->
157   let (v1, v2) = toRvar2 s_r (Rcont Nz s_r) (i_r, ts_r)
158       e1' = unBe (e1 v2 v1)
159                  (i_r, Ms (rType i_r s_r) $
160                        Ms (rType i_r (Rcont Nz s_r)) ts_r)
161       e2' = unBe (e2  (toRvar (Rcont Nz s_r) (i_r, ts_r)))
162                  (i_r, Ms (rType i_r (Rcont Nz s_r)) ts_r)
163   in KBletrec f_n x_n (rType i_r s_r) e1' e2'
164
165 toRaux (Kapp s_r a b) = ToRe $ \g_r@(i_r,_) ->
166   KBapp (rType i_r s_r) (unBv a g_r) (unBv b g_r)
167
168 toRaux (Klet_prim n p (ToRv v1) (ToRv v2) e) = ToRe $ \g_r@(i_r, ts_r) ->
169   let e' = unBe (e (toRvar Rint (i_r, ts_r)))
170   in KBlet_prim n p (v1 g_r) (v2 g_r) (e' (i_r, (Ms Rint ts_r)))
171
172 toRaux (Kif0 a b c) = ToRe $ \g_r ->
173   KBif0 (unBv a g_r) (unBe b g_r) (unBe c g_r)
174
175 toRaux (Klet_pair n s_r t_r (ToRv v1) (ToRv v2) e) =
176   ToRe $ \g_r@(i_r, ts_r) ->
177     let e' = unBe (e (toRvar (Rpair s_r t_r) (i_r, ts_r)))
178     in KBlet_pair n (rType i_r s_r) (rType i_r t_r)
179                   (v1 g_r) (v2 g_r)
180                   (e' (i_r, (Ms (rType i_r (Rpair s_r t_r)) ts_r)))
181
182 toRaux (Klet_fst n s_r (ToRv v) e) = ToRe $ \g_r@(i_r, ts_r) ->
183   let e' = unBe (e (toRvar s_r g_r))
184   in KBlet_fst n (rType i_r s_r) (v g_r) (e' (i_r, (Ms (rType i_r s_r) ts_r)))
185
186 toRaux (Klet_snd n s_r (ToRv v) e) = ToRe $ \g_r@(i_r, ts_r) ->
187   let e' = unBe (e (toRvar s_r g_r))
188   in KBlet_snd n (rType i_r s_r) (v g_r) (e' (i_r, (Ms (rType i_r s_r) ts_r)))
189
190 toRaux (Khalt (ToRv v)) = ToRe $ \g_r -> KBhalt (v g_r)
191
192
193 toR :: (forall a. ExpK a) -> ExpKb (Z, ())
194 toR x = unBe ((iterOK toRaux) x) (Nz, MO)
195
196
197 --------------------------------------------------------------------------------
198 -- Conversion to normal de Bruijn types
199
200 toBv :: NatRep i -> EnvRep ts ->
201            ValKb (i, ts) t -> ValKb (i, Benv i ts) (Btype i t)
202 toBv i_r ts_r (Kvar i) = Kvar (toBi i_r i)
203 toBv i_r ts_r (Knum n) = Knum n
204
205 toBe :: NatRep i -> EnvRep ts ->
206            ExpKb (i, ts) -> ExpKb (i, Benv i ts)
207 toBe i_r ts_r (KBlet n s_r v e) =
```

```
208     KBlet n (bType i_r s_r) (toBv i_r ts_r v) (toBe i_r (Ms s_r ts_r) e)
209  toBe i_r ts_r (KBletrec n1 n2 s_r e1 e2) =
210     KBletrec n1 n2
211              (bType i_r s_r)
212              (toBe i_r (Ms s_r (Ms (Rcont Nz s_r) ts_r)) e1)
213              (toBe i_r (Ms (Rcont Nz s_r) ts_r) e2)
214  toBe i_r ts_r (KBapp s_r v1 v2) =
215    KBapp (bType i_r s_r)  (toBv i_r ts_r v1)  (toBv i_r ts_r v2)
216
217  toBe i_r ts_r (KBlet_poly_funR n1 n2 t_r e1 e2) =
218     KBlet_poly_fun n1 n2  (bType (Ns i_r) t_r)
219                    (case lemma_benv_shift i_r ts_r of
220                       Equiv -> (toBe (Ns i_r) (Ms t_r (shiftEnvR i_r ts_r)) e1))
221                    (toBe i_r (Ms (Rcont (Ns Nz) t_r) ts_r) e2)
222
223  toBe i_r ts_r (KBpoly_appR s_r t_r v1 v2) =
224     KBpoly_app (bType (Ns i_r) s_r) (bType i_r t_r)
225                (toBv i_r ts_r v1)
226                (case lemma_btype_subst i_r s_r t_r of
227                   Equiv -> toBv i_r ts_r v2)
228
229  toBe i_r ts_r (KBlet_pair n t1_r t2_r v1 v2 e) =
230     KBlet_pair n (bType i_r t1_r) (bType i_r t2_r)
231                (toBv i_r ts_r v1)
232                (toBv i_r ts_r v2)
233                (toBe i_r (Ms (Rpair t1_r t2_r) ts_r) e)
234
235  toBe i_r ts_r (KBlet_fst n t_r v e) =
236     KBlet_fst n (bType i_r t_r)
237               (toBv i_r ts_r v)
238               (toBe i_r (Ms t_r ts_r) e)
239
240  toBe i_r ts_r (KBlet_snd n t_r v e) =
241     KBlet_snd n (bType i_r t_r)
242               (toBv i_r ts_r v)
243               (toBe i_r (Ms t_r ts_r) e)
244
245  toBe i_r ts_r (KBlet_prim n op v1 v2 e) =
246     KBlet_prim n op
247                (toBv i_r ts_r v1)
248                (toBv i_r ts_r v2)
249                (toBe i_r (Ms Rint ts_r) e)
250
251  toBe i_r ts_r (KBif0 v e1 e2) =
252    KBif0 (toBv i_r ts_r v)
253          (toBe i_r ts_r e1)
254          (toBe i_r ts_r e2)
255
256  toBe i_r ts_r (KBhalt v) =
257    KBhalt (toBv i_r ts_r v)
258
259
```

```
260 toBi :: NatRep i -> Index ts t -> Index (Benv i ts) (Btype i t)
261 toBi _ I0 = I0
262 toBi i_r (Ix i) = Ix (toBi i_r i)
263
264
265
266 ------------------------------------------------------------------------------
267 -- Support functions for the conversion to de Bruijn indices
268
269 toRvar :: TypeRep s -> (NatRep i, EnvRep ts) -> ToR a (V s)
270 toRvar s_r ((i_r::NatRep i), tb) =
271   ToRv (\(i'_r::NatRep i', tb') ->
272      let d_i = subtractN i'_r i_r
273      in Kvar (make_index (Ms (rType i'_r s_r)
274                              (uEnvI d_i i_r tb))
275                         tb'))
276
277 toRvar2 :: TypeRep s -> TypeRep t -> (NatRep i, EnvRep ts)
278              -> (ToR a (V s), ToR a (V t))
279 toRvar2 s_r t_r (i_r, tb) =
280    (ToRv (\(i_r, tb') -> Kvar (make_index (Ms (rType i_r s_r)
281                                          (Ms (rType i_r t_r) tb)) tb')),
282     ToRv (\(i_r, tb') -> Kvar (make_index2 (Ms (rType i_r s_r)
283                                           (Ms (rType i_r t_r) tb)) tb')))
284
285 data Sub ts ts' where
286   Sub_refl :: Sub ts ts
287   Sub_inc  :: Sub ts ts' -> Sub ts (s, ts')
288
289 ctx_length :: EnvRep ts -> Int
290 ctx_length M0 = 0
291 ctx_length (Ms _ r) = 1 + ctx_length r
292
293 make_sub :: EnvRep ts -> EnvRep ts' -> Sub ts ts'
294 make_sub tb tb' =
295   let d = ctx_length tb' - ctx_length tb
296       reduce :: forall ts ts' . Int -> EnvRep ts -> EnvRep ts' -> Sub ts ts'
297       reduce d r tb'@(Ms _ r')
298          | d == 0  = case envEqual r tb' of
299                        Just Equiv -> Sub_refl
300          | d > 0   = Sub_inc (reduce (d-1) r r')
301   in reduce d tb tb'
302
303 get_index :: Sub (s, ts) ts' -> Index ts' s
304 get_index Sub_refl = i0
305 get_index (Sub_inc s) = Ix (get_index s)
306
307 get_index2 :: Sub (s, (t, ts)) ts' -> Index ts' t
308 get_index2 Sub_refl = (Ix i0)
309 get_index2 (Sub_inc s) = Ix (get_index2 s)
310
311 make_index :: EnvRep (s, ts) -> EnvRep ts' -> Index ts' s
```

```
312 make_index tb tb' = get_index (make_sub tb tb')
313
314 make_index2 :: EnvRep (s, (t, ts)) -> EnvRep ts' -> Index ts' t
315 make_index2 tb tb' = get_index2 (make_sub tb tb')
316
317
318 --------------------------------------------------------------------------------
319 -- Lemmas
320
321 lemma_rtype_subst :: NatRep i -> TypeRep s -> TypeRep t
322    -> Equiv (Rtype i (Subst s t Z))
323             (SubstR (Rtype (S i) s) (Rtype i t) i)
324 lemma_rtype_subst i_r s_r t_r =
325   case typesEqual (rType i_r (substT s_r t_r Nz))
326                   (substR (rType (Ns i_r) s_r) (rType i_r t_r) i_r)
327     of Just Equiv -> Equiv
328
329 lemma_benv_shift :: NatRep i -> EnvRep ts ->
330            Equiv (ShiftEnv (Benv i ts))
331                  (Benv (S i) (ShiftEnvR i ts))
332 lemma_benv_shift i_r ts_r =
333    case envEqual (shift_env (bEnv i_r ts_r))
334                  (bEnv (Ns i_r) (shiftEnvR i_r ts_r))
335      of Just Equiv -> Equiv
336
337 lemma_btype_subst :: NatRep i -> TypeRep s -> TypeRep t
338    -> Equiv (Btype i (SubstR s t i))
339             (Subst (Btype (S i) s) (Btype i t) Z)
340 lemma_btype_subst i_r s_r t_r =
341   case typesEqual (bType i_r (substR s_r t_r i_r))
342                   (substT (bType (Ns i_r) s_r) (bType i_r t_r) Nz)
343     of Just Equiv -> Equiv
344
```

LC.hs

```
1  {-# OPTIONS -XGADTs #-} {-
2
3    Closure-converted language
4
5  -}
6
7  module LC (
8    ValC(..), ExpC(..)
9  ) where
10
11 import Tp
12
13 -----------------------------------------------------------------------------
14 -- Closure-Converted Language
15
16 data ValC g t where
17   Cvar     :: Index ts t                              -> ValC (i, ts) t
18
19   Cfix     ::     NatRep k
20                -> TypeRep t
21                -> ExpC (k, (t, (Cont k t, ())))
22                -> ValC (i, ts) (Cont k t)
23
24   Ctp_app  :: NatRep k -> TypeRep s -> TypeRep t ->
25               ValC (i, ts) (Cont (S k) s)  -> ValC (i,ts) (Cont k (Subst s t k))
26
27   Cpack    :: TypeRep s -> TypeRep t
28               -> ValC (i, ts) (Subst s t Z) -> ValC (i, ts) (Exists s)
29
30   Cnum     :: Int                                      -> ValC ts Int
31
32   -- used only temporarily during closure conversion
33   Cproj    :: ValC ts (Tup s) -> Index s t             -> ValC ts t
34
35
36 data ExpC ts where
37   Clet      :: Name -> TypeRep s ->
38                ValC (i, ts) s -> ExpC (i, (s, ts)) -> ExpC (i, ts)
39   Cunpack   ::    TypeRep s
40                -> ValC (i, ts) (Exists s)
41                -> ExpC (S i, (s, ShiftEnv ts))
42                -> ExpC (i, ts)
43
44   Clet_tup  :: Name -> EnvRep t
45                     -> MapT (ValC (i, ts)) t
46                     -> ExpC (i, (Tup t, ts))
47                     -> ExpC (i, ts)
48   Clet_proj  :: Name -> TypeRep t -> ValC (i, ts) (Tup s)
49                     -> Index s t
50                     -> ExpC (i, (t, ts))
51                     -> ExpC (i, ts)
```

```
52    Clet_prim :: Name -> PrimOp
53                    -> ValC (i, ts) Int -> ValC (i, ts) Int -> ExpC (i, (Int, ts))
54                                                            -> ExpC (i, ts)
55
56    Capp      :: ValC ts (Cont Z s) -> ValC ts s -> ExpC ts
57
58    Cif0      :: ValC ts Int -> ExpC ts -> ExpC ts          -> ExpC ts
59
60    Chalt     :: ValC ts t                                  -> ExpC ts
61
```

CC.hs

```
1  {-# OPTIONS -fglasgow-exts -fallow-undecidable-instances #-} {-
2
3    Closure conversion
4
5  -}
6
7  module CC (
8    cc
9  ) where
10
11 import Tp
12 import LKb
13 import LC
14
15
16 --------------------------------------------------------------------------------
17 -- Closure conversion [TYPES]
18
19 type family Ctype t
20 type instance Ctype Int        = Int
21 type instance Ctype (Cont i t) = CLOSURE i (Ctype t)
22 type instance Ctype (t1, t2)   = Tup (Ctype t1, (Ctype t2, ()))
23 type instance Ctype (Var n)    = Var n
24
25 type family Cenv ts
26 type instance Cenv () = ()
27 type instance Cenv (s, ts) = (Ctype s, Cenv ts)
28
29 type PAIR s t = Tup (s, (t, ()))
30
31 type CLOSURE k t = Exists (PRE_CLO k t)
32 type PRE_CLO k t = PAIR (Cont k (PAIR (U k (S Z) t) (Var k)))
33                         (Var Z)
34
35 -- type synonyms reified as functions
36
37 cType :: TypeRep t -> TypeRep (Ctype t)
38 cType Rint = Rint
39 cType (Rcont i s) = Rexists (rPair (Rcont i (rPair (uT i (Ns Nz) (cType s))
40                                                     (Rvar i)))
41                                    (Rvar Nz))
42 cType (Rpair t1 t2) = rPair (cType t1) (cType t2)
43 cType (Rvar i) = Rvar i
44
45 cEnv :: EnvRep ts -> EnvRep (Cenv ts)
46 cEnv M0 = M0
47 cEnv (Ms s_r ts_r) = Ms (cType s_r) (cEnv ts_r)
48
49 preCloT :: NatRep k -> TypeRep t -> TypeRep (PRE_CLO k t)
50 preCloT k_r t_r =
51   Rtup (Ms (Rcont k_r (Rtup (Ms (uT k_r (Ns Nz) t_r)
```

```
52                              (Ms (Rvar k_r)
53                              M0))))
54         (Ms (Rvar Nz)
55          M0))
56
57  closureT :: NatRep k -> TypeRep t -> TypeRep (CLOSURE k t)
58  closureT k_r t_r = Rexists (preCloT k_r t_r)
59
60
61  --------------------------------------------------------------------------------
62  -- Closure conversion [TERMS]
63
64  cc :: ExpKb (Z, ()) -> ExpC (Z, ())
65  cc e =  cc_e Nz M0 e M0
66
67
68  type CCe i ts   =
69     (forall cs' . MapT (ValC (i, cs')) (Cenv ts)  -> ExpC (i, cs'))
70  type CCv i ts t =
71     (forall cs' . MapT (ValC (i, cs')) (Cenv ts)  -> ValC (i, cs') (Ctype t))
72
73  --------------- values
74
75  cc_v :: NatRep i -> EnvRep ts -> ValKb (i, ts) t -> CCv i ts t
76  cc_v _ _ (Kvar i) = (\m -> lookupT m (tr i))
77  cc_v _ _ (Knum n) = (\m -> Cnum n)
78
79
80  --------------- expressions
81
82  cc_e :: NatRep i -> EnvRep ts -> ExpKb (i, ts) -> CCe i ts
83
84  cc_e i_r ts_r (KBlet n s_r v e) =
85     \m -> Clet n (cType s_r)
86                  (cc_v i_r ts_r v m)
87                  (cc_e i_r (Ms s_r ts_r) e (Ms (Cvar i0) (mapT shift_v m)))
88
89
90  cc_e (i_r::NatRep i)
91       (ts_r::EnvRep ts)
92       exp@(KBletrec _ _ s_r
93                     (f::ExpKb (i, (t, (Cont Z t, ts))))
94                     x) =
95    case mkMap i_r ts_r (fvs_e ts_r exp) of
96      EnvMap (env_r::EnvRep env) m0 env ->
97        let p_r = Ms (Rcont Nz (Rtup (Ms (cType s_r) (Ms
98                                       (Rtup (cEnv env_r)) M0)))) (Ms
99                     (Rtup (cEnv env_r)) M0)
100
101           co :: forall g.
102                    ValC g (PAIR (Cont Z (PAIR (Ctype t)
103                                            (Tup (Cenv env))))
```

```
104                               (Tup (Cenv env))) ->
105                       ValC g (PAIR (Cont Z (PAIR (Subst (U Z (S Z)) (Ctype t))
106                                                          (Tup (Cenv env)) Z)
107                                                   (Tup (Cenv env))))
108                               (Tup (Cenv env)))
109              co v = case lemma_closure Nz (cType s_r) (Rtup (cEnv env_r)) of
110                       Equiv -> v
111
112              raw_code :: forall cs'. ValC (i, cs')
113                                        (Cont i (PAIR (Ctype t)
114                                                          (Tup (Cenv env))))
115              raw_code =
116                case (lemma_add_z i_r, lemma_uenv_z Nz (cEnv env_r)) of
117                  (Equiv, Equiv) ->
118                         Cfix i_r
119                              (rPair (cType s_r) (Rtup (cEnv env_r)))
120                              (Clet_proj "arg" (cType s_r) (Cvar i0) i0 $
121                               Clet_proj "env" (Rtup (cEnv env_r)) (Cvar i1) i1 $
122                               Clet_tup "p" p_r
123                                        (Ms (tp_app_multi i_r i_r Nz
124                                                          (rPair (cType s_r)
125                                                                (Rtup (cEnv env_r)))
126                                                          (Cvar i3))
127                                            (Ms (Cvar i0) M0)) $
128                               Clet "clo" (closureT Nz (cType s_r))
129                                          (Cpack (preCloT Nz (cType s_r))
130                                                 (Rtup (cEnv env_r))
131                                                 (co (Cvar i0))) $
132                               openEnv (cEnv env_r) i2 $
133                               let m' = Ms (Cvar i3) $
134                                        Ms (Cvar i0) $
135                                          mapT (\i -> Cproj (Cvar i2) i) m0
136                               in cc_e i_r
137                                       (Ms s_r (Ms (Rcont Nz s_r) ts_r))
138                                       f m')
139          in (\m ->
140              case equiv2 (lemma_uenv_z Nz (cEnv env_r)) (lemma_add_z i_r) of
141                Equiv2 ->
142                  Clet_tup "env" (cEnv env_r)
143                           (cc_tup i_r ts_r m env) $
144                  Clet_tup "p" p_r
145                           (Ms (tp_app_multi i_r i_r Nz
146                                             (rPair (cType s_r)
147                                                    (Rtup (cEnv env_r)))
148                                             raw_code )
149                               (Ms (Cvar i0) M0)) $
150                    Clet "clo" (closureT Nz (cType s_r))
151                               (Cpack (preCloT Nz (cType s_r))
152                                      (Rtup (cEnv env_r))
153                                      (co (Cvar i0))) $
154                          (cc_e i_r (Ms (Rcont Nz s_r) ts_r)
155                                x
```

```
156                               (Ms (Cvar i0)
157                                   (mapT (shift_v . shift_v . shift_v) m))))
158
159
160
161 cc_e i_r ts_r (KBapp s_r
162                      (v1::ValKb (i, ts) (Cont Z s))
163                      (v2::ValKb (i, ts) s)) =
164    \(m::MapT (ValC (i, cs')) (Cenv ts)) ->
165      case (lemma_subst_u s_r, lemma_cenv_u ts_r) of
166         (Equiv, Equiv) ->
167          Cunpack (preCloT Nz (cType s_r)) (cc_v i_r ts_r v1 m) $
168          Clet_proj "_f"   (Rcont Nz (rPair (shift_tr (cType s_r))
169                                             (Rvar Nz)))
170                               (Cvar i0) i0 $
171          Clet_proj "_env" (Rvar Nz) (Cvar i1) i1 $
172          Clet_tup  "p" (Ms (shift_tr (cType s_r))
173                               (Ms (Rvar Nz) M0))
174                      (Ms (cc_v (Ns i_r)
175                               (shift_env ts_r)
176                               (tp_shift_valK v2)
177                               (mapT (shift_v . shift_v . shift_v)
178                                     (shift_map m))) $
179                       Ms (Cvar i0) $ M0) $
180          Capp (Cvar i2) (Cvar i0)
181
182 cc_e (i_r::NatRep i)
183      (ts_r::EnvRep ts)
184      exp@(KBlet_poly_fun _ _  s_r
185                     (f::ExpKb (S i, (t, Uenv Z (S Z) ts)))
186                     (e::ExpKb (i,   (Cont (S Z) t, ts)))) =
187   let ts'_r =  uEnv Nz (Ns Nz) ts_r
188       vs ::MapT BoolT (Uenv Z (S Z) ts)
189       vs = tailT (fvs_e (Ms s_r ts'_r) f)
190   in
191   case mkMap i_r ts_r (coerce_bool_map ts_r vs) of
192     EnvMap (env_r::EnvRep env)
193            (m0::MapT (Index (Cenv env)) (Cenv ts))
194            (env:: MapT (ValKb (i, ts)) env) ->
195        let
196           raw_code :: forall cs'. ValC (i, cs')
197                                    (Cont (Add i (S Z))
198                                          (PAIR (Ctype t)
199                                                (Tup (Uenv Z (S Z)
200                                                           (Cenv env)))))
201           raw_code =
202              case lemma_succ i_r of
203                Equiv ->
204                  case lemma_cenv_u ts_r of
205                     Equiv ->
206                      Cfix (Ns i_r)
207                           (rPair (cType s_r)
```

```
208                                  (Rtup (uEnv Nz (Ns Nz) (cEnv env_r))))
209                           (Clet_proj "arg" (cType s_r) (Cvar i0) i0 $
210                            Clet_proj "env" (Rtup (uEnv Nz (Ns Nz) (cEnv env_r)))
211                                             (Cvar i1) i1 $
212                            openEnv (uEnv Nz (Ns Nz) (cEnv env_r)) i0 $
213                            let  m' = Ms (Cvar i1) $
214                                      mapT (\i -> Cproj (Cvar i0) i)
215                                                        (shift_index_map m0) in
216
217                                 cc_e (Ns i_r)
218                                      (Ms s_r (uEnv Nz (Ns Nz) ts_r))
219                                      f
220                                      m')
221
222
223         in (\(m::MapT (ValC (i, cs')) (Cenv ts)) ->
224              case lemma_add_z i_r of
225              Equiv ->
226               case lemma_subst (cType s_r)  (Rtup (cEnv env_r)) of
227                Equiv ->
228                 Clet_tup "env"
229                          (cEnv env_r)
230                          (cc_tup i_r ts_r m env) $
231                 mkClosure (Ns Nz) (cType s_r) (Rtup (cEnv env_r))
232                           (tp_app_multi i_r i_r (Ns Nz)
233                                         (rPair (cType s_r)
234                                                (Rtup (uEnv Nz (Ns Nz)
235                                                           (cEnv (env_r)))))
236                                         raw_code)
237                           (Cvar i0) $
238                   cc_e i_r (Ms (Rcont (Ns Nz) s_r) ts_r)
239                      e
240                      (Ms (Cvar i0) $
241                          mapT (shift_v . shift_v . shift_v) m))
242
243
244 cc_e i_r ts_r (KBpoly_app (s_r:: TypeRep s)
245                           (t_r:: TypeRep t)
246                           (v  :: ValKb (i, ts) (Cont (S Z) s))
247                           (w  :: ValKb (i, ts) (Subst s t Z))) =
248  \(m::MapT (ValC (i, cs')) (Cenv ts)) ->
249   case lemma_tp_app s_r t_r of
250    Equiv ->
251     case lemma_cenv_u ts_r of
252      Equiv ->
253        Cunpack (preCloT (Ns Nz) (cType s_r)) (cc_v i_r ts_r v m) $
254        Clet_proj "_f" (Rcont (Ns Nz) (Rtup (Ms (uT (Ns Nz) (Ns Nz) (cType s_r))
255                             (Ms (Rvar (Ns Nz)) M0))))
256                       (Cvar i0) i0 $
257        Clet_proj "_env" (Rvar Nz) (Cvar i1) i1 $
258        Clet_tup "_p" (Ms (cType (uT Nz (Ns Nz) (substT s_r t_r Nz))) $
259                       Ms (Rvar Nz) $ M0)
```

```
260                        (Ms (cc_v (Ns i_r)
261                            (shift_env ts_r)
262                            (tp_shift_valK w)
263                            (mapT (shift_v . shift_v . shift_v)
264                                  (shift_map m))) $
265                         Ms (Cvar i0) M0) $
266        Capp (Ctp_app Nz (rPair (uT (Ns Nz) (Ns Nz) (cType s_r))
267                                  (Rvar (Ns Nz)))
268                           (cType t_r) (Cvar i2))
269             (Cvar i0)
270
271 cc_e i_r ts_r (KBlet_pair n t1_r t2_r v1 v2 e) =
272    \m -> Clet_tup n (Ms (cType t1_r) (Ms (cType t2_r) M0))
273                     (Ms (cc_v i_r ts_r v1 m) $
274                      Ms (cc_v i_r ts_r v2 m) M0)
275                     (cc_e i_r (Ms (Rpair t1_r t2_r) ts_r) e
276                           (Ms (Cvar i0) (mapT shift_v m)))
277
278 cc_e i_r ts_r (KBlet_fst n t_r v e) =
279    \m -> Clet_proj n (cType t_r) (cc_v i_r ts_r v m) I0
280                      (cc_e i_r (Ms t_r ts_r) e (Ms (Cvar i0) (mapT shift_v m)))
281
282 cc_e i_r ts_r (KBlet_snd n t_r v e) =
283    \m -> Clet_proj n (cType t_r) (cc_v i_r ts_r v m) (Ix I0)
284                      (cc_e i_r (Ms t_r ts_r) e (Ms (Cvar i0) (mapT shift_v m)))
285
286 cc_e i_r ts_r (KBlet_prim n p v1 v2 e) =
287    \m -> Clet_prim n p (cc_v i_r ts_r v1 m) (cc_v i_r ts_r v2 m)
288                      (cc_e i_r (Ms Rint ts_r) e (Ms (Cvar i0) (mapT shift_v m)))
289
290 cc_e i_r ts_r (KBif0 v e1 e2) =
291    \m -> Cif0 (cc_v i_r ts_r v m) (cc_e i_r ts_r e1 m) (cc_e i_r ts_r e2 m)
292
293 cc_e i_r ts_r (KBhalt v) = \m -> Chalt (cc_v i_r ts_r v m)
294
295
296 --------------- tuples
297
298 cc_tup ::     NatRep i -> EnvRep ts
299            -> MapT (ValC (i, cs')) (Cenv ts)
300            -> MapT (ValKb (i, ts)) env
301            -> MapT (ValC (i, cs')) (Cenv env)
302 cc_tup _ _ _ M0 = M0
303 cc_tup i_r ts_r m (Ms s ts) = Ms (cc_v i_r ts_r s m) (cc_tup i_r ts_r m ts)
304
305
306 -----------------------------------------------------------------------------
307 -- Free variables
308
309 data BoolT t = BoolT Bool
310
311
```

```
312 fvs_v :: EnvRep ts -> ValKb (i, ts) t -> MapT BoolT ts
313 fvs_v ts_r (Kvar i) = updateT (falseMap ts_r) i (BoolT True)
314 fvs_v ts_r (Knum _) = falseMap ts_r
315
316 fvs_e :: EnvRep ts -> ExpKb (i, ts) -> MapT BoolT ts
317 fvs_e tb (KBlet _ s_r e1 e2) = zipWithT orT (fvs_v tb e1)
318                                              (tailT (fvs_e (Ms s_r tb) e2))
319 fvs_e tb (KBletrec _ _ s_r f e) =
320    zipWithT orT (tailT (tailT (fvs_e (Ms s_r (Ms undefined tb)) f)))
321                 (tailT (fvs_e (Ms undefined tb) e))
322 fvs_e tb (KBapp _ e1 e2) = zipWithT orT (fvs_v tb e1) (fvs_v tb e2)
323 fvs_e tb (KBlet_poly_fun _ _ s_r e1 e2) =
324     zipWithT orT
325              (coerce_bool_map tb
326                                 (tailT (fvs_e (Ms undefined (shift_env tb)) e1)))
327              (tailT (fvs_e (Ms undefined tb) e2))
328 fvs_e tb (KBpoly_app s_r t_r v1 v2) =
329     zipWithT orT (fvs_v tb v1) (fvs_v tb v2)
330 fvs_e tb (KBif0 e1 e2 e3) =
331   zipWithT orT (zipWithT orT (fvs_v tb e1) (fvs_e tb e2))
332                (fvs_e tb e3)
333
334 fvs_e tb (KBlet_prim _ _ e1 e2 e3) =
335   zipWithT orT (zipWithT orT (fvs_v tb e1) (fvs_v tb e2))
336                (tailT (fvs_e (Ms Rint tb) e3))
337 fvs_e tb (KBlet_pair _ s_r t_r v1 v2 e) =
338            zipWithT orT (zipWithT orT (fvs_v tb v1) (fvs_v tb v2))
339                         (tailT (fvs_e (Ms (Rpair s_r t_r) tb) e))
340 fvs_e tb (KBlet_fst _ s_r e1 e2) = zipWithT orT (fvs_v tb e1)
341                                                (tailT (fvs_e (Ms s_r tb) e2))
342 fvs_e tb (KBlet_snd _ s_r e1 e2) = zipWithT orT (fvs_v tb e1)
343                                                (tailT (fvs_e (Ms s_r tb) e2))
344 fvs_e tb (KBhalt e) = fvs_v tb e
345
346
347 fvs_tup :: EnvRep ts -> MapT (ValKb (i, ts)) t -> MapT BoolT ts
348 fvs_tup tb M0 = falseMap tb
349 fvs_tup tb (Ms v tup) =
350   let v_m   = fvs_v tb v
351       tup_m = fvs_tup tb tup
352   in zipWithT orT v_m tup_m
353
354 tailT :: MapT c (s, ts) -> MapT c ts
355 tailT (Ms _ t) = t
356
357 zipWithT :: (forall t . c1 t -> c2 t -> c3 t)
358               -> MapT c1 ts -> MapT c2 ts -> MapT c3 ts
359 zipWithT _ M0 _ = M0
360 zipWithT f (Ms c t) (Ms c' t') = Ms (f c c') (zipWithT f t t')
361
362 orT :: BoolT s -> BoolT s -> BoolT s
363 orT (BoolT a) (BoolT b) = BoolT (a || b)
```

```
364
365 falseMap :: EnvRep ts -> MapT BoolT ts
366 falseMap M0 = M0
367 falseMap (Ms _ t) = Ms (BoolT False) (falseMap t)
368
369 -- safe if ts and ts' are the same length
370 coerce_bool_map :: EnvRep ts' -> MapT BoolT ts -> MapT BoolT ts'
371 coerce_bool_map M0 M0 = M0
372 coerce_bool_map (Ms _ ts'_r) (Ms (BoolT b) bs) =
373   Ms (BoolT b) (coerce_bool_map ts'_r bs)
374
375
376 --------------------------------------------------------------------------------
377 -- Index map construction
378
379 data EnvMap_aux env0 i cs0 cs where
380   EnvMap_aux ::    (Cat env0 env ~ env') =>
381             EnvRep env
382          -> EnvRep env'
383          -> MapT (Index (Cenv env')) (Cenv ts)
384          -> MapT (ValKb (i, ts0)) env
385          -> EnvMap_aux env0 i ts0 ts
386
387 mkMap_aux ::
388          NatRep i
389       -> EnvRep ts
390       -> MapT BoolT       ts -- free variables
391       -> MapT (Index ts0) ts -- indices
392       -> EnvRep env0
393       -> EnvMap_aux env0 i ts0 ts
394 mkMap_aux _ _ M0 M0 e0_r =
395   case cat_nil e0_r of Equiv -> EnvMap_aux M0 e0_r M0 M0
396
397 mkMap_aux i_r (Ms _ ts_r) (Ms (BoolT False) bs) (Ms _ is) env0_r =
398   case mkMap_aux i_r ts_r bs is env0_r of
399     EnvMap_aux env_r env'_r m t ->
400       EnvMap_aux env_r env'_r (Ms undefined m) t
401
402
403 mkMap_aux i_r (Ms t_r ts_r)
404           (Ms (BoolT True) bs)
405           (Ms (i::Index ts t) is)
406           (env0_r::EnvRep env0) =
407   let i_ :: Index (Cat env0 (t,())) t
408       i_ = newIndex env0_r
409       env1_r = catT env0_r (Ms t_r M0)
410   in case mkMap_aux i_r ts_r bs is env1_r of
411        EnvMap_aux (env_r::EnvRep env) (env'_r::EnvRep env') m t ->
412          case lemma_cat_assoc env0_r (Ms t_r M0) env_r of
413            Equiv ->
414             case lemma_cat_cenv env0_r (Ms t_r env_r) of
415              Equiv ->
```

```
416                  EnvMap_aux (Ms t_r env_r)
417                             env'_r
418                             (Ms (tr (weaken_index env_r i_)) m)
419                             (Ms (Kvar i) t)
420
421  data EnvMap i ts where
422    EnvMap :: EnvRep env
423           -> MapT (Index (Cenv env)) (Cenv ts)
424           -> MapT (ValKb (i, ts)) env
425           -> EnvMap i ts
426
427  mkMap :: NatRep i -> EnvRep ts -> MapT BoolT ts -> EnvMap i ts
428  mkMap i_r ts_r fs =
429    case mkMap_aux i_r ts_r fs (mkIndices fs) M0 of
430      EnvMap_aux env_r env'_r m t ->
431        case cat_nil env'_r of
432          Equiv -> EnvMap env'_r
433                          m t
434
435  mkIndices :: MapT c ts -> MapT (Index ts) ts
436  mkIndices M0 = M0
437  mkIndices (Ms _ tb) = Ms I0 (shift_is (mkIndices tb))
438
439
440  --------------------------------------------------------------------------------
441  -- Closures formation
442
443  mkClosure ::
444               NatRep k -> TypeRep t -> TypeRep env
445            -> ValC (i, ts) (Cont k (PAIR t (U Z k env)))
446            -> ValC (i, ts) env
447            -> (forall t'. ExpC (i, (CLOSURE k t, (t', ts))))
448            -> ExpC (i, ts)
449  mkClosure (k_r::NatRep k) (t_r::TypeRep t) (env_r::TypeRep env)
450             f (env::ValC (i, ts) env) body =
451    let f' :: ValC (i, ts) (Cont k (PAIR (Subst (U k (S Z) t) env k)
452                                         (U Z k env)))
453        f' = case local_lemma1 of Equiv -> f
454    in case lemma_u_z Nz env_r of
455         Equiv ->
456          case local_lemma2 of
457            Equiv ->
458              Clet_tup "p" (Ms (Rcont k_r
459                                    (rPair (substT (uT k_r (Ns Nz) t_r)
460                                                   env_r k_r)
461                                           (uT Nz k_r env_r))) $
462                              Ms env_r $ M0)
463                          (Ms f' (Ms env M0)) $
464              Clet "clo" (closureT k_r t_r)
465                           (Cpack (preCloT k_r t_r)
466                                  env_r
467                                  (Cvar i0)) $
```

```
468             body
469
470   where local_lemma1 :: Equiv (Subst (U k (S Z) t) env k) t
471         local_lemma1 =
472           case typesEqual (substT (uT k_r (Ns Nz) t_r) env_r k_r) t_r of
473             Just Equiv -> Equiv
474
475         local_lemma2 ::
476           Equiv (CMP k k (Var k) (U Z k env) (Var (Pred k)))
477                 (U Z k env)
478         local_lemma2 =
479           lemma_cmp_eq k_r (undefined :: TypeRep (Var k))
480                            (undefined :: TypeRep (U Z k env))
481                            (undefined :: TypeRep (Var (Pred k)))
482
483
484 --------------------------------------------------------------------------------
485 -- Multiple type applications
486
487 type family MultApp j c
488 type instance MultApp (S j) (Cont (S k) t) =
489               MultApp j     (Cont k     (Subst t (Var j) k))
490 type instance MultApp Z t = t
491
492 multAppT :: NatRep j -> TypeRep c -> TypeRep (MultApp j c)
493 multAppT (Ns j_r) (Rcont (Ns k_r) t_r) =
494     multAppT j_r (Rcont k_r (substT t_r (Rvar j_r) k_r))
495 multAppT Nz t = t
496
497
498 multApp ::      NatRep j -> NatRep k -> TypeRep t
499              -> ValC (i, ts) (Cont k t)
500              -> ValC (i, ts) (MultApp j (Cont k t))
501 multApp (Ns j_r) (Ns k_r) t_r v = multApp j_r k_r (substT t_r (Rvar j_r) k_r)
502                                           (Ctp_app k_r t_r (Rvar j_r) v)
503 multApp Nz k_r _ e = e
504
505 lemma_mult_app ::
506   NatRep i -> NatRep k -> TypeRep t ->
507   Equiv (MultApp i (Cont (Add i k) t))
508         (Cont k t)
509 lemma_mult_app i_r k_r t_r =
510   case typesEqual (multAppT i_r (Rcont (addT i_r k_r) t_r))
511                   (Rcont k_r t_r) of
512     Just Equiv -> Equiv
513
514 -- note j and i are the same!
515 tp_app_multi ::      NatRep j
516                   -> NatRep i
517                   -> NatRep k
518                   -> TypeRep t
519                   -> ValC (j, ts) (Cont (Add i k) t)
```

```
520                   -> ValC (j, ts) (Cont k t)
521 tp_app_multi j_r i_r k_r t_r v =
522   case lemma_mult_app i_r k_r t_r of
523     Equiv -> multApp i_r (addT i_r k_r) t_r v
524
525
526 --------------------------------------------------------------------------------
527 -- Shifting and update
528
529 tr :: Index ts t -> Index (Cenv ts) (Ctype t)
530 tr I0 = I0
531 tr (Ix i) = Ix (tr i)
532
533 shift_is :: MapT (Index ts0) ts -> MapT (Index (s, ts0)) ts
534 shift_is M0 = M0
535 shift_is (Ms i m) = Ms (Ix i) (shift_is m)
536
537 shift_v :: ValC (i, ts) t -> ValC (i, (s, ts)) t
538 shift_v v = u_v M0 undefined undefined v
539
540 shift_e :: TypeRep s -> ExpC (i, ts) -> ExpC (i, (s, ts))
541 shift_e s_r e = u_e M0 s_r undefined e
542
543 shift_index_map::
544       MapT (Index env)
545            ts
546   ->  MapT (Index (Uenv Z (S Z) env))
547            (Uenv Z (S Z) ts)
548 shift_index_map M0 = M0
549 shift_index_map (Ms i is) = Ms (shift_index i) (shift_index_map is)
550     where shift_index :: Index env t ->  Index (Uenv Z (S Z) env) (U Z (S Z) t)
551           shift_index I0 = I0
552           shift_index (Ix i) = Ix (shift_index i)
553
554 u_v ::    EnvRep ts
555        -> TypeRep s  -- never analyzed
556        -> EnvRep ts' -- never analyzed
557        -> ValC (i, Cat ts ts') t
558        -> ValC (i, Cat ts (s, ts')) t
559 u_v ts_r s_r ts'_r v =
560   let u_e_ = u_e ts_r s_r ts'_r
561       u_v_ = u_v ts_r s_r ts'_r
562   in case v of
563     Cvar i -> Cvar (u_i ts_r s_r ts'_r i)
564     Cfix k t_r e -> Cfix k t_r e
565     Ctp_app k_r s_r t_r v -> Ctp_app k_r s_r t_r (u_v_ v)
566     Cpack s_r t_r v -> Cpack s_r t_r (u_v_ v)
567     Cnum n -> Cnum n
568     Cproj v i -> Cproj (u_v_ v) i
569
570 u_e ::    EnvRep ts
571        -> TypeRep s  -- never analyzed
```

```
572         -> EnvRep ts' -- never analyzed
573         -> ExpC (i, Cat ts ts') -> ExpC (i, Cat ts (s, ts'))
574 u_e ts_r s_r ts'_r e =
575   let u_e_ = u_e ts_r s_r ts'_r
576       u_v_ = u_v ts_r s_r ts'_r
577   in case e of
578     Clet n t_r v e -> Clet n t_r
579                          (u_v_ v) (u_e (Ms t_r ts_r) s_r ts'_r e)
580     Cunpack t_r v e ->
581        case (lemma_cat_u0 ts_r ts'_r,
582              lemma_cat_u1 ts_r s_r ts'_r) of
583          (Equiv, Equiv) ->
584            Cunpack t_r (u_v_ v)
585                    (u_e (Ms t_r (uEnv Nz (Ns Nz) ts_r))
586                         (uT Nz (Ns Nz) s_r)
587                         (uEnv Nz (Ns Nz) ts'_r) e)
588     Capp v1 v2 -> Capp (u_v_ v1) (u_v_ v2)
589     Clet_tup n t_r tup e -> Clet_tup n t_r
590                                   (u_t ts_r s_r ts'_r tup)
591                                   (u_e (Ms (Rtup t_r) ts_r) s_r ts'_r e)
592     Clet_proj n t_r v i e -> Clet_proj n t_r (u_v_ v) i
593                                     (u_e (Ms t_r ts_r) s_r ts'_r e)
594     Clet_prim n p v1 v2 e -> Clet_prim n p (u_v_ v1) (u_v_ v2)
595                                     (u_e (Ms Rint ts_r) s_r ts'_r e)
596     Cif0 v e1 e2 -> Cif0 (u_v_ v) (u_e_ e1) (u_e_ e2)
597     Chalt v -> Chalt (u_v_ v)
598
599 u_t::     EnvRep ts
600         -> TypeRep s  -- never analyzed
601         -> EnvRep ts' -- never analyzed
602         -> MapT (ValC (i, Cat ts ts')) t
603         -> MapT (ValC (i, Cat ts (s, ts'))) t
604 u_t _ _ _ M0 = M0
605 u_t ts_r s_r ts'_r (Ms v ts) =
606    Ms (u_v ts_r s_r ts'_r v)
607       (u_t ts_r s_r ts'_r ts)
608
609 u_i ::     EnvRep ts
610          -> TypeRep s   -- never analyzed
611          -> EnvRep ts'   -- never analyzed
612          -> Index (Cat ts ts') t
613          -> Index (Cat ts (s, ts')) t
614 u_i M0 s0_r ts'_r  i = Ix i            -- i > |ts|
615 u_i (Ms s_r ts_r) s0_r  ts'_r i =
616   case eq1 ts_r s_r ts'_r i of
617     Ix j -> case eq2 ts_r s_r ts'_r of
618               Equiv -> Ix (u_i ts_r s0_r ts'_r j)
619     I0 -> I0
620    where  eq1 :: EnvRep ts -> TypeRep s -> EnvRep ts'
621                       -> Index (Cat (s, ts) ts') t
622                       -> Index (s, Cat ts ts')   t
623           eq1 ts_r s_r ts'_r i = i
```

```
624             eq2 :: EnvRep ts  -> TypeRep s -> EnvRep ts'
625                         -> Equiv (Cat (s, ts) ts')
626                      (s, Cat ts ts')
627             eq2 ts_r s_r ts'_r = Equiv
628
629
630   tp_shift_valK :: ValKb (i, ts) s -> ValKb (S i, ShiftEnv ts) (Shift s)
631   tp_shift_valK (Kvar i) = Kvar (tr_index_shift i)
632   tp_shift_valK (Knum n) = Knum n
633
634   tp_shift_expK :: ExpKb (i, ts) -> ExpKb (S i, ShiftEnv ts)
635   tp_shift_expK (KBlet n s_r v e) = KBlet n (shift_tr s_r)
636                                             (tp_shift_valK v)
637                                             (tp_shift_expK e)
638   tp_shift_expK (KBapp t_r v1 v2) = KBapp (shift_tr t_r) (tp_shift_valK v1)
639                                                          (tp_shift_valK v2)
640   tp_shift_expK (KBlet_prim n p v1 v2 e) =
641      KBlet_prim n p (tp_shift_valK v1) (tp_shift_valK v2)
642                 (tp_shift_expK e)
643   tp_shift_expK (KBif0 v e1 e2) =
644      KBif0 (tp_shift_valK v) (tp_shift_expK e1) (tp_shift_expK e2)
645   tp_shift_expK (KBhalt v) = KBhalt (tp_shift_valK v)
646
647   tr_index_U ::     NatRep k
648                  -> NatRep i
649                  -> Index ts t
650                  -> Index (Uenv k (S i) ts) (U k (S i) t)
651   tr_index_U _ i_r I0 = I0
652   tr_index_U k_r i_r (Ix i) = Ix (tr_index_U k_r i_r i)
653
654
655   tp_shift_valC :: ValC (i, ts) s -> ValC (S i, ShiftEnv ts) (Shift s)
656   tp_shift_valC (Cvar i) = Cvar (tr_index_shift i)
657   tp_shift_valC (Cproj v i) = Cproj (tp_shift_valC v)
658                                     (tr_index_shift i)
659
660   shift_map ::     MapT (ValC (i, ts0))            ts
661                 -> MapT (ValC (S i, ShiftEnv ts0)) (ShiftEnv ts)
662   shift_map M0 = M0
663   shift_map (Ms a m) = Ms (tp_shift_valC a)
664                           (shift_map m)
665
666
667   --------------------------------------------------------------------------------
668   -- Elimination of projections as values
669
670   openEnv ::
671       EnvRep env0
672    -> Index ts (Tup env0)
673    -> ExpC (i, ts)
674    -> ExpC (i, ts)
675   openEnv env_r i (e::ExpC (i, ts)) =
```

```
676     project_env (Cvar i) env_r (mkIndices env_r)
677                 (elim_proj_e M0
678                              env_r
679                              undefined
680                              i
681                              (shift_e_multi env_r e))
682
683  project_env ::    ValC (i, ts0) (Tup env0)
684                 -> EnvRep env
685                 -> MapT (Index env0) env
686                 -> ExpC (i, Cat env ts0)
687                 -> ExpC (i, ts0)
688  project_env _ M0 M0 e = e
689  project_env env_var (Ms t_r ts_r) (Ms i m') e =
690      project_env
691           env_var
692           ts_r
693           m'
694           (Clet_proj ("x" ++ show (toInt i)) t_r
695                      (shift_v_multi ts_r env_var)
696                      i
697                      e)
698    where toInt :: forall ts t . Index ts t -> Int
699          toInt I0 = 0
700          toInt (Ix i) = 1 + toInt i
701
702  elim_proj_v  ::
703      EnvRep ts
704   -> EnvRep env -- for type checking, never analyzed
705   -> EnvRep ts0
706   -> Index ts0 (Tup env)
707   -> ValC (i, Cat ts (Cat env ts0)) t
708   -> ValC (i, Cat ts (Cat env ts0)) t
709  elim_proj_v ts_r env_r ts0_r i v =
710    let e_e = elim_proj_e ts_r env_r ts0_r i
711        e_v = elim_proj_v ts_r env_r ts0_r i
712    in case v of
713      Cproj (Cvar j) k ->
714        case cmp_indices (shift_i_multi ts_r (shift_i_multi env_r i)) j of
715          Nothing -> Cproj (Cvar j) k
716          Just Equiv -> Cvar (shift_i_multi ts_r (weaken_index_multi ts0_r k))
717      Cvar v -> Cvar v
718      Cfix n_r t_r e -> Cfix n_r t_r e
719      Ctp_app k_r s_r t_r v -> Ctp_app k_r s_r t_r (e_v v)
720      Cpack s_r t_r v -> Cpack s_r t_r (e_v v)
721      Cnum n -> Cnum n
722
723  elim_proj_e  ::
724      EnvRep ts
725   -> EnvRep env -- for type checking, never analyzed
726   -> EnvRep ts0
727   -> Index ts0 (Tup env)
```

```
728  -> ExpC (i, Cat ts (Cat env ts0))
729  -> ExpC (i, Cat ts (Cat env ts0))
730 elim_proj_e (ts_r::EnvRep ts) (env_r::EnvRep env) (ts0_r::EnvRep ts0) i e =
731   let e_e = elim_proj_e ts_r env_r ts0_r i
732       e_v = elim_proj_v ts_r env_r ts0_r i
733   in case e of
734     Clet n t_r v e -> Clet n t_r (e_v v)
735                            (elim_proj_e (Ms t_r ts_r)
736                                         env_r ts0_r i e)
737     Cunpack t_r v e ->
738                 case lemma_cat_u2 ts_r env_r ts0_r of
739                      Equiv ->
740                        Cunpack t_r (e_v v)
741                                (elim_proj_e (Ms t_r
742                                                   (uEnv Nz (Ns Nz) ts_r))
743                                              (uEnv Nz (Ns Nz) env_r)
744                                              (uEnv Nz (Ns Nz) ts0_r)
745                                              (tr_index_U Nz Nz i)
746                                              e)
747     Capp v1 v2 -> Capp (e_v v1) (e_v v2)
748     Clet_tup n t_r tup e -> Clet_tup n t_r
749                                   (elim_proj_t ts_r env_r ts0_r i tup)
750                                   (elim_proj_e (Ms (Rtup t_r) ts_r)
751                                                env_r ts0_r i e)
752     Clet_proj n t_r v j e -> Clet_proj n t_r (e_v v) j
753                                     (elim_proj_e (Ms t_r ts_r)
754                                                  env_r ts0_r i e)
755     Clet_prim n p v1 v2 e -> Clet_prim n p (e_v v1) (e_v v2)
756                                     (elim_proj_e (Ms Rint ts_r)
757                                                  env_r ts0_r i e)
758     Cif0 v e1 e2 -> Cif0 (e_v v) (e_e e1) (e_e e2)
759     Chalt v -> Chalt (e_v v)
760
761 elim_proj_t  ::
762     EnvRep ts
763  -> EnvRep env -- for type checking, never analyzed
764  -> EnvRep ts0
765  -> Index ts0 (Tup env)
766  -> MapT (ValC (i, Cat ts (Cat env ts0))) t
767  -> MapT (ValC (i, Cat ts (Cat env ts0))) t
768 elim_proj_t _ _ _ _ M0 = M0
769 elim_proj_t ts_r env_r ts0_r i (Ms v tup) =
770   Ms (elim_proj_v ts_r env_r ts0_r i v)
771      (elim_proj_t ts_r env_r ts0_r i tup)
772
773 cmp_indices :: Index ts s -> Index ts t -> Maybe (Equiv s t)
774 cmp_indices I0 I0 = Just Equiv
775 cmp_indices (Ix i) (Ix j) = cmp_indices i j
776 cmp_indices I0 (Ix j) = Nothing
777 cmp_indices (Ix i) I0 = Nothing
778
779 weaken_index_multi ::
```

```
780       EnvRep ts   -- for type checking, never analyzed
781    -> Index env t
782    -> Index (Cat env ts) t
783  weaken_index_multi _ I0 = I0
784  weaken_index_multi env_r (Ix i) = Ix (weaken_index_multi env_r i)
785
786  shift_v_m :: MapT z ts -> ValC (i, ts0) t -> ValC (i, Cat ts ts0) t
787  shift_v_m M0 v = v
788  shift_v_m (Ms _ m) v = shift_v (shift_v_m m v)
789
790  shift_i :: Index ts t -> Index (s, ts) t
791  shift_i i = Ix i
792
793  shift_i_m :: MapT z ts -> Index ts0 t -> Index (Cat ts ts0) t
794  shift_i_m M0 i = i
795  shift_i_m (Ms _ m) i = Ix (shift_i_m m i)
796
797  shift_e_multi :: EnvRep env -> ExpC (i, ts) -> ExpC (i, Cat env ts)
798  shift_e_multi M0 e = e
799  shift_e_multi (Ms s_r env_rep) e =
800    shift_e s_r (shift_e_multi env_rep e)
801
802  shift_v_multi :: EnvRep env -> ValC (i, ts) t -> ValC (i, Cat env ts) t
803  shift_v_multi M0 v = v
804  shift_v_multi (Ms _ env_rep) v = shift_v (shift_v_multi env_rep v)
805
806  shift_i_multi :: EnvRep env -> Index ts t -> Index (Cat env ts) t
807  shift_i_multi M0 i = i
808  shift_i_multi (Ms _ env_rep) i = Ix (shift_i_multi env_rep i)
809
810
811  ------------------------------------------------------------------------------
812  -- Compound proof objects
813
814  -- Equiv2 combines two proofs of type equality.  This is useful to get
815  -- GHC to swallow two type assumptions simultaneously, as sometimes
816  -- required to satisfy the type checker.
817
818  data Equiv2 a b c d where
819    Equiv2 :: (a ~ b, c ~ d) => Equiv2 a b c d
820
821  equiv2 :: Equiv a b -> Equiv c d -> Equiv2 a b c d
822  equiv2 Equiv Equiv = Equiv2
823
824
825  ------------------------------------------------------------------------------
826  -- Lemmas
827
828  lemma_add_z :: NatRep i -> Equiv i (Add i Z)
829  lemma_add_z Nz = Equiv
830  lemma_add_z (Ns i) = case lemma_add_z i of Equiv -> Equiv
831
```

```
832 lemma_succ :: NatRep i -> Equiv (S i) (Add i (S Z))
833 lemma_succ Nz = Equiv
834 lemma_succ (Ns i) = case lemma_succ i of Equiv -> Equiv
835
836 lemma_cenv_u ::
837   EnvRep ts ->
838     Equiv (Cenv (Uenv Z (S Z) ts))
839           (Uenv Z (S Z) (Cenv ts))
840 lemma_cenv_u ts_r =
841   case envEqual (cEnv (uEnv Nz (Ns Nz) ts_r))
842                 (uEnv Nz (Ns Nz) (cEnv ts_r)) of
843     Just Equiv -> Equiv
844
845 lemma_ctype_subst ::
846    TypeRep s -> TypeRep t
847       -> Equiv (Subst (Ctype s) (Ctype t) Z)
848                (Ctype (Subst s t Z))
849 lemma_ctype_subst s_r t_r =
850    case typesEqual (substT (cType s_r) (cType t_r) Nz)
851                    (cType (substT s_r t_r Nz))
852      of Just Equiv -> Equiv
853
854 lemma_subst_u :: TypeRep s -> Equiv (Ctype (U Z (S Z) s)) (U Z (S Z) (Ctype s))
855 lemma_subst_u s_r =
856    case typesEqual (cType (uT Nz (Ns Nz) s_r))
857                    (uT Nz (Ns Nz) (cType s_r))
858      of Just Equiv -> Equiv
859
860 lemma_ctype_u ::
861   NatRep i -> NatRep j -> TypeRep t ->
862      Equiv (Ctype (U i j t))
863            (U i j (Ctype t))
864 lemma_ctype_u i_r j_r t_r =
865    case typesEqual (cType (uT i_r j_r t_r))
866                    (uT i_r j_r (cType t_r))
867      of Just Equiv -> Equiv
868
869 lemma_cat_cenv ::
870    EnvRep env0 -> EnvRep env ->
871     Equiv (Cat (Cenv env0) (Cenv env))
872           (Cenv (Cat env0 env))
873 lemma_cat_cenv env0_r env_r =
874   case envEqual (catT (cEnv env0_r) (cEnv env_r))
875                 (cEnv (catT env0_r env_r))
876     of Just Equiv -> Equiv
877
878 lemma_cat_u0 ::
879      EnvRep ts
880   -> EnvRep ts'
881   -> Equiv (Cat (Uenv Z (S Z) ts) (Uenv Z (S Z) ts'))
882            (Uenv Z (S Z) (Cat ts ts'))
883 lemma_cat_u0 M0 _ = Equiv
```

```
884 lemma_cat_u0 (Ms _ ts_r) ts'_r =
885   case lemma_cat_u0 ts_r ts'_r of Equiv -> Equiv
886
887 lemma_cat_u1 ::
888       EnvRep ts
889   -> TypeRep s
890   -> EnvRep ts'
891   -> Equiv (Cat (Uenv Z (S Z) ts) (U Z (S Z) s, Uenv Z (S Z) ts'))
892            (Uenv Z (S Z) (Cat ts (s, ts')))
893 lemma_cat_u1 M0 s_r _ = Equiv
894 lemma_cat_u1 (Ms _ ts_r) s_r ts'_r =
895   case lemma_cat_u1 ts_r s_r ts'_r of Equiv -> Equiv
896
897 lemma_cat_u2 ::
898       EnvRep ts -> EnvRep env -> EnvRep ts0
899   -> Equiv (Cat (Uenv Z (S Z) ts) (Cat (Uenv Z (S Z) env) (Uenv Z (S Z) ts0)))
900            (Uenv Z (S Z) (Cat ts (Cat env ts0)))
901 lemma_cat_u2 ts_r env_r ts0_r =
902   case lemma_cat_u0 env_r ts0_r of
903     Equiv ->
904       case lemma_cat_u0 ts_r (catT env_r ts0_r) of
905         Equiv -> Equiv
906
907 lemma_tp_app ::
908   forall s t. TypeRep s -> TypeRep t ->
909      Equiv (Subst (U (S Z) (S Z) (Ctype s)) (Ctype t) Z)
910            (Ctype (U Z (S Z) (Subst s t Z)))
911 lemma_tp_app s_r t_r =
912   case typesEqual (substT (uT (Ns Nz) (Ns Nz) (cType s_r)) (cType t_r) Nz)
913                   (cType (uT Nz (Ns Nz) (substT s_r t_r Nz)))
914      of Just Equiv -> Equiv
915
916 lemma_closure ::
917    NatRep k -> TypeRep t -> TypeRep env ->
918    Equiv (Subst (PAIR (U k (S Z) t) (Var k)) env k)
919          (PAIR t (U Z k env))
920 lemma_closure k_r t_r env_r =
921   case typesEqual (substT (rPair (uT k_r (Ns Nz) t_r) (Rvar k_r)) env_r k_r)
922                   (rPair t_r (uT Nz k_r env_r))
923    of Just Equiv -> Equiv
924
925 lemma_cmp_eq::
926      NatRep k
927   -> TypeRep a -> TypeRep b -> TypeRep c  -- ignored
928   -> Equiv (CMP k k a b c) b
929 lemma_cmp_eq Nz _ _ _ = Equiv
930 lemma_cmp_eq (Ns k_r) a b c=
931   case lemma_cmp_eq k_r a b c of
932      Equiv -> Equiv
933
934 lemma_subst :: TypeRep t -> TypeRep t'
935              ->  Equiv t
```

```
936                        (Subst (U (S Z) (S Z) t) t' (S Z))
937 lemma_subst t_r t'_r =
938             case typesEqual t_r
939                             (substT (uT (Ns Nz) (Ns Nz) t_r) t'_r (Ns Nz))
940               of Just Equiv -> Equiv
941
942 lemma_u_z :: NatRep k -> TypeRep t -> Equiv t (U k Z t)
943 lemma_u_z k_r t_r =
944   case typesEqual t_r (uT k_r Nz t_r)
945     of Just Equiv -> Equiv
946
947 lemma_uenv_z :: NatRep k -> EnvRep ts -> Equiv ts (Uenv k Z ts)
948 lemma_uenv_z k_r ts_r =
949   case envEqual ts_r (uEnv k_r Nz ts_r)
950     of Just Equiv -> Equiv
951
```

LH.hs

```
1  {-# OPTIONS -XGADTs #-} {-
2
3    Linearized language
4
5  -}
6
7  module LH (
8    ProgramH(..), CodeBlockH(..), ValH(..), ExpH(..)
9  ) where
10
11 import Tp
12
13
14 data ProgramH where
15   Hletrec :: MapT (CodeBlockH fs) fs -> ExpH (Z, (), fs) -> ProgramH
16
17 data CodeBlockH g t where
18   Hblock ::   TypeRep (Cont k t)
19            -> ExpH (k, (t, ()), fs)
20            -> CodeBlockH fs (Cont k t)
21
22 data ValH g t where
23   Hvar      :: Index ts t                  -> ValH (i, ts, fs) t
24   Hlam      :: Index fs t                  -> ValH (i, ts, fs) (Closed t)
25
26   Hdisclose :: ValH g (Closed t)           -> ValH g t
27
28   Htp_app   :: NatRep k -> TypeRep s -> TypeRep t ->
29                ValH (i, ts, fs) (Cont (S k) s)
30                                        -> ValH (i,ts,fs) (Cont k (Subst s t k))
31   Hpack     :: TypeRep s -> TypeRep t ->
32                ValH (i, ts, fs) (Subst s t Z) -> ValH (i, ts, fs) (Exists s)
33   Hnum      :: Int                         -> ValH g Int
34
35
36 data ExpH g where
37   Hlet      :: Name -> TypeRep s ->
38                    ValH (i,ts,fs) s -> ExpH (i,(s, ts),fs) -> ExpH (i,ts,fs)
39   Hunpack   ::    TypeRep s
40                -> ValH (i, ts, fs) (Exists s)
41                -> ExpH (S i, (s, ShiftEnv ts), fs)
42                -> ExpH (i, ts, fs)
43
44   Hlet_tup  :: Name -> EnvRep t
45                     -> MapT (ValH (i, ts, fs)) t
46                     -> ExpH (i, (Tup t, ts), fs)
47                     -> ExpH (i, ts, fs)
48   Hlet_proj :: Name -> TypeRep t -> ValH (i, ts, fs) (Tup s)
49                     -> Index s t
50                     -> ExpH (i, (t, ts), fs)
51                     -> ExpH (i, ts, fs)
```

```
52    Hlet_prim :: Name -> PrimOp
53              -> ValH (i,ts,fs) Int -> ValH (i,ts,fs) Int -> ExpH (i,(Int, ts),fs)
54                                                           -> ExpH (i, ts, fs)
55
56    Happ      :: ValH g (Cont Z s) -> ValH g s               -> ExpH g
57
58    Hif0      :: ValH g Int -> ExpH g -> ExpH g              -> ExpH g
59
60    Hhalt     :: ValH g t                                    -> ExpH g
61
```

Hoist.hs

```
1  {-# OPTIONS -fglasgow-exts #-}
2
3  module Hoist (
4    hoist
5  ) where
6
7  import Tp
8
9  import LC
10 import LH
11
12
13 --------------------------------------------------------------------------------
14 -- Hoisting
15
16 -------------------- values
17
18 data CollectV hs0 i ts t where
19   CollectV ::    EnvRep hs
20               -> ValH (i, ts, Cat hs0 hs) t
21               -> MapT (CodeBlockH (Cat hs0 hs)) hs
22               -> CollectV hs0 i ts t
23
24 collectV ::    EnvRep hs0
25             -> MapT (ValH (i, ts, hs0)) ts
26             -> ValC (i, ts) t
27             -> CollectV hs0 i ts t
28
29 collectV hs0_r m (Cvar i) =
30    case cat_nil hs0_r of Equiv -> CollectV M0 (lookupT m i) M0
31
32
33 collectV (hs0_r::EnvRep hs0) m
34          (Cfix i_r t_r (f::ExpC (i, (t, (Cont i t, ()))))) =
35   let i = newIndex hs0_r
36       hs1_r = catT hs0_r (Ms (Rcont i_r t_r) M0)
37   in case collectE hs1_r
38                    (Ms (Hvar i0) (Ms (Hvar i1) M0))
39                    f of
40        CollectE (hs2_r::EnvRep hs2)
41                 (f' ::ExpH (i, (t, (Cont i t, ())),
42                             Cat (Cat hs0 (Cont i t, ())) hs2))
43                 tail ->
44          case lemma_cat_assoc hs0_r (Ms (Rcont i_r t_r) M0) hs2_r of
45            Equiv ->
46              CollectV (Ms (Rcont i_r t_r) hs2_r)
47                       (Hdisclose (Hlam (weaken_index hs2_r i)))
48                       (Ms (Hblock (Rcont i_r t_r)
49                                   (Hlet "f" (Rcont i_r t_r)
50                                         (Hdisclose
51                                            (Hlam (weaken_index hs2_r i))) $
```

```
52                                   Hlet "x" t_r (Hvar i1) $
53                                      (weaken_exp_ts (Ms t_r (Ms (Rcont i_r t_r)
54                                                         M0))
55                                                  (Ms t_r M0) f')))
56                          tail)
57
58  collectV hs0_r m (Cnum i) = CollectV M0 (Hnum i) M0
59
60  collectV hs0_r m (Cpack s_r t_r v) =
61    case collectV hs0_r m v of
62      CollectV hs1_r v' tup1 ->
63        CollectV hs1_r (Hpack s_r t_r v') tup1
64
65  collectV hs0_r m (Ctp_app k_r s_r t_r e) =
66    case collectV hs0_r m e of
67      CollectV hs1_r e' tup1 ->
68        CollectV hs1_r (Htp_app k_r s_r t_r e') tup1
69
70
71  -------------------- tuples
72
73  data CollectT hs0 i ts t where
74    CollectT ::    EnvRep hs
75                -> MapT (ValH (i, ts, Cat hs0 hs)) t
76                -> MapT (CodeBlockH (Cat hs0 hs)) hs
77                -> CollectT hs0 i ts t
78
79  collectT ::    EnvRep hs0
80              -> MapT (ValH (i, ts, hs0)) ts
81              -> MapT (ValC (i, ts)) t
82              -> CollectT hs0 i ts t
83  collectT hs0_r m M0 = CollectT M0 M0 M0
84  collectT hs0_r m (Ms e1 es) =
85    case collectV hs0_r m e1 of
86      CollectV hs1_r e1' tup1 ->
87        case collectT (catT hs0_r hs1_r)
88                      (mapT (weaken_val_hs hs1_r) m) es of
89          CollectT hs2_r es' tup2 ->
90            case lemma_cat_assoc hs0_r hs1_r hs2_r of
91              Equiv ->
92                CollectT (catT hs1_r hs2_r)
93                         (Ms (weaken_val_hs hs2_r e1') es')
94                         (cmb_tup hs0_r hs1_r hs2_r tup1 tup2)
95
96
97  -------------------- expressions
98
99  data CollectE hs0 i ts where
100   CollectE ::    EnvRep hs
101               -> ExpH (i, ts, Cat hs0 hs)
102               -> MapT (CodeBlockH (Cat hs0 hs)) hs
103               -> CollectE hs0 i ts
```

```
104
105 collectE ::    EnvRep hs0
106             -> MapT (ValH (i, ts, hs0)) ts
107             -> ExpC (i, ts)
108             -> CollectE hs0 i ts
109
110 collectE (hs0_r::EnvRep hs0)
111          m (Clet n s_r v (e::ExpC (i, (s, ts)))) =
112   case collectV hs0_r m v of
113     CollectV (hs1_r::EnvRep hs1) v' tup1 ->
114       case collectE (catT hs0_r hs1_r)
115                     (Ms (Hvar i0)
116                         (mapT (shift_val_ts . (weaken_val_hs hs1_r)) m)) e of
117         CollectE (hs2_r::EnvRep hs2)
118                  (e'::ExpH (i, (s, ts), Cat (Cat hs0 hs1) hs2)) tup2 ->
119           case lemma_cat_assoc hs0_r hs1_r hs2_r of
120             Equiv ->
121               CollectE (catT hs1_r hs2_r)
122                        (Hlet n s_r (weaken_val_hs hs2_r v') e')
123                        (cmb_tup hs0_r hs1_r hs2_r tup1 tup2)
124
125 collectE (hs0_r::EnvRep hs0)
126          (m::MapT (ValH (i, ts, hs0)) ts)
127          (Cunpack t_r v
128                   (e::ExpC (S i, (s, ShiftEnv ts)))) =
129   case collectV hs0_r m v of
130     CollectV (hs1_r::EnvRep hs1) v' tup1 ->
131        case collectE (catT hs0_r hs1_r)
132                     (Ms (Hvar i0)
133                         (mapT (shift_val_ts . weaken_val_hs hs1_r)
134                               (shift_map m)))
135
136                     e of
137          CollectE (hs2_r::EnvRep hs2)
138                   (e'::ExpH (S i, (s, ShiftEnv ts), Cat (Cat hs0 hs1) hs2))
139                   tup2 ->
140           case lemma_cat_assoc hs0_r hs1_r hs2_r of
141             Equiv ->
142               CollectE (catT hs1_r hs2_r)
143                        (Hunpack t_r (weaken_val_hs hs2_r v') e')
144                        (cmb_tup hs0_r hs1_r hs2_r tup1 tup2)
145
146 collectE hs0_r m (Capp e1 e2) =
147   case collectV hs0_r m e1 of
148     CollectV hs1_r e1' tup1 ->
149       case collectV (catT hs0_r hs1_r)
150                     (mapT (weaken_val_hs hs1_r) m) e2 of
151         CollectV hs2_r e2' tup2 ->
152           case lemma_cat_assoc hs0_r hs1_r hs2_r of
153             Equiv ->
154               CollectE (catT hs1_r hs2_r)
155                        (Happ (weaken_val_hs hs2_r e1') e2')
```

```
156                         (cmb_tup hs0_r hs1_r hs2_r tup1 tup2)
157
158 collectE (hs0_r::EnvRep hs0) m (Clet_prim n op v1 v2 (e::ExpC (i, (Int, ts)))) =
159   case collectV hs0_r m v1 of
160     CollectV (hs1_r::EnvRep hs1) v1' tup1 ->
161        case collectV (catT hs0_r hs1_r)
162                      (mapT (weaken_val_hs hs1_r) m) v2 of
163          CollectV (hs2_r::EnvRep hs2) v2' tup2 ->
164            case collectE (catT (catT hs0_r hs1_r) hs2_r)
165                          (Ms (Hvar i0)
166                            (mapT (shift_val_ts . (weaken_val_hs hs2_r .
167                                                   weaken_val_hs hs1_r)) m)) e of
168              CollectE (hs3_r::EnvRep hs3)
169                       (e'::ExpH (i, (Int, ts), Cat (Cat (Cat hs0 hs1) hs2) hs3))
170                       tup3 ->
171                case (lemma_cat_assoc (catT hs0_r hs1_r) hs2_r hs3_r,
172                      lemma_cat_assoc hs0_r hs1_r (catT hs2_r hs3_r)) of
173                  (Equiv, Equiv) ->
174                      CollectE (catT hs1_r (catT hs2_r hs3_r))
175                               (Hlet_prim n op
176                                           (weaken_val_hs hs3_r $
177                                              weaken_val_hs hs2_r v1')
178                                           (weaken_val_hs hs3_r v2') e')
179                               (cmb_tup hs0_r hs1_r (catT hs2_r hs3_r) tup1 $
180                                cmb_tup (catT hs0_r hs1_r) hs2_r hs3_r
181                                        tup2 tup3)
182
183
184 collectE (hs0_r::EnvRep hs0) m (Cif0 v e1 e2) =
185   case collectV hs0_r m v of
186     CollectV (hs1_r::EnvRep hs1) v' tup1 ->
187        case collectE (catT hs0_r hs1_r)
188                      (mapT (weaken_val_hs hs1_r) m) e1 of
189          CollectE (hs2_r::EnvRep hs2) e1' tup2 ->
190            case collectE (catT (catT hs0_r hs1_r) hs2_r)
191                          (mapT (weaken_val_hs hs2_r . weaken_val_hs hs1_r) m)
192                          e2 of
193             CollectE (hs3_r::EnvRep hs3) e2' tup3 ->
194              case (lemma_cat_assoc (catT hs0_r hs1_r) hs2_r hs3_r,
195                    lemma_cat_assoc hs0_r hs1_r (catT hs2_r hs3_r)) of
196                (Equiv, Equiv) ->
197                  CollectE (catT hs1_r (catT hs2_r hs3_r))
198                           (Hif0 (weaken_val_hs hs3_r $
199                                  weaken_val_hs hs2_r v')
200                                 (weaken_exp_hs hs3_r e1') e2')
201                           (cmb_tup hs0_r hs1_r (catT hs2_r hs3_r)
202                                    tup1 $
203                            cmb_tup (catT hs0_r hs1_r) hs2_r hs3_r
204                                    tup2 tup3)
205
206
207
```

```
208 collectE (hs0_r::EnvRep hs0) m (Clet_tup n t_r tup (e::ExpC (i, (Tup t, ts)))) =
209   case collectT hs0_r m tup of
210     CollectT (hs1_r::EnvRep hs1) tup' tup1 ->
211       case collectE (catT hs0_r hs1_r)
212                     (Ms (Hvar i0)
213                         (mapT (shift_val_ts . (weaken_val_hs hs1_r)) m)) e of
214         CollectE (hs2_r::EnvRep hs2)
215                  (e'::ExpH (i, (Tup t, ts), Cat (Cat hs0 hs1) hs2)) tup2 ->
216            case lemma_cat_assoc hs0_r hs1_r hs2_r of
217              Equiv ->
218               CollectE (catT hs1_r hs2_r)
219                        (Hlet_tup n t_r
220                                  (weaken_tup_hs hs2_r tup')
221                                  e')
222                        (cmb_tup hs0_r hs1_r hs2_r tup1 tup2)
223
224 collectE (hs0_r::EnvRep hs0) m (Clet_proj n s_r v i (e::ExpC (i, (t, ts)))) =
225   case collectV hs0_r m v of
226     CollectV (hs1_r::EnvRep hs1) v' tup1 ->
227       case collectE (catT hs0_r hs1_r)
228                     (Ms (Hvar i0)
229                         (mapT (shift_val_ts . (weaken_val_hs hs1_r)) m)) e of
230         CollectE (hs2_r::EnvRep hs2)
231                  (e'::ExpH (i, (t, ts), Cat (Cat hs0 hs1) hs2)) tup2 ->
232           case lemma_cat_assoc hs0_r hs1_r hs2_r of
233             Equiv ->
234               CollectE (catT hs1_r hs2_r)
235                        (Hlet_proj n s_r  (weaken_val_hs hs2_r v') i e')
236                        (cmb_tup hs0_r hs1_r hs2_r tup1 tup2)
237
238 collectE hs0_r m (Chalt v) =
239   case collectV hs0_r m v of
240     CollectV hs1_r v' tup1 ->
241       CollectE hs1_r (Hhalt v') tup1
242
243
244 hoist :: ExpC (Z, ()) -> ProgramH
245 hoist e =
246   case collectE M0 M0 e of
247     CollectE hs_r e' tup ->
248       case cat_nil hs_r of
249         Equiv -> Hletrec tup e'
250
251
252 --------------------------------------------------------------------------------
253 -- Index manipulations, shifting, weakening, etc.
254
255 shift_val_ts :: ValH (i, ts, hs) t -> ValH (i, (s, ts), hs) t
256 shift_val_ts (Hvar i) = Hvar (Ix i)
257 shift_val_ts (Hlam i) = Hlam i
258 shift_val_ts (Hnum n) = Hnum n
259
```

```
260 shift_tup_ts :: MapT (ValH (i, ts, hs)) t -> MapT (ValH (i, (s, ts), hs)) t
261 shift_tup_ts M0 = M0
262 shift_tup_ts (Ms v t) = Ms (shift_val_ts v) (shift_tup_ts t)
263
264
265 -------------------- weakening on hs
266
267 weaken_tup_hs :: EnvRep hs -> MapT (ValH (i, ts, hs0)) t
268                            -> MapT (ValH (i, ts, Cat hs0 hs)) t
269 weaken_tup_hs _ M0 = M0
270 weaken_tup_hs hs_r (Ms v t) =
271   Ms (weaken_val_hs hs_r v) (weaken_tup_hs hs_r t)
272
273 weaken_val_hs :: EnvRep hs -> ValH (i, ts, hs0) t -> ValH (i, ts, Cat hs0 hs) t
274 weaken_val_hs hs_r (Hvar i) = Hvar i
275 weaken_val_hs hs_r (Hlam i) = Hlam (weaken_index hs_r i)
276 weaken_val_hs hs_r (Hnum n) = Hnum n
277 weaken_val_hs hs_r (Htp_app k_r s_r t_r v) =
278   Htp_app k_r s_r t_r (weaken_val_hs hs_r v)
279 weaken_val_hs hs_r (Hpack s_r t_r v) = Hpack s_r t_r (weaken_val_hs hs_r v)
280 weaken_val_hs hs_r (Hdisclose v) = Hdisclose (weaken_val_hs hs_r v)
281
282 weaken_exp_hs :: EnvRep hs -> ExpH (i, ts, hs0) -> ExpH (i, ts, Cat hs0 hs)
283 weaken_exp_hs hs_r (Hlet n s_r v e) =
284    Hlet n s_r (weaken_val_hs hs_r v) (weaken_exp_hs hs_r e)
285 weaken_exp_hs hs_r (Hunpack t_r v e) = Hunpack t_r (weaken_val_hs hs_r v)
286                                                    (weaken_exp_hs hs_r e)
287 weaken_exp_hs hs_r (Happ v1 v2) =
288    Happ (weaken_val_hs hs_r v1) (weaken_val_hs hs_r v2)
289 weaken_exp_hs hs_r (Hlet_prim n p v1 v2 e) =
290    Hlet_prim n p (weaken_val_hs hs_r v1) (weaken_val_hs hs_r v2)
291                  (weaken_exp_hs hs_r e)
292 weaken_exp_hs hs_r (Hif0 v e1 e2) =
293    Hif0 (weaken_val_hs hs_r v) (weaken_exp_hs hs_r e1) (weaken_exp_hs hs_r e2)
294 weaken_exp_hs hs_r (Hlet_tup n t_r v e) =
295    Hlet_tup n t_r (weaken_tup_hs hs_r v) (weaken_exp_hs hs_r e)
296 weaken_exp_hs hs_r (Hlet_proj n t_r v i e) =
297    Hlet_proj n t_r (weaken_val_hs hs_r v) i (weaken_exp_hs hs_r e)
298 weaken_exp_hs hs_r (Hhalt v) =
299    Hhalt (weaken_val_hs hs_r v)
300
301 -------------------- weakening on ts
302
303 weaken_tup_ts :: EnvRep ts0 -> EnvRep ts
304                            -> MapT (ValH (i, ts0, hs)) t
305                            -> MapT (ValH (i, Cat ts0 ts, hs)) t
306 weaken_tup_ts _ _ M0 = M0
307 weaken_tup_ts ts0_r ts_r (Ms v t) =
308   Ms (weaken_val_ts ts0_r ts_r v) (weaken_tup_ts ts0_r ts_r t)
309
310
311 weaken_val_ts ::    EnvRep ts0
```

```
312                  -> EnvRep ts
313                  -> ValH (i, ts0, hs) t
314                  -> ValH (i, Cat ts0 ts, hs) t
315 weaken_val_ts ts0_r ts_r (Hvar i) = Hvar (weaken_index ts_r i)
316 weaken_val_ts ts0_r ts_r (Hlam i) = Hlam i
317 weaken_val_ts ts0_r ts_r (Hnum n) = Hnum n
318 weaken_val_ts ts0_r ts_r (Hpack s t v) = Hpack s t (weaken_val_ts ts0_r ts_r v)
319 weaken_val_ts ts0_r ts_r (Htp_app k_r s_r t_r v) =
320   Htp_app k_r s_r t_r (weaken_val_ts ts0_r ts_r v)
321 weaken_val_ts ts0_r ts_r  (Hdisclose v) = Hdisclose (weaken_val_ts ts0_r ts_r  v)
322
323 weaken_exp_ts ::    EnvRep ts0
324                  -> EnvRep ts
325                  -> ExpH (i, ts0, hs)
326                  -> ExpH (i, Cat ts0 ts, hs)
327 weaken_exp_ts ts0_r ts_r (Hlet n s_r v e) =
328    Hlet n s_r (weaken_val_ts ts0_r ts_r v)
329              (weaken_exp_ts (Ms undefined ts0_r) ts_r e)
330 weaken_exp_ts ts0_r ts_r (Hunpack t_r v e) =
331    Hunpack t_r (weaken_val_ts ts0_r ts_r v)
332               (case lemma_cat_shift ts0_r ts_r of
333                  Equiv -> (weaken_exp_ts (Ms undefined (shift_env  ts0_r))
334                                          (shift_env ts_r) e))
335
336 weaken_exp_ts ts0_r ts_r (Happ v1 v2) =
337    Happ (weaken_val_ts ts0_r ts_r v1) (weaken_val_ts ts0_r ts_r v2)
338 weaken_exp_ts ts0_r ts_r (Hlet_prim n p v1 v2 e) =
339    Hlet_prim n p (weaken_val_ts ts0_r ts_r v1)
340                  (weaken_val_ts ts0_r ts_r v2)
341                  (weaken_exp_ts (Ms Rint ts0_r) ts_r e)
342 weaken_exp_ts ts0_r ts_r (Hif0 v e1 e2) =
343    Hif0 (weaken_val_ts ts0_r ts_r v)
344         (weaken_exp_ts ts0_r ts_r e1)
345         (weaken_exp_ts ts0_r ts_r e2)
346 weaken_exp_ts ts0_r ts_r (Hlet_tup n t_r v e) =
347    Hlet_tup n t_r (weaken_tup_ts ts0_r ts_r v)
348                   (weaken_exp_ts (Ms undefined ts0_r) ts_r e)
349 weaken_exp_ts ts0_r ts_r (Hlet_proj n t_r v i e) =
350    Hlet_proj n t_r (weaken_val_ts ts0_r ts_r v) i
351                    (weaken_exp_ts (Ms undefined ts0_r) ts_r e)
352 weaken_exp_ts ts0_r ts_r (Hhalt v) =
353    Hhalt (weaken_val_ts ts0_r ts_r v)
354
355 weaken_tuple :: EnvRep hs1 -> MapT (CodeBlockH hs0) hs ->
356                               MapT (CodeBlockH (Cat hs0 hs1)) hs
357 weaken_tuple hs1_r M0 = M0
358 weaken_tuple hs1_r (Ms (Hblock t_r e) t) =
359    Ms (Hblock t_r (weaken_exp_hs hs1_r e))
360       (weaken_tuple hs1_r t)
361
362
363 tp_shift_valH :: ValH (i, ts, fs) s -> ValH (S i, ShiftEnv ts, fs) (Shift s)
```

```
364 tp_shift_valH (Hvar i) = Hvar (tr_index_shift i)
365 tp_shift_valH (Hlam i) = Hlam i
366
367 tp_shift_tupleH ::    MapT (ValH (i,   ts, fs)) t
368                    -> MapT (ValH (S i, ShiftEnv ts, fs)) (ShiftEnv t)
369 tp_shift_tupleH M0 = M0
370 tp_shift_tupleH (Ms v vs) = Ms (tp_shift_valH v) (tp_shift_tupleH vs)
371
372
373 shift_map ::     MapT (ValH (i, ts0,            fs)) ts
374               -> MapT (ValH (S i, ShiftEnv ts0, fs)) (ShiftEnv ts)
375 shift_map M0 = M0
376 shift_map (Ms a m) = Ms (tp_shift_valH a)
377                         (shift_map m)
378
379 cat_tup ::   MapT (CodeBlockH hs) hs1  -> MapT (CodeBlockH hs) hs2
380           -> MapT (CodeBlockH hs) (Cat hs1 hs2)
381 cat_tup M0         t2 = t2
382 cat_tup (Ms (Hblock t_r e1) t1) t2 = Ms (Hblock t_r e1) (cat_tup t1 t2)
383
384 cmb_tup ::
385      EnvRep hs0 -> EnvRep hs1 -> EnvRep hs2
386   -> MapT (CodeBlockH (Cat hs0 hs1))           hs1
387   -> MapT (CodeBlockH (Cat (Cat hs0 hs1) hs2)) hs2
388   -> MapT (CodeBlockH (Cat (Cat hs0 hs1) hs2)) (Cat hs1 hs2)
389 cmb_tup hs0_r hs1_r hs2_r tup1 tup2 =
390   cat_tup (weaken_tuple hs2_r tup1) tup2
391
392
393 ----------------------------------------------------------------------------
394 -- Lemmas
395
396 lemma_cat_shift ::    EnvRep ts0
397                    -> EnvRep ts
398                    -> Equiv (ShiftEnv (Cat ts0 ts))
399                             (Cat (ShiftEnv ts0) (ShiftEnv ts))
400 lemma_cat_shift (M0::EnvRep ts0) (r::EnvRep ts)  =
401   case (cat_nil r, cat_nil (shift_env r)) of
402     (Equiv, Equiv) -> Equiv
403 lemma_cat_shift (Ms t_r ts0_r) (ts_r::EnvRep ts)  =
404   case lemma_cat_shift ts0_r ts_r of
405     Equiv -> Equiv
406
```

TAL.hs

```
1  {-# OPTIONS -fglasgow-exts #-} {-
2
3     Typed assembly language (TAL) syntax
4
5  -}
6
7  module TAL (
8    ValT(..), CodeBlockT(..), Instr(..), ProgramT(..),
9    Update, Sub(..), updateA
10 ) where
11
12 import Tp
13
14
15 data ProgramT where
16   Tprog :: EnvRep cs
17         -> MapT (CodeBlockT cs) cs
18         -> Instr cs (Code Z ())
19         -> ProgramT
20
21 data CodeBlockT g t where
22   Tblock ::   TypeRep (Code k rs)
23            -> Instr cs (Code k rs)
24            -> CodeBlockT cs (Code k rs)
25
26 data ValT csrs t where
27   Treg    :: Index rs t                       -> ValT (cs, i, rs) t
28   Tlabel  :: Index cs t                       -> ValT (cs, i, rs) t
29   Ttp_app :: NatRep k -> EnvRep s -> TypeRep t ->
30              ValT g (Code (S k) s)           -> ValT g (Code k (SubstEnv s t k))
31   Tpack   :: TypeRep s -> TypeRep t ->
32              ValT g (Subst s t Z)            -> ValT g (Exists s)
33   Tnum    :: Int                              -> ValT g Int
34
35
36 -- well-typed instructions sequences
37 --   cs = type of the labels / code blocks
38 --   i  = # of type variables in scope
39 --   rs = type of the registers
40 data Instr cs t where
41   ARITH ::     PrimOp
42             -> NatRep d                                         {- rd -}
43             -> Index rs Int                                     {- rs -}
44             -> ValT (cs, i, rs) Int                             {- v  -}
45             -> Instr cs (Code i (Update rs d Int))
46             -> Instr cs (Code i rs)
47
48   BNZ   ::     Sub rs t
49             -> Index rs Int                                     {- r  -}
50             -> ValT (cs, i, rs) (Code Z t)                      {- v  -}
51             -> Instr cs (Code i rs)
```

```
52              -> Instr cs (Code i rs)
53
54   MV     ::     NatRep d                                        {- rd -}
55              -> ValT (cs, i, rs) t
56              -> Instr cs (Code i (Update rs d t))
57              -> Instr cs (Code i rs)
58
59   UNPACK ::     NatRep d                                        {- rd -}
60              -> ValT (cs, i, rs) (Exists s)                     {- v  -}
61              -> Instr cs (Code (S i) (Update (ShiftEnv rs) d s))
62              -> Instr cs (Code i rs)
63
64   MKTUP ::      NatRep d                                        {- rd -}
65              -> MapT (ValT (cs, i, rs)) t
66              -> Instr cs (Code i (Update rs d (Tup t)))
67              -> Instr cs (Code i rs)
68
69   LD     ::     NatRep d                                        {- rd -}
70              -> Index rs (Tup tup)                              {- rs -}
71              -> Index tup t
72              -> Instr cs (Code i (Update rs d t))
73              -> Instr cs (Code i rs)
74
75   JMP    ::     Sub rs t
76              -> ValT (cs, i, rs) (Code Z t)                     {- v  -}
77              -> Instr cs (Code i rs)
78
79   HALT   ::     Instr cs (Code i (t, rs))
80
81
82 type family Update rs i t
83 type instance Update (s, ts) Z t     = (t, ts)
84 type instance Update ()      Z t     = (t, ())
85 type instance Update (s, ts) (S n) t = (s, Update ts n t)
86
87 updateA :: EnvRep ts -> NatRep i -> TypeRep t -> EnvRep (Update ts i t)
88 updateA (Ms _ ts_r) Nz t_r         = Ms t_r ts_r
89 updateA M0          Nz t_r         = Ms t_r M0
90 updateA (Ms s_r ts_r) (Ns n_r) t_r = Ms s_r (updateA ts_r n_r t_r)
91
92 data Sub rs' rs where
93   S0 :: Sub rs' ()
94   Sx :: Sub rs' rs -> Sub (s, rs') (s, rs)
95
```

CG.hs

```
{-# OPTIONS -fglasgow-exts -fallow-undecidable-instances #-} {-

   Code generation

-}

module CG (
  cg
) where

import Tp

import LH
import TAL

--------------------------------------------------------------------------------
-- Translation [TYPES]

type family Ttype t
type instance Ttype (Cont k t) = Code k (Ttype t, ())
type instance Ttype (Exists t) = Exists (Ttype t)
type instance Ttype (Var v)    = Var v
type instance Ttype (Tup t)    = Tup (Tenv t)
type instance Ttype Int        = Int
type instance Ttype (Closed t) = Ttype t

type family Tenv ts
type instance Tenv ()      = ()
type instance Tenv (s, ts) = (Ttype s, Tenv ts)


-- type families reified as term-level functions

tType :: TypeRep t -> TypeRep (Ttype t)
tType (Rcont k t_r) = Rcode k (Ms (tType t_r) M0)
tType (Rexists t_r) = Rexists (tType t_r)
tType (Rvar v)   = Rvar v
tType (Rtup tup)    = Rtup (tEnv tup)
tType Rint = Rint
tType (Rclosed t)   = tType t

tEnv :: EnvRep t -> EnvRep (Tenv t)
tEnv M0 = M0
tEnv (Ms t_r ts_r) = Ms (tType t_r) (tEnv ts_r)


--------------------------------------------------------------------------------
-- Translation [TERMS]

--------------- values

```

```
52  cg_val ::    MapT (Index rs) (Tenv ts)
53            -> MapT (Index cs) (Tenv fs)
54            -> ValH (i, ts, fs) t
55            -> ValT (cs, i, rs) (Ttype t)
56  cg_val m_r m_l (Hnum n) = Tnum n
57  cg_val m_r  _  (Hvar i) = Treg    (lookupT m_r (tr_a i))
58  cg_val _   m_l (Hlam i) = Tlabel (lookupT m_l (tr_a i))
59  cg_val m_r m_l (Hpack s_r t_r v) =
60    case lemma_ttype_subst Nz s_r t_r of
61      Equiv -> Tpack (tType s_r) (tType t_r) (cg_val m_r m_l v)
62  cg_val m_r m_l (Htp_app k_r s_r t_r v) =
63    case lemma_ttype_subst k_r s_r t_r of
64      Equiv -> Ttp_app k_r (Ms (tType s_r) M0) (tType t_r) (cg_val m_r m_l v)
65  cg_val m_r m_l (Hdisclose v) = cg_val m_r m_l v
66
67
68  --------------- expressions
69
70  data CG cs0 i rs where
71    CG ::    EnvRep cs
72          -> MapT (CodeBlockT (Cat cs0 cs)) cs
73          -> Instr (Cat cs0 cs) (Code i rs)
74          -> CG cs0 i rs
75
76  cg_exp ::    NatRep i
77            -> EnvRep ts
78            -> EnvRep cs0
79            -> EnvRep rs
80            -> MapT (Index cs0) (Tenv fs)
81            -> MapT (Index rs)  (Tenv ts)
82            -> ExpH (i, ts, fs)
83            -> CG cs0 i rs
84
85  cg_exp i_r ts_r cs0_r rs_r c_m r_m
86         (Hlet _ s_r (v::ValH (i,ts,fs) s) e) =
87    case fresh rs_r (tType s_r) of
88      Fresh r_n rs'_r reg_ix ->
89        case cg_exp i_r (Ms s_r ts_r) cs0_r rs'_r c_m
90                    (Ms reg_ix (mapT (update_index r_n (tType s_r)) r_m))
91                    e of
92          CG cs_r cs instr ->
93            CG cs_r
94               cs
95               (MV r_n (cg_val r_m (mapT (weaken_index cs_r) c_m) v) $
96                instr)
97
98  cg_exp i_r ts_r cs0_r rs_r c_m r_m
99         (Hlet_prim  _ op (v1::ValH (i,ts,fs) Int)
100                         (v2::ValH (i,ts,fs) Int) e) =
101   case fresh rs_r Rint of
102     Fresh r_n rs'_r reg_ix ->
103       case cg_exp i_r (Ms Rint ts_r) cs0_r rs'_r c_m
```

```
104                   (Ms reg_ix (mapT (update_index r_n Rint) r_m))
105                   e of
106         CG cs_r cs instr ->
107           case update_twice rs_r r_n Rint Rint of
108             Equiv ->
109               CG cs_r
110                  cs
111                  (MV r_n (cg_val r_m (mapT (weaken_index cs_r) c_m) v1) $
112                   ARITH op r_n reg_ix
113                         (cg_val r_m (mapT (weaken_index cs_r) c_m) v2) $
114                   instr)
115
116 cg_exp _ _ cs0_r (Ms _ rs_r) c_m r_m
117        (Happ (v1::ValH (i,ts,fs) (Cont Z t))
118              (v2::ValH (i,ts,fs) t)) =
119   let s_r :: TypeRep (Code Z (Ttype t, ())) = undefined
120       v1' :: ValH (i,ts,fs) (Cont Z t) = v1
121   in case fresh rs_r s_r of
122        Fresh r_n s'_r
123              reg_ix ->
124          case cat_nil cs0_r of
125            Equiv ->
126              CG M0 M0
127                 (MV (Ns r_n) (cg_val r_m c_m v1) $
128                  MV Nz       (cg_val (mapT (update_index (Ns r_n) s_r) r_m)
129                                      c_m v2) $
130                  JMP (Sx S0)
131                      (Treg (Ix reg_ix)))
132
133 cg_exp _ _ cs0_r M0 c_m r_m
134        (Happ (v1::ValH (i,ts,fs) (Cont Z t))
135              (v2::ValH (i,ts,fs) t)) =
136    let t_r :: TypeRep (Ttype t) = undefined
137    in case cat_nil cs0_r of
138         Equiv ->
139           CG M0 M0
140              (-- since rs = (), v1 can't be in a register, it must be a label
141               MV Nz (cg_val r_m c_m v2) $
142               JMP (Sx S0)
143                   (cg_val (mapT (update_index Nz t_r) r_m) c_m v1))
144
145 cg_exp i_r ts_r cs0_r (rs_r::EnvRep rs) c_m r_m
146        (Hif0 v e1 e2) =
147   let c_r = Ms (Rcode i_r rs_r) M0 in
148   case fresh rs_r Rint of
149     Fresh (r_n::NatRep n) s'_r reg_ix ->
150      let rs'_r = updateA rs_r r_n Rint
151      in case cg_exp i_r ts_r cs0_r rs_r c_m r_m e2 of
152         CG cs_r cs instr2 ->
153           case cg_exp i_r ts_r (catT cs0_r cs_r)
154                       rs'_r
155                       (mapT (weaken_index cs_r) c_m)
```

```
156                       (mapT (update_index r_n Rint) r_m) e1 of
157             CG cs'_r cs' instr1 ->
158               case (lemma_cat_assoc4b cs0_r cs_r cs'_r c_r,
159                     lemma_cat_assoc cs0_r cs_r cs'_r) of
160                 (Equiv, Equiv) ->
161                   CG (catT (catT cs_r cs'_r) c_r)
162                      (weaken_seg c_r $
163                         cat_map3 (weaken_seg cs'_r cs)
164                                  cs'
165                                  (weaken_seg cs'_r
166                                      (Ms (Tblock (Rcode i_r rs_r) instr2) M0)))
167                      (MV r_n
168                          (cg_val r_m
169                                  (mapT (weaken_index
170                                            (catT (catT cs_r cs'_r) c_r))
171                                        c_m)
172                                  v) $
173                       BNZ (mkSub rs_r r_n)
174                           reg_ix
175                           (tp_app_multi i_r i_r rs_r
176                               (Tlabel (mkNewIndex cs0_r cs_r cs'_r rs_r)))
177                           (weaken_instr c_r instr1))
178
179 cg_exp i_r ts_r cs0_r rs_r c_m r_m
180        (Hlet_tup _ (tup_r::EnvRep t) tup e) =
181   let t_r :: TypeRep (Ttype (Tup t))
182       t_r = tType (Rtup tup_r)
183   in case fresh rs_r t_r of
184        Fresh (r_n::NatRep n) rs'_r reg_ix ->
185          case cg_exp i_r (Ms (Rtup tup_r) ts_r) cs0_r rs'_r c_m
186                      (Ms reg_ix (mapT (update_index r_n t_r) r_m))
187                      e of
188            CG cs_r
189               cs
190               instr ->
191              CG cs_r
192                 cs
193                 (MKTUP r_n
194                        (mapA (cg_val r_m (mapT (weaken_index cs_r) c_m))
195                              tup)
196                  instr)
197
198 cg_exp i_r ts_r cs0_r (rs_r::EnvRep rs) c_m r_m
199        (Hlet_proj _ t_r
200                   (v::ValH (i, ts, fs) (Tup s))
201                   (i::Index s t) e) =
202   let s_r :: TypeRep (Tup (Tenv s)) = undefined
203   in case fresh rs_r s_r of
204        Fresh (r_n::NatRep n)
205              rs'_r
206              reg_ix ->
207          case update_twice rs_r r_n (undefined :: TypeRep (Tup (Tenv s)))
```

```
208                                     (undefined :: TypeRep (Ttype t)) of
209           Equiv ->
210             case cg_exp i_r (Ms t_r ts_r) cs0_r
211                          (updateR rs_r r_n (tType t_r))
212                          c_m
213                          (Ms (update_same_index r_n (tType t_r) reg_ix)
214                              (mapT (update_index r_n (tType t_r)) r_m))
215                          e of
216                CG cs_r cs
217                   instr ->
218                     CG cs_r
219                        cs
220                        (MV r_n (cg_val r_m (mapT (weaken_index cs_r) c_m) v) $
221                         LD r_n reg_ix (tr_a i)
222                         instr)
223
224
225 cg_exp i_r ts_r cs0_r rs_r c_m r_m
226        (Hunpack t_r (v::ValH (i,ts,fs) (Exists s)) e) =
227   let s_r :: TypeRep (Ttype s)
228       s_r = error "w"
229   in
230   case fresh (uEnv Nz (Ns Nz) rs_r) s_r of
231     Fresh r_n rs'_r reg_ix ->
232       case lemma_tenv_u ts_r of
233         Equiv ->
234           case cg_exp (Ns i_r) (Ms t_r (uEnv Nz (Ns Nz) ts_r)) cs0_r
235                       rs'_r
236                       c_m
237                       (Ms reg_ix (mapT (update_index r_n s_r)
238                                        (shift_map r_m)))
239                   e of
240             CG cs_r cs instr ->
241               CG cs_r
242                 cs
243                 (UNPACK r_n (cg_val r_m (mapT (weaken_index cs_r) c_m) v) $
244                         instr)
245
246 cg_exp _ _ cs0_r rs_r c_m r_m
247        (Hhalt v) =
248   CG M0 M0
249      (case (rs_r, cat_nil cs0_r) of
250         (M0, Equiv)     -> MV Nz (cg_val r_m c_m v) $
251                            HALT
252         (Ms _ _, Equiv) -> MV Nz (cg_val r_m c_m v) $
253                            HALT)
254
255
256 --------------- programs
257
258 data CG_tup cs0 fs where
259   CG_tup ::
```

```
260       EnvRep cs
261    -> MapT (CodeBlockT (Cat cs0 cs)) (Tenv fs)
262    -> MapT (CodeBlockT (Cat cs0 cs)) cs
263    -> CG_tup cs0 fs
264
265 cg :: ProgramH -> ProgramT
266 cg (Hletrec es (e::ExpH (Z, (), fs0))) =
267   let fs0_r :: EnvRep fs0 = mkEnvRep es
268       ixsA :: MapT (Index (Tenv fs0)) (Tenv fs0)
269       ixsA = mk_indices (tEnv fs0_r)
270
271       cg_tup ::    forall fs cs.
272                    MapT (CodeBlockH fs0) fs
273                 -> MapT (Index fs0) fs -- indices
274                 -> EnvRep cs
275                 -> CG_tup (Cat (Tenv fs0) cs) fs
276       cg_tup M0 M0 cs_r =
277          case cat_nil cs_r of
278            Equiv -> CG_tup M0 M0 M0
279       cg_tup (Ms (Hblock (cont_r@(Rcont i_r s_r))
280                         (e::ExpH (k, (t, ()), fs0)))
281                  (es::MapT (CodeBlockH fs0) fs1))
282              (Ms i   (is::MapT (Index fs0) fs1))
283              (cs_r::EnvRep cs) =
284        let cs0_r :: EnvRep (Tenv fs0)
285            cs0_r = tEnv (fs0_r) in
286         case cg_exp i_r (Ms s_r M0)
287                    (catT (tEnv fs0_r) cs_r)
288                    (Ms (tType s_r) M0)
289                    (mapT (weaken_index cs_r) ixsA)
290                    (Ms I0 M0)
291                    e of
292           CG (cs'_r::EnvRep cs')
293              (instrs ::MapT (CodeBlockT (Cat (Cat (Tenv fs0) cs) cs')) cs')
294              (instr::Instr (Cat (Cat (Tenv fs0) cs) cs')
295                             (Code k (Ttype t, ()))) ->
296             case cg_tup es is (catT cs_r cs'_r) of
297               CG_tup (cs''_r::EnvRep cs'')
298                      (root_instrs::
299                       MapT (CodeBlockT (Cat (Cat (Tenv fs0) (Cat cs cs')) cs''))
300                             (Tenv fs1))
301                      (extra_instrs::
302                       MapT (CodeBlockT (Cat (Cat (Tenv fs0) (Cat cs cs')) cs''))
303                             cs'') ->
304                 case (lemma_cat_assoc4 cs0_r cs_r cs'_r cs''_r,
305                       lemma_cat_assoc4a cs0_r cs_r cs'_r cs''_r) of
306                   (Equiv, Equiv) ->
307                     CG_tup (catT cs'_r cs''_r)
308                            (Ms (Tblock (tType cont_r)
309                                        (weaken_instr cs''_r instr))
310                                root_instrs)
311                            (cat_map (weaken_seg cs''_r instrs)
```

```
312                                      extra_instrs)
313    in case cg_tup es (mk_indices fs0_r)
314            M0 of
315         CG_tup (cs_r::EnvRep cs)
316                (root_instrs::
317                  MapT (CodeBlockT (Cat (Cat (Tenv fs0) ()) cs))
318                       (Tenv fs0))
319                (extra_instrs::
320                  MapT (CodeBlockT (Cat (Cat (Tenv fs0) ()) cs)) cs) ->
321           case cg_exp Nz M0 (catT (tEnv fs0_r) cs_r) M0
322                       (mapT (weaken_index cs_r) ixsA)
323                       M0 e of
324             CG (cs'_r::EnvRep cs')
325                (instrs :: MapT (CodeBlockT (Cat (Cat (Tenv fs0) cs) cs')) cs')
326                (instr  :: Instr (Cat (Cat (Tenv fs0) cs) cs') (Code Z ())) ->
327               case cat_nil (tEnv fs0_r) of
328                 Equiv ->
329                   Tprog ((catT (catT (tEnv fs0_r) cs_r) cs'_r))
330                         (cat_map3 (weaken_seg cs'_r root_instrs)
331                                   (weaken_seg cs'_r extra_instrs)
332                                   instrs)
333                         instr
334
335
336  --------------------------------------------------------------------------------
337  -- Index manipulations
338
339  data FreshReg ts t where
340    Fresh ::   NatRep n
341            -> EnvRep (Update ts n t)
342            -> Index (Update ts n t) t
343            -> FreshReg ts t
344
345  fresh :: EnvRep ts -> TypeRep t -> FreshReg ts t
346  fresh M0 t_r = Fresh Nz (Ms t_r M0) I0
347  fresh (Ms s_r ts_r) t_r =
348    case fresh ts_r t_r of
349      Fresh n_r ts'_r i ->
350        Fresh (Ns n_r) (Ms s_r ts'_r) (Ix i)
351
352
353  mk_indices :: EnvRep fs -> MapT (Index fs) fs
354  mk_indices M0 = M0
355  mk_indices (Ms _ m) = Ms I0 (mapT Ix (mk_indices m))
356
357  mkSub :: EnvRep rs -> NatRep n -> Sub (Update rs n Int) rs
358  mkSub M0 _ = S0
359  mkSub (Ms _ m) (Ns n) = Sx (mkSub m n)
360
361
362  shift_map ::    MapT (Index rs)             ats
363              -> MapT (Index (ShiftEnv rs))  (ShiftEnv ats)
```

```
364 shift_map M0 = M0
365 shift_map (Ms i m) = Ms (tr_index_shift i) (shift_map m)
366
367 mapA :: (forall t . c t -> d (Ttype t)) -> MapT c ts -> MapT d (Tenv ts)
368 mapA f M0 = M0
369 mapA f (Ms e tb) = Ms (f e) (mapA f tb)
370
371 tr_a :: t' ~ Ttype t => Index ts t -> Index (Tenv ts) t'
372 tr_a I0 = I0
373 tr_a (Ix i) = Ix (tr_a i)
374
375 updateR ::   EnvRep rs -> NatRep n -> TypeRep t
376           -> EnvRep (Update rs n t)
377 updateR (Ms s_r rs_r) Nz t_r = Ms t_r rs_r
378 updateR M0            Nz t_r = Ms t_r M0
379 updateR (Ms s_r rs_r) (Ns n) t_r = Ms s_r (updateR rs_r n t_r)
380
381 update_twice ::
382      EnvRep rs -> NatRep n -> TypeRep s -> TypeRep t
383   -> Equiv (Update rs n t)
384            (Update (Update rs n s) n t)
385 update_twice M0 Nz _ _ = Equiv
386 update_twice (Ms _ rs_r) Nz s_r t_r = Equiv
387 update_twice (Ms _ rs_r) (Ns n) s_r t_r =
388   case update_twice rs_r n s_r t_r of Equiv -> Equiv
389
390 -- safe if the index is NOT the same as n!
391 update_index :: NatRep n -> TypeRep s -> Index ts t -> Index (Update ts n s) t
392 update_index Nz     s_r (Ix i) = Ix i
393 update_index (Ns n) s_r (Ix i) = Ix (update_index n s_r i)
394 update_index (Ns n) s_r I0     = I0
395
396 -- safe if the index IS the same as n!
397 update_same_index ::    NatRep n
398                      -> TypeRep s
399                      -> Index ts t
400                      -> Index (Update ts n s) s
401 update_same_index Nz s_r I0 = I0
402 update_same_index (Ns n) s_r (Ix i) = Ix (update_same_index n s_r i)
403
404
405 cat_map :: MapT c ts1  -> MapT c ts2 -> MapT c (Cat ts1 ts2)
406 cat_map M0         t2 = t2
407 cat_map (Ms e1 t1) t2 = Ms e1 (cat_map t1 t2)
408
409 cat_map3 ::
410      MapT c ts0
411   -> MapT c ts
412   -> MapT c ts'
413   -> MapT c (Cat (Cat ts0 ts) ts')
414 cat_map3 m0 m m' = cat_map (cat_map m0 m) m'
415
```

```
416
417 weaken_val :: EnvRep cs' -> ValT (cs, i, rs) t -> ValT (Cat cs cs', i, rs) t
418 weaken_val cs' (Tlabel l) = Tlabel (weaken_index cs' l)
419 weaken_val _ (Treg r) = Treg r
420 weaken_val _ (Tnum n) = Tnum n
421 weaken_val cs' (Tpack s_r t_r v) = Tpack s_r t_r (weaken_val cs' v)
422 weaken_val cs' (Ttp_app i_r s_r t_r v) = Ttp_app i_r s_r t_r (weaken_val cs' v)
423
424 weaken_block :: EnvRep cs' -> CodeBlockT cs rs -> CodeBlockT (Cat cs cs') rs
425 weaken_block cs' (Tblock t_r k) = Tblock t_r (weaken_instr cs' k)
426
427
428 weaken_instr :: EnvRep cs' -> Instr cs rs -> Instr (Cat cs cs') rs
429 weaken_instr cs' v =
430   let v_ = weaken_val cs'
431       i_ = weaken_instr cs'
432   in case v of
433     ARITH p rd rs v k -> ARITH p rd rs (v_ v) (i_ k)
434     BNZ sub r v k     -> BNZ sub r (v_ v) (i_ k)
435     LD  rd rs i k     -> LD rd rs i (i_ k)
436     MKTUP rd tup k    -> MKTUP rd (mapT v_ tup) (i_ k)
437     MV rd v k         -> MV rd (v_ v) (i_ k)
438     UNPACK rd v k     -> UNPACK rd (v_ v) (i_ k)
439     JMP sub v         -> JMP sub (v_ v)
440     HALT              -> HALT
441
442 weaken_seg ::    EnvRep cs'
443               -> MapT (CodeBlockT cs) ts
444               -> MapT (CodeBlockT (Cat cs cs')) ts
445 weaken_seg cs'_r m = mapT (weaken_block cs'_r) m
446
447 mkEnvRep :: forall fs0 fs . MapT (CodeBlockH fs0) fs -> EnvRep fs
448 mkEnvRep M0 = M0
449 mkEnvRep (Ms (Hblock s_r _) m) = Ms s_r (mkEnvRep m)
450
451 mkNewIndex ::
452      EnvRep cs0 -> EnvRep cs -> EnvRep cs' -> EnvRep rs
453   -> Index (Cat (Cat cs0 (Cat cs cs'))
454                 (Code i rs, ()))
455            (Code i rs)
456 mkNewIndex cs0_r cs_r cs'_r rs_r =
457   newIndex (catT cs0_r (catT cs_r cs'_r))
458
459
460 --------------------------------------------------------------------------------
461 -- Multiple type applications
462
463 type family MultApp j c
464 type instance MultApp (S j) (Code (S k) t) =
465               MultApp j     (Code k     (SubstEnv t (Var j) k))
466 type instance MultApp Z t = t
467
```

```
468 multAppT :: NatRep j -> TypeRep c -> TypeRep (MultApp j c)
469 multAppT (Ns j_r) (Rcode (Ns k_r) t_r) =
470     multAppT j_r (Rcode k_r (substEnv t_r (Rvar j_r) k_r))
471 multAppT Nz t = t
472
473 lemma_mult_app ::
474   NatRep k -> EnvRep t ->
475   Equiv (MultApp k (Code k t))
476         (Code Z t)
477 lemma_mult_app k_r t_r =
478   case typesEqual (multAppT k_r (Rcode k_r t_r))
479                   (Rcode Nz t_r) of
480     Just Equiv -> Equiv
481
482 multApp ::      NatRep j -> NatRep k -> EnvRep t
483              -> ValT (cs, i, ts) (Code k t)
484              -> ValT (cs, i, ts) (MultApp j (Code k t))
485 multApp (Ns j_r) (Ns k_r) t_r v =
486      multApp j_r k_r (substEnv t_r (Rvar j_r) k_r)
487                      (Ttp_app k_r t_r (Rvar j_r) v)
488 multApp Nz k_r _ e = e
489
490 tp_app_multi ::      NatRep j
491                   -> NatRep k
492                   -> EnvRep t
493                   -> ValT (cs, j, ts) (Code k t)
494                   -> ValT (cs, j, ts) (Code Z t)
495 tp_app_multi j_r k_r t_r v =
496   case lemma_mult_app k_r t_r of
497     Equiv ->
498        multApp k_r k_r t_r v
499
500
501 -----------------------------------------------------------------------------
502 -- Lemmas
503
504 lemma_ttype_subst ::
505    NatRep k -> TypeRep s -> TypeRep t ->
506      Equiv (Subst (Ttype s) (Ttype t) k)
507            (Ttype (Subst s t k))
508 lemma_ttype_subst k_r s_r t_r =
509   case typesEqual (substT (tType s_r) (tType t_r) k_r)
510                   (tType (substT s_r t_r k_r)) of
511     Just Equiv -> Equiv
512
513 lemma_tenv_u ::
514   EnvRep ts -> Equiv (Tenv (Uenv Z (S Z) ts))
515                      (Uenv Z (S Z) (Tenv ts))
516 lemma_tenv_u ts_r =
517   case envEqual (tEnv (uEnv Nz (Ns Nz) ts_r))
518                 (uEnv Nz (Ns Nz) (tEnv ts_r))
519    of Just Equiv -> Equiv
```

```
520
521 lemma_cat_assoc4 ::
522      EnvRep a -> EnvRep b -> EnvRep c -> EnvRep d
523   -> Equiv (Cat (Cat a b) (Cat c d))
524            (Cat (Cat a (Cat b c)) d)
525 lemma_cat_assoc4 a b c d =
526   case lemma_cat_assoc a b c of
527     Equiv ->
528       case lemma_cat_assoc (catT a b) c d of
529         Equiv -> Equiv
530
531 lemma_cat_assoc4a ::
532      EnvRep a -> EnvRep b -> EnvRep c -> EnvRep d
533   -> Equiv (Cat (Cat a b) (Cat c d))
534            (Cat (Cat (Cat a b) c) d)
535 lemma_cat_assoc4a a b c d =
536   case lemma_cat_assoc a b c of
537     Equiv ->
538       case lemma_cat_assoc (catT a b) c d of
539         Equiv -> Equiv
540
541 lemma_cat_assoc4b ::
542      EnvRep a -> EnvRep b -> EnvRep c -> EnvRep d
543   -> Equiv (Cat a (Cat (Cat b c) d))
544            (Cat (Cat (Cat a b) c) d)
545 lemma_cat_assoc4b a b c d =
546   case lemma_cat_assoc a b c of
547     Equiv ->
548       case lemma_cat_assoc a (catT b c) d of
549         Equiv -> Equiv
550
```

Main.hs

```
1  {-# OPTIONS -fglasgow-exts #-} {-
2
3     Compiler driver
4
5  -}
6
7  module Main (
8    compile
9  ) where
10
11 import Tp
12 import Src
13 import TAL
14
15 import CPS
16 import ToB
17 import CC
18 import Hoist
19 import CG
20
21 compile :: (forall a. Exp a t) -> ProgramT
22 compile = cg . hoist . cc . toB . cps
23
```