Verified Optimizations for Functional Languages

Zoe Paraskevopoulou

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Andrew W. Appel

November 2020

Abstract

Coq is one of the most widely adopted proof development systems. It allows programmers to write purely functional programs and verify them against specifications with *machine-checked* proofs. After verification, one can use Coq's extraction plugin to obtain a program (in OCaml, Haskell, or Scheme) that can be compiled and executed. However, bugs in either the extraction function or the compiler of the extraction language may cause the executable to exhibit unexpected behaviors, rendering source-level verification futile.

A verified compiler is a compiler whose output provably preserves the semantics of the source language. CertiCoq is a verified compiler, currently under development, for Coq's specification language, Gallina. CertiCoq targets Clight, a subset of the C language, that can be compiled with the CompCert verified compiler to obtain a *certified* executable, bridging the gap between the formally verified source program and the compiled target program.

In this thesis, I present the implementation and verification of CertiCoq's optimizing middle-end pipeline. CertiCoq's middle end consists of seven different transformations and is responsible for efficiently compiling an untyped purely functional intermediate language to a subset of the same language, which can be readily compiled to a first-order, low-level intermediate language. CertiCoq's middle-end pipeline performs crucial optimizations for functional languages including closure conversion, uncurrying, shrink-reductions and inlining. It advances the state of the art of verified optimizing compilers for functional languages by implementing more efficient closure-allocation strategies.

For proving CertiCoq correct, I develop a framework based on the technique of logical relations, making novel technical contributions. I extend logical relations with notions of relational preconditions and postconditions that facilitate reasoning about the resource consumption of programs simultaneously with functional correctness. I demonstrate how this enables reasoning about preservation of non-terminating behaviors, which is not supported by traditional logical relations. Moreover, I develop a novel, lightweight technique that allows logical-relation proofs to be composed in order to obtain a top-level compositional compiler correctness theorem. This technique is used to obtain a separate compilation theorem that guarantees that programs compiled separately through CertiCoq, perhaps using different sets of optimizations, can be safely linked at the target level. Lastly, I use the framework to prove that CertiCoq's closure conversion is not only functionally correct but also safe for time and space, meaning that it is guaranteed to preserve the asymptotic time and space complexity of the source program.

Acknowledgements

First and foremost, I would like to thank my advisor Andrew Appel. I am deeply grateful to him for the ample support and guidance throughout the years of my PhD. He always gave me the freedom to pursue my ideas, while encouraging me and valuing my work. His dedication, his vision for machine-checked proofs, and his capacity for acceptance and understanding will always be an inspiration. I am also thankful to him for his detailed feedback during the process of writing this dissertation.

I want to thank the members of my thesis committee Amal Ahmed, Dave Walker, Aarti Gupta and Zachary Kincaid. I am especially thankful to the readers of my dissertation, Amal Ahmed and Dave Walker, for undertaking the laborious task of reading my dissertation and for providing helpful and detailed feedback.

I want to thank the members of the CertiCoq team for the fruitful collaboration, the interesting discussions, and their help with technical issues.

I am grateful to Leonidas Lampropoulos, Yannick Forster and Anvay Grover for their valuable feedback on this dissertation.

I am thankful to Anvay Grover and Katja Vassilev for trusting me to supervise them during their senior thesis work and mentor them at the beginning of their academic journey. I wish them the very best in their future endeavors.

I am very thankful to John Li for his contributions to the verification of the $\lambda_{\texttt{ANF}}$ pipeline.

I want to sincerely thank Nikhil Swamy, Jonathan Protzenko, and Tahina Ramananandro for hosting me as an intern at MSR Redmond for two consecutive summers and giving me the opportunity to work on interesting problems. It was a great pleasure working with them.

I am deeply grateful to my undergraduate thesis advisor, Cătălin Hrițcu, and my master thesis advisor, Deeepak Garg. I feel extremely lucky to have been advised by them. Their high research standards, their genuine enthusiasm for research, and their attention to technical detail has shaped me as a researcher. Working with them provided me with inspiration and technical knowledge that was invaluable for my graduate studies.

I wand to thank my labmates at the Princeton programming languages lab. Their presence made my years as a PhD student more pleasant and more interesting.

I want to thank the stuff of the Computer Science Department and the Graduate School for helping make our graduate studies as smooth as possible. I am especially thankful to Nicki Mahler for her help in coordinating the defense and the submission of this thesis.

I am deeply thankful to the Seibel Foundation and the Seeger Center for Hellenic Studies for the generous scholarships during my PhD studies.

Last but not least, I want to thank all of my family and friends in Greece and in the USA for their presence, their support and all the moments we have shared. I'm especially thankful to my mother, Maria, and my father Aris. This thesis is dedicated to my grandmother, Zoe, for being an endless source of love and support throughout my life.

This material is based upon work supported by the National Science Foundation under Grants CCF-1407794 and CCF-1521602.

To my Grandma.

Στη Γιαγιά μου.

Contents

List of Figures

Chapter 1

Introduction

A formal proof in a formal system of logic is a sequence of formulas expressed in the formal language of the system, such that each formula in the sequence is either an axiom or it can be derived by previous formulas in the sequence using an inference rule. A formal proof can be *machine checked*: given a computer representation of a formal system and a proof written in it, a computer program can decide if the proof is valid. This is in contrast with usual mathematical proofs that argue about the truth of a statement using informal meta-mathematical language, and can conceal logical fallacies, or "proof bugs". The idea that computer programs can, and should, be used to verify the validity of mathematical proofs dates back to the early days of computing and led to the development of algorithms, languages, and tools that allow mathematicians and computer scientists to express, mechanically verify, or even automatically derive mathematical proofs [93, 136, 94].

Computer programs, just as proofs, can have bugs that prevent them from carrying out a computation correctly. With *formal verification*, we can reason about the absence of bugs in programs by using machine checked proofs to prove that they satisfy a formal specification. Therefore, we transfer the trust from the *implementation* to the *specification*. We trust such proofs because the soundness of the formal system has been demonstrated with a metatheoretical proof argument, giving us confidence that we cannot construct a proof for an absurd proposition. Nevertheless, the proof-checking program can be an intricate piece of software and as such, it can itself have bugs that may allow invalid proofs to be constructed. In addition, after the software is verified against its specification, it will be compiled to machine code and executed on a machine. For the specification to be met by the executable program, the executable is assumed *to have the same the computational behavior as the source program*, as this is prescribed by the semantics of the source language. Consequently, when we carry out machine-checked proofs about programs, we have to trust not only that the proof checker implements the formal system correctly, but also that the compiler and the runtime system correctly compile and execute the program. These, together with the formal specification of the verified program, constitute the *trusted computing base* (TCB) of formal verification. Of course, we can remove the implementations of the proof-checking program and compiler from the TCB, by mechanically verifying them with respect to the aforementioned specifications.

In this thesis, I explore the design, implementation and verification of a large part of the CertiCoq verified compiler for Coq: one of the most widely used proof assistants today that enables formally verified software to be built and run. In the process of building and verifying (part of) the CertiCoq compiler, I make technical contributions to the design space of verified compilers and the available proof techniques for proving compilers correct. In the rest of this introduction, I will give an overview of the Coq proof assistant, a brief introduction to compiler correctness and compilation of functional languages, and an overview of the work presented in this thesis.

1.1 The Coq Proof Assistant

A proof assistant is software comprising a proof checking program and an interactive proof-development environment that helps the user construct a proof. Coq [37], as well as a number of other proof assistants, is based on dependent type theory. In the type theoretic approach of proof-checking, the problem of proof checking reduces to the problem of type-checking. This is made possible by the so-called *propositions-as-types* interpretation [19, 135]: formulas and proofs in logical systems can be interpreted as types and typing derivations in type systems. The logical formalism on which Coq is based is the Calculus of Inductive Constructions (CiC) [38, 110]. In Coq, a logical proposition is a type and a proof of this proposition is a lambda-term that inhabits this type, known as the *proof term*. Types and proofs are written in the same specification language, Gallina. Gallina can be used to write computations *and* proofs about computations, both expressed as pure functional programs. The Coq kernel, which is implemented in OCaml, is used to type-check Gallina programs The TCB of Coq includes the Coq kernel as well as the OCaml compiler that is used to compile the implementation, the OCaml runtime system, and the C compiler that compiled the runtime system.

Certified Programming with Coq. But how does one go about building actual verified software in Coq? Coq itself provides an interpreter for Gallina terms, which is a part of the Coq kernel and implements term normalization required for type-checking. This, though, is an inefficient way of running software built in Coq. The Coq toolchain features an extraction mechanism [90, 89] that erases non-computational content from a Gallina program, *i.e.*, proofs and (dependent) types, and prints out a functional program (written in OCaml, Haskell or Scheme). A program that has been formally verified in Coq can be extracted to one of these languages, compiled, and executed.

Extraction is a particularly useful tool that enables practical software verification. It has been used to verify realistic, certified software, such as a verified C compiler [83], an operating system kernel [55], and a web browser kernel [66]. However, both the Coq extraction program and the compiler of the languages into which Coq is extracted are unverified; hence, they belong to the TCB of the formally verification in Coq.

Bugs have been found in all parts of Coq's TCB: the Coq kernel,[1] the extraction mechanism[2] and the OCaml compiler, defeating the purpose of formally verifying software in Coq.

CertiCoq [8] aims to reduce the TCB of Coq formal developments by providing a compilation pipeline which is itself written and verified in Coq. CertiCoq targets Clight, a large subset of the C language, that can be translated to machine code via the CompCert verified compiler [84]. Targeting Clight gives us the opportunity to take advantage of CompCert's already proved-correct optimizations for an imperative, low-level language (*e.g.*, register allocation), and focus on the efficient compilation of a higher-order functional language to a first-order representation. Moreover, by compiling to Clight we can target all of the architectures that CompCert already does (PowerPC, ARM, RISC-V and x86). Apart from providing a verified extraction pipeline for Coq programs, CertiCoq could be used in the future in combination with the MetaCoq [123, 124] project to further reduce the TCB of Coq programs. MetaCoq provides a Coq implementation of the Coq kernel in Coq and a proof that it correctly implements type checking in the Calculus of Inductive Constructions[3] correctly). Currently, there is no way of extracting and running the verified implementation of the type checker: extracting the verified type checker produces an ill-typed term [124]. Compiling MetaCoq's type checker with CertiCoq could be used to produce *a certified type checker for Coq*.

It is worth mentioning that no one can ever hope for a provably foolproof proof assistant, one without a TCB. First, from Göedel's incompleteness theorems, the soundness of the underlying proof system (assuming it is expressive enough) cannot be proved in the system itself. We can, nonetheless, have an implementation of the proof checking algorithm that is proved correct with respect to the specification of the theory (*e.g.*, the MetaCoq typechecker). Even so, the knot cannot be tied: a verified implementation of the theory needs a verified compiler and a verified compiler needs a verified implementation of theory. It is clear that we should start with some amount of trust [131]. But from a security standpoint, bugs in the Coq kernel or the OCaml compiler are much harder to be exploited if these programs are used only to proof check and compile specific programs (like the Coq kernel and the CertiCoq compiler), as opposed to arbitrary programs picked by an adversary.

[1] A list of critical bugs in the Coq kernel compiled by the Coq developers can be found in `https://github.com/coq/coq/blob/master/dev/doc/critical-bugs/`

[2] Extraction erases dependent Coq types by introducing unsafe OCaml operations, a frequent source of bugs in extracted Coq programs. Extracted Coq programs sometimes may fail to typecheck in the extraction language or even cause segmentation faults and other runtime errors. Examples of such bugs can be found in Coq's bug tracker `https://github.com/coq/coq/labels/part%3A%20extraction`. Xuan *et al.* [82] and Sozeau *et al.* [124] report verified artifacts for which extraction fails to generate a valid OCaml file. In Chapter 8, one of the benchmarks used for the evaluation of CertiCoq also fails to compile when extracted to OCaml because of a type error.

[3] Or, more accurately, in the Polymorphic, Cumulative Calculus of Inductive Constructions (PCUIC).

1.2 Verified Compilation

Verified compilation is an idea that dates back to the 1960s; it has been studied for its own sake or in combination with source-level formal verification frameworks in order to decrease their TCB. The first attempt to prove a compiler correct is attributed to McCarthy and his graduate student Painter [95] who provide a correctness proof for a simple compiler for arithmetic expressions. Although the compiler is proved correct with pen and paper, their ultimate goal was the mechanization of compiler correctness proofs. Half a decade later, Milner and Weyhrauch [98] built a mechanically verified compiler for an ALGOL-like language. The proof was carried out in Milner's LCF framework, the predecessor of the HOL-family proof assistants, which is based on Scott's logic for computable functions [97].

Since then, there has been remarkable progress in the area of verified compilation, both in the scale of the compilers that are being verified and in the development of mathematical techniques that are used to carry out compiler correctness proofs. CompCert and CakeML, briefly described below, are two of the most influential verified compilers. They provide verified, optimizing pipelines for practical source languages that are comparable to industrial-strength compilers.

- **CompCert** [83, 86] is a landmark in modern compiler verification and demonstrates that practical, optimizing compilers can be formally verified with machine-checked proofs. CompCert achieves comparable performance to GCC optimization level 1, with CompCert being roughly 10% slower than `gcc -O1` on a PowerPC architecture. The TCB of CompCert includes (i) the formal specification of the source and target language semantics (ii) the toolchain with which CompCert is itself compiled (*i.e.*, Coq's extraction mechanism and the OCaml compiler), and (iii) the kernel of the Coq proof assistant. Since CompCert's first development, several extensions have been made to support concurrent computations (CompCertTSO [133]), different notions of compositional compiler correctness (*e.g.*, SepCompCert [73], CompCompCert [127], CompCertX [54], CompCertM [122]), preservation of stack-space bounds [27], and preservation of cryptographic constant-time behavior [20].

  CompCert is important for demonstrating not only that verification in this scale is feasible but also worthwhile. In a study that tested various C compilers with Csmith, a randomized test-case generation tool, CompCert was the only compiler for which Csmith could not find any miscompilation errors [140].[4]

- **CakeML** [130, 129] is a verified, optimizing compiler for a large subset of the ML language. The implementation and verification of CakeML are carried out

[4] This study discovered a few bugs in the unverified portions of CompCert's front end. The bugs were fixed and, in addition, the CompCert team has extended the verified portion of CompCert's front end to include a verified parser [71]. Another bug was found that was due to misunderstanding of the PowerPC semantics. The authors of the study write: "As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors." [140, Section 3.1]

in the HOL4 theorem prover. CakeML is unique in that it is bootstrapped inside the logic of the HOL theorem prover [79]. Using the technique of proof-producing synthesis [103], the compiler definition, expressed in HOL, is translated to a CakeML program, represented as abstract syntax in HOL, producing a proof that the semantics of the CakeML program correspond to the HOL definition of the compiler. Then, the application of the compiler to its CakeML abstract-syntax representation can be evaluated within the logic to produce binary code that has provably the same operational behavior as the CakeML abstract-syntax definition of the compiler.

The Spectrum of Verified Compilation. Broadly speaking, a compiler is correct if the compiled program preserves the meaning of the source program. The source program can be assigned meaning abstractly, in a machine-independent way, by a formal semantics (*e.g.*, structural operational semantics). A compiler also assigns meaning to a source program by mapping it to a target program. The target program can be executed on a machine, giving *concrete* meaning to the source program. The meaning of the target program can also be described abstractly, with a formal semantics that provides a mathematical model of the concrete behavior of the program. The goal of compiler correctness is to prove that for any source program, the target program obtained by compilation correctly models the behavior of the source program, according to the formal semantics of the source and target languages. This is typically captured by a *behavioral refinement* specification: a compiler is correct if the observable behaviors prescribed by the formal semantics of the target program are included in the observable behaviors prescribed by the formal semantics of the source program. This guarantees that all the behaviors that can be exhibited by the target program are valid behaviors of the source program.

There are different degrees to which the semantic definition of the source language captures the behavior of the program. When a program runs on a computer, we can observe far more than the result of the computation (or the absence thereof). We can observe how much time the program needs to compute the result, how much memory it consumes, how it affects the CPU usage, the power consumption, the microarchitectural state, and so on. Different compiler correctness specifications can vary vastly in both the set of behaviors that are modeled by the semantic definitions used and the assumptions under which they guarantee preservation of meaning. Some aspects of frequently used compiler correctness notions are outlined below.

- **Whole-program correctness** is the most straight forward notion of compiler correctness. It asserts that when a program is compiled as a whole, the meaning of the program is preserved. A whole program is a program that is closed, that is, it has no references outside its own code. The initial version of the CompCert development as well as the CakeML compiler are verified with respect to whole-program compilation.

  However, real programs are hardly ever stand-alone units. In most real-world situations, program modules are compiled separately and linked together at the target level. A program might use libraries that are compiled separately with the same or different compilers (or the same compiler with different optimization levels), or call a routine written in an entirely different language through a foreign-function interface (FFI). In these situations whole-program compiler correctness does not apply.

- **Compositional compiler correctness** extends whole-program correctness by allowing the specification of the behavior of program modules that are compiled separately and linked at the target level. Compositional compiler correctness, as the name suggests, is concerned with proving semantic preservation for each of the compiled modules *separately* and composing these specifications to derive a specification for the whole target program, after linking. Compositional compiler correctness is a challenging verification problem and different solutions have been suggested in the literature that vary in strength, complexity and domain of application. There is a spectrum of compositional compiler correctness theorems, depending on the assumptions that are made about the separately compiled modules. Examples of compositionally verified compilers along this spectrum include:

  - SepCompCert [73] allows linking of modules that have been compiled separately with CompCert, but perhaps using different optimizations levels.
  - PILSNER [104], a verified compiler for an ML-like language, supports linking with modules that are expressible in the same source language. This includes programs compiled with PILSNER, programs that are written in the same language but compiled with other compilers (verified with the same methodology), and even hand-written target programs that provably refine the behavior of some source program.
  - CompCertM [122] supports multi-language linking, *i.e.*, linking of modules that are compiled from different source languages.

  Patterson and Ahmed [108] characterize the spectrum of compositional compiler correctness and provide a general statement for compositional compiler correctness.

- **Preservation of quantitative properties.** So far, we have implicitly assumed that the compiler correctness theorem asserts that the compiler preserves only the result of the computation.[5] However, programmers commonly reason about the asymptotic behavior of a program with respect to computational resources according to some cost model for the source language. Failure to preserve properties such as the asymptotic running time or space of a program are considered bugs in compilers. Such compiler bugs are not uncommon: JavaScript's V8 closure representation can prevent memory from being reclaimed by the garbage collector, introducing memory leaks [51, 46]. A bug in

[5] Depending on the language, the result of a computation can be either a value (common in pure functional languages), a machine state, or a trace of events (common in languages with imperative features and side-effects).

> GCC [78] can prevent objects in C++ from being properly destroyed in certain situations, also resulting memory leaks.
>
> In recent years, there has been an increasing interest in verifying that compilers preserve resource bounds. CompCert's proof has been extended to include verification of stack-space bounds [26] and guarantee that memory consumption is preserved through compilation [24]. In functional languages, which are commonly garbage collected, the problem of defining a space cost model and verifying that it is preserved through compilation becomes significantly harder. It has been studied for individual transformations [99, 107, 28] and also, very recently, for an end-to-end verified compiler [53].

The correctness theorem for the CertiCoq asserts that both terminating and divergent behaviors of the source program are preserved to the target program. Although well-typed Gallina programs are always terminating, the middle-end pipeline of CertiCoq aims to provide a general compilation pipeline for functional languages that allow nonterminating source programs to be expressed. The theorem of CertiCoq applies to whole-program compilation as well as separate compilation of programs through pipelines that go through the same series of intermediate languages. To obtain a compositional compiler correctness theorem, I develop a novel technique for lightweight compositional compiler correctness. The framework supports compositional compiler correctness with the restriction that the pipelines involved must go through the same series of intermediate languages.

In the second part of the thesis, I extend the proof framework so that it can be used to show that time and space bounds are preserved by transformations. I apply this extended framework to show that the closure conversion pass of CertiCoq is safe for time and space.

1.3 Compiling Functional Languages

A compiler for Gallina not only has to be correct, but it should also generate efficient code. Specifically, the performance a Gallina program compiled with CertiCoq and CompCert should not be much worse than the performance of the program when it is extracted to OCaml and compiled it with OCaml compiler. An optimizing compiler, however, is a strictly more complicated program than an nonoptimizing one. And as software becomes more complicated, so does its proof of correctness. To that end, the design of a verified compiler should facilitate both verification and production of efficient code.

Functional languages, like Gallina, support *first-class functions*, allowing functions to be treated as any other form of data in the program. In such languages, functions can be higher-order: they can take functions as arguments or return functions as a result. Moreover, functions can be assigned to variables or stored in data structures, need not be named, and can be defined at any point inside a program. First-class functions are both a blessing and a curse: they add a lot of expressiveness to the source language but they introduce implementation difficulties and have been associated with high runtime costs. When compiling functional languages to low-level imperative languages without such features, like assembly, first-class functions need to be implemented using first-order representations.[6] But the combination of higher-order, nested functions with lexical scoping complicates the runtime representation of functions that now have to become converted to *closures*. In addition, multi-argument functions in higher-order languages are *curried*, taking one argument at a time and returning a function that accepts the next argument. Compilers for functional languages put a lot of effort in optimizing the representation of functions by performing efficient closure-conversion transformations that optimize the representation of closures, and uncurrying transformations that introduce proper multi-argument functions.

Closure Conversion. When functions are nested they can refer to variables that are not their own parameters or local variables, but *free* variables that are defined in the scope of one of the enclosing functions. Since functions can flow (through parameter passing or function return) to arbitrary scopes where their environment is not available, their representation needs to contain information about the values of their free variables. Compilers commonly do that with a closure-conversion transformation[7] that compiles functions to *closures*: pairs of the function code and the function environment that contains the values of the free variables of the function. Closures are an expensive representation for both time and space: they need to be constructed, stored in the heap of the program, accessed multiple times, and garbage collected. To produce efficient code, compilers must carefully pick the functions that need to be closure converted and their closure representations. In Chapter 4, I discuss in more detail efficient closure allocation strategies.

Uncurrying. In the higher-order view of functions, functions are *curried*: a multi-argument function is just a function-returning function that takes only one argument at a time, allowing the function to be partially applied. When compiling a functional language, multi-argument functions should be introduced to avoid the performance overhead of successively applying n-ary functions and allocating a closure for each intermediate function return.

This thesis is concerned with building and verifying an optimizing middle-end pipeline for CertiCoq that is aimed at efficiently compiling an untyped, higher-order functional language (with first-class, curried functions) to subset of the language that is closure-less and flat-scoped. In this subset of the language, functions that are passed as parameters and returned as results can be implemented solely using function pointers and do not require any additional runtime information. In this

[6] Technically, some limited form of higher-order functions can be implemented in low-level languages like C and assembly using function pointers. I consider this a first-order representation since the object that is passed around is effectively an address.

[7] Alternatively, some compilers (*e.g.*, MLton [139]) perform a defunctionalization pass [41].

thesis, I refer to this subset of the language as first-order since it can be mapped directly to a first-order language.

The optimizing pipeline uses an A-normal form (ANF)[8] intermediate language (henceforth $\lambda_{\texttt{ANF}}$). Through a series of modular $\lambda_{\texttt{ANF}}$-to-$\lambda_{\texttt{ANF}}$ transformations, the pipeline compiles away first-class functions, introduces multi-argument functions, implements efficient closure allocation strategies, and performs additional code simplification and optimization. The pipeline is built around the following design principles.

Principle 1: Separation of Concerns. The CompCert compiler for C provides verified and reasonably efficient code generation for a first-order imperative language. By targeting CompCert, we can focus on the task of efficiently compiling a higher-order functional language to a first-order representation.

Principle 2: Compilation by Program Transformation. Following a *compilation-by-program-transformation* approach [75] we employ a number of transformations to compile a higher-order language to a closure-less, flat scope subset of *the same* language. The output of the $\lambda_{\texttt{ANF}}$ pipeline can be the translated to different back ends. So far, CertiCoq only has a C back end but targeting other back ends, like LLVM [81] or WebAssembly [58],[9] would also be possible.

Principle 3: Modularity. Optimizations in the $\lambda_{\texttt{ANF}}$ pipeline are the result of the interaction of small, cooperating program transformations rather than monolithic all-at-once transformations. Apart from facilitating maintenance and extensibility, this design is particularly suitable to formal verification.

1.4 Summary of Contributions

In this thesis, I present the implementation and verification of a multi-pass optimizing pipeline for the CertiCoq compiler. The optimizing pipeline operates on the $\lambda_{\texttt{ANF}}$ intermediate representation and consists of seven distinct $\lambda_{\texttt{ANF}}$ program transformations. The $\lambda_{\texttt{ANF}}$ pipeline advances the state of the art in optimizing verified compilers, by implementing more efficient strategies for closure allocation.

The verified pipeline aims to provide a practical, verified optimizing pipeline for compiling functional languages. The $\lambda_{\texttt{ANF}}$ intermediate representation is a general intermediate representation that can be used to compile meaningful subsets of general-purpose functional languages like ML and OCaml. With modest extensions (described in Chapter 3) it can scale to more diverse language features like references and arrays. The output of the $\lambda_{\texttt{ANF}}$ middle-end pipeline can be compiled to different backends. The modular and extensible design of the $\lambda_{\texttt{ANF}}$ pipeline and the $\lambda_{\texttt{ANF}}$ verification framework allows new optimizations to be easily added and verified.

[8] A more in depth discussion about functional intermediate representations is given in Chapter 3.

[9] WebAssembly currently lucks support for memory management, although there are proposals for adding such feature [45, Chapter 4]. Currently, to target WebAssembly from CertiCoq, we would have to implement a garbage collector in some other language (*e.g.*, C) and compile it to Wasm.

To verify the pipeline I develop a framework based on logical relations that involves the following novel technical contributions.

- The logical relations used are extended with relational pre- and postconditions that can be used to specify instensional aspects of the source and target computations in addition to their extensional behavior. I demonstrate how this enables us to reason about the resource consumption of compiled programs simultaneously with functional correctness.

- The framework supports reasoning about preservation of divergence. In their standard formulation, logical relations do not support reasoning about preservation of nonterminating behaviors. I show how the relational postcondition enables us to extend the reasoning to nonterminating source programs.

- The framework supports lightweight verification of separate compilation with a novel technique for composing logical relations. This technique can be used to reason about linking programs compiled through the same series of intermediate languages (but not necessarily the same transformations). It does not require any modification to the logical relations, the theorem statements, and the proofs of each transformation.

The logical-relation framework has been used by myself and others to verify the $\lambda_{\texttt{ANF}}$ transformations of the CertiCoq compiler. The correctness theorem of the $\lambda_{\texttt{ANF}}$ pipeline supports whole-program compilation, as well as separate compilation of programs compiled with different sets of optimizations.

In the second part of the thesis (Chapter 7), I use the logical-relation framework to show that the closure conversion transformation of CertiCoq is not only functionally correct but also *safe for space*. This is the first space-safety proof of a closure-conversion transformation.

The results presented in this thesis are all mechanized in the Coq proof assistant. The rest of the thesis is structured as follows:

- In Chapter 2, I give a broad overview of the CertiCoq compiler, including the transformations before the $\lambda_{\texttt{ANF}}$ pipeline and the code generation phase.

- In Chapter 3, I present the $\lambda_{\texttt{ANF}}$ intermediate representation and its semantics.

- In Chapter 4, I describe the transformations of $\lambda_{\texttt{ANF}}$ pipeline and the optimizations they achieve.

- In Chapter 5, I describe the logical-relation framework and the compositional compiler correctness framework, which is built on top of the logical relation framework.

- In Chapter 6, I demonstrate the correctness results for the $\lambda_{\texttt{ANF}}$ pipeline.

- In Chapter 7, I extend the semantics of the intermediate representation to make the memory model of the language explicit. I adapt the logical relation to the new semantics and extend it to facilitate reasoning about the resource consumption of programs. I apply the framework to show that closure conversion is safe for space. This chapter was previously published as *Closure Conversion is Safe for Space* by Paraskevopoulou and Appel [107].

- In Chapter 8, I provide an experimental evaluation of the CertiCoq compiler. I compare the performance of Gallina programs compiled with CertiCoq to the performance the same programs extracted to OCaml and compiled with the OCaml compiler. I also evaluate the performance improvement of CertiCoq's optimizing closure strategies.

- In Chapter 9, I conclude by summarizing the presented work and discussing open questions and possible future directions.

The verification of CertiCoq is a product of collaborative research. In Section 2.1.6, I give an detailed account of individual contributions to the development of CertiCoq.

Chapter 2

CertiCoq Overview

In this chapter I give an account of the overall architecture of the CertiCoq compiler. Although the rest of the thesis can be read and understood independently of CertiCoq, this chapter aims to provide the context in which the presented work fits in. The goal of the CertiCoq project is to build a verified and efficient compilation pipeline from Coq all the way to assembly language, decreasing the trusted computing base of formal verification in the Coq proof assistant. CertiCoq compiles a pure functional program to Clight [25], a front-end intermediate representation of the CompCert compiler that is a subset of the C language. The Clight program can be then compiled to different architectures using the CompCert verified compiler.

There are three reasons for targeting C and compiling with CompCert instead of targeting machine language directly. First, we can take advantage of a mature verified compiler and its already proved-correct optimizations. Second, we target all of the architectures that CompCert targets (PowerPC, ARM, RISC-V, x86 and x86-64), while we only deal with platform-independent aspects of compilation. Third, by emitting C we can link Coq programs with C programs that allow them to interact in meaningful ways with their environment. Gallina, Coq's specification language, is a pure functional language: it can perform computations but it cannot interact in any way with its environment. For the Coq program to get inputs and return its output to the environment linking with some external drivers is required. In Coq's extraction pipeline this is achieved by linking the extracted Coq code with programs written in the extraction language. In CertiCoq, this is done by linking with C programs and using a foreign function interface (FFI) that provide facilities to construct and destruct values in the format expected by the compiled program. In addition, this architecture allows verified Coq programs to be part of larger unverified code bases, providing a way to formally verify only critical components of a system.

This thesis is about only a part of CertiCoq: the part that compiles efficiently a pure, untyped functional program to a first-order representation, which can be directly compiled to a low-level imperative language. This is achieved with a multi-pass optimizing pipeline (the $\lambda_{\texttt{ANF}}$ middle-end pipeline) that consists of a number of source-to-source transformations. Before the $\lambda_{\texttt{ANF}}$ pipeline, CertiCoq employs a few passes that perform simplification steps and an A-normal form (ANF) or continuation-passing style (CPS) transformation to convert the program to the $\lambda_{\texttt{ANF}}$ intermediate representation. The output of the $\lambda_{\texttt{ANF}}$ pipeline is then directly translated to C. The compiled program is linked with a garbage collector, which is itself written and verified in C [138], using the Verified Software Toolchain (VST) [11]. In the next section I go over the different parts of CertiCoq in greater detail, and I explain how the $\lambda_{\texttt{ANF}}$ pipeline fits in the CertiCoq pipeline.

2.1 The CertiCoq Pipeline and Runtime

2.1.1 The Pipeline

The CertiCoq pipeline is shown in Figure 2.1. The boxes represent the transformations that are applied to the code. The text above the arrows indicates the intermediate representation before and after a transformation is applied. All boxes represent one-pass transformations except for the $\lambda_{\texttt{ANF}}$ pipeline that is a multi-pass optimization pipeline (represented by a double border).

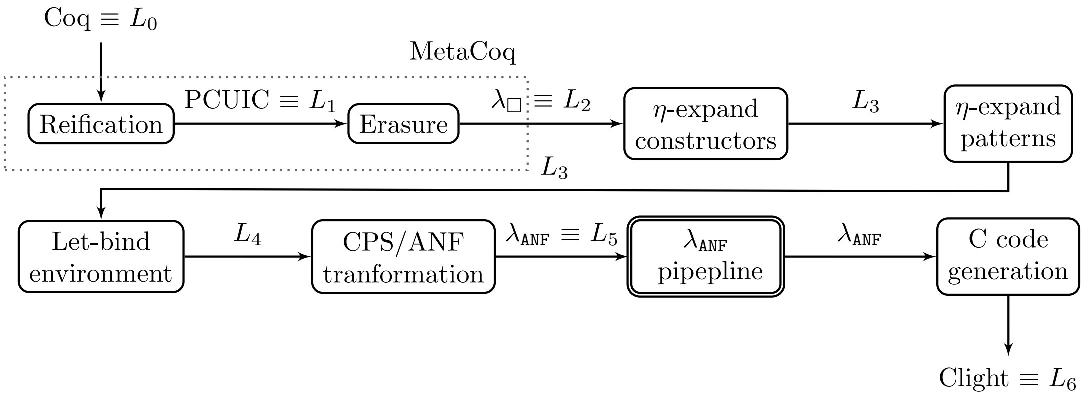


Figure 2.1: The CertiCoq pipeline.

Reification. Starting from a Coq program, we use Template-Coq [9], a metaprogramming library for Coq that is part of the MetaCoq project [123, 124], to obtain its deeply embedded representation inside Coq. The abstract syntax of reified Coq programs represents terms in the Polymorphic, Cumulative Calculus of Inductive Constructors (PCUIC), the core calculus of Coq. The language of PCUIC is a pure, dependently typed lambda calculus, extended with projections, case analysis, let-bindings, and (co)recursion operators (fix and cofix). Its static (typing judgment) and dynamic semantics are formalized as part of the MetaCoq project.

Erasure. The next step erases non-computational content from a PCUIC term (*i.e.*, proofs and types) to produce a $\lambda_{\square}$ term (pronounced lambda-box). We use a verified erasure procedure that is part of the MetaCoq project. After erasure, proofs and types are replaced by a new language constructor, namely □. After erasure, a □ can only be eliminated in function-application position. A □ can be applied to any argument and, according to its semantics, the application will evaluate to a □. The □ constructor is propagated through the CertiCoq pipeline and eventually gets replaced with a function that consumes every argument that is being passed (that is, $\texttt{fix}\ f\ x = f$) during CPS/ANF conversion.

Eta expansion of constructors. The next transformation in the pipeline takes care of converting partially applied constructors to fully applied constructors and removing the parameters of inductive types from constructors. Making constructors fully applied is a necessary compilation step since in the low-level representation of constructed values, there is no notion of partial constructor application. Furthermore, this steps allows to remove parameters of an inductive type, which are computationally irrelevant, from the constructors of the inductive type.

Full application is achieved by eta-expanding constructor applications: constructors are wrapped in lambdas that fully apply the constructor to the number of arguments that it expects. For instance, consider a partial application of the `cons` constructor of the `list` data type: `cons nat 1`. The constructor is applied first to the type parameter of the inductive type and then the element 1. It expects one more argument, the tail of the list, which is not provided. After the transformation, it will be converted to the partial function application: $(\lambda\texttt{hd}.\ \lambda\texttt{tl}.\ \texttt{cons hd tl})\ \texttt{1}$. The constructor is now fully applied and the type parameter has been removed. The redundant abstractions and applications that are generated by the transformation when an argument of the constructor is already applied will be eliminated during the $\lambda_{\texttt{ANF}}$ pipeline. For instance, the above example will be statically evaluated to $\lambda\ \texttt{tl}.\ \texttt{cons 1 tl}$.

Eta expansion of patterns. In L_2, the patterns in the branches of a match do not explicitly bind the arguments of the constructors: the expressions in the branches of the match are expected to be functions. For instance, in a pattern $\texttt{cons} \Rightarrow \texttt{e}$ of a match that discriminates a list of type `list a`, `e` is expected to have type $\texttt{a} \to \texttt{list a} \to \texttt{b}$, where `b` is the return type of the match. This transformation will make sure that `e` is in the form $\lambda\texttt{hd}.\ \lambda\texttt{tl}.\ \texttt{e}'$. This allows the following transformation to easily strip the lambdas and add the bindings to the pattern. This is achieved by converting `e` to $\lambda\texttt{hd}.\ \lambda\texttt{tl}.\ \texttt{e hd tl}$. Again, redundant lambdas (introduced when `e` is already a lambda abstraction) will be eliminated by the $\lambda_{\texttt{ANF}}$ pipeline.

Let binding of environment. The current working version of CertiCoq will produce whole programs by prepending all external definitions to the beginning of the compiled program. More precisely, when a Coq program is reified with Template-Coq a pair is produced: a term together with its environment (*i.e.*, a list of all definitions that are referenced by the reified program). Earlier transformations have to be applied to both the term and the environment. This transformation will explicitly let-bind the environment at the beginning of the term, so that a whole program is produced. In addition, it strips the lambdas from the expressions in the branches of a match, and introduces patterns that explicitly bind the arguments of the constructor.

ANF/CPS transformation. After erasure, eta expansion of constructors and patterns, and let-binding of environment, the code is translated via ANF or CPS transformations to the $\lambda_{\texttt{ANF}}$ intermediate representation. This transformation is also responsible for moving from De Bruijn indices (used by all previous intermediate representations) to named binders. In addition, it removes (again) the explicit bindings from the patterns of a pattern match moving to patterns that do not bind constructor arguments. In $\lambda_{\texttt{ANF}}$ the arguments of a constructor must be explicitly projected. The transformation introduce explicit projections at the beginning of the expression of each branch that project the arguments of a constructor and bind them to variables.

$\lambda_{\texttt{ANF}}$ **pipeline.** The $\lambda_{\texttt{ANF}}$ pipeline, which is the subject of this thesis, is responsible for efficiently compiling a pure lambda calculus with (mutually) recursive functions, constructors, projections, and pattern matching to a first-order representation. The $\lambda_{\texttt{ANF}}$ optimizing pipeline is described in detail in Chapter 4. The transformations performed include closure conversion, lambda lifting, inlining, uncurrying, shrinking (a transformation that performs shrink reductions [15, 23]), and dead parameter elimination.

The input of the $\lambda_{\texttt{ANF}}$ pipeline is a purely functional program with nested function definitions, closures (*i.e.*, functions with free variables), and unary functions and applications. The output of the $\lambda_{\texttt{ANF}}$ pipeline is a program with multi-argument functions that are closed and defined at the top-level of the program that has only two levels of scope: the global one containing all names of the top-level function, and the local scope of each top-level function. In this program higher-order functions can be implemented simply with function pointers and closures are constructed explicitly in the code.

The $\lambda_{\texttt{ANF}}$ pipeline aims to be more general than the previous CertiCoq compilation pipeline and can be seen independently from both the front-end and the code-generation (back-end) parts of the compiler. One could easily write and (perhaps less easily) verify code generators for other low-level representations such as LLVM [81] or WebAssembly [58].[1], would also be possible. The same goes for the front end of CertiCoq that has been subject to many changes since the start of the CertiCoq project, and will perhaps change more as CertiCoq is being actively developed. For example, although currently CertiCoq compiles whole programs, the $\lambda_{\texttt{ANF}}$ pipeline is built and verified with separate compilation in mind and does not assume that the input will be a whole program.

C-code generation. The last phase of the CertiCoq compiler is the C-code generation that translates the output of the $\lambda_{\texttt{ANF}}$ pipeline to Clight. Coq values of inductive types are represented in the heap of the C program following similar to the OCaml object format (Section 2.1.2) and are stored in an isolated memory chunk, the CertiCoq heap. Functions are represented as C functions and are being passed around as

[1] As mentioned in the introduction, WebAssembly currently lucks support for memory management [45, Chapter 4]. To target WebAssembly from CertiCoq, we would have to implement a garbage collector in some other language (*e.g.*, C) and compile it to Wasm.

function pointers. The code generation phase (for the CPS subset of $\lambda_{\texttt{ANF}}$) is described in detail in Savary Bélanger's thesis [21]. The code generator is also responsible for generating calls to a garbage collector in appropriate sites. The generated program will be then linked with the implementation of the garbage collector (Section 2.1.3) and compiled with a C compiler.

2.1.2 Representation of Coq types in C

A Coq value of an inductive type is represented in C as an unsigned long integer[2], that represents either a pointer to an address in the CertiCoq heap or an integer. Values are represented in memory using the OCaml memory model [61, Chapter 21]. A nullary constructor has an unboxed representation con siting of an integer that uniquely identifies each unboxed constructor in an inductive type. Constructors of non-zero arity are boxed and are represented as as pointers to the second word of a *block* in the CertiCoq heap. A block consists of a word-sized header, followed by n contiguous values that represent the constructor arguments, where n is the arity of the constructor (after removing parameters of the inductive type). The header of the block stores information about the size of the block, its garbage collection state, and the tag of the constructor, that uniquely identifies a boxed constructor of an inductive type. The representation of a boxed value is shown schematically in Figure 2.2.

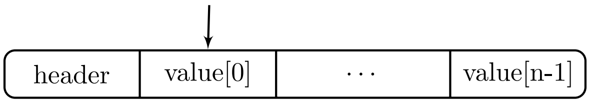


Figure 2.2: A block in the CertiCoq heap.

Boxed and unboxed values are distinguished using the last bit of their representation that is always 1 for integers and 0 for pointers.

2.1.3 Garbage Collection

CertiCoq uses a generational collector to deallocate dead objects from the CertiCoq heap. In essence, the CertiCoq heap is an array of generations, each of them being a linear memory region. The generated C code performs a garbage collection test to determine if there is enough memory for all the allocations of the program until the next garbage collection test, and invokes the garbage collector if there is not. Garbage collection tests are placed upon function entry and after the return of (non-tail) function calls in the body of a function.

As usual, the garbage collector determines the live portion of the heap by following the chain of memory references from the set of live roots, which is the set of live

[2] CertiCoq can generate C in any of several C compiler configurations, in which pointers may be 32-bit or 64-bit, and the corresponding unsigned integer type may be unsigned long or unsigned long long. In this discussion, we will use a "typical" C configuration in which pointers are 64 bits, and represented as `unsigned long long` integers (also 64 bits).

variables (parameters and local variables) at a particular point in the execution of the program. In traditional implementations of garbage-collected languages, the set of live variables is found in the call stack of the program where the arguments of a function call and local variables are stored. However, we do not have access to the runtime environment of the C program, and therefore, we don't have any way to parse the C call stack. To circumvent this issue, we implement a *shadow stack* [96, 60], which is a linked list threaded through the call stack of the C program. Each element of this list is a *shadow stack frame* which is defined as local variable in the stack frame of each active function. The shadow stack scheme works in the following way. Each activation of a function will allocate in its stack a new shadow stack frame and will link it to the shadow stack frame of the caller, which is passed to the function as a parameter. Before each intermediate function call and garbage-collector call, we push the variables that are live (*i.e.*, variables that occur free in the rest of the program) in the current shadow stack frame and pop them upon function return. At the end of the function body, before the function returns or performs a tail call, the current stack frame is discarded. When the garbage collector is called, it traverses the linked shadow stack frames, that reflect the current call stack of the execution, in order to find the set of live roots.

2.1.4 Foreign Function Interface

CertiCoq provides a foreign function interface that allows the user of the code that has been compiled to C from Coq to construct and destruct the C representation of a Coq datatype. Using the foreign function interface the user can construct inputs for the compiled Coq program and inspect the output by, *e.g.*, writing iterators that print out the result.

As an example, consider the `list` datatype in Coq:

```
Inductive list (A : Type) : Type:=
| nil : list A
| cons : A -> list A -> list A
```

CertiCoq will automatically generate the following C functions to manipulate the C representation of the Coq `list` type.

```
// Constructors
unsigned long long make_list_nil(void);
unsigned long long make_list_cons(unsigned long long,
                                  unsigned long long,
                                  unsigned long long *);
// Eliminators
unsigned int get_list_tag(unsigned long long);
struct nil_args *get_nil_args(unsigned long long);
struct cons_args *get_cons_args(unsigned long long);

// Struct representation of types
struct nil_args { };
struct cons_args {
  unsigned long long cons_arg_0;
  unsigned long long cons_arg_1;
};
```

The function `make_list_nil` returns the (unboxed) representation of the `nil` constructor. The function `make_list_cons` takes three arguments: the head and tail arguments of `cons` and a pointer to the address where the value should be stored. It returns the pointer to the constructed block, which by convention is the address where the first argument of the constructor is (*i.e.*, the input address offsetted by a word).

The function `get_list_tag` inspects a value and returns the tag of its outermost constructor, which is the order in which it appears in the inductive type (in this example, 0 for `nil` or 1 for `cons`. The functions `cons_args` and `nil_args` will return the arguments of constructor put in a struct (also automatically generated) that has as many fields as the constructor arguments.

2.1.5 Verification

Although this thesis presents an end-to-end verified, optimizing pipeline that is used in the CertiCoq compiler, the end-to-end verification of the CertiCoq compiler is work in progress. At the time when this thesis is written, the following are still ongoing research.

- Completion of some front-end proofs and the code-generation proof.
- An end-to-end theorem for CertiCoq, composing the correctness theorem of the $\lambda_{\texttt{ANF}}$ pipeline with those of individual transformations.
- An end-to-end theorem for CertiCoq + CompCert, composing the correctness theorems of CertiCoq and CompCert.
- The connection of the verified garbage collection specification with the specification of the correctness of the C-code generator.
- Extension of CertiCoq to handle separate compilation.

2.1.6 Collaboration

CertiCoq is the product of collaboration of many researchers across different institutions. In the current working version of CertiCoq the following contributions were made.

- Reification and erasure are part of the MetaCoq project and are implemented and verified by the MetaCoq team [123, 124].
- The Eta expansion of constructors and patterns phases are implemented and verified by Randy Pollack.
- The L_3 to L_4 transformation is implemented and verified by Matthieu Sozeau.

- The CPS transformation is implemented and currently being verified by Anvay Grover. The ANF translation was implemented by me and it is not currently verified.

- The proof framework that is used by myself and others to prove correct the $\lambda_{\texttt{ANF}}$ transformations, was designed and mechanized in Coq by me.

- The overall design of the $\lambda_{\texttt{ANF}}$ pipeline is my own research. This includes $\lambda_{\texttt{ANF}}$ and its semantics, presented in this thesis. $\lambda_{\texttt{ANF}}$ is based on a CPS intermediate representation initially designed by Andrew Appel and Greg Morrisett. The implementation and verification of the $\lambda_{\texttt{ANF}}$ transformations was divided as follows:

  - Lambda lifting, closure conversion, hoisting, and inlining was implemented and verified by me. The implementation of inlining was based on an earlier implementation by Savary Bélanger. John Li proved that the closure conversion program is correct with respect to the inductive definition of closure conversion.
  - The initial implementation of uncurrying is due to Greg Morrisett and was later adapted by me and John Li. John Li did the correctness proof of uncurrying.
  - Dead parameter elimination was initially implemented by Katja Vassilev. It was later adapted and currently being proved correct by me.
  - Olivier Savary Bélanger implemented and verified the shrinking transformation as part of his PhD thesis [22, 21]. The early implementation and proof of shrinking only handled the CPS subset of the current $\lambda_{\texttt{ANF}}$ IR. They were later extended to the full fragment of $\lambda_{\texttt{ANF}}$ by me.

- The code generation phase was implemented and verified by Savary Bélanger as part of his PhD thesis [21], and extended by Matthew Weaver. Again, the implementation (but not the verification) was later extended by me to apply to the full $\lambda_{\texttt{ANF}}$ language. These extensions include the implementation of the shadow stack and the relevant extensions to the garbage collector.

- The implementation of the foreign function interface was done by Joomy Korkut.

2.2 Running CertiCoq

Currently, there are two ways of running the CertiCoq compiler on a Coq program: by evaluating the application of the compiler within Coq or by extracting the compiler to OCaml and compiling it with the OCaml compiler.

Evaluation in Coq. CertiCoq can be invoked on a Coq program within the Coq interactive environment and evaluated using the term reduction mechanism that is part of the Coq kernel. To do so, one can use the Template-Coq plugin to quote a Gallina program and evaluate the application of the compiler on the quoted program. The output, a Clight AST, can be then printed to a file (with the use of a Coq plugin) and compiled with a C compiler. However, evaluating the application of CertiCoq within Coq can be very inefficient even for small programs and it will raise a stack overflow for larger programs.

The CertiCoq plugin. Alternatively, the CertiCoq compiler can be extracted to OCaml, compiled with the OCaml compiler, and invoked within Coq as a Coq plugin. The extracted CertiCoq compiler can be invoked using the command

```
CertiCoq Compile program.
```

where `program` is the identifier of the program intended for compilation. The command will generate a C file with the compiled code and a C file that contains the corresponding glue code. By default, the compiler will use the CPS translation. To compile a program with the direct ANF translation the flag `-direct` can be used.

Bootstrapping Ideally, the CertiCoq compiler will be used to compile itself, removing the dependency on the unverified extraction plugin and the OCaml compiler. Currently, there are technical issues that prevent us from quoting and using the erasure function on the erasure function itself. Therefore, at the moment, it is not possible to compile CertiCoq with CertiCoq. If these issues are overcome, we could possibly have a bootstrapped compiler.

To bootstrap CertiCoq, we could evaluate the application of the CertiCoq compiler on the quoted CertiCoq compiler using the term reduction mechanism that is part of the Coq kernel. Evaluating such term inside Coq will be costly for time and memory, but for a sufficiently large stack limit it could, in principle, terminate. That would allow us to obtain a compiled version of the compiler that does not depend on the unverified extraction.

As outlined in the introduction, it is not be possible to completely remove the OCaml compiler from the TCB of formal verification in Coq. However, bootstrapping CertiCoq would allow us to completely remove the unverified extraction program from the TCB of formal verification in Coq.

Chapter 3

Intermediate Representation

In this chapter, I describe the $\lambda_{\texttt{ANF}}$ intermediate representation that is used throughout the CertiCoq middle end, after the CPS/ANF conversion until the C-code generation phase. The next chapter, that describes the $\lambda_{\texttt{ANF}}$ transformations in detail, makes use of the formal syntax of the language. In Chapter 5, I will use the semantics presented here to define the relational framework that is used to verify the pipeline transformations.

3.1 Functional Intermediate Representations: CPS *vs*. ANF

The choice of the intermediate representations (IRs) in a compiler pipeline is key to the design of a good compiler. Compilers usually employ a number of intermediate representations, gradually moving from higher-level to lower-level IRs. For instance, the IR that is used for type-checking, which is commonly one of the very first steps in the front end of a compiler for a typed language, is a high level representation that preserves almost all of the constructs of the source language. As high level abstractions are compiled away, the intermediate representations become more and more low level, until there is a direct correspondence between the IR and the instruction set architecture (ISA) of the target machine. Two common low-level functional intermediate representations, suited for the purposes of the $\lambda_{\texttt{ANF}}$ pipeline of CertiCoq, are the CPS (*continuation-passing style*) and the ANF (*administrative normal form*) languages. In fact, the question of whether CPS or a direct-style intermediate representation, like ANF, is more adequate for compilation is a long-standing debate in the programming languages research community [48, 18, 76, 92, 47, 36].

Continuation-passing style is a way of writing programs that makes control flow and data flow explicit: all intermediate computations have to be passed as arguments to their *continuation* that specifies what is the next computation to be performed. As a result, all intermediate results are named, and functions never return: they simply pass the control to their continuation that is passed as an extra argument. CPS as a compiler intermediate representation was first used in RABBIT [125], the first compiler for Scheme and one of the earliest implementations of a higher-order, lexically scoped functional language. Other examples of compilers that use a CPS intermediate representation is the ORBIT [2] compiler for Scheme, early implementations of the SML/NJ compiler [10] compiler, and the SML.NET compiler [76]. From a compiler implementation perspective the use of CPS has many advantages. CPS encodes the evaluation strategy in the structure of the term, and hence, as Plotkin famously proven by Plotkin's indifference theorem [112], the result of evaluation of CPS terms is independent of the evaluation strategy used. As a result, full beta reduction is sound in CPS, which gives strictly more opportunities to perform inlining optimizations compared to direct-style representations where inlining has to respect evaluation strategy of the source. In addition, a lot of useful optimizations, like tail-call elimination, amount to function inlining in a CPS intermediate representation [10], making CPS a very appealing IR to perform optimizations. In a CPS-converted program no function ever returns, it just passes the control to its continuation. Therefore, CPS lends itself to a simple implementation by a stackless abstract machine. Put differently, CPS has a very simple implementation using heap-allocated activation records, that eliminates the need for a runtime stack, making both the implementation of the compiler and the runtime simpler.

Despite its virtues, CPS is not a very popular choice among modern compilers. Some possible reasons are that direct-style programs can be read and understood more easily, and some optimizations, like common subexpression elimination, are much harder to perform on CPS. Another reason is that in the straightforward implementation of CPS there is no call stack and therefore all continuation closures must be heap allocated. Heap allocated activation records are traditionally seen as more expensive than stack allocated ones due to worse locallity of reference and its increased need for garbage collection.[1]

ANF is an alternative direct-style intermediate representation, proposed by Flanagan *et al*. [48], who suggested that ANF can be just as good as CPS for a compiler intermediate representation and also avoid some of the nuances related to CPS. Flanagan *et al*. observe that performing CPS conversion, beta normalization, and the inverse CPS transformation, results in a program in A-normal form, that can be also obtained by performing a set of A-normalization steps directly at the source term. ANF is similar to CPS in that all intermediate expressions are named, making some aspects of the control and data flow explicit, but without making explicit the notion of continuations. Although its creators argue that ANF captures *the essence of continuations*, and optimizations in ANF can be expressed just as naturally as in CPS, this later proved to be inaccurate [76, 115]. In particular, the ANF representation is not closed under beta reduction, which complicates the application of function inlining. The problem is that after performing a beta reduction, the resulting term is not necessarily in ANF form, and bringing it back to ANF requires performing some

[1] A recent study [47] suggests that the performance of stack *vs*. heap allocated activation records depends on the language features. For sequential languages without advanced control features the traditional C-like stack implementation seems to be the most efficient choice. Though it is possible for heap allocated activation records in combination with generational garbage collection to have performance comparable to that of stack allocated records [18], it involves complicated implementation techniques [120, 17].

extra normalization steps. The situation gets more complicated when the language involves conditionals or case analysis constructs where renormalizing the term introduces a *join point*, which is a local continuation that captures the continuation of the conditional and avoids code duplication. I will come back to the inlining problem of ANF when I discuss the inlining transformation of $\lambda_{\texttt{ANF}}$ in Chapter 4. Moreover, unlike CPS, in ANF function calls cannot be implemented simply as immediate jumps and the implementation requires a runtime call stack. Despite these shortcomings, ANF has gained a lot of popularity in compilers. Notably the Glasgow Haskell Compiler (GHC) uses ANF extended with a special syntactic construct that represents join points as second class continuations allowing for more optimizations and more efficient compilation [92].

For the CertiCoq middle-end pipeline we choose an ANF intermediate representation. However, our ANF representation is syntactically a superset of CPS and therefore we can obtain $\lambda_{\texttt{ANF}}$ program with either direct-style or CPS conversion to $\lambda_{\texttt{ANF}}$. The transformations of the $\lambda_{\texttt{ANF}}$ pipeline are designed to apply to both direct and CPS programs, so that the $\lambda_{\texttt{ANF}}$ pipeline is independent of whether direct or CPS transformation was used earlier in the pipeline. Of course, choosing to compile with CPS or direct-style ANF transformation suffers from the respective drawbacks of each representation. If the compiled program is in direct style, the application of function inlining is more restricted. On the other hand, compiling programs by converting them to CPS form results in a large number of heap allocated closures and produces less efficient code. The same code generation and the runtime implementations can handle both ANF and CPS converted programs. Extending the code generation phase and the runtime to handle ANF programs in addition to CPS was a non-trivial task that required implementing a shadow stack. In Chapter 8, I compare the performance of the code generated with direct-style ANF transformation with the performance of code generated with CPS transformation, contributing with one more datapoint to the CPS *vs*. ANF debate.

The rest of this chapter describes the syntax of the $\lambda_{\texttt{ANF}}$ intermediate representation and useful syntactic definitions (Section 3.2), and the semantics of $\lambda_{\texttt{ANF}}$ and some of its useful properties (Section 3.3). The definition of the semantics is particularly interesting from a technical prospective: it is a big step semantics tailored to allow us to prove divergence preservation and it is parameterized by abstract notions of resource and traces.

3.2 Syntax

Figure 3.1 shows the syntax of $\lambda_{\texttt{ANF}}$. We fix two countably infinite sets `Var` and `Constr` from which we draw names for variables and constructors. In the formal Coq development we use the set of positive numbers for both sets. In $\lambda_{\texttt{ANF}}$ all intermediate computations must be explicitly bound to a variable and all operations of the IR can only refer to variables and cannot contain nested operations. Expressions in $\lambda_{\texttt{ANF}}$ belong to one of the following syntactic categories.

$$
\begin{array}{llllll}
(\textit{Variables}) & x, y & \in & \texttt{Var} & & \\
(\textit{Constructors}) & \texttt{C} & \in & \texttt{Constr} & & \\
(\textit{Expressions}) & e & \in & \texttt{Exp} & ::= & \texttt{let}\ x = \texttt{C}(\overline{y})\ \texttt{in}\ e \quad \text{Constructor} \\
 & & & & \mid & \texttt{let}\ x = y.i\ \texttt{in}\ e \quad \text{Projection} \\
 & & & & \mid & \texttt{case}\ y\ \texttt{of}\ [\texttt{C}_i\ \to\ e_i]_{i \in I} \quad \text{Case} \\
 & & & & \mid & \texttt{fun}\ f\ \overline{x} = e_1\ \texttt{in}\ e_2 \quad \text{Function def.} \\
 & & & & \mid & \texttt{let}\ x = f\ \overline{y}\ \texttt{in}\ e \quad \text{Function call} \\
 & & & & \mid & f\ \overline{x} \quad \text{Tail call} \\
 & & & & \mid & \texttt{ret}(x) \quad \text{Return} \\
(\textit{Values}) & v & \in & \texttt{Val} & ::= & \texttt{C}(\overline{v}) \mid \texttt{Clo}(\sigma, \texttt{fun}\ f\ \overline{x} = e) \\
(\textit{Environments}) & \sigma & \in & \texttt{Env} & = & \texttt{Var} \rightharpoonup \texttt{Val} \\
(\textit{Contexts}) & \mathcal{E} & \in & \texttt{Ctx} & ::= & [\cdot] \\
 & & & & \mid & \texttt{let}\ x = \texttt{C}(\overline{y})\ \texttt{in}\ \mathcal{E} \\
 & & & & \mid & \texttt{let}\ x = y.i\ \texttt{in}\ \mathcal{E} \\
 & & & & \mid & \texttt{fun}\ f\ \overline{x} = e\ \texttt{in}\ \mathcal{E} \\
 & & & & \mid & \texttt{let}\ x = f\ \overline{y}\ \texttt{in}\ \mathcal{E} \\
 & & & & \mid & \texttt{fun}\ f\ \overline{x} = \mathcal{E}\ \texttt{in}\ e \\
 & & & & \mid & \texttt{case}\ x\ \texttt{of}\ [\overline{\texttt{C}\ \to\ e}, \texttt{C}_j\ \to\ \mathcal{E}, \overline{\texttt{C}\ \to\ e'}]
\end{array}
$$

Figure 3.1: The syntax of $\lambda_{\texttt{ANF}}$.

Constructors. This construct applies a constructor to a list of arguments (that are syntactically restricted to variables) and binds the resulting value to the variable x.

Projections. Similarly, this construct projects the i-th argument of a (constructed) value and binds it to x.

Case analysis. Case analysis discriminates the value of a variable against a finite set of constructors. Note that case analysis will only discriminate the outermost constructor of the scrutinee but it will not bind the arguments of the constructor to variables. The code of each branch is responsible for projecting the arguments of the constructor with explicit projections. In typed source languages, it is common to combine these into a `match` construct that is more easily typechecked; but our untyped $\lambda_{\texttt{ANF}}$ has lower level combinational primitives to facilitate optimization and code generation.

Function Definition. This construct defines a – possibly recursive – function. In the mechanized Coq development, $\lambda_{\texttt{ANF}}$ supports mutually recursive functions as well. To keep presentation simple, here I restrict the language to only include single recursive functions.

Function Application. We distinguish two cases for function application: tail and non-tail function calls. Tail calls are leaves in the abstract syntax tree and execution will never return to the caller after the tail call. In non-tail function calls, the control is returned to the caller after the function is finished executing and the result of function application is bound to a variable This is exactly the difference between programs in CPS and ANF forms: when we translate to the CPS subset of $\lambda_{\texttt{ANF}}$, we never generate non-tail function applications — and our transformations and optimizations never introduce nontail calls into a program with only tail calls.

Return. This construct terminates the execution of an expression by returning the value of a variable.

It is also useful to define the following types.

Environments and values. To define the semantics of the language, we need notions of values (*i.e.*, the result of evaluating a computation) and environments (*i.e.*, an evaluation state that keeps track of the intermediate results). A value can be either a constructor applied to a (possibly empty) list of values, or a *closure*, a pair of a function together with an environment. The environment is a partial map from variables to values.

Contexts. We define a notion of 1-hole contexts that we use to formulate various definitions. Context application, written $\mathcal{E}[e]$, forms an expression by filling e in the hole position of $\mathcal{E}$. It is useful to restrict the notion of contexts to *binding* contexts: contexts that are strictly linear (no case analysis) with the hole at tail position. A binding context (written in darker font in Figure 3.1), is a context without case-analysis nodes or function-definition nodes with the hole at the function body. A binding context $\mathcal{E}$ satisfies the predicate $\mathsf{binding\_ctx}(\mathcal{E})$.

For binding contexts, we can define an interpretation relation that given an environment, evaluates a binding context and returns a new environment that contains the bindings of that context. For example, consider a binding context $\texttt{let}\ x = \texttt{C}_0()\ \texttt{in}\ \texttt{let}\ y = \texttt{C}_\text{S}(x)\ \texttt{in}\ \texttt{let}\ z = y.1\ \texttt{in}\ [\,\cdot\,]$ where $\texttt{C}_0$ is the constructor for zero and $\texttt{C}_\text{S}$ the constructor for successor. It can be evaluated in an environment ρ to obtain $\rho[x \mapsto \texttt{C}_0(), y \mapsto \texttt{C}_\text{S}(\texttt{C}_0()), z \mapsto \texttt{C}_0()]$. This is not possible for unrestricted contexts; we cannot, *e.g.*, interpret a context that has a case-analysis node at the top level.

Many $\lambda_{\texttt{ANF}}$ transformations generate binding contexts.[2] It is useful to use the context interpretation relation, which composes with the evaluation relation of the language,[3] to reason about this code.

Notice that by requiring all subcomputations to be bound to variables, the structure of the functional program resembles a low-level imperative language. We can easily image replacing $\texttt{let}\ x = \ldots\ \texttt{in}\ e$ constructs with assignments and sequencing operations $x := \ldots; e$, to obtain a representation that is very close to assembly language.

[2] For example, closure conversion generates such contexts to project the code and environment for the closure pair

[3] This property is formally stated in Section 3.3.3.

In fact, the only thing that prevents us from translating $\lambda_{\texttt{ANF}}$ directly to assembly language (or another low-level IR, which in our case is Clight) is the presence of nested functions that refer to bindings outside of the scope of their own definition. This is one of the purposes of the $\lambda_{\texttt{ANF}}$ pipeline: efficiently compile away nested functions to top-level function definitions, allocating closures when needed. After compilation we will have a program that is in the form $\texttt{fun}\ f_1\ \overline{x_1} = e_1\ \texttt{in}\ \ldots \texttt{fun}\ f_n\ \overline{x_n} = e_n\ \texttt{in}\ e_{\texttt{main}}$, where $e_{i\in[1,n]}$ and $e_{\texttt{main}}$ do not contain function definitions and the top-level function definitions do not have any free variables, except from references to top-level functions. This first-order subset of $\lambda_{\texttt{ANF}}$ has the property that functions can be implemented solely with function pointers. The compiler program can now be compiled to C[4] or assembly (though for the latter, we would also need register allocation to produce efficient code).

3.2.1 Useful Definitions

Bound variables. An occurrence of a variable is bound if it is the binder of let expression, a function name, or a function argument. The scope of a binder x in a let expression $\texttt{let}\ x = \ldots\ \texttt{in}\ e$ is the expression e. In a function definition $\texttt{fun}\ f\ \overline{x} = e_1\ \texttt{in}\ e_2$ the scope of the function name is both of the expressions e_1 and e_2, and the scope of the function arguments $\overline{x}$ is just the function body e_1. The set of bound variables of an expression e (resp. context $\mathcal{E}$) is denoted as $\mathsf{bv}(e)$ (resp. $\mathsf{bv}(\mathcal{E})$). Although formally these are two distinct definitions, to keep the paper presentation simple I overload the same notation.

Free variables. A variable x is free in an expression e if it occurs in a non-bound position and it is not in the scope of a binder with the same name. The set of free variables of an expression e (resp. context $\mathcal{E}$) is denoted as $\mathsf{fv}(e)$ (resp. $\mathsf{fv}(\mathcal{E})$). A term that has no free variables satisfies $\mathsf{closed}(e) \overset{\text{def}}{=} (\mathsf{fv}(e) \subseteq \emptyset)$

Unique bindings. An expression e (resp. context $\mathcal{E}$) has unique bindings, denoted as $\mathsf{ub}(e)$ (resp. $\mathsf{ub}(\mathcal{E})$), if all of its bound variables are distinct.

Substitution. I denote $e\{y/x\}$ the substitution of variable x with variable y in expression e. Note, that in $\lambda_{\texttt{ANF}}$ it is only possible to substitute variables with other variables, and not expressions.

Well-scopedness. $\lambda_{\texttt{ANF}}$ programs are expected to adhere to some syntactic restriction that will be preserved by $\lambda_{\texttt{ANF}}$ transformations. In particular, a well scoped $\lambda_{\texttt{ANF}}$ term e ($\mathsf{well\_scoped}$(e)) has the property of unique bindings (ub(e)) and the set of its free variables is disjoint from the set of its bound variables ($\mathsf{fv}(e) \cap \mathsf{bv}(e) = \emptyset$).

[4] Provided that the C compiler can handle efficient tail calls without growing the stack. Twentieth century C compilers (generally) could not do this, but 21st-century C compilers can: gcc does a very impressive job on tail calls, clang/LLVM does a good job, and CompCert does a rudimentary job, allowing tail calls only when the callee has exactly the same number of parameters as the caller.

3.3 Semantics

The semantic definitions of the source and target languages are central to a compiler correctness theorem. To show that some aspect of computation is preserved, it must be captured by the semantics of both the source and target language. For the correctness of the $\lambda_{\texttt{ANF}}$ pipeline we are interested in preserving two properties: (i) If the source program terminates, so does the target program and the results of the computation are logically related. (ii) If the source program diverges, so does the target program. In Coq all functions are required to be terminating, which is syntactically enforced by checking that each recursive call is made to a structurally smaller argument. This is required to prevent logical inconsistencies (a non-terminating function can have an arbitrary return type and hence it can be used to prove false) and also to keep type checking decidable (type checking uses term normalization: if the normalization function can diverge then so can type checking). We are nevertheless interested in showing that divergence is preserved: the $\lambda_{\texttt{ANF}}$ could act as a general-purpose verified optimizing pipeline for functional languages if extended with additional constructs (*e.g.*, primitive operations). In addition, preservation of nonterminating behaviors is an interesting problem from a compiler correctness prospective and I wish to demonstrate the generality of the approaches presented in this thesis.

3.3.1 Big-Step *vs.* Small-Step Semantics.

Both small-step and big-step semantics are extensively used in programming languages theory. To motivate the choices in the semantic definition of $\lambda_{\texttt{ANF}}$, let us look at some important differences of small-step and big-step semantics.

Big-step (or natural) semantics. Big-step semantics relates a program with the final result of its evaluation, and therefore, it can only capture terminating executions of a program. A big-step semantics is particularly convenient for compiler correctness proofs, but, in its basic formulation, it can only be used to show correctness of compilation only for terminating programs.

Small-step semantics. A small-step semantics is defined using a notion of a one-step reduction that can be used to form finite or infinite reduction sequences. Therefore small-step semantics can be used to model terminating, diverging, and stuck executions of programs. Small-step semantics can vastly complicate compiler correctness proofs [88], but they allow proving that divergence is preserved.

Typically, a compiler correctness statement formulated using small-step semantics states that, starting from related initial states, whenever the source programs takes a step, the target program takes zero or more steps and the two resulting states are also related. Since the transformation might reduce the number of steps, then there might be steps in the source that do not correspond to any steps in the target (*e.g.*, think of function inlining). Notice, however, that this correctness statement does not guarantee divergence preservation: infinite reduction steps in the source might correspond to a finite number of reduction steps in the target. This is commonly referred to as the *stuttering problem* [86] and can be avoided by requiring that some measure function over source program states is strictly decreasing whenever the target performs zero steps. This provides an upper bound to the source steps that do not correspond to a reduction step in the target. Therefore, one can derive that infinite reduction sequences in the source correspond to infinite reduction sequences in the target.

There are different approaches in the literature that can be used to reconcile big-step semantics and diverging programs.

Coinductive big-step semantics. One approach to model diverging executions is to use an additional coinductive big-step judgment to form infinite evaluation derivations [39, 88]. The weakness of this approach is that two separate semantic preservation proofs have to be done: one for terminating and one for diverging executions. To avoid writing two separate evaluation judgments, *coevaluation* [88, 33] can be used that interprets the standard evaluation relation coinductively. This is viewed as a less satisfactory solution as it does not capture all nonterminating executions and it is not very well behaved in semantic preservation proofs.

Functional big-step semantics. The CakeML compiler handles divergence preservation by using functional big-step semantics [105]. The semantics is fuel-based and, similar to ours, throws an out-of-time exception when the fuel value is not enough to evaluate the next execution step. Unlike the semantics of $\lambda_{\texttt{ANF}}$, it is defined as an interpreter rather than an inductive definition. CakeML uses functional big-step semantics to show that nontermination is preserved: if the source term times out with some fuel value then so does the target term for the same fuel value. Of course, this does not always hold: a program may timeout for some fuel value f, but after program optimization, f might be enough to execute the target program. CakeML works around that by introducing a special `Tick` instruction whose only effect is to reduce the fuel. It is introduced by transformations whenever some instruction that winds down the clock is optimized away. The drawback is that an additional `Tick` construct is introduced in the IRs of the compiler and is propagated all the way to the back end of the compiler. An additional tick-erasure pass has to be made at the final compilation step, and of course the corresponding proof. The proof is a backward simulation stating that the behavior of a program with ticks erased is subsumed by the behavior with a program with ticks.

My solution to the divergence-preservation problem is inspired by both the CakeML approach and the solution to the small-step stuttering problem. Similar to CakeML, I use a fuel-based big-step semantics that will time-out whenever there is not enough fuel. However, I avoid the use of ticks by showing that whenever the source execution times out for some fuel value c the target times out for some fuel value c', such that $c \leq f(c')$, for some function f. It is then easy to show that if f has the property[5] $f(x) \leq f(y) \Rightarrow x \leq y$ then divergence is preserved.

Finding an upper bound for the execution cost of the source in terms of the execution cost of the target is not always easy. To that end, apart from the fuel, the semantics keeps track of the an execution *trace* that provides additional information

[5] This inequality holds iff f is strictly monotonic.

for (*e.g.*, number of a particular type of evaluation steps) that can be used to express the upper bound. The trace will be used to show that the inlining transformation preserves divergence. For the rest of the transformations using just fuel is enough.

3.3.2 Formal Definition

To define our notion of semantics we use a fuel-based big-step definition. The evaluation relation is written $(\sigma, e)\ {}^{f}\!\Downarrow^{t}\ r$ and reads as follows. Given a configuration $(\sigma, e) \in \texttt{Env} \times \texttt{Exp}$ and a fuel value f, evaluation produces a result r that can be either a value or an out-of-time exception:

$$r ::= \mathsf{Res}(v) \mid \mathsf{OOT}$$

Furthermore, it returns an execution trace t that captures certain aspects of the computation. Notice that both the trace and the fuel have a similar purpose: they profile aspects of the computation that are not related to the final result. Their difference is that the semantics will inspect the fuel value and it will return an OOT exception if there is not enough fuel to carry out the computation. On the other hand, the trace is never inspected and it is constructed "on the side" of the main computation.

Monoids

In the semantic definition both notions of fuel and traces are abstract, and can be instantiated later on with different cost and trace models. I achieve that by parameterizing the semantics with two commutative monoids: $\langle \mathcal{F}, \langle + \rangle_{\mathcal{F}}, \langle 0 \rangle_{\mathcal{F}} \rangle$ and $\langle \mathcal{T}, \langle + \rangle_{\mathcal{T}}, \langle 0 \rangle_{\mathcal{T}} \rangle$ for the fuel and the trace type respectively. For each of them we have that the binary operation is associative and commutative and has the zero as its identity element. Each monoid gives rise to a preorder:

$$x \leq_S y \overset{\text{def}}{=} \exists z,\ y = z \,\langle + \rangle_S\, x \quad \text{with } S \in \{\mathcal{F}, \mathcal{T}\}$$

Furthermore, I assume that there is a way of generating "elementary" elements for the two sets $\mathcal{F}$ and $\mathcal{T}$. I assume two functions $\langle \cdot \rangle_{\mathcal{F}} : \text{exp} \to \mathcal{F}$ and $\langle \cdot \rangle_{\mathcal{T}} : \text{exp} \to \mathcal{T}$ that map expressions of the language to $\mathcal{F}$ and $\mathcal{T}$. The fuel and the trace monoids are the free monoids generated by the codomains of $\langle \cdot \rangle_{\mathcal{F}}$ and $\langle \cdot \rangle_{\mathcal{T}}$ respectively.

For the $\mathcal{F}$ monoid, I also assume a strict partial order $<_{\mathcal{F}}$, which is used by the semantics to check if there is enough fuel to perform an execution step. The ordering

$<_{\mathcal{F}}$ must satisfy the following axioms.[6]

$$
\begin{aligned}
&\forall\ x\ y\ z,\ x <_{\mathcal{F}} y \rightarrow y <_{\mathcal{F}} z \rightarrow\ x <_{\mathcal{F}} z && \textit{(Transitivity)}\\
&\forall\ x,\ x \not<_{\mathcal{F}} x && \textit{(Irreflexivity)}\\
&\forall\ x,\ x \not<_{\mathcal{F}} \langle 0\rangle_{\mathcal{F}} && \textit{(}\langle 0\rangle_{\mathcal{F}}\textit{ is the least element)}\\
&\forall\ e\ x,\ \langle 0\rangle_{\mathcal{F}} <_{\mathcal{F}} \langle e\rangle_{\mathcal{F}} && \textit{(}\langle e\rangle_{\mathcal{F}}\textit{ is strictly greater than }\langle 0\rangle_{\mathcal{F}}\textit{)}\\
&\forall\ x\ y\ z,\ x <_{\mathcal{F}} y \leftrightarrow x \,\langle +\rangle_{\mathcal{F}}\, z <_{\mathcal{F}} y \,\langle +\rangle_{\mathcal{F}}\, z && \textit{(}\langle +\rangle_{\mathcal{F}}\textit{ preserves and reflects}\\
& && \textit{the ordering)}\\
&\forall\ x\ y, x <_{\mathcal{F}} y \vee y \leq_{\mathcal{F}} x && \textit{(Decidability)}
\end{aligned}
$$

When referring to the binary operation, the identity element, the generator function, or the orders of the monoids I will often drop the subscript when it is clear from the context to which of the two monoids I am referring.

In Chapter 5, where I set up the relational framework for proving correctness I will also make use of a homomorphism $\uparrow : \mathcal{F} \rightarrow \mathbb{N}$ that must satisfy the following.

$$
\begin{aligned}
&\forall\ x\ y,\ \uparrow\!(x \,\langle +\rangle_{\mathcal{F}}\, y) = \uparrow\! x + \uparrow\! y && \textit{(}\uparrow\textit{ preserves }\langle +\rangle_{\mathcal{F}}\textit{)}\\
&\forall\ x\ y,\ \uparrow\!(\langle 0\rangle_{\mathcal{F}}) = 0 && \textit{(}\uparrow\textit{ preserves }\langle 0\rangle_{\mathcal{F}}\textit{)}
\end{aligned}
$$

Monoids (especially partial commutative monoids) have been traditionally used to give a general model to both resources and traces in the semantics of programming languages. For example, resource monoids are used in Iris [72] as a generic way to express protocols on shared state and in cost analysis [62] to express generically the composition of resource consumption.

The fuel and trace monoids are kept abstract during semantic preservation proofs. They are only need to be instantiated at the top-level, when one wants to derive the divergence preservation theorem.

Example 3.1 (Fuel Monoid)
For the purposes of this thesis, I will instantiate the fuel monoid with $\langle \mathbb{N}, +, 0\rangle$. There are different generators than can be used, that define a different cost model for the language. In order to show divergence preservation, it suffices to consider $\langle e\rangle \overset{\text{def}}{=} 1$.

Example 3.2 (Trace Monoid)
I will use the trace monoid to profile the number of different kind of steps that a program takes. These are then used to form an upper bound for fuel consumption of the source program. I will use the trace monoid $\langle \mathbb{N} \times \mathbb{N}, +, (0, 0)\rangle$, where $+$ is pairwise addition. The first component keeps track of the non-application steps, while

[6] I could have also taken $x <_{\mathcal{F}} y \overset{\text{def}}{=} \exists z, z \neq \langle 0\rangle_{\mathcal{F}} \ \wedge\ y = z \,\langle +\rangle_{\mathcal{F}}\, x$. This would be sufficient to prove transitivity, but I would again have to make assumptions about the underlying structure so that all the required axioms are provable.

the second one the application steps. The corresponding generator is:

$$\begin{array}{ll}
\langle \texttt{let } x = \mathtt{C}(\overline{y}) \texttt{ in } e \rangle \overset{\text{def}}{=} (1,0) & \langle \texttt{let } x = y.i \texttt{ in } e \rangle \overset{\text{def}}{=} (1,0) \\
\langle \texttt{case } y \texttt{ of } [\mathtt{C}_i \ \rightarrow \ e_i]_{i \in I} \rangle \overset{\text{def}}{=} (1,0) & \langle \texttt{fun } f\ \overline{x} = e_1 \texttt{ in } e_2 \rangle \overset{\text{def}}{=} (1,0) \\
\langle \texttt{let } x = f\ \overline{y} \texttt{ in } e \rangle \overset{\text{def}}{=} (0,1) & \langle f\ \overline{y} \rangle \overset{\text{def}}{=} (0,1) \\
\langle \texttt{ret}(x) \rangle \overset{\text{def}}{=} (1,0) &
\end{array}$$

Semantics

The inductive definition of the semantics is shown in Figure 3.2. The $\Downarrow$ relation is defined simultaneously with an auxiliary relation $\downarrow$. Intuitively, the rules of $\Downarrow$ are responsible for the fuel and trace profiling, whereas $\downarrow$ is responsible for performing the evaluation step. This avoids obfuscating the evaluations rule of each constructor with additional premises that manipulate the fuel.

The first rules are the introduction (rule Constr) and elimination (rules Proj and Case) of constructed values are straightforward. Similarly the rule Fun introduces a closure value, and rules Let-app, Let-app-OOT and App eliminate a closure value with function application. Rule Let-app prescribes what happens when the function body evaluates to a value, whereas rule Let-app-OOT prescribes what happens when evaluating the function body results in an OOT. Rule Ret terminates a computation by returning the value that corresponds to the returned variable. Rule OOT throws an OOT exception if the value of the fuel is less than the fuel required for the evaluation of the outermost constructor of the expression. Rule Step invokes the auxiliary relation for the evaluation of the topmost constructor, and adjusts the values of the fuel and the trace.

For evaluation of closed programs in the empty environment I will write $e\ {}^{c}\!\Downarrow^{t}\ r$, dropping the fuel and trace superscripts when they are irrelevant.

Divergence

Diverging programs are defined using the evaluation relation. In particular, a program diverges if for all values of the fuel, evaluation raises an OOT exception with some trace value.

$$(\sigma, e) \Uparrow \overset{\text{def}}{=} \forall\ c, \exists f,\ (\sigma, e)\ {}^{c}\!\Downarrow^{f}\ \mathsf{OOT}$$

Context Interpretation

The evaluation semantics extend naturally to the interpretation of binding contexts. The relation is written

$$(\sigma, \mathcal{E})\ {}^{c}\!\triangleright^{t}\ o$$

where

$$o ::= \mathsf{Res}(\sigma) \mid \mathsf{OOT}$$

$$\frac{\sigma(\overline{y}) = \overline{v} \qquad (\sigma[x \mapsto \mathtt{C}(\overline{v})], e)\ {}^{c}\!\Downarrow^{t}\ r}{(\sigma, \mathtt{let}\ x = \mathtt{C}(\overline{y})\ \mathtt{in}\ e)\ {}^{c}\!\downarrow^{t}\ r}\ \textsc{Constr}$$

$$\frac{\sigma(y) = \mathtt{C}(v_1, \ldots, v_j, \ldots, v_n) \qquad (\sigma[x \mapsto v_j], e)\ {}^{c}\!\Downarrow^{t}\ r}{(\sigma, \mathtt{let}\ x = y.j\ \mathtt{in}\ e)\ {}^{c}\!\downarrow^{t}\ r}\ \textsc{Proj}$$

$$\frac{\sigma(x) = \mathtt{C}_i(\overline{v}) \qquad (\sigma, e_i)\ {}^{c}\!\Downarrow^{t}\ r}{(\sigma, \mathtt{case}\ x\ \mathtt{of}\ \{\mathtt{C}_i\ \rightarrow\ e\}_{i \in I})\ {}^{c}\!\downarrow^{t}\ r}\ \textsc{Case}$$

$$\frac{(\sigma[f \mapsto \mathtt{Clo}(\sigma, \mathtt{fun}\ f\ \overline{x} = e_1)], e_2)\ {}^{c}\!\Downarrow^{t}\ r}{(\sigma, \mathtt{fun}\ f\ \overline{x} = e_1\ \mathtt{in}\ e_2)\ {}^{c}\!\downarrow^{t}\ r}\ \textsc{Fun}$$

$$\frac{\begin{array}{c} \sigma(f) = \mathtt{Clo}(\sigma_g, \mathtt{fun}\ g\ \overline{z} = e_g) \\ \sigma(\overline{y}) = \overline{v} \qquad (\sigma_g[\overline{z} \mapsto \overline{v}][g \mapsto \mathtt{Clo}(\sigma_g, \mathtt{fun}\ g\ \overline{z} = e_g)], e_g)\ {}^{c_1}\!\Downarrow^{t_1}\ \mathsf{Res}(v_1) \\ (\sigma[x \mapsto v_1], e)\ {}^{c_2}\!\Downarrow^{t_2}\ r \end{array}}{(\sigma, \mathtt{let}\ x = f\ \overline{y}\ \mathtt{in}\ e)\ {}^{c_1 \langle + \rangle c_2}\!\downarrow^{t_1 \langle + \rangle t_2}\ r}\ \textsc{Let-app}$$

$$\frac{\begin{array}{c} \sigma(f) = \mathtt{Clo}(\sigma_g, \mathtt{fun}\ g\ \overline{z} = e_g) \\ \sigma(\overline{y}) = \overline{v} \qquad (\sigma_g[\overline{z} \mapsto \overline{v}][g \mapsto \mathtt{Clo}(\sigma_g, \mathtt{fun}\ g\ \overline{z} = e_g)], e_g)\ {}^{c}\!\Downarrow^{t}\ \mathsf{OOT} \end{array}}{(\sigma, \mathtt{let}\ x = f\ \overline{y}\ \mathtt{in}\ e)\ {}^{c}\!\downarrow^{t}\ \mathsf{OOT}}\ \textsc{Let-app-oot}$$

$$\frac{\begin{array}{c} \sigma(f) = \mathtt{Clo}(\sigma_g, \mathtt{fun}\ g\ \overline{z} = e_g) \\ \sigma(\overline{y}) = \overline{v} \qquad (\sigma_g[\overline{z} \mapsto \overline{v}][g \mapsto \mathtt{Clo}(\sigma_g, \mathtt{fun}\ g\ \overline{z} = e_g)], e_g)\ {}^{c}\!\Downarrow^{t}\ r \end{array}}{(\sigma, f\ \overline{y})\ {}^{c}\!\downarrow^{t}\ r}\ \textsc{App}$$

$$\frac{\sigma(x) = v}{(\sigma, \mathtt{ret}(x))\ {}^{\langle 0 \rangle}\!\downarrow^{\langle 0 \rangle}\ \mathsf{Res}(v)}\ \textsc{Ret} \qquad\qquad \frac{i < \langle e \rangle_{\mathcal{F}}}{(\sigma, e)\ {}^{i}\!\Downarrow^{\langle 0 \rangle}\ \mathsf{OOT}}\ \text{OOT}$$

$$\frac{(\sigma, e)\ {}^{c}\!\downarrow^{t}\ r}{(\sigma, e)\ {}^{c \langle + \rangle \langle e \rangle}\!\Downarrow^{t \langle + \rangle \langle e \rangle}\ r}\ \textsc{Step}$$

Figure 3.2: Evaluation semantics of $\lambda_{\mathtt{ANF}}$.

Given an environment σ, it interprets the binding context $\mathcal{E}$ to obtain a result o that can be either an environment or an out-of-time exception. A context cannot be interpreted if it is not a binding context. Analogous to $\Downarrow$ and $\downarrow$, $\rhd$ is defined simultaneously with an auxiliary relation $\blacktriangleright$. The rules closely follow the rules of the evaluation relation and are not shown.

3.3.3 Properties of the Semantics

I describe some important properties of the semantics. The proofs of these theorems are fully mechanized and hence omitted, unless they elucidate some particular methodology.

The following lemma asserts that every computation times out with an empty trace when evaluated with zero fuel.

Lemma 3.3 (Evaluation with zero fuel)
For all σ and e, we have $(\sigma, e)\ {}^{\langle 0\rangle}\!\Downarrow^{\langle 0\rangle}\ \mathsf{OOT}$.

The semantics is deterministic in the following sense: whenever we have two terminating executions of the same configuration, both the trace and the fuel must be the same.

Lemma 3.4 (Determinism (termination))
Let $(\sigma, e)\ {}^{c}\!\Downarrow^{t}\ \mathsf{Res}(v)$ and $(\sigma, e)\ {}^{c'}\!\Downarrow^{t'}\ \mathsf{Res}(v')$ be two terminating evaluation derivations. Then $c = c'$, $v = v'$ and $t = t'$.

Furthermore, if a configuration terminates with some fuel a trace value then for all strictly smaller values of fuel there exists some trace, which is a subtrace of the original one, such that the configuration times out.

Lemma 3.5 (Evaluation with less fuel)
Let $(\sigma, e)\ {}^{c}\!\Downarrow^{t}\ \mathsf{Res}(v)$ be a terminating derivation. Then for all $c' <_{\mathcal{F}} c$ there exists a trace $t' \leq_{\mathcal{T}} t$ such that $(\sigma, e)\ {}^{c'}\!\Downarrow^{t'}\ \mathsf{OOT}$.

Notice that it is not possible to obtain an evaluation derivation for values of fuel strictly greater than the one of the terminating execution.

Lemma 3.6 (Evaluation with more fuel)
Let $(\sigma, e)\ {}^{c}\!\Downarrow^{t}\ \mathsf{Res}(v)$ be a terminating derivation. Then for all c' such that $c <_{\mathcal{F}} c'$, trace t', and result r, the is no derivation $(\sigma, e)\ {}^{c'}\!\Downarrow^{t'}\ r$.

Incomplete evaluations are monotonic in the fuel value in the sense that given an OOT-evaluation of a program, we know that for any smaller value of the fuel the evaluation will time out giving a subtrace of the original trace.

Lemma 3.7 (Monotonicity (OOT))
Let $(\sigma, e)\ {}^{c}\!\Downarrow^{t}\ \mathsf{OOT}$ be an OOT-derivation. Then for all $c' \leq_{\mathcal{F}} c$ there exists a trace $t' \leq_{\mathcal{T}} t$ such that $(\sigma, e)\ {}^{c'}\!\Downarrow^{t'}\ \mathsf{OOT}$.

Given two OOT-evaluations of the same configuration, we know that the one with the smaller fuel will also produce a smaller trace.

Lemma 3.8 (Monotonicity (OOT))
Let $(\sigma, e)\ {}^{c}\!\Downarrow^{t}\ \mathsf{OOT}$ and $(\sigma, e)\ {}^{c'}\!\Downarrow^{t'}\ \mathsf{OOT}$. Then if $c' \leq_{\mathcal{F}} c$ we have that $t' \leq_{\mathcal{T}} t$.

Given two evaluations of the same configuration with the same fuel, we know that the results and traces must also be the same.

Lemma 3.9 (Evaluation with the same fuel (determinism))
Let $(\sigma, e)\ {}^{c}{\Downarrow}^{t}\ r$ and $(\sigma, e)\ {}^{c}{\Downarrow}^{t'}\ r'$ be two evaluation derivations. Then $r = r'$ and $t = t'$.

The context interpretation relation has similar properties.

We can also formally state the divergence preservation theorem that was described earlier.

Lemma 3.10 (Divergence preservation)
Let f be a function $\mathcal{F} \to \mathcal{F}$ such that $f(x) \leq_{\mathcal{F}} f(y) \Rightarrow x \leq_{\mathcal{F}} y$. Assume that for two configurations (σ_1, e_1) and (σ_2, e_2) we know that if $(\sigma_1, e_1)\ {}^{c_1}{\Downarrow}^{t_1}\ \mathsf{OOT}$ then there exist c_2 and t_2 such that $(\sigma_2, e_2)\ {}^{c_2}{\Downarrow}^{t_2}\ \mathsf{OOT}$ and $c_1 \leq f(c_2)$. Then if $(\sigma_1, e_1) \Uparrow$ we have that $(\sigma_2, e_2) \Uparrow$.

Proof Let c be a fuel value. We must show that there exists t such that $(\sigma_2, e_2)\ {}^{c}{\Downarrow}^{t}\ \mathsf{OOT}$. From the hypothesis that e_1 is a diverging program, we know that $(\sigma_1, e_1)\ {}^{f(c)}{\Downarrow}^{t_1}\ \mathsf{OOT}$ for some t_1. Therefore, we can derive that $(\sigma_2, e_2)\ {}^{c_2}{\Downarrow}^{t_2}\ \mathsf{OOT}$ for some c_2 and t_2 such that $f(c) \leq f(c_2)$. But from the hypothesis about f we have that $c \leq c_2$. From lemma 3.7 we obtain $t \leq t_2$ such that $(\sigma_2, e_2)\ {}^{c}{\Downarrow}^{t}\ \mathsf{OOT}$. ■

We can also state how the evaluation relation composes with the interpretation of a binding context. If a context $\mathcal{E}$ is interpreted in environment σ as a new environment σ' and an expression e evaluates to a result in the environment σ', then the expression $\mathcal{E}[e]$ evaluates to the same result in the initial environment $\mathcal{E}$.

Lemma 3.11 (Composition)
Let $(\sigma, \mathcal{E})\ {}^{c_1}{\triangleright}^{t_1}\ \mathsf{Res}(\sigma')$ and $(\sigma', e)\ {}^{c_2}{\Downarrow}^{t_2}\ r$. Then $(\sigma, \mathcal{E}[e])\ {}^{c_1 \langle + \rangle c_2}{\Downarrow}^{t_1 \langle + \rangle t_2}\ \mathsf{Res}(v)$.

In the interpretation of a context $\mathcal{E}$ times out with some fuel value c, then for any expression e so does $\mathcal{E}[e]$ for the same fuel value.

Lemma 3.12 (Composition (OOT))
Let $(\sigma, \mathcal{E})\ {}^{c}{\triangleright}^{t}\ \mathsf{OOT}$. Then $(\sigma, \mathcal{E}[e])\ {}^{c}{\Downarrow}^{t}\ \mathsf{OOT}$.

Given an evaluation of $\mathcal{E}[e]$ for some binding context $\mathcal{E}$, we can decompose it to the interpretation of $\mathcal{E}$ and the evaluation of e. We distinguish two cases: the evaluation of expression $\mathcal{E}[e]$ returns a result and the evaluation of $\mathcal{E}[e]$ times out. In the former case the interpretation of $\mathcal{E}$ terminates with a new environment and the evaluation of e terminates in the new environment.

Lemma 3.13 (Decomposition)
Let $(\sigma, \mathcal{E}[e])\ {}^{c}{\Downarrow}^{t}\ \mathsf{Res}(v)$ for some binding context $\mathcal{E}$. Then there exist fuel values c_1 and c_2 and trace values t_1 and t_2 such that $(\sigma, \mathcal{E})\ {}^{c_1}{\triangleright}^{t_1}\ \mathsf{Res}(\sigma')$, $\quad (\sigma', e)\ {}^{c_2}{\Downarrow}^{t_2}\ \mathsf{Res}(v)$, $c = c_1 \langle + \rangle_{\mathcal{F}}\, c_2$, and $c = t_1 \langle + \rangle_{\mathcal{T}}\, t_2$.

The latter case is more complicated. If the evaluation of $\mathcal{E}[e]$ times out for some fuel c, then either the interpretation of $\mathcal{E}$ times out for the same fuel, or the interpretation of $\mathcal{E}$ terminates for some smaller fuel value and the evaluation of e times out for the remaining fuel.

Lemma 3.14 (Decomposition (OOT))
Let $(\sigma, \mathcal{E}[e])\ {}^{c}{\Downarrow}^{t}\ \mathsf{OOT}$ for some binding context ctx. Then either $(\sigma, \mathcal{E})\ {}^{c}{\triangleright}^{t}\ \mathsf{OOT}$ or there exist c_1 and c_2 and trace values t_1 and t_2 such that $(\sigma, \mathcal{E})\ {}^{c_1}{\triangleright}^{t_1}\ \mathsf{Res}(\sigma')$, $(\sigma', e)\ {}^{c_2}{\Downarrow}^{t_2}\ \mathsf{OOT}$, $\quad c = c_1 \langle + \rangle_{\mathcal{F}}\, c_2$, and $c = t_1 \langle + \rangle_{\mathcal{T}}\, t_2$.

3.4 Conclusion

In this chapter I presented the syntax and semantics of $\lambda_{\texttt{ANF}}$, the intermediate representation on which CertiCoq optimizations are performed. The big-step semantics of $\lambda_{\texttt{ANF}}$ is designed to enable tracking runtime information about the resource consumption of programs through abstract fuel and trace monoids. The semantics is fuel-based, facilitating reasoning about nonterminating behaviors.

The $\lambda_{\texttt{ANF}}$ language can serve a general compilation target for functional languages. With small extensions (primitive operations and datatypes) it could handle source languages with more features (*e.g.*, references) and be used to compile reasonable fragments of languages like OCaml.

Chapter 4

The $\lambda_{\texttt{ANF}}$ Optimizing Pipeline

4.1 Overview

The $\lambda_{\texttt{ANF}}$ optimizing pipeline captures the *essence of compiling a pure functional language.* Its input is a pure functional program with higher-order functions, nested lexical scoping, and curried functions (meaning that they expect exactly one user argument — and a continuation argument if CPS transformation is used). The output of the $\lambda_{\texttt{ANF}}$ pipeline, while still in (a subset of) the $\lambda_{\texttt{ANF}}$ language, can be easily compiled to a low-level, first-order intermediate representation. In this subset of $\lambda_{\texttt{ANF}}$ used at the back end of the pipeline, functions can be implemented simply with function pointers, available in assembly or C. The $\lambda_{\texttt{ANF}}$ pipeline compiles away lexically nested functions, by explicitly introducing closures, introduces multi-argument functions, and simplifies the code with a series of optimization passes. However, merely compiling away higher-order functions by introducing closures will not generate efficient code. Crucially, the optimizing $\lambda_{\texttt{ANF}}$ pipeline generates efficient function calls by implementing specialized closure-allocation and parameter-passing strategies for known functions.

The $\lambda_{\texttt{ANF}}$ follows a compilation-by-transformation [75, 69] approach: it uses many small and modular same-language transformations to optimize the code and compile away features that are not supported by the target language. Optimizations of $\lambda_{\texttt{ANF}}$ take heavy advantage of the "cascade effect": performing a code simplification often exposes new opportunities for optimization. The $\lambda_{\texttt{ANF}}$ transformations can be divided into two categories:

- Simplification passes that make static reductions in the code. These consist of ***inlining*** that performs static beta reductions, and ***shrinking*** [15, 23] that performs shrink reductions: projection folding, case folding, dead code elimination, and inlining (of functions that are called exactly once). Such transformations are crucial to simplify administrative redexes introduced by the transformations in the next category.

- Transformations that change the calling strategies of functions: ***uncurrying***, ***lambda lifting***, ***closure conversion***, and ***dead parameter elimination***. Uncurrying introduces multi-argument functions, closure conversion compiles nested lexical scoping into flat scoping, lambda lifting turns free variables into parameters in order to eliminate closures when this is possible, and dead parameter elimination removes parameters that are not needed (that can be initially present in the code or introduced by other transformations, *e.g.*, closure conversion). The simplification transformations are called multiple times, between other $\lambda_{\texttt{ANF}}$ transformations, and perform administrative reductions that help keep these transformations simple. For example, by doing separate inlining passes we can express both uncurrying and lambda lifting as local transformations. Shrinking statically reduced administrative redexes introduced by closure conversion, and dead parameter elimination removes useless environment parameters. These allow us to keep closure-conversion transformation very simple: our closure-conversion transformation will blindly closure convert each function. Useless closures will be eliminated by other transformations.

Before proceeding with explaining further the individual $\lambda_{\texttt{ANF}}$ transformations and their interactions, I give some background on some common compilation techniques for efficient closure implementation.

4.2 Closure Strategies

Efficient implementation of closures is essential for good performance of functional code. Heap-allocated closures are expensive for function creation (an environment and a closure pair must be allocated in the heap), function call (the heap must be accessed to project the code and environment from the closure pair) and function execution (the heap must be accessed to fetch the values of free variables of a function). Furthermore, closures increase heap allocation and stress the garbage collector. Although heap allocated closures are required to fully support higher-order functions, not every function needs to be represented with a heap-allocated closure at run time. The RABBIT compiler for Scheme [125] was the first compiler for a functional higher-order language to investigate efficient closure-allocation strategies. It demonstrated that function calls in languages with first-class functions need not be expensive and it inspired the subsequent generation of compilers for functional languages including the ORBIT compiler for Scheme [2] and the SML/NJ compiler [10]. Here, I review some common closure allocation strategies that enable efficient code generation.

First-class functions can *escape* their original scope of definition by being passed as arguments, returned as results of functions or stored in data structures. The presence of nested functions with lexical scoping however, functions can capture references to their environment that may not be available when they are called. The general solution to that is to represent a function as heap-allocated closure: a code pointer together with an environment that holds the values of the free variables of the function. However, not all functions need the full generality of closures, and the overhead of heap-allocated closures can often be avoided. We say that a function escapes *downwards* if it escapes only through parameter passing, and that it escapes *upwards* if it is being returned as a result. A function is *known* if all of is call sites are known. We can now distinguish the following cases for closure allocation.

- **Known functions with no free variables.** A known function with no free variables does not, generally, need a closure. It does not have free variables and since it does not escape it cannot flow to the same application position where a function with free variables does. Therefore its closure can be safely eliminated.

- **Known functions with free variables.** When a function is known but has free variables, these variables can be stored in registers by being passed as parameters to the function. Since all call sites of the function are known, they can be modified accordingly to pass free variables as extra parameters. This eliminates closure allocation and the memory accesses associated with fetching free variables. The transformation that turns free variables to extra parameters is known as lambda lifting [67]. When the number of arguments of the function after lambda lifting exceeds the number of the available registers in the machine then a closure may be used to avoid register spilling.

  Storing closure environments in registers is a common technique for CPS [125, 2, 10] and direct-style compilers [69, 87]. Lambda lifting is used by both GHC [69] and OCaml's Flambda optimization pipeline [87, Chapter 21] to reduce closure allocation. In both cases, lambda lifting is reported to improve the performance of some programs and worsen the performance of some other programs. In Chapter 8, I evaluate several design aspects of lambda lifting, some of which were never investigated before, identifying potential sources of overhead. Our resulting lambda lifting transformation never worsens the performance of programs in our benchmark suite compiled with ANF. Furthermore, we observe substantial speedup in certain programs.

- **Downward escaping functions.** The closure of functions that only escape downwards can be stack allocated. This is possible because downward escaping functions have a limited extent: a downward escaping function cannot be called after its original scope of definition is no longer active. This is better understood if we consider a function that only escapes through parameter passing. Such function cannot be called after the function to which it is passed as argument has returned.

  In languages like ALGOL and Pascal functions can only escape downwards. This allows the compilers to enforce a stack discipline for closure allocation and avoid the need for garbage collection. In the general case, where upward and downward escaping functions coexist, deciding where to allocate each closure (stack or heap) requires intraprocedural escape analysis [52] to approximate the set of downward escaping functions.

  Note, that in presence of first-class continuations (`callcc`) downward functions can have an unlimited extent, which complicates stack allocation of downward escaping functions.

  CertiCoq does not implement the stack allocation strategy.

- **Upward escaping functions.** Upward escaping functions require the full generality of heap-allocated closures. Such functions can have unlimited extent and therefore can be called after their original scope of definition has been deactivated. For closures of upward escaping functions a deallocation point cannot be statically determined and therefore they must be heap allocated and garbage collected.

- **Known functions called from nested functions.** Generally, a known function that is called from a function that escapes inherits the closure strategy of the function that calls it. That is, a function that is called from an escaping function is considered itself escaping, and its closure must be contained in the closure environment of the calling function. An exception to this rule is when the closure environment of the callee is a subset of the environment of the closure of the caller (which is trivially true if the callee is a closed function). In this case, all free variables of the function are available at the time of the call and they can be passed as arguments.

- **Functions with both known and escaping occurrences.** A function may both be applied at a known call site and escape. In such case, we want the known function application to avoid using a closure, even though a closure must be allocated when the function escapes. This is achieved by creating two instances of each function [10]: one that is used at known call sites and a second one, that is a wrapper around the known function, and is used at escaping positions. In this way, an optimized entry point is created for known calls to the function, while unknown calls must go through its closure-converted wrapper.

The CertiCoq compiler implements the above closure strategies, with the exception of the stack allocated closures.

Closure environment representations. Closures can also be optimized for their environment representation. Even though the closure-pair layout must be uniform for all functions that can flow into the same application positions, this is not true for the layout of the closure environment whose representation is private to each closure and its layout can be different for each function. The RABBIT, ORBIT compilers for Scheme and early implementations of the SML/NJ compiler used *linked closure environments*, where nested functions share parts of the closure environment with their enclosing function to improve space performance and optimize closure creation time. However, this representation was later shown to be unsafe for space [10], meaning that it can introduce space leaks, worsening the asymptotic space complexity of a program. A common alternative, which is also used in CertiCoq, is to use *flat environments* that contain exactly the free variables of each function and are safe for space. In Chapter 7, where I prove the space safety of CertiCoq's closure conversion, I give concrete examples of different environment representations and a counterexample for space safety for the linked environment representation.

Closure-environment sharing optimizations that are also safe for space are possible and have been employed by the SML/NJ compiler [120, 121].

4.3 Transformations

In this section, I describe each $\lambda_{\texttt{ANF}}$ optimization and the subtleties that it involves. I also describe the combined effect that is achieved by the coordination of the $\lambda_{\texttt{ANF}}$ transformations. The $\lambda_{\texttt{ANF}}$ pipeline is shown in Figure 4.1. Between transformations, we iteratively perform an inline pass followed by a shrinking pass, until no more redexes in the program can be reduced. It is important to sequence these two transformations in this way in order to enable cascading optimizations.[1]

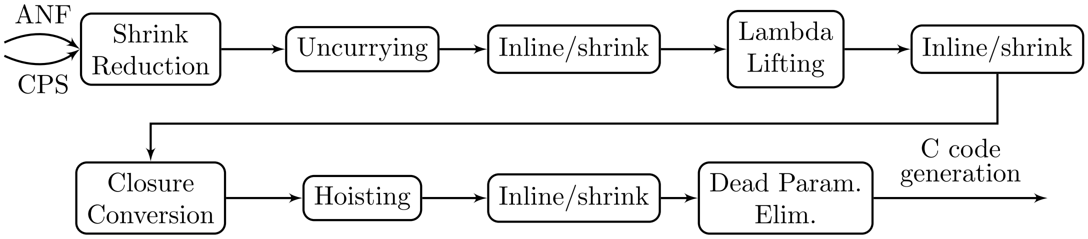


Figure 4.1: The $\lambda_{\texttt{ANF}}$ optimizing pipeline.

Presentation conventions. I do not explicitly provide the code or the relational specification of each transformation, I instead give a pointer to the online code repository where the code can be found. I describe the effect that the transformation has as a set of local rewrite steps. This often is informal as not every transformation can be described as a set of local rewrite steps (*e.g.*, closure conversion). In addition, the actual transformations will require alpha renamings to ensure the resulting term is well scoped. Recall from Section 3.2.1 that a term is well scoped (denoted $\mathsf{well\_scoped}(e)$) if it has unique bindings and its free variables are distinct from its bound variables. For the purposes of the presentation, I will allow reusing the same binder names, ignoring alpha renaming.

4.3.1 Shrinking[2]

The pipeline starts with a shrinking transformation pass that performs static reductions and simplifications that are guaranteed to never increase the size of the code (hence its name). Shrinking performs simple dead code elimination (*i.e.*, without liveness analysis), case folding, projection folding, and inlining of functions that are called exactly once. These steps are called shrink reductions. Shrinking is guaranteed to eliminate all administrative redexes after CPS conversion [22], significantly reducing the code size. CertiCoq's shrinking transformation for CPS is explained in detail in Savary Bélanger's thesis [21]. Here, I review the functionality of shrinking and I describe its extension to $\lambda_{\texttt{ANF}}$.

[1] Despite its iterative nature, this pass has minimal effect on the performance of the compiler.

[2] CertiCoq's shrinking transformation was implemented and verified for CPS by Savary Bélanger [15, 22]. The implementation and proof were later extended by me to apply to the full $\lambda_{\texttt{ANF}}$ language.

Dead code elimination. Shrinking will remove let-bindings that are not explicitly used in the rest of the program. In particular the transformation will perform the following reductions.

$$\begin{array}{c} \texttt{let}\ x = \texttt{C}(\overline{y})\ \texttt{in}\ e \\ \text{or} \\ \texttt{let}\ x = y.i\ \texttt{in}\ e \\ \text{or} \\ \texttt{fun}\ x\ \overline{y} = e'\ \texttt{in}\ e \end{array} \quad \rightsquigarrow \quad e \quad \text{if } x \notin \mathsf{fv}(e)$$

Note that, in general, the following reduction cannot be performed in a non-terminating language.[3]

$$\texttt{let}\ x = f\ \overline{y}\ \texttt{in}\ e \quad \rightsquigarrow \quad e \quad \text{if } x \notin \mathsf{fv}(e)$$

The problem is that application of f might never terminate, in which case the input program will diverge and the target might not necessarily do so. One however, can activate this reduction step if the input is restricted to terminating languages like Coq.

Projection folding. Shrinking will statically evaluate projections whenever the value of the destructed variable is statically known. The following reduction will evaluate $\texttt{let}\ z = x.i\ \texttt{in}\ e$ to $e\{y_i/x\}$ if the value of z is statically known to be a constructor whose i-th argument is y_i.

$$\begin{array}{l} \texttt{let}\ x = \texttt{C}(y_1, \ldots, y_n)\ \texttt{in}\ \mathcal{E}[\texttt{let}\ z = x.i\ \texttt{in}\ e] \\ \qquad\qquad\qquad\qquad \rightsquigarrow \\ \qquad \texttt{let}\ x = \texttt{C}(y_1, \ldots, y_n)\ \texttt{in}\ \mathcal{E}[e\{y_i/x\}] \qquad \text{if } x \notin \mathsf{bv}(\mathcal{E}) \text{ and } y_i \notin \mathsf{bv}(\mathcal{E}) \cup \mathsf{bv}(e) \end{array}$$

Capturing of y_i must be avoided by requiring that $y_i \notin \mathsf{bv}(\mathcal{E}) \cup \mathsf{bv}(e)$. This is always satisfied by well-scoped programs. Note that in a conventional lambda-calculus, where the y_i can be expressions representing computations, projection-folding (and other shrink optimizations) can increase work by duplicating computations; but in CPS or ANF, the y_i are just variables, and no computation is duplicated.

Case folding. Similarly, case constructs will be statically evaluated whenever the scrutinee is statically known to be a constructed value whose constructor appears in the discriminating patterns. In such a case, $\texttt{case}\ x\ \texttt{of}\ [\texttt{C}_i\ \rightarrow\ e_i]_{i \in I}$ becomes e_j when x is known to be a constructed value with constructor tag $\texttt{C}_j$ with $j \in I$.

$$\begin{array}{l} \texttt{let}\ x = \texttt{C}_j(\overline{y})\ \texttt{in}\ \mathcal{E}[\texttt{case}\ x\ \texttt{of}\ [\texttt{C}_i\ \rightarrow\ e_i]_{i \in I}] \\ \qquad\qquad\qquad\qquad \rightsquigarrow \\ \qquad\qquad \texttt{let}\ x = \texttt{C}_j(\overline{y})\ \texttt{in}\ \mathcal{E}[e_j] \qquad\qquad \text{if } x \notin \mathsf{bv}(\mathcal{E}) \end{array}$$

[3] A typed Gallina program always terminates. $\lambda_{\texttt{ANF}}$ proofs do not assume that the source is a terminating program; the $\lambda_{\texttt{ANF}}$ pipeline could be used to compile non-terminating languages as well.

Function Inlining. Known calls to nonrecursive functions that are called exactly once will become statically evaluated by inlining the function body at the place of the call. Choosing to inline only functions that are used once allows us to delete the function definition once the inlining has happened. Therefore, renaming the binders in the inlined expression is not necessary to preserve well-scopedness. Essentially, extending shrinking to the full $\lambda_{\texttt{ANF}}$ involves inlining let-bound function calls. I don't give the rewrite rules for shrink-inline, as they are similar to the rewrite rules for inlining that I describe next.

The shrinking algorithm can perform many cascading optimizations in the same linear-time pass (or $O(n \log n)$ time in a language without mutable arrays, where lookup tables must be implemented as purely functional search trees). Often a shrink reduction might enable another shrink reduction of a different kind. For example, a projection folding might reveal some dead code. Removing the dead code can enable inlining of a function that was previously applied more than once.[4] Shrinking will be applied repeatedly until the program is in shrink-normal form. Shrinking will also perform the necessary substitutions of variables as it traverses the term so that a second traversal is not needed.

The shrinking phase is crucial not only to simplify redexes that the user wrote in source code, but to reduce administrative redexes from the CPS (or ANF) transformation, and to simplify the result of other optimizations such as closure conversion and lambda lifting. To achieve that, shrinking is called multiple times in the $\lambda_{\texttt{ANF}}$ pipeline.

4.3.2 Inlining[5]

The inlining transformation will inline known calls to non-recursive functions. It is different from shrink-inlining in that it can inline functions that might be called multiple times in the input program. The choice of which functions to inline (inline heuristic) is a parameter to the transformation and its correctness is orthogonal to this choice. Currently, inlining will inline functions that are marked for inlining by other transformations (uncurrying and lambda lifting) and functions that are sufficiently small (to avoid code blowup). The inlining transformation will alpha-rename the binders of the inlined code with fresh names to ensure that the resulting term is well scoped.

There are two cases of functions calls to inline: tail-calls and let-bound calls. The former is rather straightforward while the latter runs into issues related to the fact that the ANF representation is not closed under beta reduction. I review both cases.

Tail-call inlining. Inlining a tail-call amounts to replacing the call with the body of the function where the formal parameters are substituted with the actual parameters.

[4] This pattern commonly arises after the closure conversion transformation.

[5] CertiCoq's inlining transformation was implemented and proved correct by me. The implementation was based on an earlier implementation by Savary Bélanger.

This is captured by the following rewrite rule (not showing the alpha-conversion that is needed after inlining).

$$\begin{array}{c} \texttt{fun}\ f\ \overline{x} = e\ \texttt{in}\ \mathcal{E}[f\ \overline{y}] \\ \rightsquigarrow \\ \texttt{fun}\ f\ \overline{x} = e\ \texttt{in}\ \mathcal{E}[e\{\overline{y}/\overline{x}\}] \quad \text{if } \overline{y} \notin \mathsf{bv}(e) \text{ and } \mathsf{fv}(e) \cap \mathsf{bv}(\mathcal{E}) = \emptyset \end{array}$$

Notice that if a function definition becomes dead after some inlinings it will be removed by subsequent passes of shrinking. The side conditions in the rewrite rule to avoid variable capture are always satisfied when the input term is well scoped.

Let-bound call inlining. Inlining of let-bound calls is more complicated. Ideally one would like to replace a known call $\texttt{fun}\ f\ \overline{x} = e_1\ \texttt{in}\ \mathcal{E}[\texttt{let}\ x = f\ \overline{y}\ \texttt{in}\ e_2]$ with $\texttt{fun}\ f\ \overline{x} = e_1\ \texttt{in}\ \mathcal{E}[\texttt{let}\ x = e_1\{\overline{y}/\overline{x}\}\ \texttt{in}\ e_2]$, however this term is not in ANF. The solution is to renormalize in-place the term $\texttt{let}\ x = e_1\{\overline{y}/\overline{x}\}$, so that it is in ANF. The situation is more complicated when e_1 involves case analysis at the outermost level of the function body. The current implementation for inlining does not handle inlining of let-bound function calls of functions that perform case analysis. This is not a fundamental limitation: I outline how our current approach can be extended to this case, as well as the reasons why this is not currently handled.

To renormalize a let-bound function application after inlining I use the helper function $\mathsf{inline\_letapp}(\cdot, \cdot)$. The function receives two arguments: the body to be inlined and an identifier, which is the original name of the let binding. It returns (optionally) a binding context and an identifier. Intuitively, after evaluating the binding context, the value of the function application will be bound to the returned identifier. The function is shown in Figure 4.2.

The cases for constructors, projections, function definitions, and let-applications are straightforward. If the recursive call succeeds then the let-binding is appended at the beginning of the context returned by the recursive call, and the returned identifier remains the same. If the recursive call fails, so does the current call. The case of case analysis is also straightforward: it always fails. More subtle are the cases of tail-call and return. If the expression to be inlined ends with a tail call then it will be converted to a let-bound call using as binder the identifier provided as input. Therefore, the result of the inlined function call will be bound at this identifier when the return context is evaluated. When the input expression is a return, then the result of the inlining function is the empty evaluation context and the returned variable, since this is the result of the function call. Conceptually, the variable that is returned by $\mathsf{inline\_letapp}(\cdot, \cdot)$ is the input variable if the expression to be inlined ends with a tail call or the variable that is being returned, if the expression ends with a return.

I can now express the rewrite rule using the $\mathsf{inline\_letapp}(\cdot, \cdot)$ function.

$$\begin{array}{cr} \texttt{fun}\ f\ \overline{x} = e_1\ \texttt{in}\ \mathcal{E}_1[\texttt{let}\ z = f\ \overline{y}\ \texttt{in}\ e_2] & \text{if } \mathsf{inline\_letapp}(e_1\{\overline{y}/\overline{x}\}, z) = \mathtt{Some}(\mathcal{E}_2, z') \\ \rightsquigarrow & \text{and } \overline{y} \notin \mathsf{bv}(e_1) \text{ and } \mathsf{fv}(e_1) \cap \mathsf{bv}(\mathcal{E}_1) = \emptyset \\ \texttt{fun}\ f\ \overline{x} = e_1\ \texttt{in}\ \mathcal{E}_1[\mathcal{E}_2[e_2\{z'/z\}]] & \text{and } z' \notin \mathsf{bv}(e_2) \text{ and } \mathsf{fv}(e_2) \setminus \{z\} \cap \mathsf{bv}(e_2) = \emptyset \end{array}$$

$$\begin{aligned}
\mathsf{inline\_letapp}(\mathtt{let}\ x = \mathtt{C}(\overline{y})\ \mathtt{in}\ e, z) &\overset{\text{def}}{=} \begin{cases} \mathtt{Some}(\mathtt{let}\ x = \mathtt{C}(\overline{y})\ \mathtt{in}\ \mathcal{E}, z') \\ \text{if } \mathsf{inline\_letapp}(e, z) = \mathtt{Some}(\mathcal{E}, z') \\ \mathtt{None} \qquad\qquad\qquad\qquad \text{otherwise.} \end{cases} \\
\mathsf{inline\_letapp}(\mathtt{let}\ x = y.i\ \mathtt{in}\ e, z) &\overset{\text{def}}{=} \begin{cases} \mathtt{Some}(\mathtt{let}\ x = y.i\ \mathtt{in}\ \mathcal{E}, z') \\ \text{if } \mathsf{inline\_letapp}(e, z) = \mathtt{Some}(\mathcal{E}, z') \\ \mathtt{None} \qquad\qquad\qquad\qquad \text{otherwise.} \end{cases} \\
\mathsf{inline\_letapp}(\mathtt{case}\ y\ \mathtt{of}\ [\mathtt{C}_i\ \rightarrow\ e_i]_{i \in I}, z) &\overset{\text{def}}{=} \mathtt{None} \\
\mathsf{inline\_letapp}(\mathtt{fun}\ f\ \overline{x} = e_1\ \mathtt{in}\ e_2, z) &\overset{\text{def}}{=} \begin{cases} \mathtt{Some}(\mathtt{fun}\ f\ \overline{x} = e_1\ \mathtt{in}\ \mathcal{E}, z') \\ \text{if } \mathsf{inline\_letapp}(e_2, z) = \mathtt{Some}(\mathcal{E}, z') \\ \mathtt{None} \qquad\qquad\qquad\qquad \text{otherwise.} \end{cases} \\
\mathsf{inline\_letapp}(\mathtt{let}\ x = f\ \overline{y}\ \mathtt{in}\ e, z) &\overset{\text{def}}{=} \begin{cases} \mathtt{Some}(\mathtt{let}\ x = f\ \overline{y}\ \mathtt{in}\ \mathcal{E}, z') \\ \text{if } \mathsf{inline\_letapp}(e, z) = \mathtt{Some}(\mathcal{E}, z') \\ \mathtt{None} \qquad\qquad\qquad\qquad \text{otherwise.} \end{cases} \\
\mathsf{inline\_letapp}(f\ \overline{x}, z) &\overset{\text{def}}{=} \mathtt{Some}(\mathtt{let}\ z = f\ \overline{x}\ \mathtt{in}\ \mathcal{E}, z) \\
\mathsf{inline\_letapp}(\mathtt{ret}(x), z) &\overset{\text{def}}{=} ([\cdot], x)
\end{aligned}$$

Figure 4.2: Inlining of let-bound calls.

The let-bound call will be replaced by the context returned from $\mathsf{inline\_letapp}(\cdot,\cdot)$ after the formal parameters are substituted with the actual parameters. In the rest of the expression, the variable z that previously bound the result of the call will be substituted by z'. As before, the side conditions are necessary to avoid erroneously capturing variables, and will always hold when the expression is well scoped.

It is worth noting that the substitutions shown in the rules above do not happen as a separate substitution pass, but in the same traversal as the inlinings.

Although inlining of expressions that include case analysis is not currently handled I outline how that could be possible. Handling inlining of case constructs involves introducing a *join point*: a local continuation that captures the continuation of the inlined call [92]. Inlining the call to f in $\mathtt{fun}\ f\ \overline{x} = e_1\ \mathtt{in}\ \mathcal{E}_1[\mathtt{let}\ z = f\ \overline{y}\ \mathtt{in}\ e_2]$ would convert the program to $\mathtt{fun}\ f\ \overline{x} = e_1\ \mathtt{in}\ \mathcal{E}_1[\mathtt{fun}\ k\ z = e_2\ \mathtt{in}\ \mathsf{cont\_app}(e_1\{\overline{y}/\overline{x}\}, k)]$ The function $\mathsf{cont\_app}(\cdot,\cdot)$ (Figure 4.3) traverses the term (first argument) and applies the continuation (second argument) to the result of the computation. Note that if the input term has no case analysis, then the continuation will be used only once and therefore will be inlined by the shrinking transformation. The result of that would be exactly the inlining strategy achieved by $\mathsf{inline\_letapp}(\cdot,\cdot)$.

It is arguable what the efficiency gain of introducing a join point is. Indeed, we are able to inline strictly more functions with this strategy, but we also introduce a new call to the continuation k. First, observe that both calls to f and k are known, so it is is likely that their free variables can be passed as parameters and no closures are used. If none of the functions requires a closure, the two calls have approximately the same efficiency (passing parameters in registers is a cheap operation so we can assume that the performance difference that arises from different number of arguments is negligible). It is also possible that one or both functions need a closure. If both functions need a closure, then inlining with join point will be faster if k has fewer free variables than f. If only k needs a closure, then it is clearly better to avoid inlining. If only f needs a closure, then inlining will be the most sensible choice. Clearly, an educated choice of whether f should be inlined using a join point requires knowing whether f or k require a closure. We did not implement this heuristic decision to keep the implementation simple.

$$
\begin{aligned}
\mathsf{cont\_app}(\mathtt{let}\ x = \mathtt{C}(\overline{y})\ \mathtt{in}\ e, k) &\overset{\text{def}}{=} \mathtt{let}\ x = \mathtt{C}(\overline{y})\ \mathtt{in}\ \mathsf{cont\_app}(e, k)\\
\mathsf{cont\_app}(\mathtt{let}\ x = y.i\ \mathtt{in}\ e, k) &\overset{\text{def}}{=} \mathtt{let}\ x = y.i\ \mathtt{in}\ \mathsf{cont\_app}(e, k)\\
\mathsf{cont\_app}(\mathtt{case}\ y\ \mathtt{of}\ [\mathtt{C}_i\ \rightarrow\ e_i]_{i \in I}, k) &\overset{\text{def}}{=} \mathtt{case}\ y\ \mathtt{of}\ \{\mathtt{C}_i\ \rightarrow\ \mathsf{cont\_app}(e_i, k)\}_{i \in I}\\
\mathsf{cont\_app}(\mathtt{fun}\ f\ \overline{x} = e_1\ \mathtt{in}\ e_2, z) &\overset{\text{def}}{=} \mathtt{fun}\ f\ \overline{x} = e_1\ \mathtt{in}\ \mathsf{cont\_app}(e_2, z)\\
\mathsf{cont\_app}(\mathtt{let}\ x = f\ \overline{y}\ \mathtt{in}\ e, k) &\overset{\text{def}}{=} \mathtt{let}\ x = f\ \overline{y}\ \mathtt{in}\ \mathsf{cont\_app}(e, k)\\
\mathsf{cont\_app}(f\ \overline{x}, k) &\overset{\text{def}}{=} \mathtt{let}\ z = f\ \overline{y}\ \mathtt{in}\ k\ z \quad \text{where } z \text{ is fresh}\\
\mathsf{cont\_app}(\mathtt{ret}(x), k) &\overset{\text{def}}{=} k\ x
\end{aligned}
$$

Figure 4.3: Inlining using a join point.

4.3.3 Uncurrying[6]

Uncurrying is responsible for transforming calls to known curried functions to calls to multi-argument functions. By default, all functions in the input of $\lambda_{\mathtt{ANF}}$ are either unary (in direct style programs) or have two arguments (in continuation-passing style programs). This produces inefficient function calls since applying a multi-argument function involves a series of successive function applications. Even worse, each of the intermediate function-returns allocates a closure.

The uncurrying transformation recognizes curried function definitions and uncurries them, one argument at a time. Uncurrying works by creating an uncurried function definition where all arguments are supplied at once. Then, it declares copies of the function that take one argument at a time as wrappers around the uncurried version. The call sites are not changed by the uncurrying transformation: all uses of the function refer to the wrapper (that has the original name of the function). Function inlining, that happens after uncurrying in the pipeline, will inline wrappers at known call sites, forcing the uncurried function to be used.

As a rewrite rule this looks like,

[6] CertiCoq's uncurrying transformation was first implemented by Greg Morrisett and was later adapted by me and John Li. John Li proved the transformation correct.

$$
\begin{array}{llll}
\mathtt{fun}\ f\ \overline{x} = & & \mathtt{fun}\ f'\ \overline{x} \mathbin{+\!\!+} \overline{y} = e_1 & \\
\quad \mathtt{fun}\ g\ \overline{y} = e_1\ \mathtt{in}\ \mathtt{ret}(g) & \rightsquigarrow & \mathtt{fun}\ f\ \overline{x} = & \text{if } g \notin \mathsf{fv}(e_1) \\
\mathtt{in}\ e_2 & & \quad \mathtt{fun}\ g\ \overline{y} = f'\ \overline{x} \mathbin{+\!\!+} \overline{y}\ \mathtt{in}\ \mathtt{ret}(g) & \\
 & & \mathtt{in}\ e_2 &
\end{array}
$$

The side condition is needed to ensure that the inner function g is not recursive, in which case the rewrite would not be sound since when the uncurried function is introduced g is no longer in scope. This uncurry pattern covers direct style programs. The pattern for CPS programs is slightly different and is shown below.

$$
\begin{array}{llll}
\mathtt{fun}\ f\ k :: \overline{x} = & & \mathtt{fun}\ f'\ \overline{x} \mathbin{+\!\!+} \overline{y} = e_1 & \\
\quad \mathtt{fun}\ g\ \overline{y} = e_1\ \mathtt{in}\ k\ g & \rightsquigarrow & \mathtt{fun}\ f\ k :: \overline{x} = & \text{if } g \notin \mathsf{fv}(e_1) \\
\mathtt{in}\ e_2 & & \quad \mathtt{fun}\ g\ \overline{y} = f'\ \overline{x} \mathbin{+\!\!+} \overline{y}\ \mathtt{in}\ k\ g & \text{and } k \notin \mathsf{fv}(e_1) \\
 & & \mathtt{in}\ e_2 &
\end{array}
$$

The difference is that in CPS f does not return g but it applies it to its continuation k. The continuation of f, which will not occur in the body e_1, is not needed as an argument in f'.

The uncurrying transformation of CertiCoq will uncurry all functions in one pass, by making a recursive call to $\mathtt{fun}\ f'\ \overline{x} \mathbin{+\!\!+} \overline{y} = e_1$. This one-pass implementation is due to John Li.

Uncurrying of escaping functions. CertiCoq does not currently handle uncurrying of escaping functions. It is not generally possible to change how parameters are passed if the function is not known as different functions might flow into the same position. Some more restricted form of uncurrying can be achieved with flow analysis. If the set of functions that flow into the same escaping position can all be uncurrried in the same way, then the call sites of the unknown function can be safely modified (if they are all known) and the uncurried functions can be used at the escaping positions instead. As an example consider the higher-order function `fold` whose first argument is a function that takes two arguments. Since this is an unknown function it will not be uncurried and in the body of `fold` it will be first applied to its first argument, returning a function that will be then applied to the second argument. If the compiler can statically prove that all of the functions that flow to the first argument of fold are curried functions, then the uncurried instances can be used as the first parameter to fold and fold can change the application of its first argument to multi-argument application.

Excessive use of unknown curried functions applications can also be mitigated by argument specialization. Argument specialization can remove higher-order arguments that are always instantiated with the same actual parameter by replacing them with the actual parameter.

Uncurrying of higher-order functions is possible as a typed-directed program transformation, where the type information can be used to determine the arity of the function [59, 43].

CakeML achieves uncurrying [106] of higher-order functions by allowing partial application in the semantics of the language.

4.3.4 Closure Conversion[7]

Closure conversion compiles nested functions with free variables into closed functions that can be moved to the top-level of the program. In Section 4.2, I explained different closure strategies that are used by CertiCoq. Despite that, closure conversion will handle all functions uniformly. Closure environments will be installed for all functions and all functions will form closure pairs together with their environment. All function calls will be handled as calls to closure-converted functions. Lambda lifting (Section 4.3.5) in combination with shrinking and dead parameter elimination will take care of implementing the specialized closure strategies. The reason for this design is to keep the implementation, specification, and correctness proof of closure conversion as simple and modular as possible.

CertiCoq's closure conversion is a $\lambda_{\texttt{ANF}}$ transformation. It is common for compilers to implement closure conversion as a cross-language transformation or as part of the code generator. In CertiCoq, having the closure conversion be part of the transformational $\lambda_{\texttt{ANF}}$ pipeline allows us to keep the closure-conversion transformation simple and modular (compared with other compilers, in which lambda lifting and hoisting are often performed as a monolithic pass). We can also run the (same, proved correct) inliner after closure conversion, where more opportunities for inlining may appear: functions are no longer nested and they are all defined at the top-level, making it easier to inline functions that had contained nested functions before closure conversion.

I show (rather informally) how closure conversion works using rewrite steps. Upon function definition, closure conversion will install an environment parameter to every function, and free variables of the function will be replaced with accesses to the environment. After the definition of the function the closure environment will be constructed (using a unique constructor tag[8] for each closure, represented as $\texttt{C}_f()$ below) that has the free variables of the function. A closure pair will be then formed whose first component is the function pointer and its second component is the closure environment. The constructor tag for the closure pair is the same for every closure (represented as $\texttt{C}_{\texttt{CC}}()$ below). To lower register pressure (*i.e.*, the number of simultaneously live local variables), the closure pair will not be formed right after the definition of the function but before its first use. A subtle point is that references to known closed functions shall never be considered free variables of another function, even if they appear to be so syntactically. In the final program, such functions will be top-level global definitions and should not end up in closure environments. Every function definition will undergo the following transformation.

[7] The implementation and semantic preservation proof of CertiCoq's closure conversion was done by me. John Li proved that the closure conversion program is sound with respect to the specification of closure conversion as an inductive relation.

[8] A closure is a heap-allocated pair of function-pointer and environment pointer, but in CertiCoq's representations of tuples any "pair" type is really an inductive data type with one 2-argument constructor; it is the tag of this constructor that is mentioned here.

$$\texttt{fun}\ f\ \overline{x} = e_1\ \texttt{in}\ e_2 \quad \rightsquigarrow \quad \begin{array}{l} \texttt{fun}\ f\ \gamma :: \overline{x} = e_1'\ \texttt{in} \\ \texttt{let}\ \Gamma = \texttt{C}_f(\overline{fv})\ \texttt{in} \\ \texttt{let}\ f_{clo} = \texttt{C}_{\texttt{CC}}(f, \Gamma)\ \texttt{in}\ e_2' \end{array}$$

In the above, $\overline{fv}$ is a list with the free variables of f without the references to known closed functions, e_1' is the closure-converted program e_1 with references to free variables replaced with accesses to the environment, and e_2' is the rest of the closure-converted program where references to f are replaced with references to the closure f_{clo} and call sites are modified as described by the following transformation.

$$f_{clo}\ \overline{x} \quad \rightsquigarrow \quad \begin{array}{l} \texttt{let}\ f_{code} = f_{clo}.1\ \texttt{in} \\ \texttt{let}\ f_{env} = f_{clo}.2\ \texttt{in} \\ f_{code}\ f_{env} :: \overline{x} \end{array}$$

After closure conversion a separate hoisting pass will move all of the function definitions to the top-level of the program. After this pass there are no nested functions in the program and there are only two scope levels: the global scope and a local scope for each function definition.

4.3.5 Lambda Lifting[9]

Lambda lifting [67] is a well-known program transformation that passes free variables of functions as extra arguments. In CertiCoq, lambda lifting happens before closure conversion and its functionality is crucial for the implementation of the closure strategies at the target program. When a function is lambda lifted, two copies are created: one that is used in known application positions and one that is used in escaping positions. Furthermore, lambda lifting will turn free variables of the known copy of the function to parameters. The escaping copy, which has exactly the same free variables as the original function, is defined as a wrapper around the known function that passes the free variables as parameters. Known calls to the original function are inlined to call the known function copy and pass its free variables as parameters.

This is summarized in the following simple rewrite step (recall that inlining will be performed in a separate pass).

$$\texttt{fun}\ f\ \overline{x} = e_1\ \texttt{in}\ e_2 \qquad \rightsquigarrow \qquad \begin{array}{l} \texttt{fun}\ f'\ \overline{fv} + \!\!\!+\ \overline{x} = \\ \quad \texttt{fun}\ f\ \overline{x} = f'\ \overline{fv} + \!\!\!+\ \overline{x}\ \texttt{in}\ e_1 \\ \texttt{fun}\ f\ \overline{x} = f'\ \overline{fv} + \!\!\!+\ \overline{x}\ \texttt{in}\ e_2 \end{array}$$

where $\overline{fv} \subseteq \mathsf{fv}(\texttt{fun}\ f\ \overline{x} = e_1)$

The function f' is the known copy of the function and $\overline{fv}$ is a subset of the free variables of the function that is chosen to be passed as parameters. After the function definition, a copy of the function f is declared that calls f' and passes the free variables as parameters. Note that now $\overline{fv}$ are free variables of f but not of f'. A copy is also

[9] The implementation and verification of CertiCoq's lambda lifting was done by me.

declared upon entering the body f' as the function might be recursive. Known calls to f can now be inlined to call f' instead.

In the rule above, $\overline{fv}$ is only required to be a subset of the free variables of a function. In practice, our lambda lifter will lambda-lift a function only if *all* of its free variables are eligible to become parameters (as in closure conversion a reference to a known closed function is not considered a free variable — this includes functions that might become closed during lambda lifting.). After lambda lifting, both copies of the function will be closure-converted. But since f' is closed its closure and closure environment will be eliminated by subsequent calls to shrinking and dead parameter elimination. The wrapper f will be closure converted and will be used only if the function escapes, otherwise it will be deleted by shrinking.

An important aspect of the transformation involves calls to recursive functions from unknown call sites. In this case, the entry to the function will happen through the closure-converted wrapper f. After entry, free variables will be projected from the environment and will be passed as parameters to the function f', and will not be projected out of the environment for the rest of the recursion. If the original closure-converted function was called instead, then free variables would be projected out of the environment at each iteration.

The above transformation is functionally correct whatever the choice of free variables and known function calls to be inlined is. However, several design decisions can be made about what function to lambda lift and which known calls to inline, which can affect the performance of the lambda lifting transformation. For best performance, not every free variable should be passed as parameter and not every known call should be inlined to call f'. These choices are crucial for the performance of the generated code but the heuristic or algorithm that makes these choices does not need to be proved correct. The design choices we investigate are summarized bellow.

Passing free variables as parameters. Generally, it might not be desirable to turn a free variable into a parameter. We observe that free variables that remain live across intermediate function calls in the body of the function in which they are free are more expensive to turn into parameters. This because the values of registers (where parameters are typically stored) that remain live across calls must be saved by the caller by pushing them into the stack before the call and popping them after the call returns.[10] I experimentally evaluate this performance trade-off in Chapter 8.

In addition, we require that the total number of parameters of a function after lambda lifting does not exceed the available registers in the machine.

Inlining of calls to lambda-lifting wrappers. If a function is called in the same scope of definition in which it was defined, then the call can be safely replaced with the call to f' since all of its free variables are in scope. The situation is more tricky when a known call to f happens from within a function g that is defined after f. If it happens that the free variables that are parameters of f' are a subset of the free variables of g then again f' can be safely called. If this is not true, then inlining

[10] This analysis assumes that no callee-save registers are used.

f inside g will increase the free variables of g. If these extra free variables cannot become parameters of g then the closure environment of g will grow (or, even worse, a previously closed function will now become a closure). In the current implementation we explore two possibilities: we can conservatively decide not to inline calls that increase the free variables of a function to avoid increase in closure allocation, or we can aggressively decide to inline all known calls. I experimentally evaluate these two approaches in Chapter 8.

Inlining known calls inside wrappers. The wrapper functions, which are used in escaping positions, will immediately call the known function. One could consider inlining the body of known functions inside the wrappers when the body of the original function is small enough (OCaml's Flambda takes a similar approach). These calls are tails calls to known functions, and can be implemented very efficiently as jumps. We chose not to inline these calls. Our experimental results suggest no improvement in performance when this inlining is performed.

4.3.6 Dead Parameter Elimination[11]

Dead parameter elimination is needed to eliminate empty closure environments of known closed functions that are by default installed by closure conversion. Recall that closure conversion installs environments to nonescaping closed functions too as it makes no distinction between known and escaping functions. Such environments will never be accessed by the code of the function; they will however be passed as arguments to recursive calls of the function. Arguments that are passed as parameters to (mutually) recursive functions – but never otherwise used – are dead and can be deleted. Such arguments are free (syntactically occur) in function bodies, but only to be passed as useless parameters. Our transformation performs liveness analysis to find which parameters are used by each function. After the set of live parameters has been computed for each function, dead parameters of known functions can be dropped and the corresponding call sites should be modified to only pass the live parameters.

For each function, the set of live parameters is the smallest set that satisfies the following.

1. If a parameter is used as a constructor argument or in a projection or as a scrutinee in case analysis or in function application position, then it is live.

2. If a parameter is used as an argument in a known function call, it is live if the corresponding formal parameter is live.

3. If a parameter is used as an argument in a unknown function call, it is live.

This set can be computed with the following algorithm. A state is kept that labels each parameter of every function as live or dead.

[11] The implementation of CertiCoq's dead parameter elimination of CertiCoq was done by Katja Vassilev and me. The verification of dead parameter elimination was done my me.

- Initialize the state by marking parameters of known functions as dead and parameter of escaping function as live.
- At each iteration, traverse the program and mark parameters that are live according to rules 1-3 above. Use the current state to determine if a parameter that is passed as an argument to another function is live.
- Stop when a fixed point is reached.

The program is traversed once for each iteration. At most, there will be as many iterations as the total number of function parameters since at least one parameter should be marked as live at each iteration. Therefore, the worst-case complexity of liveness analysis is $O(n^2)$, where n is the size of the program. Since the CertiCoq algorithm is implemented using only purely functional data structures, its worst-case complexity will be $O(n^2 \log n)$. This algorithm can be easily extended to full dead code elimination, that removes other dead bindings as well as dead parameters.

4.4 Compilation by Example

In this section, I provide a bigger example that elucidates how the $\lambda_{\texttt{ANF}}$ pipeline achieves the closure strategies described in Section 4.2. In the example presented here I will deviate slightly from the syntax of $\lambda_{\texttt{ANF}}$ by allowing nested constructor applications in order to keep the code shorter and more readable. I will also use the standard list notation (`::` for cons and `[]` for nil).

Consider the following program where variables `y` and `z` are bound in the local scope but are free in function `interleave`.

```
fun interleave l =
  case l of
  | [] => []
  | x :: l => x :: y :: z :: (interleave l)
 end
in

let l1 = [1, 2 , 3] in
let l = [l1, l1, l1] in

let l1' = interleave l1 in
let l' = map interleave l in
```

The recursive function `interleave` receives a list as input and inserts `y` and `z` after each element of the list. This function is later called on list `l1` and also on each element of the list `l` (which is a list of lists) with the use of the higher-order function `map`. For simplicity, assume that `map` is defined earlier in the program and it is not being compiled separately.

Let us also assume that the program is already uncurried and that the next transformation to take place is lambda lifting. Both of the free variables of `interleave` can be lambda lifted. A new function, interleave_known, is declared that takes as extra arguments the variables y and z. Then interleave is defined in terms of the new function, both inside the function body and immediately after the function definition.[12]

```
fun interleave_known l x y =
  fun interleave l = interleave_known l x y in
  case l of
   | [] => []
   | x :: l => x :: y :: z :: (interleave l)
 end
in
fun interleave l = interleave_known l x y in

let l1 = [1, 2, 3] in
let l = [l1, l1, l1] in

let l1' = interleave l1 in
let l' = map interleave l in
```

The two known calls to interleave can be inlined so that interleave_known is called. The definition of interleave inside the body of the function will be deleted by the shrinking transformation since it is not used anymore. The call to map, assuming that map is defined earlier in the program and "splitted" in the same way, will also be replaced with map_known. It is a closed function, so no extra parameters are passed. This leaves us with the following code.

```
fun interleave_known l x y =
  case l of
   | [] => []
   | x :: l => x :: y :: z :: (interleave_known l x y)
 end
in
fun interleave l = interleave_known l x y in

let l1 = [1, 2 , 3] in
let l = [l1, l1, l1] in

let l1' = interleave_known l1 x y in
let l' = map_known interleave l in
```

Next closure conversion will run. It will closure-convert each function, regardless of their free variables and their status as escaping or known. I use record notation for the closure and environment records. The comments in the code indicate where closures are created and applied.

[12] In the actual implementation, the local variables x and y of interleave_known would have to be renamed to avoid violating the "global unique bindings" property of our intermediate representation.

```
fun interleave_known l x y env =
  case l of
  | [] => []
  | x :: l => x :: y :: z :: (interleave_known l x y env)
 end
in
(* closure of interleave_known is created *)
let interleave_known_env = { } in
let interleave_known_clo = { interleave_known , interleave_known_env }

fun interleave l env =
  let x = env.1 in
  let y = env.2 in

  (* application of interleave_known *)
  let interleave_known = interleave_known_clo.1 in
  let env' = interleave_known_clo.2 in
  interleave_known l x y env'
in
(* closure of interleave is created *)
let interleave_env = { x , y } in
let interleave_clo = { interleave , interleave_env }

let l1 = [1, 2 , 3] in
let l = [l1, l1, l1] in

(* application of interleave_known *)
let interleave_known = interleave_known_clo.1 in
let env = interleave_known_clo.2 in
let l1' = interleave_known l1 x y env in

(* application of map *)
let map_known = map_known_clo.1 in
let env = map_known_clo.2 in
let l' = map_known interleave_clo l env in
```

In the above code all functions are closure-converted and all applications are converted so that they project the code and the environment from the closure pair. However the closures of `interleave_known` and `map_known` are statically known, so the projections of code and environment will be folded by shrinking. After that `interleave_known_clo` is dead code so it will be removed.

```
fun interleave_known l x y env =
  case l of
  | [] => []
  | x :: l => x :: y :: z :: (interleave_known l x y env)
  end
in
let interleave_known_env = { } in

fun interleave l env =
  let x = env.1 in
  let y = env.2 in

  (* application of interleave_known *)
  interleave_known l x y interleave_known_env
in
(* closure of interleave is created *)
let interleave_env = { x , y } in
let interleave_clo = { interleave ,  interleave_env }

let l1 = [1, 2 , 3] in
let l = [l1, l1, l1] in

let l1' = interleave_known l1 x y interleave_known_env in

(* application of map *)
let l' = map_known interleave_clo l map_env in
```

Now only the escaping function `interleave` has a closure. However, the functions `interleave_known` and `map_known` still have an empty environment parameter, which is never accessed but only passed as an argument at every call to these functions. Dead parameter elimination will remove these. Then `interleave_known_env` defined right after `interleave_known` will not be used anymore and will be removed by the shrinking pass. Here's the final program.

```
fun interleave_known l x y =
  case l of
  | [] ⇒ []
  | x :: l ⇒ x :: y :: z :: (interleave_known l x y)
  end
in

fun interleave l env =
  let x = env.1 in
  let y = env.2 in

  (* application of interleave_known *)
  interleave_known l x y
in
(* closure of interleave is created *)
let interleave_env = { x , y } in
let interleave_clo = { interleave ,  interleave_env }

let l1 = [1, 2 , 3] in
let l = [l1, l1, l1] in

let l1' = interleave_known l1 x y in

(* application of map *)
let l' = map_known interleave_clo l in
```

In the above program, the function `interleave_known` has no closure, it just receives two extra arguments with the values of its free variables. All known calls to `interleave` now call `interleave_known`. The function `interleave` gets closure converted, and its closure is passed as the first argument of `map`. The function will be invoked from within the body of map, and after projecting `x` and `y` from the environment, it will call `interleave_known`.

The above example illustrates how calls to known functions can be compiled more efficiently than calls to escaping functions, by eliminating closures of nonescaping functions. It also illustrates how this is achieved using a naive closure conversion algorithm, with the use of a separate lambda lifting pass, shrinking and dead parameter elimination.

4.5 Related Work

In this section, I compare the optimizations performed by the $\lambda_{\texttt{ANF}}$ pipeline of CertiCoq with those of other verified compilers for functional languages. The verified compilers I consider are the PILSNER [104], Œuf[102], CakeML [106] and Lambda Tamer [34]. I also compare CertiCoq's optimizing pipeline with similar pipelines in GHC and the OCaml compiler.

4.5.1 Optimizations in Other Verified Compilers

PILSNER [104] is a compositionally verified compiler for an ML-like source language to an idealized assembly language. PILSNER translates the source language to a CPS intermediate representation where it performs optimizations and then it generates target code. The optimizations that are performed by PILSNER are function inlining (of top-level functions only), dead code elimination (without liveness analysis), contification, hoisting of let-bindings out of function definitions to avoid recomputation during loops, and common subexpression elimination.

PILSNER does not introduce multiple arguments and functions remain curried. It also does not try to eliminate closures of known (user) functions. Calls will always access the closure record to fetch the code pointer. Continuations (that escape only downwards and therefore have a limited extent) are stack allocated, and free variable accesses are lookups in the stack, therefore no heap allocated closure is needed. PILSNER also performs a simple contification [49] optimization that will turn functions that are always called with the same continuation into continuations, causing them to be stack allocated.

Œuf [102] is a prototype compiler implementation from a subset of Gallina to CompCert's Cminor. As a prototype implementation, Œuf does not perform any optimizations. Multi-argument functions are always curried and all functions are closure converted. Calls to known functions always enter through the code pointer in the closure pair.

CakeML [129] is the most mature verified compiler for a functional language. It offers the most advanced optimizations for known function calls [106] among the compilers considered in this section. Compared to CertiCoq, CakeML performs fewer optimizations for closure strategies but it manages to uncurry calls to unknown functions, which CertiCoq currently does not. CakeML optimizes calls to known functions that are closed, by avoiding extracting the function pointer from the closure record. However, CakeML will always allocate closures for known functions with free variables. On the other hand CakeML, unlike CertiCoq, does optimize curried function applications to unknown calls. In CakeML, all functions, regardless of their status as known or escaping, are uncurried. Multi-argument applications of unknown functions are implemented using a *mismatch* semantics that allows partial application. A runtime check is performed to determine if the numbers of actual parameters matches the arity of the function. If a function is applied to exactly as many arguments as it expects, then its body is evaluated avoiding allocating closures for the intermediate results of the uncurried application. If a function is applied to fewer arguments than it expects, then a closure is allocated. If a function is applied to more arguments than it expects, the body is evaluated and the result is applied to the remaining arguments.

To achieve these optimizations, CakeML uses an intermediate representation, ClosLang, that is more complicated than $\lambda_{\texttt{ANF}}$. It distinguishes two types of calls, C-style calls for optimized known functions, and ML-style calls. Similarly, it has three different kinds of function definitions: local anonymous functions, local recursive functions, and a global immutable code table for closed known functions. In order to recognize known calls that can be optimized the compiler performs additional flow analysis (which CertiCoq has avoided by "splitting" functions into known and escaping instances).

Lambda Tamer [34] is a verified compiler for an imperative higher-order language. It uses a naive closure-conversion transformation that does not optimize closures of known functions.

4.5.2 Compilation-by-Transformation in other Compilers

Many modern compilers use a transformational approach for compilation. Here, I review the transformational pipelines of GHC and OCaml, that have a similar design with that of $\lambda_{\texttt{ANF}}$.

GHC. GHC's Core-to-Core optimization pipeline [69, 68] consists of many small modular passes that are used to optimize the code. The pipeline has a *simplifier*, that performs code simplifications including inlining, constant folding, eta-expansion, and case-of-case transformation. These transformations are performed simultaneously to take advantage of the cascade effect. This pass corresponds to our shrinking/inlining loop. The Core-to-Core pipeline performs a few other global transformations, including strictness analysis, argument specialization, lambda lifting and static argument transformation (*i.e.*, lambda dropping that removes arguments of functions and introduces free variables). As in our $\lambda_{\texttt{ANF}}$ pipeline, the simplifier will run between optimization passes to remove administrative redexes.

GHC's lambda lifting [69] is, just like ours, selective and will not lambda lift every function. The selection is based on whether lambda-lifting a function will increase closure allocation in one of the callers (we consider this but also other design parameters). GHC's lambda lifting will not split functions into known and unknown instances. Lambda-lifted functions will be partially applied before an escaping occurrence.[13] Because of our splitting strategy, our lambda-lifting transformation can make a more fine-grained choice about which functions to lambda-lift (since different calls to the same functions can call either the lambda-lifted or the original function, *i.e.*, the wrapper). Lambda lifting in GHC improves the performance of some programs but worsens the performance of others. The selective approach mitigates the overhead.

Flambda. OCaml's Flambda optimization pipeline [87, Chaper 21] has a similar design. It features a pass that performs inlining and code simplification (constant folding, dead code elimination). Other transformations include argument specialization, code motion transformations, removal of unused arguments, and lambda lifting.

Flambda's lambda lifting works in a similar way to ours by splitting a function into a known and an escaping instance, which is defined as a wrapper around the known instance. The known wrapper will be inlined at known call sites. To mitigate the overhead of unknown calls going through the wrappers, if a function is small enough Flambda will duplicate it and will use the original function at escaping occurrences and the lambda-lifted function at known call sites. However, this will miss the opportunity to eliminate closures of escaping recursive functions. In our lambda lifting implementation the free variables will be projected just once out of the environment, when they are first called. Flambda's lambda lifting does not always improve performance. It is not clear whether Flambda's lambda lifting implements any heuristic decision about which functions to lambda lift or if it follows an all-or-nothing approach.

[13] We cannot do that because partial application of multi-argument functions in not supported by the semantics of $\lambda_{\texttt{ANF}}$.

4.6 Conclusion

In this chapter, I presented the $\lambda_{\texttt{ANF}}$ optimizing pipeline of CertiCoq. The pipeline performs various code simplification transformations to optimize the code. These optimizations include dead code elimination, case and projection folding, function inlining and dead parameter elimination. Most importantly, the $\lambda_{\texttt{ANF}}$ pipeline optimizes known function calls by implementing uncurrying and efficient closure strategies. The design of the $\lambda_{\texttt{ANF}}$ pipeline follows a compilation-by program-transformation approach: a number of simple and small program transformations are composed and produce an optimized target that can be readily compiled to C or other first-order, low-level representations. Optimizations are decomposed into small and simple transformations that are proved correct individually and they are then recomposed to achieve optimizations that in other compilers are expressed as monolithic passes. In the next chapter, I introduce the proof framework that is used to verify the $\lambda_{\texttt{ANF}}$ pipeline.

Chapter 5

Relational Proof Framework

To carry out a correctness proof for a program transformation, it is often convenient to set up a more general relation and show that the input and the output of the transformation inhabit the relation. First, let us examine the compiler correctness specification that we wish to establish for the $\lambda_{\texttt{ANF}}$ pipeline. A compiler is correct if the observable behaviors exhibited by the target program are included in observable behaviors of the source program. For the language in question, $\lambda_{\texttt{ANF}}$, a program has two possible observable behaviors: it may terminate yielding a result or it may diverge. We shall therefore prove that whenever the source program terminates producing a result so does the target, and two results are observationally the same, and whenever the source diverges so does the target program.[1]

Definition 5.1 (Behavioral refinement in $\boldsymbol{\lambda_{\texttt{ANF}}}$)
We say that program e' refines the behavior of program e, written[2] $e \supseteq_B e'$, *iff*

$$\begin{array}{ll} (e \Downarrow \mathsf{Res}(v) \Rightarrow \exists\, v',\ e' \Downarrow \mathsf{Res}(v')\ \wedge\ v \approx v') \quad \wedge & \text{(termination)} \\ e \Uparrow \quad \Rightarrow \quad e' \Uparrow & \text{(divergence)} \end{array}$$

The relation $\approx$ asserts that two $\lambda_{\texttt{ANF}}$ values are observationally the same if they are both constructed values with the same constructor tag and pairwise related arguments, or if they are both function values.

$$\frac{\text{if } m = n \text{ and } \texttt{C}_1 = \texttt{C}_2 \text{ and } \forall i, v_i \approx v_i'}{\texttt{C}_1(v_1, \ldots, v_m) \approx \texttt{C}_2(v_1', \ldots, v_n')}$$

$$\frac{}{\texttt{Clo}(\sigma, \texttt{fun}\ f\ \overline{x} = e) \approx \texttt{Clo}(\sigma', \texttt{fun}\ g\ \overline{y} = e')}$$

A compiler is correct if $\forall\ e,\ e \supseteq_{\texttt{B}} \texttt{comp}(e)$.

[1] This describes a *forward simulation.* This is enough to show semantic preservation for deterministic languages. In presence of nondeterminism one must show a *backward simulation.* See Leroy [86] for a relevant discussion.

[2] I will also use $(\sigma, e) \supseteq_{\texttt{B}} (\sigma', e')$ for evaluation in a non-empty environment.

In the above statement any two functions are considered observationally the same. For programs that run in isolation this is sufficient. A whole program is expected to have a first-order type, an inductive datatype whose values (represented in memory as data structures) can be traversed with appropriate knowledge of constructor representations. In contrast, in ML-like languages, function values are not considered "intensionally observable", one does not expect to examine their internals and print them out. To observe the result of a program when this is a function, the program must be linked with another program that applies this function. I will discuss linking extensively in the next section.

Notice that the refinement relation is not a symmetric relation: a program that has no evaluation derivation—*i.e.*, a program that is "stuck"—is trivially refined by any target program. This specification allows the compiler to map a stuck source program to any target program, permitting undefined behavior whenever the semantics of the source program is ill-defined.[3]

Now that we know *what* to prove, we should examine *how* to prove it. Of course, one can attempt to do this semantic-preservation proof by reasoning directly about the semantics of the source and target languages. The semantic refinement cannot be proved directly because it asserts nothing about related functions and hence, it does not provide a strong enough induction hypothesis. One would have to strengthen the specification by relating function values too. This is typically done syntactically by using the compilation function itself: two functions are related if the target function is the compilation of the source function. Such *syntactic* simulations depend on the transformation that is being proved correct. Now imagine attempting to prove the correctness of the $\lambda_{\texttt{ANF}}$ pipeline using syntactic simulations. One would have to use seven different syntactic value relations! Changes in the transformation would result in changes in the corresponding syntactic value relation. Modifications in the semantics would incur modifications in all seven proofs.

The ad-hoc nature of reasoning with syntactic simulations can be avoided by setting up a relation between the source and target languages that relates values *semantically* and provides a more principled way of reasoning. Such relations act as a proxy between the semantics of the languages and the correctness proof. They come with a proof theory, often in form of *compatibility* lemmas or equational rules, that enable compositional reasoning about program refinement. Modifying the semantics requires adjusting the theory of the relation, but generally not the correctness proofs of transformations. In addition, semantic relations can be used for proving the correctness of more than one transformation, facilitating proof reuse. Examples of such relations in the literature include logical relations [16, 3], bisimulations [111, 63], and more advanced relations that combine best of both worlds [65, 104]. For the correctness of the $\lambda_{\texttt{ANF}}$ pipeline we chose the method of logical relations. But before getting into the details of the technical framework, let us first discuss some desirable properties of a relation that is used for compiler correctness.

[3] By virtue of type safety, we known that the Gallina programs we are compiling are safe and semantically well-defined. Restricting the compiler specification to source programs with well-defined semantics is standard in compiler correctness theorems [86].

5.1 Relations for Compiler Correctness

What is a good relation for compiler correctness? At the very least, a relation for compiler correctness should imply the behavioral refinement that we interested in establishing, which in our case is definition 5.1. A relation that has this property is said to be *adequate*. If we are only interested in compilation of whole programs adequacy is enough. We can establish behavioral refinement for each transformation separately, using the relation of choice, and then compose the individual refinement proofs[4] to derive the top-level correctness theorem.

However, the situation gets more complicated when we want to reason about semantic preservation for programs that are compiled separately and linked at the target level. To have a framework to talk about separate compilation and linking it is useful to formalize a notion of linking $\lambda_{\texttt{ANF}}$ programs.

Definition 5.2 (Linking $\boldsymbol{\lambda}_{\texttt{ANF}}$ programs)
Let e_{client} be a $\lambda_{\texttt{ANF}}$ program with exactly one external reference x and e_{lib} be a closed $\lambda_{\texttt{ANF}}$ program. The linking operator substitutes the reference x with the result of evaluating the expression e_{lib}.

$$[x \mapsto e_{\text{lib}}]e_{\text{client}} \overset{\text{def}}{=} \texttt{fun}\ f\ [] = e_{\text{lib}}\ \texttt{in}\ \texttt{let}\ x = f\ []\ \texttt{in}\ e_{\text{client}}$$

The linking operator can be generalized to more than one external reference.

Intuitively, we can think of the linking operator as a closing substitution. Only that in $\lambda_{\texttt{ANF}}$ we cannot simply substitute an identifier for an expression or simply write $\texttt{let}\ x = e_{\text{lib}}\ \texttt{in}\ e_{\text{client}}$ for reasons related to the syntactic restrictions of ANF, described in Section 4.3—so we use a zero-arity function.

Note: Although in the case of $\lambda_{\texttt{ANF}}$ the source and target languages of compilation are the same, the linking specification and the following discussion generalize to a cross-language setting as well.

5.1.1 Reasoning About Linking

Verification of a compiler with respect to not only whole-program compilation but also separate compilation of program components is referred to as *compositional compiler correctness*. There are different notions of compositional compiler correctness,[5] depending on how broad the notion of linking is. A compositional correctness theorem may support:

1. Linking of programs produced by exactly the same compiler.

2. Linking of programs produced by different configurations of the same compiler. Such configuration can use different optimization passes.

[4] Program refinement is a transitive relation.

[5] For an extensive discussion of the spectrum of compositional compiler correctness the reader can look at Patterson and Ahmed [108].

3. Linking of programs produced by different compilers but are compiled from the same source language.

4. Linking of programs produced by different compilers and are written in different source languages. Such programs may or may not be expressible in the same source language.

In this chapter, I will limit the discussion to linking programs that are compiled from the same source language. The following statement captures correctness of linking for cases 1-3 above.

Definition 5.3 (Correctness of linking)
Let $\mathtt{comp}_1$ and $\mathtt{comp}_2$ be compilers between the same source and target languages. A program compiled using $\mathtt{comp}_1$ can be safely linked with a program compiled using $\mathtt{comp}_2$ if $[x \mapsto e']e \supseteq_{\mathtt{B}} [x \mapsto \mathtt{comp}_2(e')]\mathtt{comp}_1(e)$.

That is, linking at the target level is correct if it refines the behavior of a whole source-level program produced by linking two programs at the source level.

Statement 5.3 trivially holds when the two source programs are compiled with the same compiler (*i.e.*, $\mathtt{comp} = \mathtt{comp}_1 = \mathtt{comp}_2$) and the compilation function commutes with linking (*i.e.*, $\mathtt{comp}([x \mapsto e'_s]e_s) = [x \mapsto \mathtt{comp}(e'_s)]\mathtt{comp}(e_s)$). Then, from the compiler correctness statement we know that $[x \mapsto e'_s]e_s \supseteq_{\mathtt{B}} \mathtt{comp}([x \mapsto e'_s]e_s)$. Using the above equality, we derive $[x \mapsto e'_s]e_s \supseteq_{\mathtt{B}} [x \mapsto \mathtt{comp}(e'_s)]\mathtt{comp}(e_s)$ that proves the case.

For the $\lambda_{\mathtt{ANF}}$ pipeline this is not generally the case. For now let's assume that we are compiling all programs with exactly the same $\lambda_{\mathtt{ANF}}$ optimizations. Depending on what optimizations are used, we might be able to prove that compilation commutes with linking but only up to alpha-conversion, as the binders will get renamed during compilation. In this case, to establish the linking correctness theorem we would have to prove that alpha-conversion preserves behavioral refinement, which requires additional proof effort. The compiler could also choose to inline the zero-arity linking function f of Definition 5.2 or closure convert f.[6] We would then have to prove additional lemmas (*e.g.*, strengthening and weakening lemmas for the semantics) to reason about such cases. Generally speaking, changing the pipeline would require us to reestablish the linking theorem, which might be a non-trivial task.

The situation gets trickier when we attempt to link programs that are compiled with different configurations of the $\lambda_{\mathtt{ANF}}$ pipeline that use different optimization passes. This is a common scenario in compilation: separately compiled programs might have been compiled with different optimization flags, or different versions of the compiler (*e.g.*, one might add an argument specialization optimization to $\lambda_{\mathtt{ANF}}$ and link programs compiled with the new transformation with programs that were compiled before adding this pass). Then there is no general recipe for proving correctness of separate compilation. Using a suitable relation to prove behavioral refinement can enable us to derive correctness of separate compilation *compositionally*, just by showing that each

[6] Our compiler eliminates closures of known closed functions, such as f, but some other pipeline might not perform this optimization.

of the pipelines inhabits the relation. In this case, we can prove a linking theorem that is independent of the compilation function and only depends on the relation that is used to prove correctness. Correctness of separate compilation for pipelines that inhabit the relation is a corollary of the linking theorem.

To make things more formal, let $\mathcal{R} \subseteq \exp \times \exp$. We call this relation *compatible with linking* if it satisfies the following.

$$\forall e_s\ e'_s\ e_t\ e'_t,\ \mathcal{R}\ (e_s, e_t)\ \Rightarrow\ \mathcal{R}\ (e'_s, e'_t)\ \Rightarrow \mathcal{R}\ ([x \mapsto e'_s]e_s, [x \mapsto e'_t]e_t)$$

Clearly, if R is also an adequate relation, then we can derive behavioral refinement for linking for any two compilers that satisfy R . This property is also referred to as *horizontal compositionality.* If $\mathcal{R}$ is also transitive, then it can be used to prove correct each intermediate optimization and then such proofs can be composed to show that the whole compiler is in $\mathcal{R}$.

Unfortunately, compiler correctness relations for higher-order languages that are adequate, compatible with linking, and transitive are extremely rare. There is only one such type of relations in the literature, *Parametric Inter-Language Simulations (PILS)*, which are used to verify the PILSNER compiler [104]. The technical framework of PILS is quite complicated, and according to the authors, the proof of transitivity is very involved.

In this thesis, I show that if we restrict linking to programs compiled by compilers that have exactly the same sequence of intermediate representations, then a compositional linking theorem can be obtained in a very lightweight and general way. In the framework that I present in this chapter, each transformation is proved correct with respect to a relation that is adequate and compatible with linking. The pipeline then inhabits the composition of all intermediate relations that are used to prove correctness. But the top-level relation is the composition of adequate and compatible relations and as such, it is also an adequate and compatible relation that enables us to prove a compositional compiler correctness theorem.

The linking theorem of $\lambda_{\texttt{ANF}}$ is similar in strength with the one of SepCompCert [73]. However the proof technique is more general, and it allows strictly more programs to be linked with each other. Unlike SepCompCert, it requires no modification to the proof of each transformation. I make a more detailed comparison between the two techniques in the related work section of this chapter (Section 5.4).

In the rest of this chapter, I present the relational framework that is used to prove correct the $\lambda_{\texttt{ANF}}$ transformation, its extension to verification of separate compilation, briefly the correctness result of each transformation, and the end-to-end, top-level theorem of the $\lambda_{\texttt{ANF}}$ pipeline. I conclude with related work about proof techniques for compiler verification.

5.2 Logical Relations

Logical relations let us avoid syntactically relating function values by using an extensional notion of function relatedness. With logical relations, two functions are related if applying them to related arguments yields semantically related results. Logical relations have a long history in programming languages and have been used to prove a wide variety of properties about lambda calculus (and related languages). Unary logical relations, which are predicates over just one program, have been used to show properties such as strong normalization [128, 50] and type safety [16, 4]. Binary logical relations are a useful tool to show program equivalence and refinement. Reynolds famously used a logical relation to show relational parametricity [114]. In compiler correctness, logical relations are being used to show behavioral refinement. Commonly, logical relations are indexed by the types of the language, but they can be used in untyped settings as well [106, 1]. For CertiCoq, we use untyped logical relations.

Step-indexing. Typically, defining a logical relation requires defining a different relation for different syntactic category of the language—*e.g.*, an expression relation, a value relation, *etc.* Let us, just for the purpose of this example, use $\prec$ for the value relation and $\preceq$ for the expression relation. Using a simple untyped lambda calculus with substitution semantics (so that we do not have to include evaluation environments in our definitions), let us try to define the value relation for functions, which states that functions are related if they map related inputs to related outputs. The following definition captures the intuition described above.

$$\lambda x_1.e_1 \prec \lambda x_2.e_2 \overset{\text{def}}{=} \forall\ v_1\ v_2, v_1 \sim v_2 \Rightarrow e_1\{v_1/x_1\} \preceq e_2\{v_2/x_2\}$$

Yet, the above definition has a problem: it is not a well-founded definition. In the recursive call of $\sim$ nothing gets strictly smaller: the values v_1 and v_2 are arbitrary values and not sub terms of the related functions. To overcome this, we use a *step index* [16]: a natural number that gets strictly smaller in recursive calls and guarantees that the recursion is well-founded. Using a step index, the above definition becomes:

$$\lambda x_1.e_1 \prec_k \lambda x_2.e_2 \overset{\text{def}}{=} \forall\ (i < k)\ v_1\ v_2, v_1 \sim_i v_2 \Rightarrow e_1\{v_1/x_1\} \preceq_i e_2\{v_2/x_2\}$$

That is, two functions are related at step index k, if for all arguments related at some *strictly* smaller step index i the bodies of the function after substituting the formal for the actual parameter, are also related at step index i. That means that in order to apply two functions to see if they are related, we have to spend one step. Intuitively, if two expressions are related at step index k then we can establish behavioral refinement for the next k steps of computation—but we do not know what happens after that. This becomes more clear with the definition of the expression relation.

$$e_1 \preceq_k e_2 \overset{\text{def}}{=} \forall\ c_1\ v_1,\ e_1\ {}^{c_1}\!\Downarrow v_1 \Rightarrow \exists\ c_2\ v_2,\ e_2\ {}^{c_2}\!\Downarrow v_2\ \wedge\ v_1 \prec_{k-c_1} v_2$$

The expression relation states that if the source program evaluates to some value in some amount of steps c_1 that is smaller than the step index, then the target program also evaluates to a value, and the two values are related for the remaining number of steps. To show that a transformation is correct we prove that the source and target program are related for *all* step indices: $\forall\ k, e_1 \preceq_k e_2$.

5.2.1 Reasoning with the Logical Relation

To reason about logical relatedness of programs we use the so-called *compatibility lemmas* of the logical relation. These lemmas state that relatedness of configurations is preserved for various transitions of the two configurations. It enables us to establish relatedness of two congifurations by the relatedness of subconfigurations.

First, let us adapt the logical relation from the previous section to the $\lambda_{\texttt{ANF}}$ language. Since the $\lambda_{\texttt{ANF}}$ semantics is environment-based, the $\lambda_{\texttt{ANF}}$ expression relation relates configurations (*i.e.*, pairs of environments and expressions). Let us also slightly change the notation that was used in the example above and use the symbol $\boldsymbol{\mathcal{E}}$ with prefix notation for the expression relation and the symbol $\boldsymbol{\mathcal{V}}$ for the value relation (which we will define later). A logical relation adapted to $\lambda_{\texttt{ANF}}$ is shown in the following definition.

$$\begin{aligned}\boldsymbol{\mathcal{E}}^k(\sigma_1, e_1)\ (\sigma_2, e_2) \ \stackrel{\text{def}}{=}\ & \forall\ c_1\ v_1,\ (\sigma_1, e_1)\ {}^{c_1}\!\Downarrow v_1 \Rightarrow \\ & \qquad\qquad \exists\ c_2\ v_2,\ (\sigma_2, e_2)\ {}^{c_2}\!\Downarrow v_2 \wedge \\ & \qquad\qquad\qquad \boldsymbol{\mathcal{V}}^{k-c_1}(v_1, v_2)\end{aligned}$$

An example of compatibility lemma is demonstrated below. Let C_1 and C_2 be binding contexts (recall, from Section 3.2, that a binding context is a linear 1-hole context with the hole at continuation position). Then we can show that any two configurations $(\sigma_1, C_1[e_1])$ and $(\sigma_2, C_2[e_2])$ are related if the configurations (σ_1', e_1) and (σ_2', e_2) are related, where σ_1' (resp. σ_2') is the environment obtained by interpreting C_1 (resp. C_2) in the environment σ_1 (resp. σ_2). This is captured by the following compatibility lemma.

Lemma 5.4 (Context application compatibility (simplified).)
Let C_1 and C_2 be binding contexts such that

- $(\sigma_1, \mathcal{E}_1) \rhd \mathsf{Res}(\sigma_1')$,
- $(\sigma_2, \mathcal{E}_2) \rhd \mathsf{Res}(\sigma_2')$, *and*
- $\boldsymbol{\mathcal{E}}^k(\sigma_1', e_1)\ (\sigma_2', e_2)$

Then we can derive that $\boldsymbol{\mathcal{E}}^k(\sigma_1, C_1[e_1])\ (\sigma_2, C_2[e_2])$.

The above statement is slightly simplified in that it ignores the fuel and resource values in the context interpretation judgment. We will come back to that in subsequent sections. The proof of this lemma uses the composition lemmas for the evaluation and context interpretation judgments of Section 3.3.3.

We state and prove such generic rules for different language constructs that we then use to establish that transformation are in the logical relation.

5.2.2 Reasoning About Divergence

We aim to use a logical relation to reason about $\lambda_{\texttt{ANF}}$ transformations. As stated in the previous section, the logical relation prescribes what happens when the source program terminates, but leaves unspecified the behavior of the target program when the source program diverges or get stuck. Yet, the refinement we wish to prove asserts that whenever the source program diverges so does the target program. As I discussed in Section 3.3, we can prove divergence preservation for a transformation if we can prove that the number of steps that the source program performs is upper bounded by some strictly monotonic function of the number of steps of the target program (Lemma 3.10). To perform such reasoning we can extend the definition of the logical relation to assert that some relation between the source and target steps of execution holds. For example, the following logical relation asserts that the target steps is always greater or equal to the source steps.

$$\begin{aligned}\boldsymbol{\mathcal{E}}^k(\sigma_1, e_1)\ (\sigma_2, e_2) \ \stackrel{\text{def}}{=}\ \ & \forall\ c_1\ v_1,\ (\sigma_1, e_1)\ {}^{c_1}\!\Downarrow v_1 \Rightarrow \\ & \quad \exists\ c_2\ v_2,\ (\sigma_2, e_2)\ {}^{c_2}\!\Downarrow v_2 \wedge \\ & \qquad c_1 \leq c_2 \wedge \\ & \qquad \boldsymbol{\mathcal{V}}^{k-c_1}(v_1, v_2)\end{aligned}$$

For two programs that satisfy the above logical relation we can prove, using Lemma 3.10, that whenever e_1 diverges so does e_2. Unfortunately, not all transformations satisfy the simple relation $c_1 \leq c_2$ for the steps of the source c_1 and the steps of the target c_2. To be able to establish the relation for programs that whose steps are related in different ways, we parameterize the logical relation with a *relational postcondition* Q that captures an arbitrary relationship between the fuel values of the two programs. Such relational postcondition can be instantiated differently for each pair of programs that we want to relate. This is reflected in the following logical relation definition.

$$\begin{aligned}\boldsymbol{\mathcal{E}}^k(\sigma_1, e_1)\ (\sigma_2, e_2)\ \{Q\} \ \stackrel{\text{def}}{=}\ \ & \forall\ c_1\ v_1,\ (\sigma_1, e_1)\ {}^{c_1}\!\Downarrow v_1 \Rightarrow \\ & \quad \exists\ c_2\ v_2,\ (\sigma_2, e_2)\ {}^{c_2}\!\Downarrow v_2 \wedge \\ & \qquad Q\ c_1\ c_2 \wedge \\ & \qquad \boldsymbol{\mathcal{V}}^{k-c_1}(v_1, v_2)\end{aligned}$$

This is not a complete definition: the definition of value relation also needs to be parameterized by a postcondition in order to pass it to the expression relation. To make this clear, let us define the value relation for closure values. Observe that the value relation we defined for untyped lambda calculus with substitution semantics relates function values. When an environment semantics is used, we need to relate closure values (*i.e.*, pairs of code and environment) in order to assign values to the free variables of the functions.

$$
\begin{array}{l}
\boldsymbol{\mathcal{V}}^k(\texttt{Clo}(\sigma_1, \texttt{fun}\ f\ \overline{x} = e_1), \texttt{Clo}(\sigma_2, \texttt{fun}\ g\ \overline{y} = e_2))\ \{Q\} \overset{\text{def}}{=} \\
\forall\ i < k\ \overline{v_1}\ \overline{v_2}, \\
\quad \boldsymbol{\mathcal{V}}^i(\overline{v_1}, \overline{v_2})\ \{Q\}\ \Rightarrow \\
\quad \mathsf{len}(\overline{x}) = \mathsf{len}(\overline{v_1})\ \Rightarrow \\
\quad \mathsf{len}(\overline{y}) = \mathsf{len}(\overline{v_2})\ \wedge \\
\quad \boldsymbol{\mathcal{E}}^i(\sigma_1', e_1)\ (\sigma_2', e_2)\ \{Q\} \\
\quad \text{where } \sigma_1' = \sigma_1[\overline{x} \mapsto \overline{v_1}, f \mapsto \texttt{Clo}(\sigma_1, \texttt{fun}\ f\ \overline{x} = e_1)] \\
\quad \text{and } \sigma_2' = \sigma_2[\overline{y} \mapsto \overline{v_2}, g \mapsto \texttt{Clo}(\sigma_2, \texttt{fun}\ g\ \overline{y} = e_2)].
\end{array}
$$

Two closure values are related if for any two lists of pairwise related values the configurations consisting of the bodies of the functions and the closure environments extended with appropriate bindings are related. The closure environments are related with bindings that bind the formal parameters to the actual parameters and the function name to the closure value, which is needed when the function is recursive. The definition also requires that the actual parameters of the source function have the same length as its formal parameters, and ensures the same for the target function, essentially requiring that related functions have the same arity. The value relation takes a postcondition as parameter which is the postcondition at which the two function bodies are related.

Now, the value relation is also parameterized by the postcondition. This postcondition is enforced at the function bodies of related closures. Going back to the expression relation, we need to instantiate the postcondition parameter of the value relation. The most obvious choice would be to use the same Q parameter for relating both the fuels of the current configurations and future executions of results (that may contain closures). This however is too restrictive: it might be true when we are considering whole program executions, but it might not hold when we consider executions of subcomputations of the two programs, which may be related at a different postcondition. By imposing the same postcondition on the current computations and on future application of the results, we are forced to relate the current configurations with the same postcondition that holds for whole function executions. However, to achieve compositional reasoning we must be able to allow the postcondition of the current configurations to vary independently when we are considering transitions of the configurations we are trying to relate.

As an example, consider the following simple program. As in previous examples, I deviate slightly from the $\lambda_{\texttt{ANF}}$ syntax for conciseness and readability. I also use standard notation for natural numbers and lists (`::` for cons and `[]` for nil).

$$
e_{\texttt{src}} \overset{\text{def}}{=} \begin{array}{l}
\texttt{fun}\ f\ x\ l = x\texttt{::}x\texttt{::}l\ \texttt{in} \\
\texttt{let}\ h = 42\ \texttt{in} \\
\texttt{let}\ t = \texttt{[]}\ \texttt{in} \\
f\ h\ t
\end{array}
$$

Assuming that all language operations take unary cost, the program runs in 5 units of time. Now consider that we inline the function f in the above program and we obtain the program $e_{\texttt{trg}}$, in which the definition of f has been inlined and deleted since is not used anymore.

$$\begin{array}{rcl} e_{\texttt{trg}} & \overset{\text{def}}{=} & \texttt{let}\ h = 42\ \texttt{in} \\ & & \texttt{let}\ t = \texttt{[]}\ \texttt{in} \\ & & h::h::t \end{array}$$

The inlined program is more efficient and executes in just 3 units of time. We now wish to prove that for any environments $\sigma_{\texttt{src}}$ and $\sigma_{\texttt{trg}}$ we have that $\boldsymbol{\mathcal{E}}^k(\sigma_{\texttt{src}}, e_{\texttt{src}})\ (\sigma_{\texttt{trg}}, e_{\texttt{trg}})\ \{Q\}$ for $Q\ (c_{\texttt{src}}, c_{\texttt{trg}}) \overset{\text{def}}{=} c_{\texttt{src}} <= c_{\texttt{trg}} + 2$. This obviously holds for the whole executions of the two programs. However, we wish to reason compositionally about the relatedness of the two programs, showing that the relatedness of the two top-level configurations follows from the relatedness of subconfigurations. For that, we can use Lemma 5.7.

Let us apply the lemma for $C_1 = \texttt{fun}\ f\ x\ l = x::x::l\ \texttt{in}\ [\,\cdot\,]$, $e_1 = e'_{\texttt{src}}$, $C_2 = [\,\cdot\,]$ and $e_2 = e_{\texttt{trg}}$, where $e'_{\texttt{src}}$ is defined below.

$$\begin{array}{rcl} e'_{\texttt{src}} & \overset{\text{def}}{=} & \texttt{let}\ h = 42\ \texttt{in} \\ & & \texttt{let}\ t = \texttt{[]}\ \texttt{in} \\ & & f\ h\ t \end{array}$$

We have that

$$(\sigma_{\texttt{src}}, \texttt{fun}\ f\ x\ l = x::x::l\ \texttt{in}\ [\,\cdot\,]) \rhd \mathsf{Res}(\sigma'_{\texttt{src}}) \quad \text{and} \quad (\sigma_{\texttt{trg}}, [\,\cdot\,]) \rhd \mathsf{Res}(\sigma_{\texttt{trg}})$$

where $\sigma'_{\texttt{src}} = \sigma_{\texttt{src}}[f \mapsto \texttt{Clo}(\sigma_{\texttt{src}}, \texttt{fun}\ f\ x\ l = x::x::l)]$.

To prove the goal, we need to show that $\boldsymbol{\mathcal{E}}^k(\sigma'_{\texttt{src}}, e'_{\texttt{src}})\ (\sigma_{\texttt{trg}}, e_{\texttt{trg}})\ \{Q\}$. But these configurations are no longer related by the logical relation for the Q we picked earlier! The source program needs 4 execution steps while the target still needs 3 executions steps, which invalidates the postcondition. Because of lack of postcondition monotonicity, we are forced to make a choice for Q for the whole program and stick with it during the whole proof, as we execute more instructions of the two programs.

To get around this issue, we decouple the postcondition to a *local postcondition* and a *global postcondition*. The local postcondition holds locally for the executions of the two programs, while the global one holds for future executions of the results. The logical relation is shown in the following definition. Q_L denotes the local postcondition and Q_G the global postcondition.

$$\begin{array}{rcl} \boldsymbol{\mathcal{E}}^k(\sigma_1, e_1)\ (\sigma_2, e_2)\ \{Q_L; Q_G\} & \overset{\text{def}}{=} & \forall\ c_1\ v_1,\ (\sigma_1, e_1)\ {}^{v_1}\!\Downarrow c_1 \Rightarrow \\ & & \quad \exists\ c_2\ v_2,\ (\sigma_2, e_2)\ {}^{v_2}\!\Downarrow c_2 \land \\ & & \quad\quad Q_L\ c_1\ c_2 \land \\ & & \quad\quad \boldsymbol{\mathcal{V}}^{k-c_1}(v_1, v_2)\ \{Q_G\} \end{array}$$

Using this definition, we can establish the postcondition of choice for the top-level program and allow it to vary when we perform local reasoning steps.

5.2.3 Fuels and Traces

So far the logical relation definitions use a natural number to denote the number of steps that a program takes. However, the $\lambda_{\texttt{ANF}}$ semantics uses an abstract notion of fuel to model the execution steps and also an abstract notion of trace to profile other aspects of the computation. Both the fuel and the trace are represented as commutative monoids. Recall the definition of fuel $\langle \mathcal{F}, \langle + \rangle_{\mathcal{F}}, \langle 0 \rangle_{\mathcal{F}} \rangle$ and trace $\langle \mathcal{T}, \langle + \rangle_{\mathcal{T}}, \langle 0 \rangle_{\mathcal{T}} \rangle$ monoids from Section 3.3.2. The fuel value acts as a virtual clock that winds down as the program is being evaluated. The program fails with an out-of-time exception (OOT) if there is not enough fuel to carry out a computation. The trace value is used to profile other information about the program's execution.

The logical relation that I will define in this chapter also treats the fuel and trace monoids abstractly. The postconditions that are used for the $\lambda_{\texttt{ANF}}$ logical relations are relations over pairs of a fuel and a trace value. The proof of each transformation is also parametric in the fuel and trace monoids and also the postcondition, which is only required to obey certain restrictions. We derive a top-level theorem by instantiating each proof with concrete fuel and trace types and also a concrete postcondition, which is typically different for each program. If the postcondition implies that the source steps are upper bounded by some strictly monotonic function of the target steps, then we can also derive divergence preservation.

Although we are only interested in relating the fuel values, the trace value is also helpful to express the bound we want to establish. To show divergence preservation, I use the trace monoid to count separately different kinds of steps that a program takes. In particular, the execution trace keeps a count of application steps and a count of nonapplication steps. This is particularly useful to express the bound for the inlining transformation (Section 6.1.1). Later on, in Chapter 7, where I extend the logical relation to reason about space consumption, I will use the trace monoid to profile the amount of memory that the program uses.

5.2.4 CertiCoq's Logical Relations

For the verification of the $\lambda_{\texttt{ANF}}$ pipeline I will use two different logical relations: one for transformations that do not globally change the way functions are represented and applied (*i.e.*, all transformations except closure conversion), and a different one for closure conversion. The first one symmetrically relates closure values with other closure values (so I will refer to it as symmetrical[7] to distinguish it from the closure-conversion relation). The second one relates closure values with closure records constructed by closure conversion. The definitions of the logical relations are shown in Figure 5.1 and Figure 5.2 respectively. Most of the definitions are the same except

[7] Not to be confused with the standard notion of a symmetric relation. The logical relation is not a symmetric relation.

for the closure value relation. The definition of the logical relation consists of an expression relation, a value relation, a variable relation, and an environment relation. We also define the auxiliary result relation since in our formalization final results of computations can be either values or out-of-time exceptions. I go through each one of these definitions separately.

Value relation

$$\boldsymbol{\mathcal{V}}^k(\mathtt{C}_1(\overline{v_1}), \mathtt{C}_2(\overline{v_2}))\ \{Q\} \overset{\text{def}}{=} \mathtt{C}_1 = \mathtt{C}_2\ \wedge\ \boldsymbol{\mathcal{V}}^k(\overline{v_1}, \overline{v_2})\ \{Q\}$$

$$\begin{array}{l}
\boldsymbol{\mathcal{V}}^k(\mathtt{Clo}(\sigma_1, \mathtt{fun}\ f\ \overline{x} = e_1), \mathtt{Clo}(\sigma_2, \mathtt{fun}\ g\ \overline{y} = e_2))\ \{Q\} \overset{\text{def}}{=} \\
\forall\ i < k\ \overline{v_1}\ \overline{v_2}, \\
\quad \boldsymbol{\mathcal{V}}^i(\overline{v_1}, \overline{v_2})\ \{Q\}\ \Rightarrow \\
\quad \mathsf{len}(\overline{x}) = \mathsf{len}(\overline{v_1})\ \Rightarrow \\
\quad \mathsf{len}(\overline{y}) = \mathsf{len}(\overline{v_2})\ \wedge \\
\quad \boldsymbol{\mathcal{E}}^i(\sigma_1', e_1)\ (\sigma_2', e_2)\ \{Q; Q\}
\end{array}$$

Where $\sigma_1' = \sigma_1[\overline{x} \mapsto \overline{v_1}, f \mapsto \mathtt{Clo}(\sigma_1, \mathtt{fun}\ f\ \overline{x} = e_1)]$
and $\sigma_2' = \sigma_2[\overline{y} \mapsto \overline{v_2}, g \mapsto \mathtt{Clo}(\sigma_2, \mathtt{fun}\ g\ \overline{y} = e_2)]$.

$\boldsymbol{\mathcal{V}}^k(v_1, v_2)\ \{Q\} \overset{\text{def}}{=} \mathtt{False}$ For all other cases.

Result relation

$$\boldsymbol{\mathcal{R}}^k(\mathsf{OOT}, \mathsf{OOT})\ \{Q\} \overset{\text{def}}{=} \mathtt{True}$$

$$\boldsymbol{\mathcal{R}}^k(\mathsf{Res}(v_1), \mathsf{Res}(v_2))\ \{Q\} \overset{\text{def}}{=} \boldsymbol{\mathcal{V}}^k(v_1, v_2)\ \{Q\}$$

$\boldsymbol{\mathcal{R}}^k(r_1, r_2)\ \{Q\} \overset{\text{def}}{=} \mathtt{False}$ For all other cases.

Expression relation

$$\begin{array}{l}
\boldsymbol{\mathcal{E}}^k(\sigma_1, e_1)\ (\sigma_2, e_2)\ \{Q_L; Q_G\} \overset{\text{def}}{=} \\
\forall c_1\ r_1\ t_1,\ \uparrow c_1 \leq k\ \Rightarrow \\
\quad (\sigma_1, e_1)\ {}^{c_1}\!\Downarrow^{t_1}\ r_1\ \Rightarrow \\
\quad \exists\ c_2\ r_2\ t_2,\ (\sigma_2, e_2)\ {}^{c_2}\!\Downarrow^{t_2}\ r_2\ \wedge\ Q_L(c_1, t_1)\ (c_2, t_2)\ \wedge\ \boldsymbol{\mathcal{R}}^{k-\uparrow c_1}(r_1, r_2)\ \{Q_G\}
\end{array}$$

Variable relation

$$\boldsymbol{\mathcal{X}}^k(x, \sigma_1)\ (y, \sigma_2)\ \{Q\} \overset{\text{def}}{=} \forall\ v_1,\ \sigma_1(x) = v_1\ \Rightarrow\ \exists\ v_2, \sigma_2(y) = v_2\ \wedge\ \boldsymbol{\mathcal{V}}^k(v_1, v_2)\ \{Q\}$$

Environment relation

$$S \vdash \boldsymbol{\mathcal{C}}^k(\sigma_1, \sigma_2)\ \{Q\} \overset{\text{def}}{=} \forall\ x \in S,\ \boldsymbol{\mathcal{X}}^k(x, \sigma_1)\ (x, \sigma_2)\ \{Q\}$$

Figure 5.1: The symmetrical logical relation.

Value relation

$$\boldsymbol{\mathcal{V}}^k_{\mathsf{CC}}(\mathtt{C}_1(\overline{v_1}), \mathtt{C}_2(\overline{v_2}))\ \{Q\} \overset{\text{def}}{=} \mathtt{C}_1 = \mathtt{C}_2\ \wedge\ \boldsymbol{\mathcal{V}}^k_{\mathsf{CC}}(\overline{v_1}, \overline{v_2})\ \{Q\}$$

$$\begin{aligned}
&\boldsymbol{\mathcal{V}}^k_{\mathsf{CC}}(\mathtt{Clo}(\sigma_1, \mathtt{fun}\ f\ \overline{x} = e_1), \mathtt{C}_{\mathsf{CC}}(\mathtt{Clo}(\sigma_2, \mathtt{fun}\ g\ \gamma :: \overline{y} = e_2), env))\ \{Q\} \overset{\text{def}}{=}\\
&\forall\ i < k\ \overline{v_1}\ \overline{v_2},\\
&\quad \boldsymbol{\mathcal{V}}^i_{\mathsf{CC}}(\overline{v_1}, \overline{v_2})\ \{Q\}\ \Rightarrow\\
&\quad \mathsf{len}(\overline{x}) = \mathsf{len}(\overline{v_1})\ \Rightarrow\\
&\quad \mathsf{len}(\overline{y}) = \mathsf{len}(\overline{v_2})\ \wedge\\
&\quad \boldsymbol{\mathcal{E}}^i_{\mathsf{CC}}(\sigma'_1, e_1)\ (\sigma'_2, e_2)\ \{Q; Q\}
\end{aligned}$$

Where $\sigma'_1 = \sigma_1[\overline{x} \mapsto \overline{v_1}, f \mapsto \mathtt{Clo}(\sigma_1, \mathtt{fun}\ f\ \overline{x} = e_1)$
and $\sigma'_2 = \sigma_2[\gamma \mapsto env, \overline{y} \mapsto \overline{v_2}, f \mapsto \mathtt{Clo}(\sigma_2, \mathtt{fun}\ g\ \overline{y} = e_2)$.

$\boldsymbol{\mathcal{V}}^k_{\mathsf{CC}}(v_1, v_2)\ \{Q\} \overset{\text{def}}{=} \mathtt{False}$ For all other cases.

Result relation

$\boldsymbol{\mathcal{R}}^k_{\mathsf{CC}}(r_1, r_1)\ \{Q\}$ Same as before.

Expression relation

$\boldsymbol{\mathcal{E}}^k_{\mathsf{CC}}(\sigma_1, e_1)\ (\sigma_2, e_2)\ \{Q_L; Q_G\}$ Same as before.

Variable relation

$\boldsymbol{\mathcal{X}}^k_{\mathsf{CC}}(x, \sigma_1)\ (y, \sigma_2)\ \{Q\}$ Same as before.

Environment relation

$S \vdash \boldsymbol{\mathcal{C}}^k_{\mathsf{CC}}(\sigma_1, \sigma_2)\ \{Q\}$ Same as before.

Figure 5.2: Logical relation for closure conversion.

Expression relation. The symmetrical expression relation (Figure 5.1) is denoted $\boldsymbol{\mathcal{E}}^k(\sigma_1, e_1)\ (\sigma_2, e_2)\ \{Q_L; Q_G\}$. The subscripted symbol $\boldsymbol{\mathcal{E}}_{\mathsf{CC}}$ (Figure 5.2) is used to denote the expression relation used for closure conversion. The first argument k is the usual step index that is needed for the well-foundedness of the definition. The next two arguments are the configurations (pairs of environments and expressions) that are being related. The last two arguments are the *local postcondition* and *global postcondition* that are relations over two pairs of fuel and trace ($Q_L, Q_G \subseteq (\mathcal{F} \times \mathcal{T}) \times (\mathcal{F} \times \mathcal{T})$). The relation asserts that if the the source configuration evaluates to some result and trace with some fuel value that is less or equal to the step index, then there exists target fuel, trace, and result values, such that the evaluation of the target configuration with the target fuel produces the target result and trace. Furthermore, the two results are related by the result relation for the remaining step indices, after subtracting the consumed fuel from the initial step index. As explained earlier, in order to support divergence preservation, we also require that the two pairs of fuel and trace are related by the local postcondition. To compare the fuel value with the step index, which is a natural number, I use the $\uparrow : \mathcal{F} \to \mathbb{N}$ homomorphism defined in Section 3.3.2.

Note: In the mechanized development and also in Chapter 7, the postconditions are relations over triples of configurations, fuel values, and trace values. This is useful to express fuel bounds that are dependent on the size of the source term. Since the bounds that are required to establish preservation of divergence are not dependent on the source configurations, in the paper presentation of the logical relations of this chapter, I use postconditions that are relations over pairs of a fuel a trace value.

Result relation. The result relation is denoted $\boldsymbol{\mathcal{R}}^k(r_1, r_2)\ \{Q\}$ (or $\boldsymbol{\mathcal{R}}_{\mathsf{CC}}$ for the closure-conversion relation). It is true whenever the two results are both out-of-time exceptions or related values, and false otherwise.

Value relation. The value relation relates values of the language and it is denoted $\boldsymbol{\mathcal{V}}^k(v_1, v_2)\ \{Q\}$ (or $\boldsymbol{\mathcal{V}}_{\mathsf{CC}}$ for the closure-conversion relation). It is the only definition that differs between the two logical relations. In both cases, two constructed values are related if they are constructed with the same constructor, and the arguments of the two constructors are of the same length and pairwise related with the value relation. Two closure values are related with $\boldsymbol{\mathcal{V}}$ at some step index k if they map lists of arguments related at some step index $i < k$ to configurations related at step index i. Each configuration consist of the body of the function part of the corresponding closure and the environment part of the closure extended with appropriate bindings. The closure environments are extended with the formal parameters of each function bound to the values of the actual parameters, and the function name bound to the closure value, which is needed for recursive functions. Recall that the closure environment is the evaluation environment at the time of function definition and contains the values of the function's free variables.

For the closure-conversion relation $\boldsymbol{\mathcal{V}}_{\mathsf{CC}}$, the value relation for closure values is different. In this relation, a closure value is not related with another closure value, but with an explicitly constructed closure record, created with the closure constructor $\mathtt{C}_{\mathtt{CC}}$. The code component of the closure record is a closure[8] and the environment component an arbitrary value. We require that the second function definition has an additional argument γ for the closure environment. The two closure values are related as before, with the only difference that in the environment of the second configuration the argument γ is mapped to the value of the environment in the closure record.

Note: I use the notation $\boldsymbol{\mathcal{V}}^k(\overline{v_1}, \overline{v_2})\ \{Q\}$ to denote that the two lists $\overline{v_1}$ and $\overline{v_2}$ have the same number of elements that are pairwise related.

Variable and environment relations. The variable relation is denoted $\boldsymbol{\mathcal{X}}^k(x, \sigma_1)\ \ (y, \sigma_2)\ \{Q\}$ (or $\boldsymbol{\mathcal{X}}_{\mathtt{CC}}$ for the closure-conversion relation). It asserts that whenever x is defined in the environment then so is y, and, furthermore, that their values are related. As with the value relation, I will also use the variable relation with lists of variables meaning that the lists have the same number of elements that are pairwise related with the variable relation. The environment relation is denoted $S \vdash \boldsymbol{\mathcal{C}}^k(\sigma_1, \sigma_2)\ \{Q\}$ (or $\boldsymbol{\mathcal{C}}_{\mathtt{CC}}$ for the closure-conversion relation) and it is defined in terms of the variable relation. It asserts that any binder x that belongs to the set S and the domain of σ_1 it also belongs to the domain of σ_2, and the values of the two environments at x are related with the value relation.

Mnemonic: $\boldsymbol{\mathcal{E}}$ is used for the ***e***xpression relation and $\boldsymbol{\mathcal{V}}$ is used for the ***v***alue relation. $\boldsymbol{C}$ is used for relating environments, which provide ***c***ontexs for the free variables of the term. $\boldsymbol{\mathcal{X}}$ is used for the variable relation, named after the ubiquitous variable name $\boldsymbol{x}$.

5.2.5 Reasoning with Local and Global Postconditions

The expression logical relation is parameterized with a local and a global postcondition. The local postcondition relates the fuel and trace values of the two program executions, whereas the global one holds for later executions of the (possibly higher-order) results. Separating the global from the local postcondition allows the local postcondition to vary independently from the global postcondition which enables compositional reasoning.

The *compatibility* lemmas of the logical relation, which we will state in the following section, enable us to reason compositionally about the executions of two programs. We state and prove these lemmas for abstract postconditions that satisfy certain requirements. In particular, will impose some conditions in the local and global postconditions that assert that the relations are preserved when different constructors of the language are evaluated.

To make this clear let us consider the context compatibility lemma (Lemma 5.7). The statement of this lemma will now use the expression relation that is extended with the local and the global postcondition:

[8] Observe that even after closure conversion function values are still closures. This is because functions are not necessarily closed yet. They may contain free variables that refer to other closed functions. After hoisting, all functions will be defined at the same mutually-recursive function bundle, and all functions become closed.

$$\begin{array}{c}
(\sigma_1, \mathcal{E}_1) \ {}^{c_1}\triangleright^{t_1} \ \mathsf{Res}(\sigma_1') \ \ \Rightarrow \\
(\sigma_2, \mathcal{E}_2) \ {}^{c_2}\triangleright^{t_2} \ \mathsf{Res}(\sigma_2') \ \ \Rightarrow \\
\boldsymbol{\mathcal{E}}^{(\sigma_1', e_1)}(\sigma_2', e_2') \ Q_1 \ \{Q_G; \ \} \ \Rightarrow \\
\boldsymbol{\mathcal{E}}^{k}(\sigma_1, C_1[e_1]) \ (\sigma_2, C_2[e_2]) \ \{Q_2; Q_G\}
\end{array}$$

Where c_1 and c_2 the fuel values and t_1 and t_2 are trace values (which were elided from the previous statement of the lemma). Q_G is the global postcondition that remains invariant throughout the execution of two configurations. Q_1 and Q_2 are the local postconditions for the two congifurations. As we argued in Section 5.2.1, the local postcondition in the assumption and the goal need not necessarily be the same.

To prove this lemma we need to make certain assumptions about Q_1 and Q_2. First, we need to be able to derive Q_2 for the fuel and trace values of the evaluation of the configurations $(\sigma_1, C_1[e_1])$ and $(\sigma_2, C_2[e_2])$ from Q_1 for the fuel and trace values of the evaluation of the configurations (σ_1', e_1') and (σ_2', e_2'). Let c, t (resp. c', t') be the fuel and trace values of the configuration (σ_1', e_1) (resp. (σ_2', e_2)) for which we know $Q_1 \ (c, t) \ (c', t')$. Then, because of the compositionality of the interpretation judgment with the evaluation judgment, we know that the fuel and trance of configuration $(\sigma_1, C_1[e_1])$ (resp. $(\sigma_2, C_2[e_2])$) will be $c\langle + \rangle_{\mathcal{F}} c_1$ and $t\langle + \rangle_{\mathcal{T}} t_1$ (resp. $c'\langle + \rangle_{\mathcal{F}} c_2$ and $t'\langle + \rangle_{\mathcal{T}} t_2$). To prove the compatibility lemma, we need to establish that $Q_2 \ (c\langle + \rangle_{\mathcal{F}} c_1, t\langle + \rangle_{\mathcal{T}} t_1) \ (c'\langle + \rangle_{\mathcal{F}} c_2, t'\langle + \rangle_{\mathcal{T}} t_2)$. This is described by the following predicate.

Definition 5.5 (Postcondition compatibility, context application)
PostCompatCtx $Q_1 \ Q_2 \ c_1 \ t_1 \ c_2 \ t_2$ *holds iff whenever*

$$Q_1(c, t) \ (c', t')$$

we also have

$$Q_2(c\langle + \rangle_{\mathcal{F}} c_1, t\langle + \rangle_{\mathcal{T}} t_1) \ (c'\langle + \rangle_{\mathcal{F}} c_2, t'\langle + \rangle_{\mathcal{T}} t_2).$$

Furthermore, if the source configuration times out during execution of C_1, we will need to provide a target fuel value that makes the target configuration time out as well. We could pick the obvious fuel value $\langle 0 \rangle_{\mathcal{F}}$, for which the target configuration trivially times out with trace $\langle 0 \rangle_{\mathcal{T}}$. But then we would need to establish that $Q_2(c, t) \ (\langle 0 \rangle_{\mathcal{F}}, \langle 0 \rangle_{\mathcal{T}})$ for some arbitrary c and t, which does not hold for the postconditions we are interested. Instead, we assume that $\texttt{fuel}(C_1) \leq_{\mathcal{F}} \texttt{fuel}(C_2)$, where $\texttt{fuel}(C)$ is the (statically known) amount of fuel that is required for the execution of the binding context C. Then, we require that the postcondition holds for the same fuel values (regardless of the trace value).[9]

Definition 5.6 (Postcondition is reflexive for fuel.)
PostRefl Q *holds iff* $Q \ (c, t) \ (c, t')$.

[9] This does not hold of the concrete postcondition that we use for inlining. Therefore, we cannot use this compatibility lemma in the proof of inlining.

Now we can state and prove the context compatibility lemma in presence of postconditions.

Lemma 5.7 (Context application compatibility.)
Assume that $\mathsf{PostRefl}\ Q_2$ *and* $\mathsf{PostCompatCtx}\ Q_1\ Q_2\ c_1\ t_1\ c_2\ t_2$.

Let C_1 *and* C_2 *be binding contexts such that*

- $\mathtt{fuel}(C_1) \leq_{\mathcal{F}} \mathtt{fuel}(C_2)$,
- $(\sigma_1, \mathcal{E}_1)\ {}^{c_1}{\triangleright}^{t_1}\ \mathsf{Res}(\sigma_1')$,
- $(\sigma_2, \mathcal{E}_2)\ {}^{c_2}{\triangleright}^{t_2}\ \mathsf{Res}(\sigma_2')$, and
- $\boldsymbol{\mathcal{E}}^k(\sigma_1', e_1)\ (\sigma_2', e_2)\ \{Q_1; Q_G\}$

Then we can derive that $\boldsymbol{\mathcal{E}}^k(\sigma_1, C_1[e_1])\ (\sigma_2, C_2[e_2])\ \{Q_2; Q_G\}$.

In the rest of this section, I will define the different conditions that we need to impose on local and global postconditions when proving compatibility lemmas for the logical relations. These rules assert that the postconditions are preserved when different constructors of the language are evaluated. In the next section, I will state more *compatibility* lemmas of the logical relation that will make use of these definitions.

In most of the compatibility lemmas, we will need to establish a postcondition when both programs have zero fuel remaining and they both time out throwing an out-of-time exception. This is captured by the following definition.

Definition 5.8 (Postcondition holds for zero)
$\mathsf{PostZero}\ Q$ *holds iff* $Q(\langle 0\rangle_{\mathcal{F}}, \langle 0\rangle_{\mathcal{T}})\ (\langle 0\rangle_{\mathcal{F}}, \langle 0\rangle_{\mathcal{T}})$.

Similarly, we need to be able to establish the postcondition when the two programs terminate by returning some (related) values.

Definition 5.9 (Postcondition holds for return)
$\mathsf{PostRet}\ Q$ *holds iff* $Q(\langle \mathtt{ret}(x)\rangle_{\mathcal{F}}, \langle \mathtt{ret}(x)\rangle_{\mathcal{T}})\ (\langle \mathtt{ret}(y)\rangle_{\mathcal{F}}, \langle \mathtt{ret}(y)\rangle_{\mathcal{T}})$.

We are also required to assume that the postcondition is preserved when both the source and target perform one evaluation step.

Definition 5.10 (Postcondition compatibility)
$\mathsf{PostCompat}\ Q_1\ Q_2\ e_1\ e_2$ *holds iff whenever*

$$Q_1(c_1, t_1)\ (c_2, t_2)$$

we also have

$$Q_2(c_1 \langle + \rangle_{\mathcal{F}} \langle e_1\rangle_{\mathcal{F}}, t_1 \langle + \rangle_{\mathcal{T}} \langle e_1\rangle_{\mathcal{T}})\ (c_2 \langle + \rangle_{\mathcal{F}} \langle e_2\rangle_{\mathcal{F}}, t_2 \langle + \rangle_{\mathcal{T}} \langle e_2\rangle_{\mathcal{T}}).$$

We use two different postconditions, Q_1 and Q_2, to allow the postconditions be different before and after preforming the evaluation step (for the reasons explained in Section 5.2.1).

We use a slightly different compatibility rule is to establish the local postcondition for programs whose outermost constructors are let-bound applications (nontail calls). In this case, from the postcondition of the function and the postcondition of the execution of the rest of the program, we derive the postcondition for the execution of the let-bound call.

Definition 5.11 (Postcondition compatibility for let-bound application) $\mathsf{PostLetApp}\ Q_1\ Q_2\ Q_3\ P\ e_1\ e_2$ *holds iff whenever*

$$Q_1(c_1, t_1)\ (c_2, t_2)\ \textit{and}\ Q_2(c_1', t_1')\ (c_2', t_2')$$

we also have

$$Q_3(c_1\langle+\rangle c_1'\langle+\rangle\langle e_1\rangle, t_1\langle+\rangle t_1'\langle+\rangle\langle e_1\rangle)\ (c_2\langle+\rangle c_2'\langle+\rangle\langle e_2\rangle, t_2\langle+\rangle t_2'\langle+\rangle\langle e_2\rangle).$$

With these definitions in hand, we can now state the compatibility lemmas for the logical relation.

5.2.6 Compatibility Lemmas

The compatibility lemmas assert that the relation is preserved during the execution of two programs.

Note: Properties that hold for both for both logical relations will be denoted with † next to the name of the theorem. Unless this symbol is used a theorem holds only for the relation that it is stated for.

The lemma for constructor states that two constructor expressions are related if 1. their variable arguments are pairwise related in the two environments, and 2. if for every pair of lists of pairwise related values the rest of the two programs are related in the environments extended to bind the let-bound variables two newly allocated constructed values. Q_G is the global postcondition while Q_1 is the local poscondition that holds for the evaluation of the continuations of the contructors, and Q_2 is the local postcondition that is eshtablished for the two expressions.

Lemma 5.12 (Compatibility (constructor) †)
Assume that $\mathsf{PostZero}\ Q_2$ *and*
$\mathsf{PostCompat}\ Q_1\ Q_2\ (\texttt{let}\ x_1 = \texttt{C}(\overline{y_1})\ \texttt{in}\ e_1)\ (\texttt{let}\ x_2 = \texttt{C}(\overline{y_2})\ \texttt{in}\ e_2)$.

If

- $\boldsymbol{\mathcal{X}}^k(\overline{y_1}, \sigma_1)\ (\overline{y_2}, \sigma_2)\ \{Q_G\}$
- $\forall\, \overline{v_1}\, \overline{v_2},\ \boldsymbol{\mathcal{V}}^k(\overline{v_1}, \overline{v_2})\ \{Q_G\}\ \Rightarrow\ \boldsymbol{\mathcal{E}}^k(e_1, \sigma_1[x_1 \mapsto C(\overline{v_1})])\ (e_2, \sigma_2[x_2 \mapsto C(\overline{v_2})])\ \{Q_1; Q_G\}$

then $\boldsymbol{\mathcal{E}}^k(\texttt{let}\ x_1 = \texttt{C}(\overline{y_1})\ \texttt{in}\ e_1, \sigma_1)\ (\texttt{let}\ x_2 = \texttt{C}(\overline{y_2})\ \texttt{in}\ e_2, \sigma_2)\ \{Q_2; Q_G\}$.

The rule for projections is in similar spirit.

Lemma 5.13 (Compatibility (projection) †)
Assume that $\mathsf{PostZero}\ Q_2$ *and*
$\mathsf{PostCompat}\ Q_1\ Q_2\ (\texttt{let}\ x_1 = y_1.i\ \texttt{in}\ e_1)\ (\texttt{let}\ x_2 = y_2.i\ \texttt{in}\ e_2)$.

If

- $\boldsymbol{\mathcal{X}}^k(y_1, \sigma_1)\ (y_2, \sigma_2)\ \{Q_G\}$
- $\forall\ v_1\ v_2,\ \boldsymbol{\mathcal{V}}^k(v_1, v_2)\ \{Q_G\}\ \Rightarrow\ \boldsymbol{\mathcal{E}}^k(e_1, \sigma_1[x_1 \mapsto v_1])\ (e_2, \sigma_2[x_2 \mapsto v_2])\ \{Q_1; Q_G\}$

then $\boldsymbol{\mathcal{E}}^k(\texttt{let}\ x_1 = y_1.i\ \texttt{in}\ e_1, \sigma_1)\ (\texttt{let}\ x_2 = y_2.i\ \texttt{in}\ e_2, \sigma_2)\ \{Q_2; Q_G\}$.

The lemma for case analysis requires that the two scrutinees are related in the environment, that the patterns are pairwise the same and that expressions of each pattern are pairwise related in the current environment.

Lemma 5.14 (Compatibility (case analysis) †)
Assume that $\mathsf{PostZero}\ Q_2$ *and*
$\mathsf{PostCompat}\ Q_1\ Q_2\ (\texttt{case}\ y_1\ \texttt{of}\ [\texttt{C}_i\ \rightarrow\ e_i]_{i \in I})\ (\texttt{case}\ y_2\ \texttt{of}\ [\texttt{C}_i\ \rightarrow\ e'_i]_{i \in I})$.

If

- $\boldsymbol{\mathcal{X}}^k(y_1, \sigma_1)\ (y_2, \sigma_2)\ \{Q_G\}$
- $\forall\ i,\ \boldsymbol{\mathcal{E}}^k(e_i, \sigma_1)\ (e'_i, \sigma_2)\ \{Q_1; Q_G\}$

then $\boldsymbol{\mathcal{E}}^k(\texttt{case}\ y_1\ \texttt{of}\ [\texttt{C}_i\ \rightarrow\ e_i]_{i \in I}, \sigma_1)\ (\texttt{case}\ y_2\ \texttt{of}\ [\texttt{C}_i\ \rightarrow\ e'_i]_{i \in I}, \sigma_2)\ \{Q_2; Q_G\}$.

Two function definitions are related if the rest of the programs are related in the current environment extended with bindings that map the function names to the closure consisting of the function bodies and the current environments.

Lemma 5.15 (Compatibility (function definition) †)
Assume that $\mathsf{PostZero}\ Q_2$ *and*
$\mathsf{PostCompat}\ Q_1\ Q_2\ (\texttt{fun}\ f_1\ \overline{x_1} = e'_1\ \texttt{in}\ e_1)\ (\texttt{fun}\ f_2\ \overline{x_2} = e'_2\ \texttt{in}\ e_2)$.

If

- $\boldsymbol{\mathcal{E}}^k(e_1, \sigma'_1)\ (e_2, \sigma'_2)\ \{Q_1; Q_G\}$
  where $\sigma'_1 = \sigma_1[f_1 \mapsto \texttt{Clo}(\texttt{fun}\ f_1\ \overline{x_1} = e'_1, \sigma_1,])$
  and $\sigma'_2 = \sigma_2[f_2 \mapsto \texttt{Clo}(\texttt{fun}\ f_2\ \overline{x_2} = e'_2, \sigma_2,])$

then $\boldsymbol{\mathcal{E}}^k(\texttt{fun}\ f_1\ \overline{x_1} = e'_1\ \texttt{in}\ e_1, \sigma_1)\ (\texttt{fun}\ f_2\ \overline{x_2} = e'_2\ \texttt{in}\ e_2, \sigma_2)\ \{Q_2; Q_G\}$.

The lemma for return simply requires that the returned identifiers are related in the current environments.

Lemma 5.16 (Compatibility (return) †)
Assume that $\mathsf{PostZero}\ Q$ *and* $\mathsf{PostRet}\ Q$

If

- $\boldsymbol{\mathcal{X}}^k(x_1, \sigma_1)\ (x_2, \sigma_2)\ \{Q_G\}$

then $\boldsymbol{\mathcal{E}}^k(\texttt{ret}(x_1), \sigma_1)\ (\texttt{ret}(x_2), \sigma_2)\ \{Q; Q_G\}$.

So far, all of the compatibility lemmas are the same for the two logical relations. The compatibility lemmas for function calls (nontail let-bound calls and tail calls) are different for the two relations, since functions are related differently by the two relations.

In the symmetrical relation, to show that two let-bound function applications are related we require that the identifiers for the applied functions as well as the two lists of arguments are related in the source and target environments. Additionally, we require that the rest of the programs are related in the current environments extended with related mappings for let-bound variables. As usual we require the postcondition to satisfy $\mathsf{PostZero}$. But unlike the previous lemma we also require (using $\mathsf{PostLetApp}$) that if the global postcondition Q_G holds for the execution of the function calls and the local postcondition Q_1 holds for the evaluation of the rest of the programs, then the local postcondition Q_2 holds for the evaluation of the two function calls.

Lemma 5.17 (Compatibility (let-bound application))
Assume that $\mathsf{PostZero}\ Q_2$ *and*
$\mathsf{PostLetApp}\ P_G\ Q_1\ Q_2\ (\texttt{let}\ x_1 = f_1\ \overline{y_1}\ \texttt{in}\ e_1)\ (\texttt{fun}\ f_2\ \overline{y_2} = e_2\ \texttt{in}\ )$.

If

- $\boldsymbol{\mathcal{X}}^k(f_1, \sigma_1)\ (f_2, \sigma_2)\ \{Q_G\}$
- $\boldsymbol{\mathcal{X}}^k(\overline{y_1}, \sigma_1)\ (\overline{y_2}, \sigma_2)\ \{Q_G\}$
- $\forall\ v_1\ v_2,\ \boldsymbol{\mathcal{V}}^k(v_1, v_2)\ \{Q_G\}\ \Rightarrow\ \boldsymbol{\mathcal{E}}^k(e_1, \sigma_1[x_1 \mapsto v_1])\ (e_2, \sigma_2[x_2 \mapsto v_2])\ \{Q_1; Q_G\}$

Then $\boldsymbol{\mathcal{E}}^k(\texttt{let}\ x_1 = f_1\ \overline{y_1}\ \texttt{in}\ e_1, \sigma_1)\ (\texttt{let}\ x_2 = f_2\ \overline{y_2}\ \texttt{in}\ e_2, \sigma_2)\ \{Q_2; Q_G\}$.

To show that two tail calls are related, we only require that the identifiers being applied and the lists of arguments are related in the two environments. We assume that the postcondition holds if the two programs time out ($\mathsf{PostZero}$) and that if the global postcondition holds for the execution of the two functions, then the local postcondition holds for the execution of the applications ($\mathsf{PostCompat}$).

Lemma 5.18 (Compatibility (tail-call))
Assume that $\mathsf{PostZero}\ Q$ *and* $\mathsf{PostCompat}\ Q_G\ Q\ (f_1\ \overline{y_1})\ (f_2\ \overline{y_2})$.

If

- $\boldsymbol{\mathcal{X}}^k(f_1, \sigma_1)\ (f_2, \sigma_2)\ \{Q_G\}$
- $\boldsymbol{\mathcal{X}}^k(\overline{y_1}, \sigma_1)\ (\overline{y_2}, \sigma_2)\ \{Q_G\}$

Then $\boldsymbol{\mathcal{E}}^k(f_1\ \overline{y_1}, \sigma_1)\ (f_2\ \overline{y_2}, \sigma_2)\ \{Q; Q_G\}$.

To relate function applications before and after closure conversion, we need to convert function calls to application of closure records that project the function and environment out of the explicitly constructed closure records. We also need to state different rules for the preservation of the postconditions. The rules are similar to those of the symmetrical relation but they account for the fuel and trace of projecting the code and the environment from the closure pair.

Definition 5.19 (Postcondition compat. for tail app., closure conversion) *Let* $c_{app} \overset{\text{def}}{=} \langle f_{code}\ f_{env} :: \overline{y_2}\rangle\langle+\rangle\langle \texttt{let}\ f_{env} = f_2.2\ \texttt{in}\ \ldots\rangle\langle+\rangle\langle\texttt{let}\ f_{code} = f_2.1\ \texttt{in}\ \ldots\rangle$ *and* $t_{app} \overset{\text{def}}{=} \langle f_{code}\ f_{env} :: \overline{y_2}\rangle\langle+\rangle\langle \texttt{let}\ f_{env} = f_2.2\ \texttt{in}\ \ldots\rangle\langle+\rangle\langle\texttt{let}\ f_{code} = f_2.1\ \texttt{in}\ \ldots\rangle$.

$\mathsf{PostAppCC}$ Q_1 Q_2 *holds iff whenever*

$$Q_1(c_1, t_1)\ (c_2, t_2)$$

we also have

$$Q_2(c_1\langle+\rangle\langle f_1\ \overline{x_1}\rangle, t_1\langle+\rangle\langle f_1\ \overline{x_1}\rangle)\ (c_2\langle+\rangle c_{app}, t_2\langle+\rangle t_{app}).$$

Definition 5.20 (Postcondition compat. for let app., closure conversion) *Let* $c_{app} \overset{\text{def}}{=} \langle\texttt{let}\ x_2 = f_{code}\ f_{env} :: \overline{y_2}\ \texttt{in}\ \ldots\rangle\langle+\rangle\langle \texttt{let}\ f_{env} = f_2.2\ \texttt{in}\ \ldots\rangle\langle+\rangle\langle\texttt{let}\ f_{code} = f_2.1\ \texttt{in}\ \ldots\rangle$
and $t_{app} \overset{\text{def}}{=} \langle\texttt{let}\ x_2 = f_{code}\ f_{env} :: \overline{y_2}\ \texttt{in}\ \ldots\rangle\langle+\rangle\langle \texttt{let}\ f_{env} = f_2.2\ \texttt{in}\ \ldots\rangle\langle+\rangle\langle\texttt{let}\ f_{code} = f_2.1\ \texttt{in}\ \ldots\rangle$.
Also, let $e_1' \overset{\text{def}}{=} \texttt{let}\ x_1 = f_1\ \overline{x_1}\ \texttt{in}\ e_1$.

$\mathsf{PosLetAppCC}$ Q_1 Q_2 Q_3 *holds iff whenever*

$$Q_1(c_1, t_1)\ (c_2, t_2)\ \textit{and}\ Q_2(c_1', t_1')\ (c_2', t_2')$$

we also have

$$Q_3(c_1\langle+\rangle c_1'\langle+\rangle\langle e_1'\rangle, t_1\langle+\rangle t_1'\langle+\rangle\langle e_1'\rangle)\ (c_2\langle+\rangle c_2'\langle+\rangle c_{app}, t_2\langle+\rangle t_2'\langle+\rangle t_{app}).$$

We can now state compatibility lemmas for function application for the closure-conversion relation. The let-bound function application lemma states that applying a function is related to a closure application (that is, applying a function after projecting the code and the environment out) if the identifiers of the function and the arguments are related in the current environment and the rest of the programs are related when the current environments are extended with related values. Notice, that in the target environment we also need to add bindings for the identifiers that are used for the code and the environment. We achieve that by using the binding-context interpretation relation.

Lemma 5.21 (Compatibility (let-bound application, closure conversion)) *Let* $e_2' = \texttt{let}\ f_{code} = f_2.1\ \texttt{in}\ \texttt{let}\ f_{env} = f_2.2\ \texttt{in}\ \texttt{let}\ x_2 = f_{code}\ f_{env} :: \overline{y_2}\ \texttt{in}\ e_2$ *and that the identifiers* f_{code} *and* f_{env} *are distinct and different from* f_2 *and* $\overline{y_2}$.

Assume that $\mathsf{PostZero}\ Q$ *and* $\mathsf{PostLetAppCC}\ Q_G\ Q_1\ Q_2$.

If

- $\boldsymbol{\mathcal{X}}^k_{\mathit{CC}}(f_1, \sigma_1)\ (f_2, \sigma_2)\ \{Q_G\}$
- $\boldsymbol{\mathcal{X}}^k_{\mathit{CC}}(\overline{y_1}, \sigma_1)\ (\overline{y_2}, \sigma_2)\ \{Q_G\}$
- $\forall\ v_1\ v_2\ \sigma_2',\ \boldsymbol{\mathcal{V}}^k_{\mathit{CC}}(v_1, v_2)\ \{Q_G\}\ \Rightarrow$ $(\sigma_2, \mathtt{let}\ f_{code} = f_2.1\ \mathtt{in}\ \mathtt{let}\ f_{env} = f_2.2\ \mathtt{in}\ [])\ \triangleright\ \mathsf{Res}(\sigma_2')\ \Rightarrow$ $\boldsymbol{\mathcal{E}}^k_{\mathit{CC}}(e_1, \sigma_1[x_1 \mapsto v_1])\ (e_2, \sigma_2'[x_2 \mapsto v_2])\ \{Q_1; Q_G\}$

Then $\boldsymbol{\mathcal{E}}^k_{\mathsf{CC}}(\mathtt{let}\ x_1 = f_1\ \overline{x_1}\ \mathtt{in}\ e_1, \sigma_1)\ (e_2', \sigma_2)\ \{Q_2; Q_G\}$.

A tail call is related to a closure-converted tail call if the identifiers for the function and the arguments are related in the current environment.

Lemma 5.22 (Compatibility (tail application, closure conversion))
Let $e_2' = \mathtt{let}\ f_{code} = f_2.1\ \mathtt{in}\ \mathtt{let}\ f_{env} = f_2.2\ \mathtt{in}\ f_{code}\ f_{env} :: \overline{y_2}$ *and assume that the identifiers* f_{code} *and* f_{env} *are distinct and different from* f_2 *and* $\overline{y_2}$.

Assume that $\mathsf{PostZero}\ Q$ *and* $\mathsf{PostAppCC}\ Q_G\ Q$.

If

- $\boldsymbol{\mathcal{X}}^k_{\mathit{CC}}(f_1, \sigma_1)\ (f_2, \sigma_2)\ \{Q_G\}$
- $\boldsymbol{\mathcal{X}}^k_{\mathit{CC}}(\overline{y_1}, \sigma_1)\ (\overline{y_2}, \sigma_2)\ \{Q_G\}$

Then $\boldsymbol{\mathcal{E}}^k_{\mathit{CC}}(f_1\ \overline{x_1}, \sigma_1)\ (e_2', \sigma_2)\ \{Q; Q_G\}$.

This concludes the compatibility lemmas for the two relations.

It is useful to define a shorthand for assuming the set of requirements over the postconditions that allow the compatibility lemmas to be proved.

Definition 5.23 (Postcondition properties shorthand)
$\mathsf{PostProperties}\ Q_G\ Q_1\ Q_2$ *holds whenever the following are satisfied.*

- $\mathsf{PostZero}\ Q_2$
- $\mathsf{PostRet}\ Q_2$
- $\mathsf{PostCompat}\ Q_1\ Q_2$
- $\mathsf{PostLetApp}\ Q_G\ Q_1\ Q_2$

5.2.7 Properties of the Logical Relations

In this section I present some important properties of the logical relations. As usual, both logical relations are (anti)monotonic in the step index, meaning that whenever the relation holds for a value of the step index it also holds for all smaller values of the step index.

Lemma 5.24 (Step index monotonicity †)
$\forall\ i \leq k,\ \boldsymbol{\mathcal{E}}^k(\sigma_1, e_1)\ (\sigma_2, e_2)\ \{Q_L; Q_G\}\ \Rightarrow\ \boldsymbol{\mathcal{E}}^i(\sigma_1, e_1)\ (\sigma_2, e_2)\ \{Q_L; Q_G\}$.

The same holds for the value, result, variable and environment relation.

We can also prove that the logical relation is monotonic in the local postcondition. Observe that this property hold because we have decoupled the local and the global postcondion. If we had used the same postcondition for the local and global postconditions, we would have been able to prove monotonicity, because the global postcondition occurs in both negative and positive in the definition of the logical relation

Lemma 5.25 (Local postcondition monotonicity †)
$\forall\ Q \supseteq Q',\ \boldsymbol{\mathcal{E}}^k(\sigma_1, e_1)\ (\sigma_2, e_2)\ \{Q; Q_G\}\ \Rightarrow\ \boldsymbol{\mathcal{E}}^i(\sigma_1, e_1)\ (\sigma_2, e_2)\ \{Q'; Q_G\}$.

The symmetrical logical relation is also reflexive. For any expression e and for any two environments that are related in the set of free variables of the expression e, then the configuration (σ_1, e) is related to the configuration (σ_2, e). Observe that such a property does not hold for the closure-conversion relation since the closure-conversion relation is inhabited only by a suitably closure-converted program.

Lemma 5.26 (Reflexivity)
Assume that $\mathsf{PostProperties}\ Q_G\ Q\ Q$ *and* $\mathsf{PostProperties}\ Q_G\ Q_G\ Q_G$ *and* $Q \supseteq Q_G$.

If $\mathsf{fv}(e) \vdash \boldsymbol{\mathcal{C}}^k(\sigma_1, \sigma_2)\ \{Q\}$ *then* $\boldsymbol{\mathcal{E}}^k(\sigma_1, e)\ (\sigma_2, e)\ \{Q_L; Q_G\}$.

Proof By induction on the step index and nested induction on the expression e. Each case follows from the corresponding compatibility lemma (therefore, we need to assume the required properties for the postconditions hold). The condition that the local postcondition implies the global postcondition is needed in order to show that adding the same two function definitions (satisfying the local postcondition by the induction hypothesis) in two related environments, gives us related environments. The condition that P_G also satisfies $\mathsf{PostProperties}$ is required by the induction hypothesis. ■

Two other crucial properties of the logical relation are adequacy and compatibility with linking (horizontal compositionality). Adequacy of the logical relation for terminating behaviors follows trivially for the definition of the expression logical relation. In order to prove adequacy for nonterminating behaviors we shall assume that the local postcondition implies that the fuel value of the source is upper bounded by some strictly monotonic function of the target fuel value.

Definition 5.27 (Postcondition, upper bound)
$\mathsf{PostUpperBound}\ Q$ *holds iff there exists a function* f *such that* $\forall\ x\ y,\ f(x) \leq_{\mathcal{F}} f(y) \Rightarrow x \leq_{\mathcal{F}} y$ *and for all fuel values* $c_1,\ c_2$ *and trace values* $t_1,\ t_2$ *if* $Q(c_1, t_1)\ (c_2, t_2)$ *then* $c_1 \leq f(c_2)$.

We can now state the adequacy property of the logical relation.

Lemma 5.28 (Adequacy †)
Assume that $\mathsf{PostUpperBound}\ Q$.

Then if $\forall\ k,\ \boldsymbol{\mathcal{E}}^k(\sigma_1, e_1)\ (\sigma_2, e_2)\ \{Q; Q_G\}$ *we have that* $(e_1, \sigma_1) \supseteq_B (e_2, \sigma_2)$.

Proof The termination case follows easily by the definition of the expression relations and by the fact that the value relation implies the value refinement $\approx$. The nontermination case follows by the divergence preservation lemma of the semantics (Lemma 3.10). ■

In addition both of the relations are compatible with linking.

Lemma 5.29 (Linking compatibility †)
Assume that $\mathsf{PostProperties}\ Q_G\ Q\ Q$.

If

- $\forall\ k\ \sigma_1\ \sigma_2,\ \boldsymbol{\mathcal{E}}^k(e_1^{\mathrm{lib}}, \sigma_1)\ (e_2^{\mathrm{lib}}, \sigma_2)\ \{Q; Q_G\}$
- $\forall\ k\ \sigma_1\ \sigma_2,\ \{x\} \vdash \boldsymbol{\mathcal{C}}^k(\sigma_1, \sigma_2)\ \{Q_G\}\ \Rightarrow\ \boldsymbol{\mathcal{E}}^k(e_1^{\mathrm{client}}, \sigma_1)\ (e_2^{\mathrm{client}}, \sigma_2)\ \{Q; Q_G\}$

Then $\forall\ k\ \sigma_1\ \sigma_2,\ \boldsymbol{\mathcal{E}}^k(\sigma_1, [x \mapsto e_1^{\mathrm{lib}}]e_1^{\mathrm{client}})\ (\sigma_2, [x \mapsto e_2^{\mathrm{lib}}]e_2^{\mathrm{client}})\ \{Q; Q_G\}$.

The above statement asserts that if the library programs are related in all possible environments, and that the client programs are related in all environments that are related in the value of the external reference x, then linking in the source is related with linking in the target.

This relational model does not have the necessary vertical compositionality properties that are required to prove that the end-to-end pipeline inhabits the relation. In the next chapter I will set up a relational framework built on top of the logical relation framework that allows us to obtain a top-level relation that is inhabited by the end-to-end pipeline and is adequate and compatible with linking.

Vertical Compositionality

Despite the fact that that the two relations do not support a general, unrestricted notion of vertical compositionality, they do support a restricted form of vertical compositionality that can be useful in certain proofs. In particular, the symmetrical logical relation is transitive given that the local and global postconditions satisfy some requirements.

Lemma 5.30 (Transitivity)
Assume that $Q_G \supseteq Q_1$, $Q_G \supseteq Q_2$ *and* $Q_1 \circ Q_2 \supseteq Q_G$.

If

- $\boldsymbol{\mathcal{E}}^k(\sigma_1, e_1)\ (\sigma_2, e_2)\ \{Q_1; Q_G\}$
- $\forall\ k,\ \boldsymbol{\mathcal{E}}^k(\sigma_2, e_2)\ (\sigma_3, e_3)\ \{Q_2; Q_G\}$

then $\boldsymbol{\mathcal{E}}^k(\sigma_1, e_1)\ (\sigma_3, e_3)\ \{Q_1 \circ Q_2; Q_G\}$

The above lemma requires that the global postcondition implies each of the local postconditions and, furthermore, that the composition of the two local postconditions imply the global postcondition. To understand what these restrictions imply assume that both local postconditions are the same as the global postconditions, which is the case for the top-level theorems of the transformations that we will want to compose. Then for the postcondition it must hold that $Q_G \circ Q_G \supseteq Q_G$. This (semi-idempotency) requirement holds for simple postconditions like $c_1 \leq c_2$, where c_1 is the fuel of the source and c_2 the fuel of the target. But it does not hold for the postconditions required by transformations that reduce the number of steps a program takes. $\lambda_{\texttt{ANF}}$ transformations such as inlining and shrinking reduce the number of steps that the program takes, and therefore we cannot use the transitivity lemma to compose $\lambda_{\texttt{ANF}}$ transformations proved correct with $\boldsymbol{\mathcal{E}}$.

We can also compose the closure-conversion relation with the symmetrical relation, but only if the closure conversion relation comes first. Again we have similar restrictions on the postconditions.

Lemma 5.31 (Composition of $\boldsymbol{\mathcal{E}}_{\texttt{CC}}$ with $\boldsymbol{\mathcal{E}}$)
Assume that $Q_G \supseteq Q_1$, $Q_G \supseteq Q_2$ *and* $Q_1 \circ Q_2 \supseteq Q_G$.

If

- $\boldsymbol{\mathcal{E}}^k_{\mathit{CC}}(\sigma_1, e_1)\ (\sigma_2, e_2)\ \{Q_1; Q_G\}$
- $\forall\ k,\ \boldsymbol{\mathcal{E}}^k(\sigma_2, e_2)\ (\sigma_3, e_3)\ \{Q_2; Q_G\}$

then $\boldsymbol{\mathcal{E}}^k_{\mathit{CC}}(\sigma_1, e_1)\ (\sigma_3, e_3)\ \{Q_1 \circ Q_2; Q_G\}$

We cannot use that lemma since some transformations after closure conversion reduce the number of steps a program takes.

5.3 Compositional Proof Framework

The compositional relational framework presented in this section is based on the observation that adequacy and compatibility with linking are closed under relation composition. That is, if we have two adequate and compatible relations, the composition of these relations will be an adequate and compatible relation. I set up a relation that is defined as the composition of the intermediate logical relations that are used to prove correctness of the individual transformations. Then, the whole pipeline can be easily show to inhabit the composition of these relations, without requiring that the individual relations support vertical compositionality. The composition of the intermediate relations is an adequate and compatible relation, therefore we can obtain the desired correctness result for the pipeline.

An important aspect of the framework is that linking is supported not only for programs that are compiled through the same $\lambda_{\mathtt{ANF}}$ transformation, but also for programs that are compiled by any $\lambda_{\mathtt{ANF}}$ pipeline that is proved correct with respect to the same logical relations. For instance, one can link $\lambda_{\mathtt{ANF}}$ programs that are compiled with optional optimizations with programs that are compiled without any optimization.[10] Or, one could link programs that are compiled with different implementations of closure conversion, as long as they are proved correct with the same logical relation.

The first ingredient of the compositional framework is the transitive closure of the symmetrical logical relation, denoted $\boldsymbol{\mathcal{E}}^+$. The definition of the relation is shown in Figure 5.3. Two expressions e_1 and e_2 are related with $\boldsymbol{\mathcal{E}}^+$ if for all environments σ_1 and σ_2 that are related in the set of free variables of e_1, and the free variables of e_1 are in the domain of σ_1, then the configurations (σ_1, e_1) and (σ_2, e_2) are related. The local and global postconditions are existentially quantified and never exposed. We do however require that they satisfy $\mathsf{PostProperties}$ (which is needed to derive compatibility with linking) and $\mathsf{PostUpperBound}$ (which is needed to derive adequacy).

$$\frac{\begin{array}{c}\mathsf{PostProperties}\ Q_G\ Q\ Q \qquad \mathsf{PostUpperBound}\ Q\\ (\forall\ \sigma_1\ \sigma_2\ k,\ \mathsf{fv}(e_1) \vdash \boldsymbol{\mathcal{C}}^k(\sigma_1,\sigma_2)\ \{Q_G\}\ \Rightarrow\ \mathsf{fv}(e_1) \in \mathtt{dom}(\sigma_1)\ \Rightarrow\\ \boldsymbol{\mathcal{E}}^k(e_1,\sigma_1)\ (e_2,\sigma_2)\ \{Q; Q_G\})\end{array}}{\boldsymbol{\mathcal{E}}^+\ e_1\ e_2}\ \textsc{Step}$$

$$\frac{\boldsymbol{\mathcal{E}}^+\ e_1\ e \qquad \boldsymbol{\mathcal{E}}^+\ e\ e_2}{\boldsymbol{\mathcal{E}}^+\ e_1\ e_2}\ \textsc{Trans}$$

Figure 5.3: The $\boldsymbol{\mathcal{E}}^+$ relation.

Observe that we require not only that the environments are related in the set of free variables of the first expression, but also that these variables are bound in the source environment. This is a necessary precondition of the correctness of some transformations, namely closure conversion and lambda lifting. If the free variables are not present in the environment then the source program might get stuck when the target program does not, or vice versa.

The transitive closure $\boldsymbol{\mathcal{E}}^+$ closure is inhabited by the same program (because $\boldsymbol{\mathcal{E}}$ is reflexive) or any program that has gone through one or more transformations proved correct with $\boldsymbol{\mathcal{E}}$. That practically means that a program that has been (*e.g.*,) shrink-reduced and inlined can be linked with a program that has been uncurried. Or, that

[10] In fact, the only $\lambda_{\mathtt{ANF}}$ transformation that is non-optional and is required for code generation is closure conversion and hoisting of function definitions to the top level.

a program that has been shrink-reduced, inlined and uncurried, can be linked with a program that has not gone through any transformation. This is reflected in the following lemma.

Lemma 5.32 (Linking compatibility of $\boldsymbol{\mathcal{E}}^{+}$)
If

- $\boldsymbol{\mathcal{E}}^{+}\ e_1^{\text{lib}}\ e_2^{\text{lib}}$
- $\boldsymbol{\mathcal{E}}^{+}\ e_1^{\text{client}}\ e_2^{\text{client}}$

Then $\boldsymbol{\mathcal{E}}^{+}\ [x \mapsto e_1^{\text{lib}}]e_1^{\text{client}}\ [x \mapsto e_2^{\text{lib}}]e_2^{\text{client}}$.

It is worth outlining the proof of the above lemma, as it exposes the technique by which composition is achieved. The difficulty is that the two uses of the $\boldsymbol{\mathcal{E}}^{+}$ relation in the assumptions might use different number of transitivity steps, and each of them can use an entirely different postcondition. However, according to Lemma 5.29, compatibility of $\boldsymbol{\mathcal{E}}$ requires the two relations that are composed have the same postconditions. The solution is to horizontally compose each intermediate relation with an identity transformation, what satisfies any postcondition (Lemma 5.26). This is reflected in the following two auxiliary lemmas.

Lemma 5.33 (Linking compatibility of $\boldsymbol{\mathcal{E}}^{+}$ (lib))
If $\boldsymbol{\mathcal{E}}^{+}\ e_1^{\text{lib}}\ e_2^{\text{lib}}$ *then* $\boldsymbol{\mathcal{E}}^{+}\ [x \mapsto e_1^{\text{lib}}]e_1^{\text{client}}\ [x \mapsto e_2^{\text{lib}}]e_1^{\text{client}}$.

Lemma 5.34 (Linking compatibility of $\boldsymbol{\mathcal{E}}^{+}$ (client))
If $\boldsymbol{\mathcal{E}}^{+}\ e_1^{\text{client}}\ e_2^{\text{client}}$ *then* $\boldsymbol{\mathcal{E}}^{+}\ [x \mapsto e_1^{\text{lib}}]e_1^{\text{client}}\ [x \mapsto e_1^{\text{lib}}]e_2^{\text{client}}$.

The two lemmas are proved by induction on the $\boldsymbol{\mathcal{E}}^{+}$ relation. Lemma 5.35 is a direct corollary of the above two lemmas. That means that whenever the client programs are related with this relation using m transitivity steps, and the library programs are related using n transitivity steps, the linked programs will be related using $m+n$ transitivity steps. $\boldsymbol{\mathcal{E}}^{+}$ is also adequate (the statement is omitted).

$$\frac{\begin{array}{c}\mathsf{PostProperties}\ Q_G\ Q\ Q \qquad \mathsf{PostUpperBound}\ Q\\ \boldsymbol{\mathcal{E}}^{+}\ e_1\ e_1'\\ (\forall\ \sigma_1\ \sigma_2\ k,\ \mathsf{fv}(e_1') \vdash \boldsymbol{\mathcal{C}}^{k}(\sigma_1,\sigma_2)\ \{Q_G\}\ \Rightarrow\ \mathsf{fv}(e_1') \in \mathtt{dom}(\sigma_1)\ \Rightarrow\\ \boldsymbol{\mathcal{E}}^{k}_{\mathtt{CC}}(e_1',\sigma_1)\ (e_2',\sigma_2)\ \{Q;Q_G\})\\ \boldsymbol{\mathcal{E}}^{+}\ e_2'\ e_2\end{array}}{\boldsymbol{\mathcal{E}}^{+}_{\mathtt{CC}}\ e_1\ e_2}\ \textsc{Compose}$$

Figure 5.4: The $\boldsymbol{\mathcal{E}}^{+}_{\mathtt{CC}}$ relation.

Using $\boldsymbol{\mathcal{E}}^{+}$ we can state the relation $\boldsymbol{\mathcal{E}}^{+}_{\mathtt{CC}}$ (Figure 5.4) that will be inhabited by the pipeline. Then $\boldsymbol{\mathcal{E}}^{+}$ is the composition of $\boldsymbol{\mathcal{E}}^{+}$, $\boldsymbol{\mathcal{E}}_{\mathtt{CC}}$, and $\boldsymbol{\mathcal{E}}^{+}$. It is inhabited by the closure conversion transformation, preceded and followed by any number of transformations proved correct by $\boldsymbol{\mathcal{E}}$. It is therefore inhabited by the $\lambda_{\mathtt{ANF}}$ pipeline.

As composition of relations that are compatible with linking and adequate, $\boldsymbol{\mathcal{E}}^{+}_{\mathtt{CC}}$ is also compatible with linking and adequate.

Lemma 5.35 (Linking compatibility of $\mathcal{E}^+_{\mathsf{CC}}$)
If

- $\mathcal{E}^+_{\mathit{CC}}\ e_1^{\mathrm{lib}}\ e_2^{\mathrm{lib}}$
- $\mathcal{E}^+_{\mathit{CC}}\ e_1^{\mathrm{client}}\ e_2^{\mathrm{client}}$

Then $\mathcal{E}^+_{\mathit{CC}}\ [x \mapsto e_1^{\mathrm{lib}}]e_1^{\mathrm{client}}\ [x \mapsto e_2^{\mathrm{lib}}]e_2^{\mathrm{client}}$.

Lemma 5.36 (Adequacy of $\mathcal{E}^+_{\mathsf{CC}}$)
Assume that $\mathsf{PostUpperBound}\ Q$.

Then if $\mathcal{E}^+_{\mathit{CC}}\ e_1\ e_2$ *and* $\mathsf{closed}(e_1)$ *we have that* $e_1 \supseteq_B e_2$.

The above framework allows us to derive correctness of separate compilation without additional proof effort. It suffices to prove that each individual transformation inhabits the logical relation. We can derive, as an easy corollary, that any pipeline comprising of transformations proved correct with the logical relations are in the $\mathcal{E}^+_{\mathsf{CC}}$ relation. From Lemma 5.35, which is proved once and for all, we obtain that we can link any programs compiled pipelines that satisfy $\mathcal{E}^+_{\mathsf{CC}}$.

Cross-language setting. The above framework generalizes to a cross-language setting with more than one IR. Conceptually, the closure-conversion logical relation behaves as a cross-language asymmetrical logical relation. If more than one IR were used then we would have to compose all cross-language relations, in the same sequence that are used in the compiler, perhaps adding an intermediate symmetrical relation for the IRs where same-language transformations happen.
3ex

This concludes the definition of the relational proof framework that we use to prove correct the $\lambda_{\mathtt{ANF}}$ transformations. In the next chapter, I present the theorem for each transformation and the top-level theorem of the $\lambda_{\mathtt{ANF}}$ pipeline. In the rest of this chapter, I present related work on relational reasoning and compositional compiler correctness.

5.4 Related Work

In this section, I survey proof techniques that are used in other verified compilers for functional languages as well as related work in compositional compiler correctness and relational reasoning about programs.

5.4.1 Other Verified Compilers for Functional Languages

CakeML is proved correct with a combination of syntactic simulations and logical relations [106, 129]. The CakeML compiler does not provide a separate compilation theorem. However, older versions of the compiler [80] were running in a REPL (read-eval-print loop) that requires the same style of reasoning as linking with programs that are compiled with the same compiler. Therefore, it is likely that a separate compilation theorem for programs compiled with exactly the same CakeML compiler could be obtained. Additional effort would be required to port the CakeML's correctness theorem to stronger notions of compositional compiler correctness.

CakeML's ClosLang optimizations [106] are verified using a logical relation. In order to show divergence preservation, the logical relation forces the step index to be the *same* for both programs. This enables showing that if the source program diverges so does the second program, but it forces the two programs to use the same amount of fuel. To enable verification of transformations that reduce the amount of steps that a program takes, the language has a special instruction, `Tick`, that has no observable effect but only decrements the fuel value. Programs that increase the amount of steps cannot be shown related in this model. This would be detrimental for the $\lambda_{\texttt{ANF}}$ pipeline since a lot of transformations introduce administrative redexes that are removed by subsequent transformations. Furthermore, and additional `Tick`-erasure pass must be proved correct before the final code generation phase. CertiCoq's proof framework overcomes these issues by allowing the steps of the two programs to be related by an arbitrary relation, the postcondition, and formally characterizing the postconditions that allow to derive divergence preservation. With this model we can relate programs that both reduce and increase the amount of steps, without the need for a `Tick` instruction.

Œuf [102] is proved correct using syntactic simulations. Unlike CertiCoq, Œuf's reification phase (which is currently unverrified) is proved correct using denotational semantics: the denotation of the reified term is equivalent of the original Gallina program.

Lambda Tamer [34] is proved correct using syntactic simulations. The novelty of the compiler lies in that the intermediate representations are formalized parametric higher-order abstract syntax (PHOAS). The correctness theorem of the compiler does not support separate compilation.

5.4.2 Compositional Compiler Correctness

SepCompCert. In terms of the strength of the linking theorem, SepCompCert [73] is the most closely related work. SepCompCert supports verified linking of programs that are compiled through CompCert using different sets of optional optimizations. SepCompCert achieves that by forcing transformations to be in lockstep in each of the linked pipelines, by padding the pipeline that does not perform an optional optimization with the identity transformation. The correctness statement of each optional optimization must be modified such that either the initial simulation holds between the source and target or the target is the same as the source. Therefore the new statement is satisfied by both the optional and the identity transformation. Then the simulations that are used to verify each pipeline are in lockstep and linking can be proved correct in the same way that linking programs compiled through exactly the same pipeline can be proved correct.

The framework presented in this chapter follows a similar idea as SepCompCert: to show Lemma 5.35 we essentially pad with the identity transformation by using the reflexivity property of the logical relation. However, the CertiCoq framework is stronger than SepCompCert in that provides a compositional compiler correctness theorem; SepCompCert's theorem is formed in terms of whole-program correctness. That has a few implications. First, the CertiCoq framework requires zero modification of correctness statements and their proofs. Second, in SepCompCert whenever a new optional transformation is added or some transformations are reordered, a new linking theorem must be proved (or at the very least the proof of the old linking theorem must modified), so that there is a theorem that covers each possible linking combination. In our case, the only thing that we need to do is to show that each pipeline we want to link with inhabits the $\boldsymbol{\mathcal{E}}^{+}_{\mathsf{CC}}$ relation. Then the linking theorem follows as an direct corollary. Lastly, the linking theorem of SepCompCert can be applied only for optional transformations that can be replaced with the identity transformation. This is not true for closure conversion. With our framework we can link two programs compiled with different closure conversion transformations granted that they are both in the $\boldsymbol{\mathcal{E}}_{\mathsf{CC}}$ relation.[11] That would not have been possible with the SepCompCert proof technique.

CompCompCert. Compositional CompCert [127] supports a general notion of linking that allows the source program to be linked with programs that are written in any of the CompCert languages. This is achieved using *interaction semantics* that defines a protocol that can model cross-language function calls. Interaction semantics require that the languages involved have the same memory model and value representation, and therefore it is not directly applicable to functional languages where closure conversion necessarily changes the value representation of functions.

CompCertM [122] introduces the RUSC (Refinement Under Self-related Contexts) relation, a lightweight proof technique for compositional compiler correctness. It supports a quite general notion of compositional compiler correctness, similar in strength with CompCompCert. Given a set of adequate and compatible relations, two programs are related under RUSC if the target program refines the behavior of the source under any context that is self-related by all the relations in the given set. The RUSC relation is adequate and has both vertical and horizontal compositionality. To link the output of two compilers proved correct with a set of adequate and horizontally composable relations it suffices to form a RUSC with the set of relations that are used to prove the compilers correct. Then correctness of linking follows by the fact that the two compilers are in RUSC and that RUSC can be composed horizontally. As in the case of interaction semantics, RUSC can only be applied to languages that share

[11] and our $\boldsymbol{\mathcal{E}}_{\mathsf{CC}}$ relation is quite general: it would permit, for example, sophisticated closure representations such as Shao and Appel's safe-for-space hybrid flat/linked closures [120].

the same notion of values and memory models, and therefore it is not clear how it could be applied to functional languages.

Parametric bisimulations and inter-language simulations. Hur *et al.* [65] develop *parametric bisimulations* (PBs) that combine aspects of Kripke logical relations and bisimulations to obtain a relational framework that can be transitively composed. The model avoids the use of step-indexing (which hinders transitivity in logical relations) by using a coinductively defined relation. A crucial technical novelty of the construction is the notions of *local* and *global knowledge. Local knowledge* captures the terms that are currently shown to be equivalent, while *global knowledge* captures all other terms that are known to be equivalent. In this relational model, two functions are related if they map arguments related by global knowledge to results that are related by local knowledge. This allows functions to be linked not only with code that satisfies the local knowledge, but rather the more general notion of global knowledge that captures, *e.g.*, programs compiled with a different proved-correct compiler. Crucially, the relation is parametric on the notion of *global knowledge* that can be instantiated with various relations.

Building on top of PBs, Neis *et al.* [104] develop *parametric inter-language simulations (PILS)* and use them in a inter-language setting to verify the Pilsner multi-pass compiler from an ML-like source language to assembly, and also Zwickel, a simple one-pass compiler between the same languages. The proof technique allows them to derive that programs compiled with Pilsner can be linked with other programs compiled with Pilsner, with programs compiled with Zwickel, and also with handwritten assembly that refines some source-level code. RTS and PILS do not admit the eta-conversion rule that is crucial for functional-language compilers (for example, CertiCoq's uncurrying transformation is based on eta-expansion). A solution has been suggested [77], but has not been incorporated into PILS. Compared to CertiCoq's logical relations, PILS support a stronger notion of compositional compiler correctness, they are, however, more technically involved. According to the authors [104, Section 4], the metatheory of RTS (including the transitivity proof) and PILS is quite complex and requires a lot of effort. A PILS-based relation could, in principle, be used to verify CertiCoq, if the solution that makes eta-conversion admissible were incorporated into the model.[12]

Multi-language Semantics. Perconti and Ahmed [109] use the notion of multi-language semantics[91] in order to support source-independent linking that poses no restriction on the programs that are linked. They define a two-pass compiler from a high-level typed language to a low-level language, that uses some intermediate language. They then define a language where the source, intermediate, and target languages can be embedded and can be used to model interoperability between them. This framework allows linking source programs with arbitrary target code that has the right (source) type. The multi-language nature of the language allows to prove

[12] The solution [77] is to allow stuttering steps by using some notion of fuel that specifies how many stuttering steps a program may take between two actual steps.

contextual equivalence between programs defined in different languages. In particular, the linking theorem asserts that linking a source program with an embedded target program with the right type is contextually equivalent with linking the compiled source program with the target program. Contextual equivalences are proved using logical relations. This technique scales beyond separate compilation and allows to model foreign-function interfaces, which is outside the scope of the framework presented in this chapter.

5.4.3 Relational Reasoning for Program Resources

Relational cost analysis [35, 30, 29, 113] is concerned with devising ways that to relate the execution cost of two programs by means of static analysis. Type systems for relational cost analysis are used to derive precise bounds on the difference of the execution cost of the two programs. Typechecking [31] can be used to mechanize these proofs. The framework of this thesis also allows to establish bounds on the resource consumption of two programs (together with relatedness of the results). Whereas relational cost analysis is targeted at source level verification of relational resource bounds, the framework of this chapter is targeted at showing that the resource consumption of programs, as defined by the semantics of the language, is preserved by program transformations.

The soundness of relational type systems for cost analysis is often demonstrated using a binary logical relation (indexed by types) that asserts that the bound obtained by means of typing indeed holds for the execution time of the two programs. For example the logical relation by Çiçek *et al.* [29] is parameterized by a bound that it is enforced on the difference of the execution times of this program. The logical relations presented in this chapter, albeit not indexed by types, both take inspiration from, and generalize the logical relations in [35, 30, 29]. The machinery presented in this thesis to track runtime and resource information and capture their relationship could be likely reused in the context of relational resource analysis to show soundness of relational resource analysis methods.

5.5 Conclusion

In this chapter, I presented the relational framework that is used in the verification of the $\lambda_{\texttt{ANF}}$ pipeline. The framework is based on step-indexed logical relations and makes a few novel technical contributions that tackle compositional compiler correctness and divergence preservation. First, the logical relation is parameterized by a novel notion of relational postcondition that is used to relate the abstract resource consumption of the two programs. The resource consumption is captured by semantics of the program using abstract trace and fuel monoids that can be instantiate in different ways. This machinery allows us to derive divergence preservation for programs that are related by the logical relation at certain postconditions that satisfy some property. Second, I show how such logical relation proofs can be composed to derive a compositional compiler correctness theorem, providing a novel, lightweight solution to a challenging problem. This compositional compiler correctness theorem can be applied to programs that are separately compiled with compilers that go through the same series of intermediate languages.

It would be interesting to examine whether this restriction can be lifted by so that the framework can be used to prove correct compilers that need not have the same internal intermediate representations. If possible, this would most likely require providing back translations between adjacent intermediate representations and proving them correct.

Another useful extension would be to define the logical relation in a language-generic way so that the same relation can be used to verify other parts of CertiCoq, extending the end-to-end formal guarantees. Language-generic logical relations exists in the literature [64] and perhaps the same ideas could be used for the logical relations presented in this chapter.

Chapter 6

Correctness of Transformations

In this chapter, I apply the relational proof framework of the previous chapter to show that $\lambda_{\texttt{ANF}}$ transformation are correct. I also give an account of the mechanized proof development showing the top-level theorems as they are stated and proved in Coq.

6.1 Correctness of $\lambda_{\texttt{ANF}}$ transformations

In this section I give a mostly high-level account the individual proofs for the $\lambda_{\texttt{ANF}}$ transformations. For the correctness of each transformation we assume that the program is well scoped. Therefore we also need to show that each transformation preserves well-scopedness.

The proofs of many transformations in the $\lambda_{\texttt{ANF}}$ pipeline are structured in layers. First, a relational specification is proved to inhabit the logical relation. Then a second theorem is proved that the implementation of the transformation satisfies its relational specification. This allows us to separate the details of semantics preservation proof from the details of implementation.

The individual top-level theorems have generally the same shape (with the exception of shrinking, which is discussed later in this section). In particular we show the following statement.

$$\begin{array}{ll}\forall\ e\ e', & \mathsf{Trans}\ e\ e' \ \Rightarrow \\ & \mathsf{well\_scoped}(e) \Rightarrow \\ & \mathsf{well\_scoped}(e') \wedge \\ & (\forall\ \sigma_1\ \sigma_2\ k,\ \mathsf{fv}(e) \vdash \boldsymbol{\mathcal{C}}^k(\sigma_1, \sigma_2)\ \{Q_G\}\ \Rightarrow \\ & \quad \mathsf{fv}(e) \in \texttt{dom}(\sigma_1)\ \Rightarrow \\ & \quad \boldsymbol{\mathcal{E}}^k(e, \sigma_1)\ (e', \sigma_2)\ \{Q; Q_G\})\end{array}$$

$\mathsf{Trans}\ e\ e'$ denotes that e' is the translation of e. The relations $\boldsymbol{\mathcal{E}}_{\texttt{CC}}$ and $\boldsymbol{\mathcal{C}}_{\texttt{CC}}$ are used in the case of closure conversion. The precondition $\mathsf{fv}(e) \in \texttt{dom}(\sigma_1)$ is needed only in the case of lambda lifting and closure conversion. In both transformations the translated program will use free variables before their first use in the source program, either to construct a closure environment or to pass as parameters. If these variables where not in the domain of the source environment, then target program would get stuck before the source did, invalidating the correctness theorem.

The main difference between the correctness theorems of each transformation is the postcondition, which is generally different for each transformation. In the remainder of this section I will refer to the particular details of the correctness theorems of each transformation including the postconditions that are proved.

Concrete fuel and trace monoids Recall from section 3.3.2 that the concrete monoid we are using for fuel is $\langle \mathbb{N}, +, 0\rangle$ with $\langle e\rangle \overset{\text{def}}{=} 1$. For the trace monoid we are using $\langle \mathbb{N}\times\mathbb{N}, +, (0,0)\rangle$ with generator:

$$\begin{array}{ll}
\langle \texttt{let}\ x = \texttt{C}(\overline{y})\ \texttt{in}\ e\rangle \overset{\text{def}}{=} (1,0) & \langle \texttt{let}\ x = y.i\ \texttt{in}\ e\rangle \overset{\text{def}}{=} (1,0) \\
\langle \texttt{case}\ y\ \texttt{of}\ [\texttt{C}_i\ \rightarrow\ e_i]_{i\in I}\rangle \overset{\text{def}}{=} (1,0) & \langle \texttt{fun}\ f\ \overline{x} = e_1\ \texttt{in}\ e_2\rangle \overset{\text{def}}{=} (1,0) \\
\langle \texttt{let}\ x = f\ \overline{y}\ \texttt{in}\ e\rangle \overset{\text{def}}{=} (0,1) & \langle f\ \overline{y}\rangle \overset{\text{def}}{=} (0,1) \\
\langle \texttt{ret}(x)\rangle \overset{\text{def}}{=} (1,0) &
\end{array}$$

.

Intuitively, the fuel monoid counts a unit of time for each execution step. The trace monoid keeps a separate count of the nonapplication steps (first component of the pair) and the application steps (second component of the pair)

6.1.1 Inlining

In the proof of inlining follows from the compatibility lemmas for each case except of the application cases. For the application cases we need to prove two logical relation lemmas for function inlining. As usual, first we need to state the assumptions that we will impose on the postcondition.

For the tail-call case, we need to be able to derive that the postcondition holds if an application is evaluated only by the source program. In this case the target program does not perform the application step, since the application was statically evaluated.

Definition 6.1 (Postcondition (tail-call inlining))
PostAppInline Q_1 Q_2 *holds iff whenever*

$$Q_1(c_1, t_1)\ (c_2, t_2)$$

we also have

$$Q_2(c_1\langle+\rangle_{\mathcal{F}}\langle f_1\ \overline{x_1}\rangle_{\mathcal{F}}, t_1\langle+\rangle_{\mathcal{T}}\langle f_1\ \overline{x_1}\rangle_{\mathcal{F}})\ (c_2, t_2)$$

The let-bound application case is more involved. Because of the extra renormalization steps that are required to inline let-bound function in ANF (Section 4.3), the target program can omit one or two steps for each inlined application. As in the tail-call inlining case, the target will not perform the step that applies the function.

Additionally, if the body of the inlined function ends with return, then this instruction will be omitted after inlining (recall the definition of inlining in Figure 4.2).

For let-bound application lemma, we also need to distinguish two cases: one for when the function call terminates and one for when the function call runs out of time. The compatibility rules for the postconditions are shown below.

Definition 6.2 (Postcondition (let-bound call inlining))
PostLetAppInline Q_1 Q_2 Q_3 *holds iff whenever*

- $Q_1(c_1, t_1)\ (c_2, t_2)$ *or* $Q_1(c_1, t_1)\ (c_2\langle+\rangle\langle\texttt{ret}(y)\rangle, t_2\langle+\rangle\langle\texttt{ret}(y)\rangle)$, *and*
- $Q_2(c_1', t_1')\ (c_2', t_2')$

we also have
$Q_3(c_1\langle+\rangle\langle e_1\rangle\langle+\rangle c_1', t_1\langle+\rangle\langle e_1\rangle\langle+\rangle t_1')\ (c_2\langle+\rangle\langle e_2\rangle\langle+\rangle c_2', t_2\langle+\rangle\langle e_2\rangle\langle+\rangle t_2')$.

Definition 6.3 (Postcondition (let-bound call inlining OOT))
PostLetAppInlineOOT Q_1 Q_2 *holds iff whenever*

$$Q_1(c_1, t_1)\ (c_2, t_2) \text{ or } Q_1(c_1, t_1)\ (c_2\langle+\rangle\langle\texttt{ret}(y)\rangle, t_2\langle+\rangle\langle\texttt{ret}(y)\rangle)$$

we also have

$$Q_2(c_1\langle+\rangle\langle e_1\rangle, t_1\langle+\rangle\langle e_1\rangle)\ (c_2\langle+\rangle\langle e_2\rangle, t_2\langle+\rangle\langle e_2\rangle).$$

In the above rules, the postcondition Q_1 is the postcondition that holds for the function body before and after translation. We distinguish two cases for Q_1: one where the fuel of the target program is c_2 and one where the fuel of the target is $c_2\langle+\rangle\langle\texttt{ret}(y)\rangle$. The former covers the case when the function body ends with a tail call, and therefore the fuel of the function body is the same before and after inlining. The latter covers the case when the function body ends with return, that will be removed during inlining, and therefore the function body before inlining will perform one extra step (captured by $\langle\texttt{ret}(y)\rangle$).

Using the above, we can now state the lemmas required for inlining. For tail calls, we have the following lemma.

Lemma 6.4 (Inlining (tail-cal))
Assume that PostZero Q_2 *and* PostAppInline Q_1 Q_2.

If

- $\forall\ (m < k)\ g\ \overline{y}\ e_1\ \overline{v}$,
  $\sigma_1(f) = \texttt{fun}\ g\ \overline{y} = e_1 \ \Rightarrow\ \sigma_1(\overline{x}) = \overline{v}\ \Rightarrow$
  $\boldsymbol{\mathcal{E}}^m(e_1, \sigma_1[\overline{y} \mapsto \overline{v}])\ (e_2, \sigma_2)\ \{Q_1; Q_G\}$

Then $\boldsymbol{\mathcal{E}}^k(f_1\ \overline{x_1}, \sigma_1)\ (e_2, \sigma_2)\ \{Q_2; Q_G\}$.

Similarly for inlining of let-bound applications we prove the following lemma.

Lemma 6.5 (Inlining (let-bound cal))
Assume that $\mathsf{PostZero}\ Q_2$, $\mathsf{PostLetAppInline}\ Q_1\ Q_2\ Q_3$
and $\mathsf{PostLetAppInlineOOT}\ Q_1\ Q_3$.

If

- $\forall\ (m < k)\ g\ \overline{y}\ e_1\ \overline{v},\ \sigma_1(f) = \texttt{fun}\ g\ \overline{y} = e_1\ \Rightarrow\ \sigma_1(\overline{x}) = \overline{v}\ \Rightarrow$
  $\boldsymbol{\mathcal{E}}^m(e_1', \sigma_1[\overline{y} \mapsto \overline{v}])\ (e_2', \sigma_2)\ \{Q_1; Q_G\}$
- $\mathsf{inline\_letapp}(e_2', x) = \texttt{Some}(\mathcal{E}, x')$
- $\forall\ \sigma_1\ \sigma_2,\ \mathsf{fv}(e_1) \vdash \boldsymbol{\mathcal{C}}^k(\sigma_1, \sigma_2)\ \{Q\}\ \Rightarrow$
  $\boldsymbol{\mathcal{E}}^k(e_1, \sigma_1[\overline{y} \mapsto \overline{v}])\ (e_2\{x'/x\}, \sigma_2)\ \{Q_2; Q_G\}$

Then $\boldsymbol{\mathcal{E}}^k(\texttt{let}\ x_1 = f_1\ \overline{x_1}\ \texttt{in}\ e_1, \sigma_1)\ (\mathcal{E}[e_2], \sigma_2)\ \{Q_3; Q_G\}$.

Let us now look at the postcondition that needs to be established for inlining. We need to find an upper bound for the execution steps of the source in terms of the steps of the target, but the difficulty is that the steps of the target are fewer than the steps of the source by at most two steps for each application that is executed in the source and not the target. Let G be the total number of inlining steps during translation, and L the number of remaining inlining steps at the current program point. To find an upper bound, consider that for each function body that is evaluated in the target there might be at most G inlining steps that have happened inside this function body, and therefore $2 * G$ extra steps in the source (G for the number of removed calls, and G for the number of removed returns). In addition we shall account for $2 * L$ steps for the function body that is currently being translated. Therefore, we want to establish the relation $c_1 \leq c_2 + 2 * G * c_2 + 2 * L$ for the execution steps of the two program. Unfortunately, this upper bound is too coarse grained and we cannot show that it satisfies the postcondition requirements for let-bound application. We prove the following more fine-grained bound using the number of total application steps that happen in the source program, which is tracked by the trace.

$$\begin{aligned}\mathsf{PostInline}\ G\ L\ (c_1, (t_1, t_1^{app}))\ (c_2, (t_2, t_2^{app}))\ \overset{\text{def}}{=}\ & c_1 \leq 2 + 2 * G * t_1^{app} + 2 * L\ \land \\ & t_1^{app} <= t_2^{app} + 2 * G * t_2^{app} + L\ \land \\ & t_2 + t_1^{app} = c_2.\end{aligned}$$

This bound satisfies all the required rules and implies the simpler bound that suffices to show divergence preservation. Being able to state this more precise upper bound is the reason why tracing of application steps is required.

6.1.2 Shrinking

The shrinking transformation is proved correct with respect to a rewrite system [22]. Shrink-reduction steps are modeled as a system of local rewrites that are shown to inhabit the logical relation. The postcondition is the same as in the inlining case, but this time G is 1 (since we are considering only one rewrite step each time) and L is 1 before the rewrite step and 0 after the rewrite step. The Lemma 6.4 and 6.5 are also used in the shrinking proof along with similar lemmas for the other shrink-reduction steps.

The shrinking program is shown to be in the transitive, congruent closure of the shrink-rewrite system. Savary Bélanger and Appel [22], do not show divergence preservation for shrinking. That permits them to use a relational model that does not require a postcondition and, as a result, it satisfies the transitivity property (Lemma 5.30). Therefore, we can show that the shrink-reduction program is in $\mathcal{E}$ (with a trivial postcondition). But when we prove divergence preservation, we use a postcondition that does not satisfy the requirements for transitivity, therefore we can no longer prove that the shrinking transformation is in $\mathcal{E}$. Instead, we show that the shrinking program is in the $\boldsymbol{\mathcal{E}}^+$ relation.

6.1.3 Uncurrying

The proof of uncurrying is also layered: we show that performing an uncurry rewrite step is in the logical relation and that the uncurrying transformation is in the transitive, congruent closure of the rewrite step. For uncurrying however, the upper bound that is proved is the following.

$$\mathsf{SimplePost}\ (c_1, (t_1, t_1^{app}))\ (c_2, (t_2, t_2^{app})) \ \stackrel{\text{def}}{=}\ c_1 \leq c_2$$

The SimplePost upper bound satisfies the requirements for transitivity and therefore uncurrying can be shown to inhabit $\mathcal{E}$.

6.1.4 Closure Conversion, Hoisting, and Lambda Lifting

The proofs of closure conversion, hoisting, and lambda lifting are similar in style. First, a relational specification of the transformation is proved to inhabit the logical relation, and then the transformation is proved to inhabit the relational specification. For closure conversion and lambda lifting the SimplePost postcondition is used. For hoisting, which reduces the number of steps, a different bound is used. Hoisting will remove nested function definitions (each in incurring unary execution cost) and will move them to the top-level in the same bundle of (potentially) mutually recursive functions (also incurring unary cost). Let again G be the number of total number of hoisted function definitions and L the number of remaining number of hoisted function definitions in the current expression. The following bound is proved for the hoisting transformation.

$$\mathsf{HoistingBound}\ G\ L\ (c_1, (t_1, t_1^{app}))\ (c_2, (t_2, t_2^{app})) \ \stackrel{\text{def}}{=}\ c_1 \leq c_2 + 2 * G * c_2 + L$$

6.2 Top-level Theorem for $\lambda_{\texttt{ANF}}$

Using the correctness proofs of individual transformations we can show that the $\lambda_{\texttt{ANF}}$ pipeline is in the $\mathcal{E}^{+}_{\mathsf{CC}}$ relation. Let $\mathsf{compile}$ be the $\lambda_{\texttt{ANF}}$ compilation function. It receives two arguments: *opt*, which determines which optimizations will be performed, and the source program e. Technically, all optimizations other than closure conversion and hoisting can be disabled. We obtain the following theorem.

Theorem 6.6 (Top-level Theorem for $\boldsymbol{\lambda}_{\texttt{ANF}}$)

$$\begin{aligned} \forall\ \mathit{opt}\ e\ e',\ \ & \mathsf{compile}\ \mathit{opt}\ e = e' \ \Rightarrow \\ & \mathsf{well\_scoped}(e) \Rightarrow \\ & \mathsf{well\_scoped}(e')\ \wedge\ \mathcal{E}^{+}_{\mathsf{CC}}\ e\ e'. \end{aligned}$$

Now assume that we use $\mathsf{compile}$ to compile the well-scoped programs e_{lib} and e_{client} with different sets of optimizations opt_1 and opt_2. From the above theorem and Lemma 5.35 (compatibility with linking) we can derive the following statement about linking the two programs.

$$\mathcal{E}^{+}_{\mathsf{CC}}\quad ([x \mapsto e_{\mathrm{lib}}]e_{\mathrm{client}})\quad ([x \mapsto \mathsf{compile}\ \mathit{opt}_1\ e_{\mathrm{lib}}](\mathsf{compile}\ \mathit{opt}_2\ e_{\mathrm{client}}))$$

Therefore from Lemma 5.36 (adequacy), we obtain that linking the two target programs refines the behavior of linking the two source programs, regardless of what optimizations were used to compile them.

$$([x \mapsto e_{\mathrm{lib}}]e_{\mathrm{client}})\ \supseteq_{\mathsf{B}}\ ([x \mapsto \mathsf{compile}\ \mathit{opt}_1\ e_{\mathrm{lib}}](\mathsf{compile}\ \mathit{opt}_2\ e_{\mathrm{client}}))$$

6.3 Coq Proof Development

The definitions and proofs of the previous chapter are all mechanized in the Coq proof assistant. In this section, I give the Coq statements of the most important top-level theorems. At the time when this thesis is submitted, the dead parameter elimination transformation is not yet proved correct. The top-level theorem disables this transformation in the pipeline.

The proofs of the $\lambda_{\texttt{ANF}}$ pipeline are constructive and do not assume any classical logic axioms.

6.3.1 Specification

The actual $\lambda_{\texttt{ANF}}$ transformations are partial functions; programs that may fail on certain inputs. $\lambda_{\texttt{ANF}}$ transformations take as input a source term and a compilation state and return a either a target term or an error message and a new compilation state. Since most of the transformations need to generate fresh binders, the compilation state provides a pool for fresh variable names that is threaded through the program. $\lambda_{\texttt{ANF}}$ transformations are of type `anf_trans = exp -> comp_data -> error exp * comp_data`, where exp is the type of $\lambda_{\texttt{ANF}}$ terms and comp_data the type of the compilation state. The type error is a sum type with two variants:

```
Inductive error (A : Type) : Type:=
  Err : string -> error A | Ret : A -> error A
```

A transformation might return a result (Ret e') or it may fail returning, an error message (Err s). In both cases, the transformation returns a new compilation state (the compilation state tracks debugging information that can be useful when a transformation fails).

The top-level correctness specification for transformations states that if the input is a well-scoped term and the next available variable is *fresh* (*i.e.*, larger than all the variables—free or bound—in the source program)[1] the transformation is returns a non-error result, the target term is also well-scoped, the new next available variable is fresh in the with respect to the target term, and the source and target term are in the $\boldsymbol{\mathcal{E}}^{+}$ relation (or in the $\boldsymbol{\mathcal{E}}^{+}_{\mathsf{CC}}$ relation in the case of closure conversion).

The following Coq definition captures correctness of "identity" transformations (*i.e.*, transformations that are optional and can be replaced with the identity transformation).

```
  Definition correct (trans :  anf_trans):=
    ∀ e c,
      well_scoped e ->
      max_var e 1 < next_var c ->
     ∃ (e' : exp) (c' : state.comp_data),
        trans e c = (Ret e', c') ∧
        well_scoped e' ∧
        max_var e' 1 < state.next_var c' ∧
        preord_exp_n e e'.
```

In the above, trans is the transformation, next_var c is the next fresh variable of the compilation state c, max_var e is the maximum variable in c, and preord_exp_n is the $\boldsymbol{\mathcal{E}}^{+}$ relation.

We can prove that if two transformations satisfy correct, so does their sequential composition.

```
  Lemma correct_compose (t1 t2 : anf_trans) :
    correct t1 ->
    correct t2 ->
    correct (fun e => e <- t1 e;; t2 e).
```

In the above x <- m ; ; p is notation for the monadic bind bind m (fun x => p) for the monadic type constructor fun A => comp_data -> error A * comp_data.

Similarly, we define correct_cc to capture the correctness of the closure conversion transformation, and any pipeline that includes closure conversion.

[1] Variables are represented as positive numbers. Fresh variables are generated by keeping track of the next available variable and increasing its value by one whenever a new name is needed.

```
Definition correct_cc (trans : anf_trans):=
 ∀ e c,
  well_scoped e ->
  max_var e 1 < next_var c ->
  ∃ (e' : exp) (c' : state.comp_data),
   trans e c = (Ret e', c') ∧
   well_scoped e' ∧
   max_var e' 1 < next_var c' ∧
   R_cc_exp e e'.
```

This definition is similar to `correct` but it uses the relation `R_cc_exp` which is the $\mathcal{E}^{+}_{\texttt{CC}}$ relation. We can prove lemmas about the sequential composition of transformations that satisfy `correct` and `correct_cc`. These are stated below.

```
Lemma correct_cc_compose_l (t1 t2 : anf_trans) :
 correct cenv t1 ->
 correct_cc cenv clo_tag t2 ->
 correct_cc cenv clo_tag (fun e => e <- t1 e;; t2 e).
```

```
Lemma correct_cc_compose_r (t1 t2 : anf_trans) :
 correct_cc cenv clo_tag t1 ->
 correct cenv t2 ->
 correct_cc cenv clo_tag (fun e => e <- t1 e;; t2 e).
```

6.3.2 Top-level Theorem

For each transformation we prove that it is either `correct` or `correct_cc`. The top-level theorems for the pipeline is obtained by composing these proofs with the lemmas presented above. The following theorem is proved for the $\lambda_{\texttt{ANF}}$ pipeline.

```
Theorem anf_pipeline_correct opts :
 dead_param_elim opts = false ->
 correct_cc cenv clo_tag (anf_pipeline opts).
```

The pipeline (`anf_pipeline`) takes as input a parameter (`opts`) that determines which transformations are enabled. In the theorem we require that the dead parameter elimination transformation is disabled because it is not yet proved correct.

6.3.3 Proof Artifact

The sources of the CertiCoq compiler can be assessed online at https://github.com/PrincetonUniversity/certicoq. At the time when this thesis is written[2] the toplevel theorems presented in this section are defined in the file https://github.com/PrincetonUniversity/certicoq/blob/master/theories/L6_PCPS/toplevel_theorems.v. In the same directory, there are also the definition of the $\lambda_{\texttt{ANF}}$ language, the semantics, the logical relation framework, the implementation the $\lambda_{\texttt{ANF}}$ transformations and the corresponding proofs.

[2] Which corresponds to commit hash `d8afa96` in the code repository.

Chapter 7

Space Safety

In this chapter I consider an extension of the proof framework that I described in the previous chapter. The extension makes use of the postcondition machinery of the logical relation to establish upper bounds on the running time and space of programs. To that end I give a new semantics to the intermediate language that formalizes the memory model of the language. I use this framework to show that the closure conversion transformation is both functionally correct and safe for time and space. I also show that the garbage collection strategy of CertiCoq is safe for space. The formalization presented in this chapter is only about the CPS subset (henceforth $\lambda_{\texttt{CPS}}$) of the $\lambda_{\texttt{ANF}}$ intermediate representation that I have used in the previous chapters. The reason for this is chronological as this framework was developed before the intermediate representation was extended to ANF. Although the formalization has not been ported to ANF, at the end of this chapter I outline how this could be done.

7.1 Introduction

Formally verified compilers [85, 80, 104] guarantee that the compiled executable behaves according to the specification of the source language. Most of the times this specification is limited to the result of the computation, as the correctness statement only specifies the extensional behavior of the program. But programmers also expect compilers to preserve programs' *intensional* properties, such as resource consumption, and failure to do so may result in performance and security leaks. At the same time, static cost analysis frameworks enable programmers to formally reason about the running time [70, 62, 137] and memory usage [132, 6, 62] of programs. But a compiler that fails to preserve resource consumption renders source-level cost analysis useless. There are few examples in the literature of program transformations certified with respect to resource consumption [40, 100], and most are limited to running time. In this chapter, I develop a general proof framework, based on logical relations, that supports reasoning about preservation of resource consumption. Inspired by a well-known example of space-safety failure—the (once widely used) linked closure conversion algorithm—I apply this framework to show that flat closure conversion is safe with respect to both time and space.

Closure conversion is used to implement static scoping in languages with nested functions: a program with nested lambdas that may reference variables nonlocal to their definition is transformed to a flat-scope program in which lambdas do not have free variables, and are packaged together with their environment, that contains the values of their free variables, to form a *closure*. Designers of optimizing compilers try to optimize the closure data structures for creation time, access depth of variables, and space usage, and a standard optimization technique is to share parts of the closure environment across multiple closures. If a closure, however, contains variables of different future lifetimes (some of which may be pointers to large data structures) then the garbage collector cannot reclaim the data until the entire closure-pair is no longer accessible; this can increase the program's memory use by an amount *not bounded by any constant factor* [10]. Closure conversion that does not increase the space (respectively, time) usage of a garbage-collected program (by more than a constant factor per program) is called *safe for space* (respectively, safe for time). In fact, shared environment representations that may leak space are still employed: the JavaScript V8 engine's environment sharing strategy, based on linked environment representations, is not safe for space [46]. Standard flat closures are safe for space, but not necessarily optimal for creation time. More efficient safe-for-space closure conversion algorithms exist [121, 120], but no one has attempted to formally reason about space safety of closure conversion.

The difficulty is, to reason formally about space consumption in garbage-collected programs, it is not enough to account for the allocated heap cells during execution. One must explicitly reason about the number of cells simultaneously live in the heap at any point during the execution of the source program, and show that this is preserved for the transformed program. Minamide [100] employs this technique to prove space preservation of the CPS transformation. Using a simulation argument, he shows that the maximum size of the reachable heap, *i.e.*, the ideal amount of space required for a program's execution, is preserved by the transformation. In closure conversion this is more challenging as it changes the shape of a program's heap data structures and lifetime much more than CPS conversion. Furthermore, we are also aiming at not just showing that the ideal space usage is preserved, but to connect the idealized space usage model (assumed by the source cost model) with a model closer to the actual implementation, by accounting for the size of the whole heap (not just the reachable part) and explicitly modeling calls to the garbage collector.

The proof uses a standard technique in proving semantics preservation: logical relations. The main technical novelty is that the logical relation imposes pre- and postconditions on the related programs. I use them to establish the (time and space) resource bounds simultaneously with functional correctness, by showing that the input and output programs of the transformation inhabit the logical relation. I show how to overcome several technical difficulties associated with logical relations. The presence of garbage collection complicates Kripke monotonicity, which states that whenever two values are related, they remain related for future states of a program's execution. Invoking a garbage collector causes heaps to not only grow during execution, but also shrink and become renamed (in the case of copying garbage collectors). I overcome this by explicitly quantifying over all future heaps in our logical relation definitions (as is common in Kripke logical relations) for the right notion of "future". I also explain why pre- and postcondition monotonicity (which in Hoare logic enables compositional reasoning with the use of weakening, strengthening, and frame rules) does not hold directly for our logical relations, and how it is possible to restore it. Finally, I show how the logical relation can be used to prove that divergence and space consumption of diverging programs is preserved (a program can run indefinitely in a bounded memory).

This is the first proof that a closure conversion transformation is safe for space. As in the previous chapters, the result is fully mechanized in the Coq proof assistant. The rest of the chapter is structured as follows. In Section 7.2 I give an overview of closure conversion and different closure environment representations, explaining why linked environments fail to preserve asymptotic complexity. In Sections 7.5 and 7.6 I give the formal definition of the language, its semantics, and the closure-conversion transformation. In Section 7.7, I describe the logical relation framework and in Section 7.8 I apply it to prove correctness of the closure conversion transformation. I conclude with related and future work (Sections 7.9 and 7.10).

7.2 Closure Representation

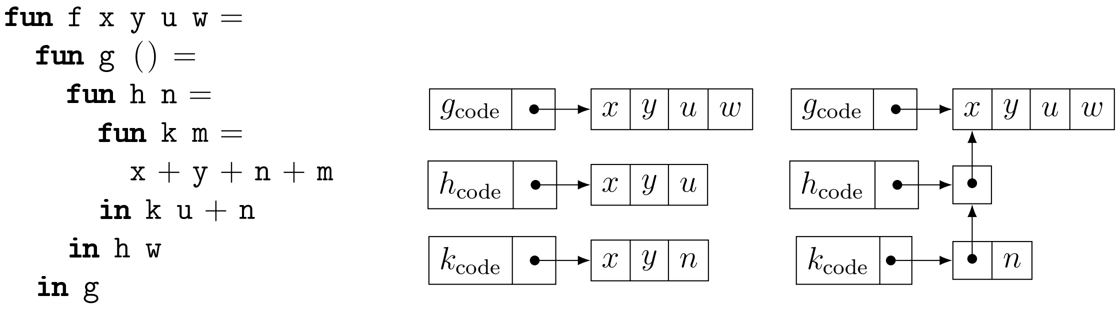
In functional languages like ML, nested functions can access variables that are nonlocal to their definition but are formal parameters or local definitions of an enclosing function. To implement accesses to nonlocal variables, compilers commonly employ a closure-conversion transformation [14, 75], in which the environment of each function, represented as a record, is passed as an extra parameter and free-variable accesses are compiled to accesses to the environment parameter. Function values are represented as *closures*, i.e. pairs of a code pointer and the environment. At application time the code and the environment components are projected out of the pair and the latter is passed as an argument to the former.

The representation of closure environments is crucial to the design of a closure conversion algorithm. Several closure representations have been proposed, each of them trying to optimize metrics such as space consumption, number of accesses to the environment, and closure creation time [121, 120, 74]. However, the choice of representation may affect the space-safety of a program.

7.2.1 Flat Closure Representation

In the *flat* representation of closures (Figure 7.1b), the environment of each function is a record that contains exactly the values of the function's free variables.

Execution time. The time overhead of flat closure conversion consists of the time needed to allocate the closure environment and pair after each function definition, which is proportional to some constant amount plus the size of the function's environment, and the time needed for fetching free-variable values from the environment, which in flat environments is always constant. Since the size of the environment is



```
fun f x y u w =
  fun g () =
    fun h n =
      fun k m =
        x + y + n + m
      in k u + n
    in h w
  in g
```

g_{code} | x | y | u | w
h_{code} | x | y | u
k_{code} | x | y | n
g_{code} | x | y | u | w
h_{code}
k_{code} | n

(a) Program with nested scopes. (b) Flat environments. (c) Linked environments.

Figure 7.1: Flat and linked closure representations. Linked closures *appear* to save space; for example, the flat closure environment for h is three words (x, y, u) but the linked representation is just one word. But linked closures are not safe for space; suppose k is live but g is not, then with flat closures w is garbage-collectible but with linked closures w is still reachable. If w is the root of a large data structure, this is significant.

equal to the number of free variables in the function's body, the total time overhead is, in the worst case, proportional to the execution time of the source by a factor that depends linearly on the size of the source program.

In the above analysis, we implicitly assume that, in the source cost model, function definitions incur constant cost. As we will see in Section 7.5, in our source cost model, function definitions incur time proportional to the number of their free variables. This decision allows us to establish a more precise bound on the running time of the target.

Execution space. To reason about the space overhead, we must think about the amount of space that is live simultaneously in the heap of the closure-converted program. Constructed values (lists, closure records, closure environments) occupy space as long as they are live, that is, as long as there is some chain of pointers (through such records) that reaches them, starting from a statically live variable of a currently active function. In CPS, of course, there is only one currently active function (in direct-style programs there is a stack of active functions). What is a statically live variable? The execution state of a function is modeled by an expression and an environment. At the beginning of function execution, the expression is the entire function body, and the environment is a finite map whose domain is the set of free variables of that function. As execution proceeds through the function, the current expression is some subexpression of the function body, and the environment may have been augmented by new local bindings. The statically live variables are the free variables of the current expression.[1]

[1] In traditional compiler terminology, statically live variables are those whose next use is before their next definition, along some path of control flow—and assuming that branches may go either way. This coincides with the notion of free variable in our CPS expressions.

Before closure conversion, our space consumption model measures exactly the amount of space that is live during the execution of the program. After closure conversion, the truly live space is *approximated* by calling the garbage collector upon each function entry and measuring the size of the actual heap. During the execution of a CPS function, the set of live records may increase (as new records are allocated by, e.g., constructor application) or decrease (as local variables live in expression e become dead in a subexpression of e). But since our CPS functions contain no internal loops—each function is a tree of control flow, of bounded height, terminating in tail calls at the leaves—the approximation error can be (at most) an additive amount proportional to the maximum path length in the execution-tree of the function.

I will, of course, formalize this argument in the remainder of this chapter.

7.2.2 Linked Closure Representation

In the flat environments in Figure 7.1b, free variables `x`, `y` and `z` appear multiple times. This increases both the time and the space that the program consumes: when creating environments for nested functions the values of their free variables must be copied from the environment of the immediately enclosing function to the newly constructed environment and, in addition, these environments can be live in the heap simultaneously, holding multiple copies of the same value.

The linked closure representation attempts to avoid this by introducing pointers from the environments of nested functions to the environments of the immediately enclosing functions. When function k is nested within function h, the linked closure for k contains (locally) only the variables free in k but *not* in h, and points to h's environment for the remaining free variables of k.

Although this representation might save time and space through sharing of free variables across function environments, it is not safe for space. In particular, any closure at some nesting depth can access the closure environments of the all the enclosing functions. This extends lifetime of variables across function calls, and can introduce memory leaks.

Figure 7.2 shows an example adapted from Appel [10]. The function *double* n creates a list with n copies of 0, and then it returns a function that computes the length of the list and that, in turn, returns another function that adds the computed length to n. We expect the space complexity of the program to be $\mathcal{O}(M)$: every call to *double* requires $\mathcal{O}(M)$ space and in the final result we store M closures, each of them taking up constant space. With linked closures, however, each closure that is created for g will maintain a pointer to the closure environment of f that in turn points to a list of length increasing from 1 to M, that cannot be reclaimed by the garbage collector, even though it is not needed. Since the final list keeps M closures, the space required for the execution of the program is $\mathcal{O}(M^2)$.

The Javascript V8 engine uses linked environments. Furthermore, it attempts to save time and space by sharing environments between all closures that are defined in the same scope. Environments are created eagerly upon entry to a scope. Although this may appear reasonable as it reduces closure-creation time, if the free variables of different functions have different (future) lifetimes, it is not safe for space. As in

```
fun double (n : ℤ) : unit → unit → ℤ =
  let l = repeat 0 n in
  fun f () =
    let m = length l in
    fun g () = m + n in
    g
  in f

fun app (n : ℤ) : list (unit → ℤ) =
  if n = 0 then []
  else double n () :: app (n − 1)
```

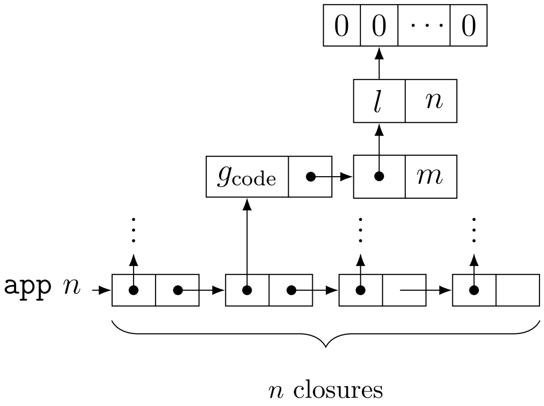


Figure 7.2: Linked closures are not safe for space. Each g closure contains (indirectly) a *different* long list l, while a flat closure for g would contain only the two integer values of m and n.

the previous case, variable lifetimes may extend past the lifetime of functions they are originally used, and lead to asymptotically worse space consumption [46, 51].

Therefore: Since unsafe-for-space closures can worsen the memory usage of a program by much more than a constant factor, it is important to use safe-for-space closure conversion.

7.2.3 The Main Theorem

The top-level corollary that we wish to establish states: Let e be a closed program in our compiler's CPS intermediate language. Let $\bar{e}$, in the same language, be the output of the closure-conversion phase. Suppose e evaluates to value v_1 and final heap H_1, using time c_1 and space m_1. Then $\bar{e}$ evaluates to value v_2 with heap H_2 using time c_2 and space m_2, such that: (v_1, H_1) relates to (v_2, H_2), $\quad c_1 \leq c_2 \leq K_{\texttt{time}} * c_1,\quad$ and $m_2 \leq m_1 + \texttt{space}_{\texttt{exp}}(e)$; where $K_{\texttt{time}}$ is a small constant and $\texttt{space}_{\texttt{exp}}(e)$ a function linear in the size of the source program e. The additive factor in the space bound accounts for the amount of allocation that happens during the evaluation of the expression e (*i.e.*, until the next function call happens or the program returns), that is the maximum that the space consumption of the target can diverge from the idealized space consumption of the source program.

In addition, a second corollary applies to programs e that diverge: $\bar{e}$ diverges, but its space consumption is guaranteed to remain within bounds.

7.3 Language and Memory Model

Our compiler CPS-converts into a continuation-passing style intermediate representation, and the back-end uses heap-allocated closures for both user functions and continuations, and no runtime stack. This greatly simplifies the interface between the compiler and garbage collector: there is no need to specify how to find roots in the stack. Such specifications can be enormously complicated [44]; furthermore, no C compiler (and certainly no verified C compiler) supports a method to keep track of root-pointers in the stack. McCreight et al. [96] show how to make a source-to-source transformation in C to keep track of roots using a shadow stack, which we employed for the $\lambda_{\texttt{ANF}}$ extension.

Let us begin the formalization with the definition of the untyped CPS lambda calculus ($\lambda_{\texttt{CPS}}$) on which we apply the closure conversion transformation. We will also formalize the heap model used by the semantics of the language that I will describe in Section 7.5.

7.3.1 Syntax

$$
\begin{array}{lrclcll}
(\textit{Variables}) & x, y & \in & \texttt{Var} & & & \\
(\textit{Constructors}) & \texttt{C} & \in & \texttt{Constr} & & & \\
(\textit{Expressions}) & e & \in & \texttt{Exp} & ::= & \texttt{let}\ x = \texttt{C}(\overline{y})\ \texttt{in}\ e & \text{Construction} \\
 & & & & \mid & \texttt{let}\ x = y.i\ \texttt{in}\ e & \text{Projection} \\
 & & & & \mid & \texttt{case}\ y\ \texttt{of}\ [\texttt{C}_i\ \rightarrow\ e_i]_{i \in I} & \text{Case} \\
 & & & & \mid & \texttt{fun}\ f\ \overline{x} = e_1\ \texttt{in}\ e_2 & \text{Function def.} \\
 & & & & \mid & f\ \overline{x} & \text{Continuation call} \\
 & & & & \mid & \texttt{ret}(x) & \text{Halt} \\
(\textit{Contexts}) & \mathcal{E} & \in & \texttt{Ctx} & ::= & [\,\cdot\,] \mid \texttt{let}\ x = \texttt{C}(\overline{y})\ \texttt{in}\ \mathcal{E} \mid \texttt{let}\ x = y.i\ \texttt{in}\ \mathcal{E} & \\
 & & & & \mid & \texttt{fun}\ f\ \overline{x} = e\ \texttt{in}\ \mathcal{E} & \\
(\textit{Locations}) & l & \in & \texttt{Loc} & & & \\
(\textit{Values}) & v & \in & \texttt{Val} & ::= & l \mid \texttt{fun}\ f = e & \\
(\textit{Environments}) & \sigma & \in & \sigma & = & \texttt{Var} \rightharpoonup \texttt{Val} & \\
(\textit{Blocks}) & b & \in & \texttt{Block} & ::= & \texttt{C}(\overline{v}) \mid \texttt{Clo}(v_1, v_2) \mid \texttt{Env}(\sigma) & \\
(\textit{Heaps}) & h & \in & \texttt{Heap} & = & \texttt{Loc} \rightharpoonup \texttt{Block} &
\end{array}
$$

Figure 7.3: Syntax and memory model of $\lambda_{\texttt{CPS}}$.

Expressions. The intermediate language (Figure 7.3) is exactly the same as $\lambda_{\texttt{ANF}}$, presented in Section 3.2, but without the let-bound application node. For clarity and completeness, I show the definition of the language here too. In the CPS language, the return construct plays the role for the halting continuation that returns the result of the evaluation to the top-level.

I also (re)define evaluation contexts $\mathcal{E}$ for the expressions of the calculus. I will use contexts to describe new code introduced by closure conversion, so we include only the relevant constructors of the language.

Values and Blocks. The notion of value in this formalization is different from the one we previously saw in Section 3.2. Values in this formalization are either pointers to heap blocks or function pointers. A heap block represents either a constructed value, a closure, or an environment. A block representing a constructor contains the constructor tag followed by pointers to the constructor arguments. Before closure conversion, closures are represented as a special type of block consisting of a pair of two values, with the first one expected to be a code pointer and the second one an environment, which is also a special type of memory block. After closure conversion, closures and their environments will be explicitly constructed by the program and will be represented as constructed values, therefore the target code will only use blocks that represent constructed values. A heap is represented as a partial map from locations to blocks.

I model function pointers as the actual code of the function: the space taken up by the code of the program is statically known and does not change during the execution. I do not account for executable code *in memory*.

Heap Implementation. In the formal development, the definitions are parameterized (using Coq's module system) by a heap implementation that satisfies an abstract interface. To prove the realizability of the abstract heap model, I provide a concrete implementation of heaps that realizes the abstract interface.

Let us move on to some definitions that will be useful later for the formalization of the source and target cost models and the garbage collector.

Free Locations. The set of locations that appear in a value can be either a singleton or an empty set:

$$\mathsf{FL}_{\texttt{Val}}(l) = \{l\} \qquad \mathsf{FL}_{\texttt{Val}}(\texttt{fun}\ f\ \overline{x} = e) = \emptyset$$

We can then define the locations that appear free in an environment in a subset S of its domain, where $\sigma[S]$ is the image of S under σ.

$$\mathsf{FL}_{\sigma}(\sigma)[S] = \bigcup_{v \in \sigma[S]} (\mathsf{FL}_{\texttt{Val}}(v))$$

We simply write $\mathsf{FL}_{\sigma}(\sigma)$ to denote the locations of an environment on its whole domain. This is a useful definition: the set $\mathsf{FL}_{\sigma}(\sigma)[\mathsf{fv}(e)]$ will be the root set of garbage collection.

Using the above definition we can give a definition for the locations that appear free inside a memory block.

$$\begin{array}{rclcrcl} \mathsf{FL}_{\texttt{Block}}(\texttt{C}(\overline{v})) & = & \bigcup_{v \in \overline{v}} (\mathsf{FL}_{\texttt{Val}}(v)) & \qquad & \mathsf{FL}_{\texttt{Block}}(\texttt{Env}(\sigma)) & = & \mathsf{FL}_{\sigma}(\sigma) \\ \mathsf{FL}_{\texttt{Block}}(\texttt{Clo}(v_1, v_2)) & = & \mathsf{FL}_{\texttt{Val}}(v_1) \cup \mathsf{FL}_{\texttt{Val}}(v_2) & & & & \end{array}$$

Heap Reachability. Given a set of locations S and a heap H, we define the set of locations that can be reached in one dereferencing step from S as follows.

$$\texttt{Post}(H)[S] = \bigcup_{b \in H[S]} (\mathsf{FL}_{\texttt{Block}}(b))$$

Then, the reachable locations from a root set S of pointers can be defined as the least fixed point of the `Post` operator.

$$\mathcal{R}(H)[S] = \bigcup_{n \in \mathbb{N}} ((\texttt{Post}(H))^n[S])$$

Heap Size. To give a formal account for the space consumption of a program, we shall first define the size of a heap. We consider values to be word-sized, as they are either heap or function pointers. First, we define the size of a heap-allocated block as the number of words that it takes up in memory. Blocks that represent constructed values have size equal to a word, that is used to represent the constructor tag, plus the number of the arguments, which are all word-sized values. The size of a block that represents a closure has a constant size equal to three words that are taken up by the tag and the two values that follow it. Lastly, environments have size equal to a word (for the tag) plus the number of bindings in the environment.

$$\texttt{size}(\texttt{C}(\overline{v})) = 1 + \texttt{length}(\overline{v}) \qquad \texttt{size}(\texttt{Clo}(v_1, v_2)) = 3 \qquad \texttt{size}(\texttt{Env}(\sigma)) = 1 + |\sigma|$$

We then define the size of the heap as the sum of the sizes of the allocated blocks.

$$\texttt{size}(H) = \sum_{l \in \texttt{dom}(H)} \texttt{size}(H(l))$$

It is also useful to define the size of the reachable portion of the heap from a root set S, by restricting the sum to only those blocks that are both in the domain of the heap and reachable from the root set.

$$\texttt{size}_{\mathcal{R}}(H)[S] = \sum_{l \in \texttt{dom}(H) \cap \mathcal{R}(H)[S]} \texttt{size}(H(l))$$

7.4 Heap Isomorphism

To specify garbage collection we first formalize a notion of heap isomorphism. After garbage collection the resulting heap will not necessarily be a subheap of the initial heap. Garbage collection algorithms may copy the contents of a location into another to compact the allocated space. Hence, the specification of garbage collection must describe that the reachable portions of the heap before and after garbage collection are equal up to an injective renaming of locations. Injectivity ensures that the same amount of sharing happens before and after garbage collection.[2]

We start by defining a relation on pairs of values and heaps that holds when two values represent the same data structure in their corresponding heaps. This relation is defined simultaneously with the corresponding relations for environments and heap blocks. These definitions recursively look up pointers in the heap to check if the represented data structures are the same. To ensure well-foundedness of the definitions in the presence of heap cycles,[3] the definitions are indexed by a natural number indicating the maximum lookup depth in the heap.

Value equivalence[4] is denoted $(v_1, H_1) \approx_\beta^n (v_2, H_2)$ where n is the lookup index and β a location renaming, represented as a total function on locations. Two values are equivalent if they are either both equivalent locations or equivalent function pointers. Two locations are equivalent if a.) they agree on the location renaming and b.) they are either both undefined or both point to equivalent blocks (denoted $(b_1, H_1) \sim_\beta^i (b_2, H_2)$ and its definition is given below). Two function pointers are equivalent if they syntactically represent the same function.

$$
\begin{array}{rcll}
(l_1, H_1) \approx_\beta^n (l_2, H_2) & \stackrel{\text{def}}{=} & l_2 = \beta(l_1) \wedge & \\
 & & \forall i < n, (b_1, H_1) \sim_\beta^i (b_2, H_2) & \text{if } H_1(l_1) = b_1 \\
 & & & \text{and } H_2(l_2) = b_2 \\
(l_1, H_1) \approx_\beta^n (l_2, H_2) & \stackrel{\text{def}}{=} & l_2 = \beta(l_1) & \text{if } l_1 \notin \texttt{dom}(H_1) \\
 & & & \text{and } l_2 \notin \texttt{dom}(H_2) \\
(v, H_1) \approx_\beta^n (v, H_2) & \stackrel{\text{def}}{=} & \texttt{True} & \text{if } v = \texttt{fun}\ f\ \overline{x} = e \\
(v_1, H_1) \approx_\beta^n (v_2, H_2) & \stackrel{\text{def}}{=} & \texttt{False} & \text{otherwise}
\end{array}
$$

We define environment equivalence and then use this to define equivalence on blocks representing closure environments. Two environments are equivalent in a set S of free variables, denoted $S \vdash (\sigma_1, H_1) \dot{\approx}_\beta^n (\sigma_2, H_2)$, if for any variable in the set, both environments map the variable to values that are equivalent in the given heaps, or both mappings are undefined.

$$
\begin{array}{rl}
S \vdash (\sigma_1, H_1) \dot{\approx}_\beta^n (\sigma_2, H_2) \stackrel{\text{def}}{=} \forall x \in S, & (\exists v_1\ v_2,\ \sigma_1(x) = v_1\ \wedge\ \sigma_2(x) = v_2 \wedge \\
 & \qquad (v_1, H_1) \approx_\beta^n (v_2, H_2)) \vee \\
 & (x \notin \texttt{dom}(\sigma_1)\ \wedge\ x \notin \texttt{dom}(\sigma_2))
\end{array}
$$

When the set S is the set of all variables we simply write $(\sigma_1, H_1) \dot{\approx}_\beta^n (\sigma_2, H_2)$.

[2] Conventional garbage collectors are injective; hash-consing collectors [13] are not.

[3] $\lambda_{\texttt{CPS}}$does not build circular data structures, but the framework is extensible to mutable references.

[4] I use the term "equivalence" only nominally for these definitions. Only if we existentially quantify the location renaming are these relations equivalences. However, we want to keep the location renaming transparent in our definitions in order to use it in the semantics.

Using the above definitions we can define block equivalence. Two heap blocks are equivalent in their corresponding heaps if they are both constructed values with the same outermost constructor and arguments that are pairwise equivalent values, or if they are both closures whose arguments are pairwise equivalent values, or if they are both equivalent environments.

$$\begin{array}{rcll}
(\mathtt{C}(v_1, \ldots v_n), H_1) \sim_\beta^n (\mathtt{C}(v'_1, \ldots v'_n), H_2) & \overset{\text{def}}{=} & \forall i, (v_i, H_1) \approx_\beta^n (v'_i, H_2) & \\
(\mathtt{Clo}(v_1, v_2), H_1) \sim_\beta^n (\mathtt{Clo}(v'_1, v'_2), H_2) & \overset{\text{def}}{=} & (v_1, H_1) \approx_\beta^n (v'_1, H_2) \wedge & \\
 & & (v_2, H_1) \approx_\beta^n (v'_2, H_2) & \\
(\mathtt{Env}(\sigma_1), H_1) \sim_\beta^n (\mathtt{Env}(\sigma_2), H_2) & \overset{\text{def}}{=} & (\sigma_1, H_1) \dot{\approx}_\beta^n (\sigma_2, H_2) & \\
(b_1, H_1) \sim_\beta^n (b_2, H_2) & \overset{\text{def}}{=} & \mathtt{False} & \text{otherwise}
\end{array}$$

That concludes the (mutually recursive) definition of value equivalence. We can now define value, environment, and block equivalence for any lookup depth simply by quantifying over all lookup indexes. To simplify future definitions, we also require that the renaming is an injective function in the set of reachable locations. I use the predicate $\mathtt{inj}_S(\beta)$ that asserts that the function β is injective in the subset S of its domain.

$$\begin{array}{rcl}
(l_1, H_1) \approx_\beta (l_2, H_2) & \overset{\text{def}}{=} & \forall\, n,\ (l_1, H_1) \approx_\beta^n (l_2, H_2) \ \wedge\ \mathtt{inj}_{\mathcal{R}(H_1)[\{l_1\}]}(\beta) \\
S \vdash (\sigma_1, H_1) \dot{\approx}_\beta (\sigma_2, H_2) & \overset{\text{def}}{=} & \forall\, n,\ S \vdash (\sigma_1, H_1) \dot{\approx}_\beta^n (\sigma_2, H_2) \ \wedge\ \mathtt{inj}_{\mathcal{R}(H_1)[\mathsf{FL}_\sigma(\sigma_1)[S]]}(\beta) \\
(b_1, H_1) \sim_\beta (b_2, H_2) & \overset{\text{def}}{=} & \forall\, n,\ (b_1, H_1) \sim_\beta^n (b_2, H_2) \ \wedge \mathtt{inj}_{\mathcal{R}(H_1)[\mathsf{FL}_{\mathtt{Block}}(b_1)]}(\beta)
\end{array}$$

Given a set of locations and a renaming of locations, two heaps are isomorphic with respect to this renaming if all locations in the set are equivalent with their renaming in their corresponding heaps.

$$S \vdash H_1 \dot{\sim}_\beta H_2 \ \overset{\text{def}}{=}\ \forall\, l \in S,\ (l, H_1) \approx_\beta (\beta(l), H_2)$$

In the following section we will use this definition to give a specification for garbage collection.

7.5 Profiling Semantics

In this section I formalize the source and target level semantics of $\lambda_{\mathtt{CPS}}$. As in chapter 3, I use environment-based, big-step operational semantics. Unlike the previously described semantics of $\lambda_{\mathtt{ANF}}$, the semantics presented in this chapter manipulate explicitly the heap of the program. Another difference is that here we need to set up a different semantics for the source and target programs of closure conversion, even though the languages are the same. The reason is that before closure conversion, whenever a function is defined, the source semantics needs to construct a closure, using the special closure block, by capturing the relevant part of the environment, and storing it in the heap using the special block type for environments. After closure conversion, the code explicitly constructs environments as ordinary constructed values, and therefore the target semantics only needs to store the pointer to the closed[5] function. The heap of target code will never use the closure and environment blocks, it will just use constructed values. Our source and target semantics profile the time and space needed for the execution of the program. Before getting into the details of our definitions, we discuss informally how we measure time and space.

Time. We use fuel-based semantics for both the source and the target: the computation times out if there are not enough units of time available. At each execution step the virtual clock winds down by the cost associated with the language constructor that is being evaluated, which is proportional to the size of the constructor. Evaluation succeeds if the provided fuel matches exactly the execution cost of the program.

Space. Our source space-cost semantics measures the size of the reachable heap at every evaluation step, and keeps track of the maximum value. That is, a program's space cost is the maximum reachable heap space during its execution. Conceptually, this is the space consumption of an ideal garbage collector strategy that collects all the unreachable pointers after every execution step.

In principle, we can use the same profiling strategy in the target in order to prove that closure conversion is safe for space. However, we opt for a different strategy in the target that models more accurately the consumed space of the target program. In particular, we keep track of the maximum size of the actual heap (instead of the reachable heap) during the execution but we run a garbage collector upon every function entry. Intuitively, this preserves the idealized measurement of space usage because only a bounded amount of allocation can occur between function calls in CPS. Functions in continuation-passing style are trees of control-flow (no loops) whose leaves are function calls, therefore the amount of heap allocation between function calls is bounded by the path-length of the function. Therefore we overapproximate the truly live data (compared to measuring at every allocation) by an additive amount proportional at worst to the size of the program. By using this profiling strategy in the target, not only we prove that closure conversion is safe-for-space but also that garbage collecting the heap upon every function entry is a safe-for-space garbage-collection strategy. Although this strategy is still idealized with respect to what our real garbage collector is doing, it allows to carry out a part of the refinement proof between the ideal model and our actual garbage collection implementation simultaneously with the space safety proof of closure conversion.

[5] The closure conversion transformation presented here is slightly different from the one presented in Chapter 4. The closure conversion transformation of this chapter always produces closed functions. The closure conversion transformation of Chapter 4 allows functions to have free variables if these refer to some know closed function. Here, we do not do that to avoid allocating special closure and environment blocks and account for the space they take in the heap. This optimization would be possible if hoisting functions at the top-level happened simultaneously with closure conversion, so that functions became immediately closed.

Theorem 7.1 (Concrete garbage collection cost (not proved in Coq)) *There is a simple garbage collector such that, if a program runs in our target semantics with time t and space A, then execution cost with the garbage collector is time linear in t and space linear in A.*

Proof Use a simple two-semispace copying collector [10, Ch. 16] in memory $M = 4A$. Each semispace has size $2A$, and when it is full, the collector copies it to the other semispace using Cheney's algorithm. Construct an alternate profiling semantics for the target language, which runs the garbage collector at each function call if the size of the heap is $\geq M/4$.

In the alternate profiling semantics, one garbage collection takes cA time, for some constant c. The program can run for at least A instructions (it takes at least one instruction to initialize each word of a newly allocated record) before the next collection. Therefore, the total time to run the program in our alternate semantics is bounded by $(c+1)t$, and the memory use of our alternate semantics is $4A$.

Finally, there is a simple simulation relation between the "ideal-garbage-collector" target semantics (formalized below) and the "real-garbage-collector" target semantics (which we do not formalize here). The simulation permits unreachable data to be absent from the ideal heap, but present (not yet collected) in the real heap. ■

The GC specification permits the use of other (more efficient) garbage collection algorithms, such as generational collection. For any such algorithm, one can formulate a "real-garbage-collector" target semantics, prove bounds on the space and time usage of the program, and prove a simulation between the ideal and the real.

7.5.1 Formal Model of Garbage Collection

In this section, I present a relational specification of ideal garbage collection. Given a root set S, the specification asserts that some heap is the collection of another for a given location renaming β.

$$\texttt{GC}_S(H_1, H_2, \beta) \overset{\text{def}}{=} S \vdash H_1 \dot{\sim}_\beta H_2 \ \wedge \ \texttt{dom}(H_2) \subseteq \mathcal{R}(H_2)[\beta(S)]$$

The two heaps must be isomorphic (up to the given renaming) in the set of the locations reachable from the root set in the initial heap (this also implies that the renaming is injective). To ensure that all the unreachable locations are collected, we require that the domain of the collected heap is a subset of the locations reachable from the root set in the collected heap. We apply the renaming in the initial root set, to find the root set of the collected heap.

To justify the specification of garbage collection, I provide an implementation in Coq of a simple garbage collection algorithm for the concrete heap implementation and I prove that it satisfies the above specification. The algorithm simply computes the set of reachable locations from the root set and restricts the domain of the heap to this set. This algorithm yields the identity renaming, since locations are not moved.

Using the above specification, we can state and prove that garbage collection respects heap equivalences. That is, if a heap is the collection of another for a given root-set then it is also the collection of an isomorphic heap for the corresponding root-set and an appropriate renaming.

Theorem 7.2 (GC respects heap equivalence)
Assume that $\mathtt{GC}_S(H_1, H_2, \beta)$ and $S \vdash H_1 \dot{\sim}_\delta H_1'$. Then $\mathtt{GC}_{\delta(S)}(H_1', H_2, \beta \circ \delta^{-1})$.

7.5.2 Operational Semantics

The big-step operational semantics judgment, written $H; \sigma; e \ {}^{f}{\Downarrow}_{l}^{m} \ r$ with $l \in \{\mathtt{src}, \mathtt{trg}\}$ for the source and target semantics respectively, states that a configuration ($\mathtt{Conf} \in \mathtt{Heap} \times \sigma \times \mathtt{Exp}$) evaluates to a result r. It is indexed by two metrics: a virtual clock f indicating the available execution steps (decreasing as the program executes), and m indicating the consumed space during execution. The result can be either a pair of a value and a heap or an out-of-time exception, written OOT, that indicates that the program does not terminate within i steps.

The semantics is parameterized with a fuel and a trace monoid, just as in Chapter 3. The fuel monoid models the execution time of the program, just as before. But unlike the previous semantics, in this semantics the trace monoid models the consumed space of the program. The carriers of both monoids are the set of natural numbers. As before we use an auxiliary relation $\downarrow$ that performs an evaluation step, while $\Downarrow$ takes care of the time and space profiling.

A program diverges if for all values of the clock it yields an out-of-time exception.

$$H; \sigma; e \Uparrow_{l}^{m} \quad \overset{\text{def}}{=} \quad \forall\, i, \exists\, m',\ H; \sigma; e \ {}^{i}{\Downarrow}_{l}^{m'} \ \mathsf{OOT} \ \wedge \ m' \leq m$$

The definition is annotated with an upper bound for memory consumption of the diverging program, which might be infinite and hence it belongs to the set $\mathbb{N} \cup \{\infty\}$.

Source Semantics. The evaluation rules for the source semantics are shown in Figure 7.4. At every step, the memory consumption is calculated by taking the maximum of the memory consumption of the rest of the program and the size of reachable heap before performing the evaluation step (rule Step). The memory consumption of a program that times out is the size of the reachable portion of the current heap (rule OOT). The memory consumption of a program just before returning is zero (rule OOT). At every evaluation step we decrease the virtual clock by the cost of each rule, which is given by the function $\mathtt{cost}(e)$. If at any point the remaining units of computation are fewer that the units required to evaluate the outermost constructor the programs terminates with an out-of-time exception.

Most rules are straightforward; here we review the slightly more complicated rules for function definition and application. The rule for function definition (rule Fun) first allocates the environment of the closure, that is the current environment restricted to those variables that appear free in the function body. This restriction is needed to have an accurate space cost model. It then allocates a closure and evaluates the continuation of the expression in the current environment extended with a mapping from the function name to the closure pointer. The cost of evaluating a

$$\frac{\sigma(\overline{y}) = \overline{v} \qquad \texttt{alloc}(\texttt{C}(\overline{v}), H_1) = (l, H_2) \qquad H_2; \sigma[x \mapsto l]; e \ {}^{c}{\Downarrow}^{m}_{\texttt{src}}\ r}{H_1; \sigma; \texttt{let}\ x = \texttt{C}(\overline{y})\ \texttt{in}\ e \ {}^{c}{\downarrow}^{m}_{\texttt{src}}\ r}\ \textsc{Constr}$$

$$\frac{\sigma(y) = l \qquad H(l) = \texttt{C}(v_1, \ldots, v_j, \ldots, v_n) \qquad H; \sigma[x \mapsto v_j]; e \ {}^{c}{\Downarrow}^{m}_{\texttt{src}}\ r}{H; \sigma; \texttt{let}\ x = y.j\ \texttt{in}\ e \ {}^{c}{\downarrow}^{m}_{\texttt{src}}\ r}\ \textsc{Proj}$$

$$\frac{\sigma(x) = l \qquad H(l) = \texttt{C}_i(\overline{v}) \qquad H; \sigma; e_i \ {}^{c}{\Downarrow}^{m}_{\texttt{src}}\ r}{H; \sigma; \texttt{case}\ x\ \texttt{of}\ \{\texttt{C}_i\ \rightarrow\ e\}_{i \in I} \ {}^{c}{\downarrow}^{m}_{\texttt{src}}\ r}\ \textsc{Case}$$

$$\frac{\begin{array}{c}\texttt{alloc}(\texttt{Env}(\sigma|_{\mathsf{fv}(\texttt{fun}\ f\ \overline{x} = e_1)}), H_1) = (l_{env}, H_2) \\ \texttt{alloc}(\texttt{Clo}(\texttt{fun}\ f\ \overline{x} = e_1, l_{env}, ,)H_2) = (l_f, H_3) \qquad H_3; \sigma[f \mapsto l_f]; e_2 \ {}^{c}{\Downarrow}^{m}_{\texttt{src}}\ r\end{array}}{H_1; \sigma; \texttt{fun}\ f\ \overline{x} = e_1\ \texttt{in}\ e_2 \ {}^{c}{\downarrow}^{m}_{\texttt{src}}\ r}\ \textsc{Fun}$$

$$\frac{\begin{array}{c}\sigma(\overline{x}) = \overline{v} \qquad \sigma(f) = l_f \qquad H_1(l_f) = \texttt{Clo}(\texttt{fun}\ g\ \overline{x} = e_1, l_{env},) \\ H_1(l_{env}) = \texttt{Env}(\sigma_f) \qquad H_1; \sigma_f[\overline{x} \mapsto \overline{v}][g \mapsto l_f]; e_1 \ {}^{c}{\Downarrow}^{m}_{\texttt{src}}\ r\end{array}}{H_1; \sigma; f\ \overline{x} \ {}^{c}{\downarrow}^{m}_{\texttt{src}}\ r}\ \textsc{App}$$

$$\frac{\sigma(x) = v}{H; \sigma; \texttt{ret}(x) \ {}^{\langle 0 \rangle}{\downarrow}^{\langle 0 \rangle}_{\texttt{src}}\ (v, H)}\ \textsc{Ret} \qquad\qquad \frac{c < \langle e \rangle_{\mathcal{F}}}{H; \sigma; e \ {}^{c}{\Downarrow}^{\langle H; \sigma; e \rangle_{\mathcal{T}}}_{\texttt{src}}\ \mathsf{OOT}}\ \mathrm{OOT}$$

$$\frac{H; \sigma; e \ {}^{c}{\downarrow}^{m}_{\texttt{src}}\ r}{H; \sigma; e \ {}^{c \langle + \rangle_{\mathcal{F}} \langle e \rangle_{\mathcal{F}}}{\Downarrow}^{m \langle + \rangle_{\mathcal{T}} \langle H; \sigma; e \rangle_{\mathcal{T}}}_{\texttt{src}}\ r}\ \textsc{Step}$$

where

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{F} & \overset{\text{def}}{=} & \mathbb{N} & \mathcal{T} & \overset{\text{def}}{=} & \mathbb{N} \\
\langle e \rangle_{\mathcal{F}} & \overset{\text{def}}{=} & \texttt{cost}(e) & \langle H; \sigma; e \rangle_{\mathcal{T}} & \overset{\text{def}}{=} & \texttt{size}_{\mathcal{R}}(H)[\mathsf{FL}_{\sigma}(\sigma)[\mathsf{fv}(e)]] \\
\langle + \rangle_{\mathcal{F}} & \overset{\text{def}}{=} & + & \langle + \rangle_{\mathcal{T}} & \overset{\text{def}}{=} & \max
\end{array}$$

and

$$\begin{array}{rcl@{\qquad}rcl}
\texttt{cost}(\texttt{let}\ x = \texttt{C}(\overline{y})\ \texttt{in}\ e) & \overset{\text{def}}{=} & 1 + \mathsf{len}(\overline{y}) & \texttt{cost}(\texttt{let}\ x = y.j\ \texttt{in}\ e) & \overset{\text{def}}{=} & 1 \\
\texttt{cost}(\texttt{case}\ x\ \texttt{of}\ \{\texttt{C}_i\ \rightarrow\ e\}_{i \in I}) & \overset{\text{def}}{=} & 1 & \texttt{cost}(\texttt{ret}(x)) & \overset{\text{def}}{=} & 1 \\
\texttt{cost}(\texttt{fun}\ f\ \overline{x} = e_1\ \texttt{in}\ e_2) & \overset{\text{def}}{=} & 1 + |\mathsf{fv}(\texttt{fun}\ f\ \overline{x} = e_1)| & \texttt{cost}(f\ \overline{x}) & \overset{\text{def}}{=} & 1 + \mathsf{len}(\overline{x})
\end{array}$$

Figure 7.4: Big-step operational semantics (source).

function declaration is a unit of time plus the number of the free variables of the function, to account for the implicit closure environment creation. The rule for function application (rule App) looks up the value of the applied function in the environment that should be a pointer to a closure in the heap. It dereferences the environment pointer of the closure to find the closure environment, which then extends by binding the formal parameters to the values of the actual parameters and the name of the function to the closure pair. It then evaluates the body of the function in the extended closure environment. Notice that the trace generator in this semantics takes as input the whole configuration, in order to compute the reachable size of the heap.

Target Semantics. The semantics for evaluating the code after closure conversion need not construct the closure pair and environment, as this is done explicitly by the code. Therefore the application and function definition are handled differently in the target semantics. We also change the way space profiling works: we profile the actual size of the heap, and we invoke the garbage collector upon function entry. As in the source semantics, evaluating each constructor incurs a cost. In this case the cost associated to the function definition will be just one unit of time since the function is closed and the semantics does not construct the closure environment.

$$\frac{H; \sigma[f \mapsto \texttt{fun}\ f\ \overline{x} = e_1]; e_2\ {}^{f}\!\Downarrow^{m}_{\texttt{trg}}\ r}{H; \sigma; \texttt{fun}\ f\ \overline{x} = e_1\ \texttt{in}\ e_2\ {}^{i}\!\downarrow^{m}_{\texttt{trg}}\ r}\ \textsc{Fun}_{\text{CC}} \qquad \frac{\sigma(x) = v}{H; \sigma; \texttt{ret}(x)\ {}^{\langle 0\rangle}\!\downarrow^{\langle 0\rangle}_{\texttt{trg}}\ (v, H)}\ \textsc{Ret}_{\text{CC}}$$

$$\frac{\begin{array}{c}\sigma(\overline{x}) = \overline{v} \qquad \sigma(f) = \texttt{fun}\ g\ \overline{x} = e \qquad \mathsf{GC}_{\mathsf{FL}_\sigma([\overline{x}\mapsto\overline{v}][g\mapsto\texttt{fun}\ g\ \overline{x}=e])[\mathsf{fv}(e)]}(H_1, H_2, \beta)\\ H_2; \beta \circ ([\overline{x} \mapsto \overline{v}][g \mapsto \texttt{fun}\ g\ \overline{x} = e]); e\ {}^{f}\!\downarrow^{m}_{\texttt{trg}}\ r\end{array}}{H_1; \sigma; f\ \overline{x}\ {}^{f}\!\downarrow^{m}_{\texttt{trg}}\ r}\ \textsc{App}_{\text{CC}}$$

$$\frac{i < \langle e\rangle_{\mathcal{F}}}{H; \sigma; e\ {}^{i}\!\Downarrow^{\langle H\rangle_{\mathcal{T}}}_{\texttt{trg}}\ \mathsf{OOT}}\ \text{OOT}_{\text{CC}} \qquad \frac{H; \sigma; e\ {}^{c}\!\downarrow^{m}_{\texttt{trg}}\ r}{H; \sigma; e\ {}^{c\langle+\rangle_{\mathcal{F}}\langle e\rangle_{\mathcal{F}}}\!\Downarrow^{m\langle+\rangle_{\mathcal{T}}\langle H\rangle_{\mathcal{T}}}_{\texttt{trg}}\ r}\ \textsc{Step}_{\text{CC}}$$

where

$$\begin{array}{rclrcl}
\mathcal{F} & \overset{\text{def}}{=} & \mathbb{N} & \mathcal{T} & \overset{\text{def}}{=} & \mathbb{N}\\
\langle e\rangle_{\mathcal{F}} & \overset{\text{def}}{=} & \texttt{cost}(e) & \langle H\rangle_{\mathcal{T}} & \overset{\text{def}}{=} & \texttt{size}(H)\\
\langle+\rangle_{\mathcal{F}} & \overset{\text{def}}{=} & + & \langle+\rangle_{\mathcal{T}} & \overset{\text{def}}{=} & \max
\end{array}$$

Figure 7.5: Big-step operational semantics (target).

Selected evaluation rules are shown in Figure 7.5. In the function definition case (rule $\textsc{Fun}_{\text{CC}}$), the environment is extended with the function name bound to function pointer before evaluating the continuation. In the application case (rule $\textsc{App}_{\text{CC}}$), the values of the actual parameters and the value of the function that is being looked up in the environment, with the latter expected to be a function pointer. A new environment is constructed by binding the names of the formal parameters to the values of the actual parameters and the function name to the function pointer. Before continuing to the evaluation of the function body, the heap is garbage collected using as roots the locations in environment bindings of the variables of the function. The garbage collector induces a location renaming that is used to rename the free location of the environment before proceeding to the evaluation of the function body (denoted by function composition).

I also define the interpretation of an evaluation context in some given heap and environment, formalized as a judgment of the form $H_1; \sigma_1; \mathcal{E} \blacktriangleright^c_l H_2; \sigma_2$, with $l \in \{\texttt{src}, \texttt{trg}\}$. The judgment asserts that the evaluation context $\mathcal{E}$ is interpreted in heap H_1 and environment σ_1, yielding a new heap H_2 and environment σ_2, using c units of time. The rules of this judgment follow closely the rules of the evaluation judgment; I do not show them here.

7.6 Closure Conversion

In this section I give a formal account of the flat closure conversion transformation on the CPS intermediate representation. The closure-converted code will explicitly construct closure environments and pairs, and will replace free variable occurrences with environment accesses. Function calls will be modified so that they destruct the closure pair and pass the closure environment as an argument to the function. After closure conversion, functions are closed and can be evaluated at the environment that only contains bindings for the formal parameters and the function name. Therefore, the post-closure-conversion semantics does not need to capture the environment at the time of function definition or look it up in the heap during function calls. After closure conversion, all function definitions can be hoisted to the top level, and there are only two levels of scope: the global scope and the scope that is local to every function. The resulting flat, closure-converted code can be directly translated to C.

I formalize closure conversion as a deductive system. The judgment $\Gamma, \Phi \vdash_{(\phi,\gamma)} e \rightsquigarrow \bar{e}$ asserts that $\bar{e}$ is the output of closure conversion of a source program e in a *local* environment Γ, that contains the names of locally bound free variables (*i.e.*, variables declared within the scope of the current function declaration) and a *global* environment Φ, that contains the names of free variables in scopes not local to the current function. The judgment is also indexed by two identifiers: ϕ (a map from function names in scope to their closure environment) and γ (the name of the formal parameter that corresponds to the closure environment in the current function). I write $e \rightsquigarrow \bar{e}$ for the closure conversion of top-level programs in which case Γ, Φ are empty and ϕ, γ have dummy values. The global environment Φ is an ordered set, that represents the order in which the values of free variables are stored in the closure environment.

I formalize an auxiliary judgment $\Gamma, \Phi \vdash_{(\phi,\gamma)} \overline{x} \triangleright \mathcal{E}, \Gamma'$ that associates a list of free variables $\overline{x}$ before closure conversion with an evaluation context $\mathcal{E}$, under which the variables should be used in closure converted code, and a possibly updated local environment Γ'. Intuitively, the evaluation context will take care of fetching the values of free variables from the environment parameter or constructing a closure pair when this is needed. Because we want to project each free variable and construct a closure pair at most once within each scope, every time a variable is fetched from the environment or a closure is constructed we update the local environment so that the next time the variable is referenced it will not be projected or constructed again. Apart from the local and global environments (Γ and Φ respectively), the judgment has two additional parameters: ϕ, which a partial map that bind function names in scope to their corresponding closure environments, and γ, which is the name of the formal parameter of the environment in the current scope, or a dummy variable if we are at the top-level scope.

$$\frac{x \in \Gamma \qquad \Gamma, \Phi \vdash_{(\phi,\gamma)} \overline{x} \rhd \mathcal{E}, \Gamma'}{\Gamma, \Phi \vdash_{(\phi,\gamma)} x :: \overline{x} \rhd \mathcal{E}, \Gamma'}\ \textsc{Local}$$

$$\frac{x_i \notin \Gamma \qquad \Phi = x_1, \ldots, x_i, \ldots, x_n \qquad \{x\} \cup \Gamma, \Phi \vdash_{(\phi,\gamma)} \overline{x} \rhd \mathcal{E}, \Gamma'}{\Gamma, \Phi \vdash_{(\phi,\gamma)} x_i :: \overline{x} \rhd \texttt{let}\ x_i = \gamma.\texttt{i}\ \texttt{in}\ \mathcal{E}, \Gamma'}\ \textsc{Global}$$

$$\frac{f \notin \Gamma \qquad f \in \texttt{dom}(\phi) \qquad \{f\} \cup \Gamma, \Phi \vdash_{(\phi,\gamma)} \overline{x} \rhd \mathcal{E}, \Gamma'}{\Gamma, \Phi \vdash_{(\phi,\gamma)} f :: \overline{x} \rhd \texttt{let}\ f = \texttt{C}_{\textsc{cc}}(f, \phi(f))\ \texttt{in}\ \mathcal{E}, \Gamma'}\ \textsc{Function}$$

$$\frac{}{\Gamma, \Phi \vdash_{(\phi,\gamma)} [] \rhd [\ \cdot\ ], \Gamma}\ \textsc{Empty}$$

Figure 7.6: Free variable judgment.

The rules of the free variable judgment are presented in Figure 7.6. If a variable is in the local scope (rule LOCAL), then it can be used as-is. The evaluation context and the local environment do not change. If a pre-closure-conversion variable is in the global environment (rule GLOBAL), then we project the value from the current environment parameter and we assign it to a binder that shadows its name. We add the variable to the local environment, so subsequent uses do not cause it to be projected out again. If a pre-closure-conversion variable is in the domain of the function environment map ϕ (rule FUNCTION), then it is a function name that must be packed together with its environment (given by the binding of the map) to construct a closure pair. Again, the local environment is extended accordingly.

We may now formalize the closure-conversion judgment (Figure 7.7). The rules for constructor, projection, pattern matching and halt are straightforward propagation rules. All they need to do is to handle the free variables correctly using the free variable judgment. The function case (rule $\textsc{cc}_{\textsc{Fun}}$), after picking a fresh name for the formal environment parameter of the closure-converted function, closure-converts the body of the function in a local environment that contains only the formal parameters, a global environment that consists of the free variables of the function definition in some particular order and a function map that has a single binding that maps function name to its formal environment parameter. The closure-converted code will then construct the closure environment. The function map will be extended with a mapping from the newly defined function to the newly constructed environment, and the continuation will be closure-converted. The application case (rule $\textsc{cc}_{\textsc{App}}$), after handling the free variables g and $\overline{x}$, projects the code pointer and the environment parameter out of the closure and performs the application passing the extra environment argument.

For recursive calls, our closure-conversion transformation will first construct the closure by packing its code with the current environment, then immediately project them out to perform the call. During compilation, these administrative redexes, that do not affect our resource-preservation proof, will be eliminated by a proved-correct shrink-reduction transformation [22].

$$\frac{\Gamma, \Phi \vdash_{(\phi,\gamma)} \overline{y} \triangleright \mathcal{E}, \Gamma' \qquad \{x\} \cup \Gamma', \Phi \vdash_{(\phi,\gamma)} e \rightsquigarrow \bar{e}}{\Gamma, \Phi \vdash_{(\phi,\gamma)} \texttt{let } x = \texttt{C}(\overline{y}) \texttt{ in } e \rightsquigarrow \mathcal{E}[\texttt{let } x = \texttt{C}(\overline{y}) \texttt{ in } \bar{e}]} \text{ CC}\textsc{Constr}$$

$$\frac{\Gamma, \Phi \vdash_{(\phi,\gamma)} y \triangleright \mathcal{E}, \Gamma' \qquad \{x\} \cup \Gamma', \Phi \vdash_{(\phi,\gamma)} e \rightsquigarrow \bar{e}}{\Gamma, \Phi \vdash_{(\phi,\gamma)} \texttt{let } x = \overline{y}.i \texttt{ in } e \rightsquigarrow \mathcal{E}[\texttt{let } x = y'.i \texttt{ in } \bar{e}]} \text{ CC}\textsc{Proj}$$

$$\frac{\Gamma, \Phi \vdash_{(\phi,\gamma)} x \triangleright \mathcal{E}, \Gamma' \qquad \Gamma', \Phi \vdash_{(\phi,\gamma)} e_i \rightsquigarrow \bar{e_i}}{\Gamma, \Phi \vdash_{(\phi,\gamma)} \texttt{case } x \texttt{ of } \{\texttt{C}_i \ \rightarrow \ e\}_{i \in I} \rightsquigarrow \mathcal{E}[\texttt{case } x \texttt{ of } \{\texttt{C}_i \ \rightarrow \ \bar{e}\}_{i \in I}]} \text{ CC}\textsc{Case}$$

$$\frac{\Gamma, \Phi \vdash_{(\phi,\gamma)} x \triangleright \mathcal{E}, \Gamma'}{\Gamma, \Phi \vdash_{(\phi,\gamma)} \texttt{ret}(x) \rightsquigarrow \mathcal{E}[\texttt{ret}(x)]} \text{ CC}\textsc{Halt}$$

$$\frac{\Gamma, \Phi \vdash_{(\phi,\gamma)} g::\overline{x} \triangleright \mathcal{E}, \Gamma'}{\Gamma, \Phi \vdash_{(\phi,\gamma)} g \ \overline{x} \rightsquigarrow \mathcal{E}[\texttt{let } g_{\text{code}} = g.\texttt{1} \texttt{ in let } g_{\text{env}} = g.\texttt{2} \texttt{ in } g_{\text{code}} \ (g_{\text{env}}::\overline{x'})]} \text{ CC}\textsc{App}$$

$$\frac{\begin{array}{c}\overline{FV} = \mathsf{fv}(\texttt{fun } f \ \overline{x} = e_1) \qquad \Gamma, \Phi \vdash_{(\phi,\gamma)} \overline{FV} \triangleright \mathcal{E}, \Gamma' \\ \overline{x}, \overline{FV} \vdash_{([f \mapsto \gamma_f], \gamma_f)} e_1 \rightsquigarrow \bar{e_1} \qquad \Gamma', \Phi \vdash_{(\phi[f \mapsto f_{\text{env}}], \gamma)} e_2 \rightsquigarrow \bar{e_2}\end{array}}{\Gamma, \Phi \vdash_{(\phi,\gamma)} \texttt{fun } f \ \overline{x} = e_1 \texttt{ in } e_2 \rightsquigarrow \mathcal{E}[\texttt{fun } f \ \gamma_f::\overline{x} = \bar{e_1} \texttt{ in let } f_{\text{env}} = \texttt{C}_{env}(\overline{FV}) \texttt{ in } \bar{e_2}]} \text{ CC}\textsc{Fun}$$

Figure 7.7: Closure conversion.

In our compiler we have a closure-conversion *program*, a Coq function. Although we could prove it correct directly, we provide this inductive definition as a relational specification for closure conversion. This allows us to reason about closure conversion without worrying about the details of the implementation. Then, we prove that closure-conversion program is correct w.r.t. the relational specification.

7.7 Logical Relation

In this section I set up the logical relation that we use to prove closure conversion correct and safe for time and space. The idea is that along with proving functional correctness we establish the cost bounds on the resources consumed by the evaluation of the source and target programs. I achieve this by parameterizing the logical relation with a pre- and a postcondition. The precondition is imposed on the initial configurations, and the postcondition is guaranteed to hold on the results of the evaluation. The logical relation extends the logical relation presented earlier and it is nonstandard in the following ways:

- It is parameterized by two pairs of pre- and postconditions. The doubling of pre- and postconditions allows us to prove monotonicity of the logical relation with respect to these and provide weakening and strengthening rules that important for compositional reasoning.

- For a certain class of pre- and postconditions, the reasoning applies to diverging sources as well: we show that divergence is preserved and the memory bound holds. This is in the same with the divergence preservation result presented in Chapter 5. The difference is that in addition to divergence preservation, I show that the memory consumption bound is satisfied by diverging programs as well.

- The logical relation is doubly indexed by a step-index that bounds the depth of recursion (crucial for the well-foundedness of our definition), and a heap lookup index that bounds the look up depth in the heap. Although one index could (seemingly) have served both purposes, we explain why it is important to decouple the two, and quantify over all lookup indexes in the fundamental theorem of the logical relation.

- The logical relation satisfies Kripke monotonicity, in that it respects heap isomorphism: when two configurations are related, they are related for all pairs of configurations that are isomorphic to the original configurations. I achieve this by quantifying over all pairs of isomorphic heap-environment pairs. This allows to maintain relatedness of the evaluation environment through the execution, regardless of how irrelevant portions of the heap may change.

Note: This logical relation is stated using the concrete fuel and trace monoids for time and space profiling and not the abstract fuel and trace monoids.

7.7.1 Configuration Relation: A Failed Attempt

We wish to define a logical relation that allows us to relate the costs of the source and target programs along with proving semantics preservation. Setting the value relation aside for now, let us try to define the configuration relation, that relates the execution of a source and a target configuration.

Following the standard pattern in step-indexed logical relations, we start by stating that for a given step index when the source evaluates to a result in steps than the step index, the target program also evaluates to a result that is related with the result of the source using the value relation. Additionally, we parameterize the relation with a *resource precondition* that is imposed on the initial configurations, and a *resource postcondition* that holds on the two pairs of the time and space consumption. The resource bounds can be program-dependent, hence we add the source configuration as a parameter to the resource postcondition, that is $\texttt{Post} \in \texttt{Conf} \to \texttt{Relation}(nat \times nat)$.

This is the initial formulation of the expression, or rather configuration, logical relation:

$$\begin{array}{l}\boldsymbol{\mathcal{E}}^k\ \{P\}\ (H_1, \sigma_1, e_1)\ (H_2, \sigma_2, e_2)\ \{Q\} \quad \stackrel{\text{def}}{=} \\ \quad \forall\ r_1\ c_1\ m_1,\ P\ (H_1, \sigma_1, e_1)\ (H_2, \sigma_2, e_2)\ \Rightarrow\ c_1 \le k\ \Rightarrow\ H_1; \sigma_1; e_1\ {}^{c_1}\Downarrow^{m_1}_{\texttt{src}}\ r_1\ \Rightarrow \\ \qquad \exists\ \sigma_2\ c_2\ m_2,\ H_2; \sigma_2; e_2\ {}^{c_2}\Downarrow^{m_2}_{\texttt{trg}}\ r_2\ \ \wedge\ \ Q\ (H_1, \sigma_1, e_1)\ (c_1, m_1)\ (c_2,\ m_2)\ \wedge \\ \qquad\qquad\qquad \boldsymbol{\mathcal{V}}^{k-c_1}\ \{P\}\ r_1\ r_2\ \{Q\}\end{array}$$

Where $\boldsymbol{\mathcal{E}}$ denotes the relation, k is the step index, P and Q the pre- and postcondition, and (H_1, σ_1, e_1) and (H_2, σ_2, e_2) the source and target configurations.

We would have been satisfied with this definition if we could prove that closure conversion inhabits it, as it entails what we wish to establish for the transformation. But technical complications prevent us from using this definition directly.

Compositional Reasoning. We want to prove compatibility lemmas that we can use in our closure-conversion proof to establish that each step of the transformation yields related programs. For instance, to prove the projection case we would need a compatibility lemma that is stated roughly as follows.

Lemma 7.3
Assume that

- $\boldsymbol{\mathcal{C}}^k\ \{y\} \vdash \{P\}\ (\sigma_1, H_1)\ (\sigma_2, H_2)\ \{Q\}$
- $\forall\ v_1\ v_2, \boldsymbol{\mathcal{V}}^k\ \{P\}\ (v_1, H_1)\ (v_2, H_2)\ \{Q\} \Rightarrow$
  $\boldsymbol{\mathcal{E}}^k\ \{P\}\ (H_1, \sigma_1[x \mapsto v_2], e_1)\ (H_2, \sigma_2[x \mapsto v_2], e_2)\ \{Q\}$

Then $\boldsymbol{\mathcal{E}}^k\ \{P\}\ (H_1, \sigma_1, \textit{\texttt{let}}\ x = y.j\ \textit{\texttt{in}}\ e_1)\ (H_2, \sigma_2, \textit{\texttt{let}}\ x = y.j\ \textit{\texttt{in}}\ e_2)\ \{Q\}$

The above variant of the lemma is problematic because it forces us to use the same pre- and postconditions before and after the evaluation of the outer constructor. This happens because the same pre- and postcondition that are enforced for the current configuration should hold also for future execution of whole functions. Looking back at the closure-conversion rules, before evaluating the right expression we should evaluate the context that causes the projection of the free variables, which (if not empty) consumes some units of time. Therefore, after evaluating this context, the initial postcondition will no longer hold. The original postcondition should be reestablished after the evaluation of the two projection constructors, allowing us to apply the induction hypothesis. To support compositional reasoning we should allow pre- and postconditions to vary in these rules. As we explain below, we can solve this issue by decoupling the pair of pre- and postconditions that are *local* and hold for the current configurations, from the *global* that hold for execution of whole functions in the environment and the result. This is the same approach that we took in Chapter 5.

Heap Monotonicity. The reader familiar with Kripke logical relations might have already identified a different problem: the logical relation does not directly satisfy the Kripke monotonicity requirement—two configurations that are related should remain related for any future execution states. Kripke monotonicity allows to maintain relatedness of the environments of evaluation during execution of the configuration, and it is crucial to prove that closure conversion inhabits the logical relation. In our case, target heaps not only grow, but also shrink and become renamed because of garbage collection. Our goal is to be able to prove that two related states (*i.e.*, pairs of environment and heap) remain related for all isomorphic environment-heap pairs:

$$\begin{array}{l}
\boldsymbol{\mathcal{E}}^k \ \{P\} \ (H_1, \sigma_1, e_1) \ (H_2, \sigma_2, e_2) \ \{Q\} \ \Rightarrow \\
\mathsf{fv}(e_1) \vdash (\sigma_1, H_1) \mathrel{\dot{\approx}_{\beta_1}} (\sigma_1', H_1') \ \Rightarrow \\
\mathsf{fv}(e_2) \vdash (\sigma_2, H_2) \mathrel{\dot{\approx}_{\beta_2}} (\sigma_2', H_2') \ \Rightarrow \\
\boldsymbol{\mathcal{E}}^k \ \{P\} \ (H_1', \sigma_1', e_1) \ (H_2', \sigma_2', e_2) \ \{Q\}
\end{array}$$

One could argue that it's enough to prove the semantics is deterministic up to heap isomorphism, in order to prove that our logical relation respects heap isomorphism. This is not the case: to do this we would have to impose additional requirements that the pre- and postcondition respect heap isomorphism. However, the concrete instantiations we are interested in are not preserved by heap isomorphism, because isomorphic heaps can have arbitrary sizes. Therefore, to achieve monotonicity, we explicitly close our logical definitions over all pairs of isomorphic environment-heap pairs.

Step Indexing. It is common in logical relations that are used to model the semantics of state, to use the step index to bound the lookup depth in the heap and avoid circularities. This technique was use by Ahmed [5, 4] to provide a stratified semantics of mutable state. Typically, this heap index is the same as the step index. In our case, we decouple these two indices. The reason is that when we carry out the proof, by induction on the step index, we will need the environment relation to hold for all heap indices, as it enforces useful heap invariants.

7.7.2 Logical Relation Definition

With all this in mind, we can now simultaneously define a value relation that relates the results of evaluation, and a configuration relation that relates the execution of two configurations. The relations are defined by induction on two indexes, the usual step index that indicates the maximum recursion depth, and the lookup index that indicates the maximum heap lookup depth.

Configuration Relation. The configuration relation in its final form appears below. Decoupling the indexes i and j is essential for the correctness proof—by quantifying over all lookup indexes we are able to relate the heaps at any depth for any given step index

$$\begin{array}{l}
\boldsymbol{\mathcal{E}}^{(k,j)}\ \{P_G; P_L\}\ (H_1, \sigma_1, e_1)\ (H_2, \sigma_2, e_2)\ \{Q_G; Q_L\} \quad \overset{\text{def}}{=} \\
\forall\ \sigma_1'\ H_1'\ \beta_1\ \sigma_2'\ H_2'\ \beta_2\ \ r_1\ c_1\ m_1, \\
\quad \mathsf{fv}(e_1) \vdash (\sigma_1, H_1) \mathrel{\dot{\approx}_{\beta_1}} (\sigma_1', H_1') \ \Rightarrow \\
\quad \mathsf{fv}(e_2) \vdash (\sigma_2, H_2) \mathrel{\dot{\approx}_{\beta_2}} (\sigma_2', H_2') \ \Rightarrow \\
\quad c1 \leq k \ \Rightarrow\ P_L\ (H_1', \sigma_1', e_1)\ (H_2', \sigma_2', e_2) \ \Rightarrow \\
\quad H_1'; \sigma_1'; e_1\ {}^{c_1}\!\Downarrow^{m_1}_{\mathtt{src}}\ r_1 \ \Rightarrow\ \mathtt{not\_stuck}(H_1'; \sigma_1'; e_1) \ \Rightarrow \\
\quad \exists\ r_2\ c_2\ m_2\ \beta, \\
\qquad H_2'; \sigma_2'; e_2\ {}^{c_2}\!\Downarrow^{m_2}_{\mathtt{trg}}\ r_2 \ \wedge \\
\qquad Q_L\ (H_1, \sigma_1, e_1)\ (c_1, m_1)\ (c_2,\ m_2) \ \wedge \\
\qquad \boldsymbol{\mathcal{V}}_{\beta}^{(k-c_1,j)}\ \{P_G\}\ r_1\ r_2\ \{Q_G\}
\end{array}$$

We additionally require that the source program cannot get stuck, that is, for any given fuel it either terminates with a result or yields an out-of-time exception. This is needed to exclude programs that may time out with small fuel values but get stuck later. Observe that this is not a requirement in the logical relations that we saw earlier. This is because the cost model we used in the previous setup assigned the same cost to every constructor of the language. In these semantics, the cost of the constructors can be different and therefore the source might time out for a fuel that allows the target to execute more steps. Knowing that the source is not stuck allows us to prove that the target can time out later than the source program.

Value Relation. The value relation (Figure 7.8) relates the result of two executions at a given step index i, lookup index j and location renaming function β, which keeps track of the mapping between source and target locations. Tracking the renaming allows us to establish that the reachable locations in the two heaps are in bijective correspondence.

Results are related as follows: An out-of-time exception relates only to an out-of-time exception. A pair of a location and a heap is related to another such pair if a.) both locations point to a constructed value in the heap, the target location agrees with the mapping of the source location of renaming function, and the arguments of the constructors are pairwise related at a strictly smaller lookup index; or b.) the source location points to a closure value and the target location points to a closure record. The closure environments must be related by the closure environment relation (shown below). In addition, for any source and target heaps isomorphic to the original ones at the root set that contains the closure environment, the function maps logically related values to logically related results, and any strictly smaller step index. The environment part of the configuration is the environment part of the closure for the source, and singleton environment mapping the environment parameter to the appropriate location for the target, both extended with the appropriate bindings for the formal parameters and the recursive function. We additionally require that the precondition holds upon function entry. This is crucial in order to instantiate the precondition premise of the configuration relation in the application case.

$$\boldsymbol{\mathcal{V}}_{\beta}^{(k,j)}\ \{P\}\ \mathsf{OOT}\ \mathsf{OOT}\ \{Q\} \ \overset{\text{def}}{=}\ \ \texttt{True}$$

$$\boldsymbol{\mathcal{V}}_{\beta}^{(k,j)}\ \{P\}\ (l_1, H_1)\ (l_2, H_2)\ \{Q\} \ \overset{\text{def}}{=} \qquad \text{If } H_1(l_1) = \mathtt{C}(\overline{v_1}) \text{ and } H_2(l_2) = \mathtt{C}(\overline{v_2}).$$

$$\begin{aligned}
&\beta(l_1) = l_2 \ \wedge \\
&\forall\ (j' < j),\ \boldsymbol{\mathcal{V}}_{\beta}^{(k,j')}\ \{P\}\ (\overline{v_1}, H_1)\ (\overline{v_2}, H_2)\ \{Q\}
\end{aligned}$$

$$\boldsymbol{\mathcal{V}}_{\beta}^{(k,j)}\ \{P\}\ (l_1, H_1)\ (l_2, H_2)\ \{Q\} \ \overset{\text{def}}{=} \qquad \text{If } H_1(l_1) = \mathtt{Clo}(\mathtt{fun}\ f_1\ \overline{x} = e_1, l_{\text{env}_1}) \text{ and } H_2(l_2) = \mathtt{C}_{\text{CC}}(\mathtt{fun}\ f_2\ \gamma\mathtt{::}\overline{x} = e_2, l_{\text{env}_2}).$$

$$\begin{aligned}
&\forall\ (j' < j),\ \boldsymbol{\mathcal{CL}}_{\beta}^{(k,j')}\ \{P\}\ (l_{\text{env}_1}, H_1)\ (l_{\text{env}_2}, H_2)\ \{Q\} \ \wedge \\
&\forall (i < k)\ H_1'\ \beta_1\ l'_{\text{env}_1}\ H_2'\ \beta_2\ l'_{\text{env}_2}\ \overline{v_1}\ \overline{v_2}, \\
&\quad (l_{\text{env}_1}, H_1) \approx_{\beta_1} (l'_{\text{env}_1}, H_1') \ \Rightarrow\ (l_{\text{env}_2}, H_2) \approx_{\beta_2} (l'_{\text{env}_2}, H_2') \ \Rightarrow \\
&\quad \mathtt{alloc}(H_1', \mathtt{Clo}(\mathtt{fun}\ f_1\ \overline{x} = e_1, l'_{\text{env}_1},)) = (l_{f_1}, H_1'') \ \Rightarrow \\
&\quad H_1'(l'_{\text{env}_1}) = \mathtt{Env}(\sigma_{f_1}) \ \Rightarrow \\
&\quad (\forall j,\ \boldsymbol{\mathcal{V}}_{\beta_2 \circ \beta \circ \beta_1^{-1}}^{(k,j)}\ \{P\}\ (\overline{v_1}, H_1')\ (\overline{v_2}, H_2')\ \{Q\}) \ \Rightarrow \\
&\quad P\ (H_1'', \sigma_1, e_1)\ (H_2', \sigma_2, e_2) \ \wedge \\
&\quad\ (\forall j,\ \boldsymbol{\mathcal{E}}^{(i,j)}\ \{P; P\ \}\ (H_1'', \sigma_1, e_1)\ (H_2', \sigma_2, e_2)\ \{Q; Q\})
\end{aligned}$$

where $\sigma_1 = \sigma_{f_1}[\overline{x} \mapsto \overline{v_1}][f_1 \mapsto l_{f_1}]$ and $\sigma_2 = [\gamma \mapsto l'_{\text{env}_2}, \overline{x} \mapsto \overline{v_2}, f_2 \mapsto \mathtt{fun}\ f_2\ \gamma\mathtt{::}\overline{x} = e_2]$.

$$\boldsymbol{\mathcal{V}}_{\beta}^{(k,j)}\ \{P\}\ (l_1, H_1)\ (l_2, H_2)\ \{Q\} \ \overset{\text{def}}{=}\ \ \texttt{False} \qquad \text{Otherwise.}$$

Figure 7.8: Value relation

Closure Environment Relation. The closure relation mentioned by the above definition is defined simultaneously with the value relation and it relates a source and a target closure environment pointers. It asserts that the locations of the two pointers agree with the location renaming and that the values of each environment are pairwise related.

$$\begin{aligned}
&\boldsymbol{\mathcal{CL}}_{\beta}^{(k,j')}\ \{P\}\ (l_1, H_1)\ (l_2, H_2)\ \{Q\} \overset{\text{def}}{=} \\
&\quad \exists \sigma, x_1, \ldots, x_n, v_1, \ldots, v_n, \\
&\qquad l_2 = \beta(l_1) \ \wedge\ H_1(l_1) = \mathtt{Env}(\sigma) \ \wedge\ H_2(l_2) = \mathtt{C}(v_1, \ldots, v_n) \ \wedge \\
&\qquad \mathtt{dom}(\sigma) = \{x_1, \ldots, x_n\} \ \wedge\ \bigwedge^{i \in [1,n]} \boldsymbol{\mathcal{V}}_{\beta}^{(k,j)}\ \{P\}\ (\sigma(x_i), H_1)\ (v_i, H_2)\ \{Q\}
\end{aligned}$$

Environment Relation. As usual, we lift the value relation to environments. The environment relation is annotated with a set of free variables and states that every variable in the set that is bound in the first environment, is also bound in the second environment and the bindings are logically related.

$$\begin{aligned}
&\boldsymbol{\mathcal{C}}_{\beta}^{(k,j)}\ S \vdash \{P\}\ (\sigma_1, H_1)\ (\sigma_2, H_2)\ \{Q\} \overset{\text{def}}{=} \\
&\quad \forall (x \in S)\ v_1,\ \sigma_1(x) = v_1 \Rightarrow \exists\ v_2,\ \sigma_2(x) = v_2 \ \wedge\ \boldsymbol{\mathcal{V}}_{\beta}^{(k,j)}\ \{P\}\ (v_1, H_1)\ (v_2, H_2)\ \{Q\}
\end{aligned}$$

7.7.3 Properties

I discuss formally some important properties of the logical relation.

Relating Nonterminating Programs. A known limitation of logical relations is the inability to reason about divergence preservation. In the usual formulation of logical relations, two programs are vacuously related if the source program doesn't terminate (or gets stuck). We address this limitation here by adopting a fuel-based semantics and raising out-of-time exceptions, which as a result of the computation is logically related only with an other out-of-time exception. We show how this allows us to prove, for a certain class of postconditions, that the translation of a nonterminating program is also a nonterminating program, whose memory consumption is within the desired bounds.

We say a resource is *downward* (resp. *upward*) *f-bounded* if for the resource consumption of the source r_{src} and the target r_{trg} we have that $f(r_{\text{src}}) \leq r_{\text{trg}}$ (resp. $r_{\text{trg}} \leq f(r_{\text{src}})$) for some function f.

Theorem 7.4 (Divergence preservation)
Assume $\boldsymbol{\mathcal{E}}^{(k,j)}$ $\{P_G; P_L\}$ (H_1, σ_1, e_1) (H_2, σ_2, e_2) $\{Q_G; Q_L\}$ and that Q_L implies that time cost is downward f-bounded for some invertible function f, and that the memory consumption is upward g-bounded for some monotonic function g. Then if, for some m_1, $H_1; \sigma_1; e_1 \Uparrow^{m_1}_{src}$ then $H_2; \sigma_2; e_2 \Uparrow^{m_2}_{trg}$ for some m_2. Furthermore, if $m_1 \neq \infty$ then $m_2 \leq g(m_1)$.

This theorem requires that the execution time of the source is bounded by the execution time of the target and the memory usage of the target is bounded by the memory usage of the source. The proof below gives some intuition about these requirements.

Proof Consider any fuel value i. We will show that the target program diverges when run with fuel i, with some memory consumption that satisfies the bound. Since the source diverges, running the source program with $f^{-1}(i)$ must return an OOT exception with some memory consumption $m_1' \leq m_1$. From the logical relation, there exists a fuel j for which the target program also returns an OOT exception and has some memory consumption m_2'. Furthermore, from the postcondition, $f(f^{-1}(i)) \leq j$, that is $i \leq j$. Hence, by monotonicity of the resource consumption of the target evaluation semantics, the target program runs out of time when run with fuel i with some memory consumption $m_2'' \leq m_2'$.

To prove the memory bound, we derive from the post condition that $m_2' \leq g(m_1')$ and therefore $m_2'' \leq g(m_1')$. By monotonicity of g and the fact that $m_1' \leq m_1$ we obtain that $m_2'' \leq g(m_1)$. ■

Local Reasoning. In its first simple form the logical relation did not support compositional reasoning. We address that by allowing a pair of local and a pair of global pre- and postconditions that allow us to restate compositional versions of the compatibility lemmas. Conceptually, by decoupling the global from the local conditions, we obtain monotonicity for the local conditions as the local postcondition can vary without affecting the global invariant that holds for the execution of whole functions. As an example, we look again at the projection case for an example of a compatibility lemma.

First, we state strengthening and weakening properties for the pre- and postconditions. For compactness, we group constructors that are described as evaluation contexts together, although these properties are meaningful only when these contexts are singleton contexts.

Definition 7.5 (Precondition strengthening, context application)
$\mathsf{PreCtx}\ \mathcal{E}_1\ \mathcal{E}_2\ e_1\ e_2\ P_1\ P_2$ *asserts that for any evaluation context interpretations* $H_1; \sigma_1; \mathcal{E}_1 \blacktriangleright^{c_1} H_1'; \sigma_1'$ *and* $H_2; \sigma_2; \mathcal{E}_2 \blacktriangleright^{c_2} H_2'; \sigma_2'$*, if*

$$P_2\ (H_1, \sigma_1, \mathcal{E}_1[e_1])\ (H_2, \sigma_2, \mathcal{E}_2[e_2])$$

holds one the configuration before interpreting the context then

$$P_1\ (H_1', \sigma_1', e_1)\ (H_2', \sigma_2', e_2)$$

holds one the final configurations.

Definition 7.6 (Postcondition weakening, context application)
$\mathsf{PostCtx}\ \mathcal{E}_1\ \mathcal{E}_2\ e_1\ e_2\ Q_1\ Q_2$ *asserts that for any any evaluation context interpretations* $H_1; \sigma_1; \mathcal{E}_1 \blacktriangleright^{c_1} H_1'; \sigma_1'$ *and* $H_2; \sigma_2; \mathcal{E}_2 \blacktriangleright^{c_2} H_2'; \sigma_2'$*, if*

$$Q_1\ (H_1', \sigma_1', e_1)\ (c, m)\ (c', m')$$

on the final configuration and its resource consumption, then

$$Q_2\ (H_1, \sigma_1, \mathcal{E}_1[e_1])\ (c + c_1, \max(m, \boldsymbol{size}_{\mathcal{R}}(H_1)[\mathsf{FL}_\sigma(\sigma)[\mathsf{fv}(\mathcal{E}_1[e_1])]]))\ (c' + c_2, m')$$

holds on the initial configuration and its resource consumption.

Above, the time consumption of the initial configurations is the time needed for evaluation of the final configurations, plus the time needed for the interpretation of the context. The memory consumption of the source is the maximum of the size of the reachable memory from the initial configurations and the memory consumption of the final configurations. The memory consumption of the target configuration is just propagated.

To establish the bounds when both programs timeout, we need to be able to derive the post condition from the precondition on the current configurations.

Definition 7.7 (Precondition entails postcondition, time-out)
$\mathsf{PrePostOOT}\ P\ Q$ *asserts that if*

$$P\ (H_1, \sigma_1, e_1)\ (H_2, \sigma_2, e_2)$$

and $c < \mathtt{cost}(e_1)$ *then*

$$Q\ (H_1, \sigma_1, e_1)\ (c, \boldsymbol{size}_{\mathcal{R}}(H_1)[\mathsf{FL}_\sigma(\sigma)[\mathsf{fv}(e_1)]])\ (c, \boldsymbol{size}(H_2))$$

We can now state and prove the projection compatibility theorem in its full generality, encompassing the weakening and strengthening rules for pre- and postconditions.

Theorem 7.8 (Projection Compatibility)
Assume that

- $\mathsf{PreCtx}\ (\boldsymbol{let}\ x = y.j\ \boldsymbol{in}\ [\ \cdot\ ])\ (\boldsymbol{let}\ x = y.j\ \boldsymbol{in}\ [\ \cdot\ ])\ e_1\ e_2\ P'_L\ P_L$
- $\mathsf{PostCtx}\ (\boldsymbol{let}\ x = y.j\ \boldsymbol{in}\ [\ \cdot\ ])\ (\boldsymbol{let}\ x = y.j\ \boldsymbol{in}\ [\ \cdot\ ])\ e_1\ e_2\ Q'_L\ Q_L$
- $\mathsf{PrePostOOT}\ P_L\ Q_L$
- $\forall\ j,\ \boldsymbol{\mathcal{C}}_{\beta}^{(k,j)}\ \{y\} \vdash \{P_G\}\ (\sigma_1, H_1)\ (\sigma_2, H_2)\ \{Q_G\}$
- $\forall\ v_1\ v_2, \boldsymbol{\mathcal{V}}_{\beta}^{(k,j)}\ \{P_G\}\ (v_1, H_1)\ (v_2, H_2)\ \{Q_G\} \Rightarrow$
  $\boldsymbol{\mathcal{E}}^{(k,j)}\ \{P_G; P'_L\}\ (H_1, \sigma_1[x \mapsto v_2], e_1)\ (H_2, \sigma_2[x \mapsto v_2], e_2)\ \{Q_G; Q'_L\}$

Then

$$\boldsymbol{\mathcal{E}}^{(k,j)}\ \{P_G; P_L\}\ (H_1, \sigma_1, \mathcal{E}_1[\boldsymbol{let}\ x = y.j\ \boldsymbol{in}\, e_1])\ (H_2, \sigma_2, \mathcal{E}_2[\boldsymbol{let}\ x = y.j\ \boldsymbol{in}\, e_2])\ \{Q_G; Q_L\}$$

Heap Size The environment relation has some useful implications for the structure of the two heaps. First, it implies that the reachable portions of the two heaps (up to depth equal to the heap lookup index) are well-formed, meaning that there are no dangling pointers. Therefore, when we quantify over all lookup indexes in the proof statement we avoid having to also assume and show the preservation of additional well-formedness assumptions, that are required for the closure conversion proof. Second, and most importantly, it implies that the set of reachable locations of the two heaps are in correspondence up to the given renaming.

The closure conversion proof uses this fact to show that directly after function entry in the source program, and garbage collection in the target program, the sizes of the heaps are related. By quantifying the environment relation over all heap depths we can derive that the data structures in the reachable portions of two heaps are in 1-1 correspondence, up to the given renaming. Therefore, we can prove that if two environments are related:

$$\forall\ j,\ \boldsymbol{\mathcal{C}}_{\beta}^{(k,j)}\ S \vdash \{P\}\ (\sigma_1, H_1)\ (\sigma_2, H_2)\ \{Q\}$$

then the size of the reachable portion of the target is upper-bounded by the size of the reachable portion of the source:

$$\mathtt{size}_{\mathcal{R}}(H_2)[\mathsf{FL}_{\sigma}(\sigma_2)[S]] <= \mathtt{size}_{\mathcal{R}}(H_1)[\mathsf{FL}_{\sigma}(\sigma_1)[S]].$$

Notice the use of less than or equal here, since nothing prevents the renaming from mapping two source locations to the same (related with both) target location.

7.8 Correctness Proof

With this machinery we can prove that closure conversion is correct and safe for space. The theorem states that the source and target programs of closure conversion are logically related for an appropriate choice of pre- and postconditions. In this section, I will give the concrete pre- and postconditions for the logical relation proof and we state the correctness theorem.

7.8.1 Time Bound

We are looking to find bounds for the execution time of the target. To handle divergent programs, the execution time of the closure converted program has to be lower-bounded by a function of the execution time of the source. This is easy: closure-conversion only adds cost anyway. To derive an upper bound, consider the running time of the closure-converted program: for each source construct there will be an overhead associated with the handling of its free variables, which is at most three steps for each free variable. Function definitions incur additional overhead for the construction of the environment, which costs just one extra step (recall that source cost semantics accounts for cost equal to the number of the free variables of the functions, but this will be zero in the target since functions are closed). For applications there is also an additional overhead equal to three steps, associated with the projections of the code and environment components of the closures and the application of the extra parameter. For each unit of time spent in the source execution, the overhead of the target is at most six units of time. Therefore the cost of evaluating the target will be at most seven times the cost of evaluating the source. The relation we wish to establish for the execution cost of the two programs is $c_{\texttt{src}} \leq c_{\texttt{trg}} \leq k_{\texttt{time}} * c_{\texttt{trg}}$, where $k_{\texttt{time}} = 7$.

7.8.2 Space Bound

To derive the space bound, consider how the sizes of the initial configurations relate immediately after the source enters a function body, and the target calls the garbage collector after function entry. The size of the target heap will be equal to its reachable portion, which will contain at most the space reachable from its original arguments and the space reachable from the environment argument (if the function has free variables and the environment is used). The latter includes the allocated space for the environment itself, and the space reachable by the free variables of the original program, whose values are exactly the fields of the environment record. The reachable portion of the source heap will contain at least the space reachable by the arguments, the space reachable by the free variables of the function, and the allocated closure block itself (code pointer + environment pointer + tag, 3 words) if the function is recursive and it is references by its own definition. This closure will be allocated for the target closure too, but unlike the source semantics that does it eagerly, the target will do it just before the function name is used; so we must account for the cost of the closure that may be allocated later in the execution of the target. We also account for the size of the environment that may be present in the target. Let $\texttt{size}_{\texttt{clo}} = 3 * |\{f\} \cap \mathsf{fv}(e_1)|$, be the size of the closure in the target heap. If $f \in \mathsf{fv}(e_1)$ it is 3 words and 0 otherwise. Also, let $\texttt{size}_{\texttt{env}} = 1 + |\mathsf{fv}(\texttt{fun}\ f\ \overline{x} = e_{\texttt{src}})|$ be the size of the environment in the target heap. When a function starts executing the two initial configurations will be related as follows.

$$\texttt{size}(H_{\texttt{trg}}) + \texttt{size}_{\texttt{clo}} \leq \texttt{size}_{\mathcal{R}}(H_{\texttt{src}})[\mathsf{FL}_{\sigma}(\sigma_{\texttt{src}})[\mathsf{fv}(e_{\texttt{src}})]] + \texttt{size}_{\texttt{env}}$$

When evaluating the target function, the heap can grow at most by a an amount proportional to the size of $e_{\texttt{src}}$, until the program returns or the next function is called and this relation in established again. Let $\texttt{space}_{\mathsf{exp}}(e_{\texttt{src}})$ be the maximum amount of allocation that can happen during the evaluation of $e_{\texttt{trg}}$. During the evaluation of the closure converted function, the size of the target heap, will remain below $\texttt{size}_{\mathcal{R}}(H_{\texttt{src}})[\mathsf{FL}_{\sigma}(\sigma_{\texttt{src}})[\mathsf{fv}(e_{\texttt{src}})]] + \texttt{size}_{\texttt{env}} + \texttt{space}_{\mathsf{exp}}(e_{\texttt{src}})$. Hence, the memory consumption of the target program will be upper bounded by the maximum of the above expression and the memory consumption of the execution of its continuation, which as a function call obeys a similar upper bound. To relate the execution cost of the two programs, observe that the size of the reachable portion of the source heap will be below the memory consumption $m_{\texttt{src}}$ of the source program, which follows directly by the way the source space consumption is calculated. Therefore, the target space consumption will be below the source consumption plus the maximum overhead incurred by the evaluation of the current function or any future function call.

$$m_{\texttt{trg}} \leq m_{\texttt{src}} + \max(\texttt{size}_{\texttt{env}} + \texttt{space}_{\mathsf{exp}}(e_{\texttt{src}}), \quad \texttt{space}_{\mathsf{heap}}(H_{\texttt{trg}}))$$

The function $\texttt{space}_{\mathsf{exp}}(\cdot)$ captures the maximum amount of allocation that can happen either during the evaluation of the closure-conversion of its argument, or during the evaluation of the closure-conversion of a function definition that is nested inside it. Its definition is shown below.

$$\begin{array}{rcl}
\texttt{space}_{\mathsf{exp}}(\texttt{let}\ x = \mathtt{C}(\overline{y})\ \texttt{in}\ e) & \overset{\text{def}}{=} & 1 + \mathsf{len}(\overline{y}) + \texttt{space}_{\mathsf{exp}}(e) \\
\texttt{space}_{\mathsf{exp}}(\texttt{let}\ x = y.j\ \texttt{in}\ e) & \overset{\text{def}}{=} & \texttt{space}_{\mathsf{exp}}(e) \\
\texttt{case}\ y\ \texttt{of}\ [\mathtt{C}_i\ \to\ e_i]_{i \in I} & \overset{\text{def}}{=} & \max\limits_{i \in I}(\texttt{space}_{\mathsf{exp}}(e_i)) \\
\texttt{space}_{\mathsf{exp}}(\texttt{fun}\ f\ \overline{x} = e_1\ \texttt{in}\ e_2) & \overset{\text{def}}{=} & \max(|\mathsf{env}| + 3 + \texttt{space}_{\mathsf{exp}}(e_2), |\mathsf{env}| + \texttt{space}_{\mathsf{exp}}(e_1)) \\
 & & \text{where } |\mathsf{env}| = 1 + \mathsf{len}(\mathsf{fv}(\texttt{fun}\ f\ \overline{x} = e_1)) \\
\texttt{space}_{\mathsf{exp}}(f\ \overline{x}) & \overset{\text{def}}{=} & 0 \\
\texttt{space}_{\mathsf{exp}}(\texttt{ret}(x)) & \overset{\text{def}}{=} & 0
\end{array}$$

Most cases are straightforward, but it is worth discussing the function definition case, which is the most complicated. We take the maximum of the allocation that can happen during the evaluation of the current expression and the maximum allocation that can happen during the evaluation of function being defined. The maximum allocation that can happen during evaluation of the current expression is size of the closure environment that will be allocated, the space of the closure that will be allocated (before its first use in this scope), and the space that will be allocated when evaluating the rest of the program. The maximum allocation that can happen during the evaluation of function being defined amounts to the size of the environment of the function plus the space that will be allocated during its evaluation.

The function $\texttt{space}_{\texttt{heap}}(\cdot)$ captures the maximum amount of allocation that can happen when evaluating any function pointer that is stored in the heap and it given by the following definition.

$$\texttt{space}_{\texttt{heap}}(H) \overset{\text{def}}{=}$$
$$\max\{\texttt{space}_{\texttt{exp}}(e) + (1 + |\textsf{fv}(\texttt{fun}\ f\ \overline{x} = e)|) \mid \exists l,\ H(l) = \texttt{fun}\ f\ \overline{x} = e\}$$

We have formulated the bound that we expect to hold for the space consumption of the two programs, but we're not done yet. These pre- and postconditions will hold only directly after function entry, not necessarily during the execution of the two functions. We need to find pre- and postconditions that capture the relation of the resources at any point during execution. Let us first state the precondition in a more general way. In particular, assume that for some parameters A and δ we have that $\texttt{size}(H_{\texttt{trg}}) + 3 * |F| \leq A + \delta$, where F are the function names in scope that have not been used yet, hence the target code has not constructed their closures and therefore we have to account for their future allocation. A stands for the size of the reachable source heap at the function entry point and δ is the extra space that has been allocated during the execution of the target code after the function entry point. If this assumption holds for the initial target heap, then the postcondition that bounds the space consumption of the target is captured by the following inequality.

$$m_{\texttt{trg}} \leq \max(A + \texttt{space}_{\texttt{exp}}(e_{\texttt{src}}) + \delta, m_{\texttt{src}} + \max(\texttt{space}_{\texttt{exp}}(e_{\texttt{src}}), \quad \texttt{space}_{\texttt{heap}}(H_{\texttt{src}})))$$

The term $A + \texttt{space}_{\texttt{exp}}(e_1) + \delta$ captures the maximum space consumption for the execution of the current function, and the term $m_{\texttt{src}} + \max(\texttt{space}_{\texttt{exp}}(e_{\texttt{src}}), \texttt{space}_{\texttt{heap}}(H_{\texttt{src}}))$ the maximum space consumption of any future execution of a function that is either a function defined in $e_{\texttt{src}}$ or a heap pointer.

We can now formally state the concrete pre- and postconditions that we wish to establish when proving closure conversion correct. The local pre- and postconditions can be defined as shown below.

$$\texttt{P}_{\texttt{L}}\ F\ A\ \delta\ (H_{\texttt{src}}, \sigma_{\texttt{src}}, e_{\texttt{src}})\ (H_{\texttt{trg}}, \sigma_{\texttt{trg}}, e_{\texttt{trg}}) \overset{\text{def}}{=} \texttt{size}(H_{\texttt{trg}}) + 3 * |F| \leq A + \delta$$

$$\texttt{Q}_{\texttt{L}}\ A\ \delta\ (H_{\texttt{src}}, \sigma_{\texttt{src}}, e_{\texttt{src}})\ (c_{\texttt{src}}, m_{\texttt{src}})\ (c_{\texttt{trg}}, m_{\texttt{trg}})\ \overset{\text{def}}{=}$$
$$c_{\texttt{src}} \leq c_{\texttt{trg}} \leq k_{\texttt{time}} * c_{\texttt{trg}} \land$$
$$m_{\texttt{trg}} \leq \max(A + \texttt{space}_{\texttt{exp}}(e_{\texttt{src}}) + \delta, m_{\texttt{src}} + \max(\texttt{space}_{\texttt{exp}}(e_{\texttt{src}}), \texttt{space}_{\texttt{heap}}(H_{\texttt{src}})))$$

Note that the local pre- and postconditions are parameterized by A, δ, and F which, will are also parameters of the closure conversion proof.

The global pre- and postconditions are instances of the local ones that hold when considering the execution of whole functions. The parameter F will be passed by the value logical relation (not shown in its definition) and will be the set of free variables of the function. The parameter δ will be instantiated with the size of the function's environment (computed using the cardinality of the set of free variables F), and parameter A with the size of the reachable portion of the heap upon function entry.

$$\begin{gathered}\mathtt{P_G}\ F\ (H_{\mathtt{src}}, \sigma_{\mathtt{src}}, e_{\mathtt{src}})\ (H_{\mathtt{trg}}, \sigma_{\mathtt{trg}}, e_{\mathtt{trg}}) \overset{\text{def}}{=} \\ \mathtt{P_L}\ F\ \mathtt{size}_{\mathcal{R}}(H_{\mathtt{src}})[\mathsf{FL}_{\sigma}(\sigma_{\mathtt{src}})[\mathsf{fv}(e_{\mathtt{src}})]]\ (1 + |F|)\ (H_{\mathtt{src}}, \sigma_{\mathtt{src}}, e_{\mathtt{src}})\ (H_{\mathtt{trg}}, \sigma_{\mathtt{trg}}, e_{\mathtt{trg}})\end{gathered}$$

$$\begin{gathered}\mathtt{Q_G}\ F\ (H_{\mathtt{src}}, \sigma_{\mathtt{src}}, e_{\mathtt{src}})\ (c_{\mathtt{src}}, m_{\mathtt{src}})\ (c_{\mathtt{trg}}, m_{\mathtt{trg}}) \overset{\text{def}}{=} \\ \mathtt{Q_L}\ \ \mathtt{size}_{\mathcal{R}}(H_{\mathtt{src}})[\mathsf{FL}_{\sigma}(\sigma_{\mathtt{src}})[\mathsf{fv}(e_{\mathtt{src}})]]\ (1 + |F|)\ (H_{\mathtt{src}}, \sigma_{\mathtt{src}}, e_{\mathtt{src}})\ (c_{\mathtt{src}}, m_{\mathtt{src}})\ (c_{\mathtt{trg}}, m_{\mathtt{trg}})\end{gathered}$$

7.8.3 Correctness

The main correctness theorem states that the input and the output programs of closure conversion are related when evaluated in related heap-environment pairs. The logical relation implies semantic preservation and establishes the resource bounds. In the proofs, I make the explicit assumption that the source program has unique bindings, and that bound variables are disjoint from the free variables of the program (*i.e.*, that the program is well scoped). As usual, in the fundamental theorem of the logical relation, I require that the environments are logically related, and maintain this invariant throughout the proof. But in the case of closure conversion this relation holds only for the local environments, that contain parameters of the current function and locally defined variables, excluding newly defined functions that have not been used yet (because the closure-converter will not build the closure for a function until its first occurrence).

Recall the closure conversion judgment: $\Gamma, \Phi \vdash_{(\phi,\gamma)} e \rightsquigarrow \bar{e}$. The environment Γ corresponds to the local variables and hence its contents will be related by the environment relation. The evaluation environment also contains the names of the functions that have been defined locally but have not been used yet (given by the set $\mathtt{dom}(\phi) \setminus \Gamma$)—and therefore the target code has not constructed their closures yet—and the free variables that are not local to the current scope (given by the global environment Φ).

Two additional environment invariants capture the relations that hold for these parts of the environment. The first relation asserts that the variables in the global environment of the source program are logically related with the values stored in the closure environment of the target program.

$$\begin{aligned}&\boldsymbol{\mathcal{FV}}_{\beta}^{(k,j)}\ \Phi; \gamma \vdash \{P\}\ (\sigma_1, H_1)\ (\sigma_2, H_2)\ \{Q\} \overset{\text{def}}{=} \\ &\qquad \exists x_1, \ldots, x_n, v_1, \ldots, v_n,\ \Phi = [x_1, \ldots, x_n]\ \wedge\ H_2(\sigma_2(\gamma)) = \mathtt{C}(v_1, \ldots, v_n)\ \wedge \\ &\qquad\quad \bigwedge\nolimits^{i \in [1,n]} \boldsymbol{\mathcal{V}}_{\beta}^{(k,j)}\ \{P\}\ (\sigma(x_i), H_1)\ (v_i, H_2)\ \{Q\}\end{aligned}$$

In the same spirit, we can formulate a relation that connects the closures in the source with the closures that have not yet been constructed in the target.

$$\begin{array}{l}\boldsymbol{\mathcal{F}}_{\beta}^{(k,j)}\ \Gamma ; \phi \vdash \{P\}\ (\sigma_1, H_1)\ (\sigma_2, H_2)\ \{Q\} \stackrel{\text{def}}{=} \\ \quad \forall\ (f \in \mathtt{dom}(\phi) \setminus \Gamma)\ l_f\ H_2',\quad \mathtt{alloc}(H_2, \mathtt{C}_{\mathrm{CC}}(\sigma_2(f); \sigma_2(\phi(f)))) = (l_f, H_2') \quad \Rightarrow \\ \qquad \boldsymbol{\mathcal{V}}_{\beta}^{(k,j)}\ \{P\}\ (\sigma_1(f), H_1)\ (l_f, H_2')\ \{Q\}\end{array}$$

The above relation asserts that the value of a function name in $\mathtt{dom}(\phi) \setminus \Gamma$ in the source environment is logically related to a freshly allocated location that points to target closure, whose code component is the value of the function name in the target environment, and the environment component is the value of the environment name ($\phi(f)$) in target environment. Maintaining this invariant allows us to establish the local environment relatedness after the target closure has been constructed and the function name has been added to the local environment.

Using the above invariants we can state and prove the fundamental theorem of our logical relation: that the closure conversion relation inhabits the logical relation for our choice of pre- and postconditions. We additionally require that the location renaming between the source and target heaps is injective at the reachable portion of its domain order to establish that the reachable portions of the two heaps are in bijection.

Theorem 7.9 (Correctness of Closure Conversion)
Assume that

- $\mathsf{well\_scoped}(e)$
- $\Gamma, \Phi \vdash_{(\phi,\gamma)} e \rightsquigarrow \bar{e}$
- $\forall\ j,\quad \boldsymbol{\mathcal{C}}_{\beta}^{(k,j)}\ \Gamma \vdash \{P_G\}\ (\sigma_{src}, H_{src})\ (\sigma_{trg}, H_{trg})\ \{Q_G\}$
- $\forall\ j,\quad \boldsymbol{\mathcal{FV}}_{\beta}^{(k,j)}\ \Phi ; \gamma \vdash \{P_G\}\ (\sigma_{src}, H_{src})\ (\sigma_{trg}, H_{trg})\ \{Q_G\}$
- $\forall\ j,\quad \boldsymbol{\mathcal{F}}_{\beta}^{(k,j)}\ \Gamma ; \phi \vdash \{P_G\}\ (\sigma_{src}, H_{src})\ (\sigma_{trg}, H_{trg})\ \{Q_G\}$

and let $F = \mathtt{dom}(\phi) \setminus \Gamma$.

Then for all $j\ A\ \delta$, $\boldsymbol{\mathcal{E}}^{(k,j)}\ \{P_G ; P_L\ F\ A\ \delta\}\ (H_{src}, \sigma_{src}, e)\ (H_{trg}, \sigma_{trg}, \bar{e})\ \{Q_G ; Q_L\ A\ \delta\}$.

The proof is by induction on the step-index and then by case analysis on the expression. Doing the proof amounts to showing that the contexts generated by the free variable judgment are related to the source environment when interpreted, and then for the cases other than application and function definition, applying the compatibility lemmas and the induction hypothesis (in inductive cases). The application case is almost immediate using the environment relation. The abstraction case requires us to show that the newly defined functions are related with $\boldsymbol{\mathcal{F}}$, which we obtain from the induction hypothesis and the fact that the size bound holds upon function entry. Finally, we need to show that the concrete pre and postconditions satisfy the compatibility requirements.

Now consider a complete program, one without free variables. If it terminates, the closure-converted program terminates, the result is correct, and the resource bound is satisfied: it is safe for space and safe for time.

Corollary 7.10 (Correctness of Closure Conversion, top-level, termination) *If* $e \rightsquigarrow \bar{e}$ *and* $e \Downarrow_{src}^{(c_1,m_1)} (v_1, H_1)$, *then there exist* v_2, H_2, c_2, m_2, β *such that*

- *the target terminates* $\bar{e} \Downarrow_{trg}^{(c_2,m_2)} (v_2, H_2)$,
- *the results are related* $\forall\ i\ j,\ \boldsymbol{\mathcal{V}}_{\beta}^{(k,j)}\ \{P_G\}\ (v_1, H_1)\ (v_2, H_2)\ \{Q_G\}$, *and*
- *the resource consumption for time and space is preserved:* $c_1 \leq c_2 \leq k_{\texttt{time}} * c_1$ *and* $m_2 \leq m_1 + \boldsymbol{space}_{exp}(e) + 1$.

If the source program diverges, the closure converted program must also diverge, and if the source program runs in bounded space then the target must run in some bounded space related to the source space by the postcondition.

Corollary 7.11 (Correctness of Closure Conversion, top-level, divergence) *If* $e \rightsquigarrow \bar{e}$ *and* $e \Uparrow_{src}^{m_1}$ *then there exists* m_2 *such that the target diverges* $\bar{e} \Uparrow_{trg}^{m_2}$ *and* $m_2 \leq m_1 + \boldsymbol{space}_{exp}(e) + 1$.

Coq Development. These results have all been fully formalized in the Coq proof assistant. The length of the main closure conversion proof is 1855 lines and the logical relation framework along with the compatibility lemmas is 1815 lines of specification code and 1921 lines of proof code.

Compositionality. I have presented a resource-safety proof for closure conversion. For a provably resource-preserving compiler we have to establish this result for all the passes of the compiler and compose the proofs to get and end-to-end theorem. Unfortunately, for the reasons explained in Chapter 5 composition cannot happen at the level of logical relations. Composition can happen using the transitive closure and exposing pre- and postconditions that are the composition of the ones used for each composed logical relation. Composing individual resource bounds will give a bound for the whole pipeline, which might however be very imprecise as individual factors will be multiplied with each other. If, for instance, two transformations are upper bounded by factors K and M, then their composition will be upper bounded by $K \times M$. A more precise bound could be achieved by using a cost model that keeps track of different steps.

7.9 Related Work

Space-safety of Program Transformations. Others have already observed that program transformations ought to be safe with respect to their resource consumption. Minamide [100] proves space-safety of the CPS transformation by showing, using a simulation argument, that CPS preserves the size of the reachable portion in the heap within a constant factor. This proof is done using a forward syntactic simulation.

Improvement theory. Sands [116, 117] defines a related concept to characterize transformations guaranteed to never worsen the asymptotic complexity of a program with respect to a particular resource. A program is called an improvement of another if its execution is more efficient with respect to a particular resource in any given program context. To show that a transformation is resource-safe it suffices to show that local steps of the transformation inhabit a particular improvement relation. This technique has been studied both in the context of time-safety and space-safety [57, 56]. Theory of space improvement has only been applied to local transformations.

After the publication of the work presented in this chapter [107], Carr [28] in his master's thesis explored the formal verification of space safety of transformations. Carr formally proves in the Coq proof assistant that a globalization transformation that hoists definitions to higher scopes is safe for space. This work is focused not only on showing a global upper bound for space safety, but establishing that this bound will also hold locally too by showing that space bounds holds between I/O events that can happen nondeterministically. The proof technique is based on syntactic simulations.

Space bounds in CakeML. After publication of this work [107], a concrete space cost semantics has been developed for CakeML programs [53]. This cost semantics allows reasoning about heap and space bounds and is proved sound with respect to end-to-end compilation. Previously, the top-level theorem of CakeML allowed a program to terminate early because of insufficient memory resources. The new top-level theorem of CakeML guarantees that if the source program is safe for space (the user can prove this assumption using the space cost semantics) then the target program will successfully terminate. Compared to the work in this chapter, the cost semantics for CakeML avoids explicit modeling of the heap (which we do here) by using a time-stamp mechanism to tag aliased values. It is also connected with a concrete garbage collection implementation, as opposed to the idealized garbage collection specification that is considered here.

Resource bound certification. Resource consumption bounds can be certified by showing that these bounds are preserved through a compilation pipeline. Crary [40] present a decidable type system capable of establishing upper bounds on resource consumption of programs and they show that these bounds are preserved all the way to assembly. They do so by introducing a virtual clock that winds down at every function application. Although this technique suffices to show preservation of time, it does not scale to space usage in a garbage collected setting. Since memory does not grow monotonically, "ticks" cannot capture space consumption.

Certified space-preserving compilers for imperative languages exists, but employ very different proof techniques, that do not directly apply to functional programs and higher order state. Carbonneaux *et al.* [26] prove that stack-space bounds established for C programs are preserved through CompCert by extending the trace preservation proof of the compiler to also prove stack space consumption. Amadio *et al.* [7] show that the *labeling* method, an instrumentation technique for monitoring resources the program consumes, is preserved through a compilation chain. The labels of the source program can be used to reason about its execution time, allowing for end-to-end bound certification. Besson *et al.* [24] extend the semantics and the proof of CompCert to verify that memory consumption is preserved.

Logical relations. The application of logical relations to correctness of program transformations has been studied widely. Our logical relation is unusual in that it supports reasoning in presence of garbage collection. Hur *et al.* [64] observed that Kripke logical relations are not directly compatible with garbage collection because of the heap monotonicity requirement that is violated in presence of garbage collection. They address that by the means of *logical memories* in which locations are never deallocated. Logical memories are connected to physical memories by a lookup table that specifies if a location is live in the physical memory and which physical location it corresponds. Our solution is to close our relation over all isomorphic heaps, hence guaranteeing that related computations will be related for future, possibly garbage collected, heaps.

The logical relation used in this chapter uses a stratified model for space, a technique introduced by Ahmed [5, 4] to provide a semantics of mutable state. We use technique to avoid circularities in chains of references in the heap. Our model is unusual in that it decouples the heap stratification index from the step index. This allows us to maintain invariants that hold for the entirety of the heap, while performing a proof by induction on the step index.

Garbage collection specification. Our formalization of garbage collection is inspired by the one presented by Morrisett *et al.* [101] in their formalization of an abstract machine for a functional language that allows the heap to be garbage collected nondeterministically. Our specification of garbage collection is similar to theirs, with the addition that we require the heap to be fully garbage-collected, which is required to prove the bounds.

Source cost analysis. Source-level cost analysis is complementary to the work presented in this paper. Most related to our work are techniques that apply to higher-order functional languages and support time and space usage [70], and garbage collection [132, 6].

Recent work in relational cost analysis is closely connected to our work as it is also concerned with relating the resource consumption of the execution of two programs. Çiçek *et al.* [35, 30, 29] develop type-and-effect refinement type systems capable of establishing precise bounds on the difference in the execution cost of two programs (or two executions of the same program). The type systems have applications in reasoning about the cost of incremental computation and reasoning about absence of side-channel attacks. Çiçek *et al.* use a logical relation to prove the soundness of the type system with respect to the concrete cost semantics of the language. We drew inspiration from their logical relation when designing the proof framework presented in this chapter.

7.10 Conclusion

I have presented the first formal proof that closure conversion with flat closure representation is safe for space (as well as correct with respect to evaluation semantics). To do so I extended the logical relation that was used to prove behavioral refinement for closure to support reasoning about intensional properties of programs along with extensional ones. Compared to known limitations of standard logical relations, this novel relation provides the ability to reason about resource consumption preservation in the presence of garbage collected heaps. The reasoning can be applied to diverging source programs as well.

Extension to $\lambda_{\texttt{ANF}}$. I briefly outline how the framework of this chapter could be adapted to the full fragment of $\lambda_{\texttt{ANF}}$. To handle let-bound function calls we would have to modify the reachable heap and garbage collection specifications to use the correct set of live roots that include variables that are live after a function returns. For that we would have to extend the semantics with a notion of "stack" to keep track of live variables across function calls since in $\lambda_{\texttt{ANF}}$ functions may return. The garbage collection specification would then have to be invoked not only after function entry, but also after function return. Of course, we would have to add a new compatibility for let-bound application and carry out the corresponding case of the closure conversion proof.

Chapter 8

Evaluation

In this chapter, I evaluate the performance of the CertiCoq compiler. I compare the performance of Gallina programs compiled with CertiCoq, using various configurations, with that of the same programs extracted to OCaml and compiled with the OCaml bytecode and native compilers. I evaluate different CertiCoq configurations:

- I compare the performance of code compiled through CertiCoq using CPS transformation with that of the code compiled through CertiCoq with direct-style ANF transformation.

- I evaluate the performance of CertiCoq-generated code using both the `clang` compiler and the CompCert compiler.

- I evaluate the performance of the code with and without the lambda-lifting closure-optimization pass. This optimization has long been known to have unpredictable behavior, resulting in improved performance in certain programs and in worse performance in others. To better understand the runtime performance of the lambda-lifted code, I evaluate the following aspects of the transformation: 1. Which known call sites are allowed to call a lambda lifted function instead of its closure. 2. Free variables that are live across calls inside the function body in which they appear free, are more expensive to pass as parameters. I run experiments with different threshold values for the number of calls during which a free variable can be live. 3. The effect of inlining lambda-lifted functions inside their escaping wrappers.

8.1 Experimental Setup

I run the experiments on a MacOS machine[1] with a 2.5 GHz Intel Core i7 processor. To compile the generated C code, I use Clang 10.0.1 and CompCert 3.7 with optimization level `-O2`. To compile the extracted OCaml code, I use OCaml 4.07.0.

To measure the execution time of each benchmark I run it for 100 times and take the average time.

[1] `x86_64-apple-darwin18.7.0`

8.2 Benchmarks

For the evaluation of CertiCoq I use a benchmark suite that consists of the following programs.

List append (`demo1`). A microbenchmark that appends two lists of booleans.

List map (`demo2`). A microbenchmark that maps boolean negation over a list of booleans (using the higher-order function `map`).

List summation (`list-sum`). A microbenchmark that constructs a list with 100 elements, all equal to 1, and computes the sum of the elements.

The rest of the benchmarks are larger programs, and some of them are parts of larger verified software developments.

VeriStar entailment checker (`vs-easy` and `vs-hard`). VeriStar [126] is a formally verified decision procedure for a decidable fragment of separation logic. We use CertiCoq to compile the VeriStar prover and evaluate the performance of deciding the validity of two entailments, an easy one `vs-easy` and a harder one `vs-hard`.

Binomial queue (`binom`). We use a verified binomial queue implementation [134, 12] to construct two queues (the first one including all the even numbers from 0 to 2000 and the second one including all the odd numbers in the same range), merge them, and find the maximum element.

Graph coloring (`color`). We use a formally verified implementation of the Kempe/Chaitin algorithm for graph coloring [32, 12] to color a graph.

SHA encryption (`SHA`). We use Secure Hash Algorithm 2 with 256 bit digest (SHA-256) to compute the cryptographic hash of a string consisting of 484 characters.

8.3 Results

8.3.1 CertiCoq CPS *vs.* CertiCoq ANF *vs.* OCaml

In Figure 8.3, I compare the performance of the code compiled with CertiCoq using CPS or direct-style ANF transformation with that of code extracted to OCaml and compiled with the OCaml bytecode compiler (`ocamlc`) and the OCaml native compiler (`ocamlopt`).

	CertiCoq ANF (ms)	CertiCoq CPS (ms)	ocamlopt (ms)	ocamlc (ms)	ANF/ocamlopt
demo1	0.036	0.052	0.017	0.034	2.073
demo2	0.012	0.015	0.005	0.010	2.138
list-sum	0.064	0.140	0.038	0.103	1.714
vs-easy	0.090	0.210	0.019	0.220	4.665
vs-hard	29.224	46.098	12.353	112.940	2.366
binom	2.654	5.201	0.627	5.197	4.233
color	17.716	27.108	-	-	-
sha-fast	22.006	50.820	15.693	69.787	1.402

Figure 8.1: CertiCoq benchmarks: CertiCoq (ANF and CPS) *vs*. OCaml (native and bytecode). OCaml numbers for color are omitted because Coq's built-in extraction generates illegal code.

CPS *vs*. ANF In all cases above, the direct-style code outperforms the continuation-passing style code. The performance of the CPS code could be improved with more sophisticated techniques to reduce heap allocation such as using callee-save registers to pass live free variables to continuations [17], and using optimally linked closure environments [121]. This is consistent with the recent results in the literature by Farvardin and Reppy [47], who observe that programs with deep recursion (such us the programs of this benchmark suite) perform better in direct-style implementations. The performance of CPS could be also improved by adding support for primitive functions in CertiCoq (such as addition of integers).

CertiCoq *vs*. OCaml In the above benchmarks, the code compiled with CertiCoq is between 1.4 and 4.7 times slower than the code generated by the OCaml native compiler. One source of overhead is the implementation of multi-argument functions in CertiCoq. CertiCoq will only uncurry known function calls, leaving escaping functions curried. Because of currying, application of unknown functions will be applied to one argument at a time, causing the allocation of a closure for each intermediate function result. In OCaml, calls to unknown function can be applied to multiple arguments, and hence a closure will be allocate only if an unknown function is partially applied [141].

CertiCoq performs significantly better than the OCaml byte code compiler for the larger benchmarks of the test suite. The two compilation pipelines have similar performance on the smaller benchmarks `demo1` and `demo2`.

8.3.2 CompCert *vs*. Clang

In Figure 8.2, I compare the performance of code compiled with CertiCoq and `clang` with code compiled with CertiCoq and CompCert (`ccomp`). The code compiled with CompCert is slower that the code compiled the Clang compiler for all the benchmarks of the test suite. At worse, code generated with CompCert runs 4.4 times slower

	ANF	ANF	Slowdown	CPS	CPS	Slowdown
	+ Clang	+ ccomp		+ Clang	+ ccomp	
	(ms)	(ms)		(ms)	(ms)	
demo1	0.037	0.039	5.9%	0.053	0.054	1.9%
demo2	0.011	0.015	45.5%	0.015	0.016	11.1%
list-sum	0.063	0.062	1.7%	0.138	0.144	4.3%
vs-easy	0.082	0.101	24%	0.200	0.239	19.2%
vs-hard	28.337	42.336	49.4%	46.484	71.369	53.5%
binom	2.622	11.629	343.4%	5.413	11.229	107.4%
color	17.085	20.341	19.1%	25.801	35.344	37%
sha-fast	21.449	29.251	36.4%	49.779	74.431	49.5%

Figure 8.2: CertiCoq benchmarks: CertiCoq + Clang *vs*. CertiCoq + CompCert.

than code generated with clang (binom), but most of the benchmarks have a more a reasonable overhead. Savary Bélanger *et al*. [119, section 6] explain some of the reasons why clang outperforms CompCert on the output of CertiCoq.

8.3.3 Lambda lifting

Lambda lifting [67] is a transformation that turns free variables of known functions into formal parameters. Lambda lifting has been used by compilers to eliminate closures for known functions. CertiCoq's lambda lifting transformation is described in Section 4.3.5. For clarity, I review its functionality here. CertiCoq will split each function into two instances: one to be used at known call sites and one to be used at escaping occurrences of the function name (parameter passing, function return, constructor argument). The known function call can be lambda-lifted, meaning that its free variables will be passed as parameters. The escaping instance will be a wrapper around the known instance and will immediately call the (possibly lambda-lifted) known function. To decide which free variables can become parameters and which functions can be lambda-lifted, we follow these rules.

- Either all free variables of a function will become parameters or none. That is, after lambda lifting a function must be closed. Partial lambda lifting of a function creates unnecessary closures, as described in Section 4.3.5.

- After lambda lifting, the number of arguments of each function should not exceed the number of available registers.

- A free variable that remains live across many calls inside the body of the function in which it appears is more expensive to pass as parameter than to store in a closure environment, and hence such free variables are not passed as parameters.

Furthermore, at each known call site we may choose to call the escaping wrapper, or inline it to call the known function. We follow two different heuristics:

- Conservative strategy: the known function instance is called only if the extra arguments are already in scope (bound in the local scope or free in the current function). This way we avoid potential growth in closure environments.

- Aggressive strategy: we inline all known calls to call the known lambda-lifted function.

Lastly, we can choose to inline the calls to the known functions from their wrappers, to avoid the cost of calling the known function from the escaping wrapper when the function is called from an unknown call site.

In our default configuration, which we base on the results of these experiments, we choose to inline all calls to known functions, to allow variables to remain live across at most one call, and not to inline the calls to known functions from wrappers.

First, I compare the performance of CertiCoq code without the lambda-lifting transformation with the performance of the code compiled with lambda lifting enabled. Then, I compare the performance of different lambda lifting configurations, based on the design parameters I described above. The benchmarks `demo1`, `demo2`, and `list-sum` remain unaffected by the transformation and are omitted from the results.

Default configuration

Figure 8.3 shows the performance of CertiCoq compiled with with the lambda lifting optimization enabled (`-O 1`) compared with CertiCoq code compiled without lambda lifting.

	ANF (ms)	ANF + LL (ms)	Speedup	CPS (ms)	CPS + LL (ms)	Speedup
`vs-easy`	0.094	0.086	8.2%	0.214	0.211	1.4%
`vs-hard`	28.115	26.457	5.9%	45.470	45.689	-0.5%
`binom`	2.614	2.551	2.4%	5.353	6.562	-22.6%
`color`	17.495	16.906	3.4%	25.942	25.557	1.5%
`sha-fast`	21.617	21.324	1.4%	49.817	49.851	0.1%

Figure 8.3: CertiCoq benchmarks: CertiCoq ANF/CPS *vs.* CertiCoq ANF/CPS + LL (Lambda Lifting).

The most significant improvement is exhibited by the VeriStar benchmark: we observe a a 8.2% speedup in `vs-easy` and 5.9% speedup in `vs-hard`. Interestingly, lambda lifting incurs a 22.6% overhead in the CPS version of `binom`. After inspection of the code, the overhead appears to be coming exclusively from eager projection of free variables from the environment of continuation closures. In lambda-lifted code, continuations are always called as unknown functions, and they always enter through the wrapper. Upon entry, all free variables will be projected out of the environment all at once, even if they are not needed for this execution (because they are used in a different branch of a case statement). For example, in `binom`, the recursive function find-max has such a continuation, causing the projection of unneeded free variables of the environment at each iteration. To avoid this issue with CPS, we could turn off lambda lifting for non-recursive functions that escape (such us continuations).

Size of the generated code. In direct code, lambda-lifting decreases the size of the generated code. For example vs-easy compiles to 54623 loc (lines of C code) with lambda lifting and to 57071 loc without. For CPS, lambda lifting increases the size of the code (71464 *vs*. 75493 for vs-easy).

Remark about the performance of lambda-lifting optimization. Lambda lifting appears to improve performance the most when it is able to find a free variable that can be passed through a chain of known calls to the function that uses it. An example of such case is a patrially applied function in the the VeriStar benchmark. This benchmark uses a comparison function from the Coq standard library that is parameterized by the type of comparison (less, equal or greater) it performs.

```
Inductive comparison : Set:=
  Eq : comparison | Lt : comparison | Gt : comparison

Fixpoint compare_cont (r : comparison) (x y : positive):= ...
```

The VeriStar function expr_cmp uses the partial application of compare_cont to compare for equality.

```
Definition compare = compare_cont Eq.

Definition expr_cmp e e' : comparison:=
  ... compare v v' ...
```

In ANF, the above example will look like,

```
let x = Eq () in
let compare = compare_cont Eq in

fun expr_cmp e e' := ... let z = compare v in let w = z v' in ...
```

In the above code, the variable compare (which is a closure) is a *free variable* of the function that uses it, and it has to be passed to it through its closure environment by each caller. This, causes compare to be a free variable of the caller of expr_cmp and so on. Furthermore, at application time, compare will be applied to its three arguments one by one, allocating a closure for each intermediate application. Now, crucially, compare_cont, which after uncurrying is just an uncurry wrapper, will be inlined, causing the application of compare to be fully applied:

```
fun expr_cmp e e' : comparison := ... compare_cont' x v v' ...
```

Where `compare_cont'` is the uncurried version of `compare_cont`.

Now, instead of the partially applied `compare`, the program will call the *fully applied* `compare_cont' x v v'`. This is already an improvement, but `x` is a free variable of `expr_cmp`, so again it has to be passed to through a chain of closure environments, causing `expr_cmp` and functions that call it to be closure converted.[2] Fortunately, with lambda lifting `x` can be passed as parameter to `expr_cmp` and all functions that call `expr_cmp` and their callers, eliminating all closures of these functions and turning them to efficient C-style calls.

Which call sites to inline?

In the following table I evaluate the performance of the conservative inlining strategy *vs*. the performance of the aggressive inlining strategy at known call sites. The conservative strategy only calls the known lambda-lifted functions when the extra parameters are either already free variables of the caller, or local variables in scope (that is, it is guaranteed to never increase the free variables of a function). The aggressive strategy inlines all known calls to they directly call the lambda lifted function. The aggressive strategy might increase closure allocation but also it exposes more opportunities for optimization.

	$\text{LL}^{\text{c}}_{\text{ANF}}$ (ms)	$\text{LL}^{\text{a}}_{\text{ANF}}$ (ms)	Speedup	$\text{LL}^{\text{c}}_{\text{CPS}}$ (ms)	$\text{LL}^{\text{a}}_{\text{CPS}}$ (ms)	Speedup
`vs-easy`	0.087	0.083	4.1%	0.214	0.197	8%
`vs-hard`	28.413	26.652	6.2%	47.391	45.554	3.9%
`binom`	2.544	2.527	0.7%	6.583	6.536	0.7%
`color`	17.277	17.017	1.5%	26.288	25.684	2.3%
`sha-fast`	21.863	21.556	1.4%	50.895	50.417	0.9%

Figure 8.4: CertiCoq benchmarks: CertiCoq ANF/CPS + LL^{c} (conservative inlining) *vs*. CertiCoq ANF/CPS + LL^{a} (aggressive inlining) .

We observe that the aggressive strategy (used by our default configuration) is more efficient. This is because the extra free variables inside the callers can be turned into parameters themselves, as described above for the `vs-easy` benchmark, completely eliminating closure allocation for these functions. Further analysis could be used to improve this heuristic decision, by approximating which free variables might increase closure allocation if passed as parameters.

Which free variables to pass as parameters?

CertiCoq's lambda lifting transformation is parametric on the number of intermediate calls during which a free variable may be live. I evaluate the performance of lambda

[2] `x` has an unboxed representation so it could, in principle, be defined as a global variable in the generated C code. This optimization is not performed by CertiCoq. However, even if CertiCoq performed this optimization, the above scenario would still apply to boxed constructors (*i.e.*, heap allocated objects).

lifting when a free variable can be live during at most 0, 1, 2, and 10 calls in order to become a parameter. The results are shown in Figure 8.5.

	LL^0 (ms)	LL^1 (ms)	LL^2 (ms)	LL^{10} (ms)
vs-easy	0.086	0.076	0.081	0.079
vs-hard	26.553	26.461	26.745	26.736
binom	2.508	2.500	2.508	2.525
color	16.942	17.081	16.909	17.053
sha-fast	21.432	21.331	21.274	21.149

Figure 8.5: CertiCoq benchmarks: comparison of lambda lifting allowing parameters to be live at most during n calls (LL^n).

We observe that there is a trade-off between turning free variables to parameter eliminating a closure environment and the number of intermediate calls in the body of the lambda-lifted function during which these variables remain live. Parameters (typically stored in registers) that remain live across calls generate memory traffic because their values have to be saved into the stack before the call and restored after the call. For the VeriStar benchmark (vs-easy and vs-hard), we observe that the performance peaks when the threshold of calls is 1. After that, the performance is worsened. For binom the code remains unaffected by this choice, hence the performance stays virtually the same. For color, the performance becomes somewhat worsened at threshold 1, but it gets improved for larger values, perhaps as more opportunities for closure elimination are exposed. The performance of sha-fast is slightly improved by increasing the threshold value.

Inlining of Lambda-lifting Wrappers

Lastly, I evaluate the effect on the performance of the code when the call from the wrapper is inlined *vs.* when it is not inlined. This inlining happens after closure conversion, to avoid exponential (in the size of the nesting depth) blowup in the size of the code.

	LL$^{\mathrm{n}}_{\mathrm{ANF}}$ (ms)	LL$^{\mathrm{i}}_{\mathrm{ANF}}$ (ms)	Speedup	LL$^{\mathrm{n}}_{\mathrm{CPS}}$ (ms)	LL$^{\mathrm{i}}_{\mathrm{CPS}}$ (ms)	Speedup
vs-easy	0.079	0.079	-0.2%	0.187	0.193	-3.1%
vs-hard	26.620	26.646	-0.1%	45.856	45.359	1.1%
binom	2.508	2.566	-2.3%	6.518	6.569	-0.8%
color	16.988	16.925	0.4%	25.598	25.498	0.4%
sha-fast	21.353	21.213	0.7%	49.632	48.955	1.4%

Figure 8.6: CertiCoq benchmarks: CertiCoq ANF/CPS + LL$^{\mathrm{n}}$ (no inlining in wrappers) *vs.* CertiCoq ANF/CPS + LL$^{\mathrm{i}}$ (inlining).

We observe that inlining of known calls inside the escaping wrappers has very little effect on performance. As expected, it also increases the size of the code. The performance behavior can be due to various reasons. First, tail calls to known functions are implemented as efficient jumps by `clang` (which performs tail-call optimization when `-O2` is enabled). Therefore such calls are already very efficient and inlining them has little performance improvement.

Moreover, many of the escaping functions are functions that have been uncurried, using the technique described in Section 4.3.3. For such functions, the lambda-lifted function has only known occurrences, some of them inside the uncurry wrapper, since escaping occurrences of this function always refer to the uncurry wrapper. Therefore, the escaping lambda-lifted wrapper is never used.[3]

Conclusion and Related Work

Based on the results of these experiments we pick our default lambda lifting configuration: 1. We inline all known calls so that they call directly the lambda-lifted functions 2. we allow variables to be live during at most one call, and 3. we do not inline calls to known lambda-lifted functions inside their escaping wrappers. Our lambda lifting approach implements more heuristics than lambda lifting in other compilers, allowing us to have a more selective lambda-lifting strategy. With our lambda lifting strategy, we have observed significant performance improvement in some of our ANF benchmarks, and no performance overhead for the set of benchmarks that were used. For CPS, the "optimization" results in worse performance, therefore it is best disabled for CPS compilation.

Flambda. OCaml's optimizing Flambda pipeline [87, Chapter 21] uses a similar lambda-lifting transformation to unbox closure environments and store free variables in registers. The optimization is turned off by default, as it may incur performance overhead in some programs. To try to mitigate the overhead, small enough functions will be duplicated by Flambda, so that two different versions of function are created (which are not defined in terms of each other in the way escaping wrappers are). The original function will be used at escaping positions and the lambda-lifted one at known positions.

This strategy does not remove the performance overhead (after all, tail-calls to known closed functions are very efficient) and, moreover, it will miss the opportunity to optimize unknown calls to recursive functions with free variables. In CertiCoq, such calls will enter through the escaping wrapper, that will project all free variables once and for all and then it will call the efficient C-style known function.[4] We

[3] The uncurry wrappers may also become lambda lifted but this will have no observable effect in the generated code. The code will always enter through the escaping wrapper, since uncurry wrappers only escape, and the lambda lifted function will always be small enough to be inlined inside the body of the lambda-lifting wrapper of an uncurry wrapper.

[4] Escaping functions that are not recursive are best not to lambda-lift to avoid eager projection of free variables.

anticipate that the lambda-lifting transformation of Flambda could be improved by implementing some of the design decisions described in this chapter.

GHC. GHC uses lambda lifting as part of its GHC Core compilation-by-transformation optimizing pipeline [68, 118]. This transformation in GHC also appears to improve the performance for certain programs but worsen it for others. The inverse transformation, lambda dropping [42] that turns parameters to free variables is also explored. Lambda lifting in GHC is performed selectively, in order to completely avoid lambda lifting functions that escape and are partially applied. However, GHC's lambda lifting will not split functions to known and escaping instances that could expose more optimization opportunities and would allow for a more flexible selective lambda-lifting strategy. GHC's selective lambda lifting approach results in modest performance improvement (1-2%).

Chapter 9

Conclusions and Future Work

I have presented the design, implementation, and verification of CertiCoq's middle end, an optimization pipeline that compiles CertiCoq's $\lambda_{\texttt{ANF}}$ intermediate language to a first-order subset of the language that can be readily compiled to first-order low-level languages.

The design of the $\lambda_{\texttt{ANF}}$ pipeline follows a compilation-by-transformation approach plugging together small, modular transformations that can be verified independently and composed in arbitrary order. That allows a lot of flexibility in fine-tuning the pipeline: experimenting with how transformations interact can lead to significant improvements in the generated code. This design allows new transformations to be easily introduced, and transformations to be reused multiple times (which is typical for inlining and shrink reduction). GHC's Core-to-Core and OCaml's Flambda (unverified) optimizing pipelines follow a similar design.

To verify the $\lambda_{\texttt{ANF}}$ pipeline I developed a proof framework based on step-indexed logical relations. The framework presented is novel in that it supports reasoning about divergence preservation (with the use of a postcondition to relate the number of steps that the two programs take). Most importantly, I show how individual logical relation proofs can be composed in order to derive a top-level theorem that supports correctness of separate compilation, providing a lightweight solution to the lack of vertical compositionality of logical relations. This technique can be used to show correctness of separate compilation when programs are compiled through pipelines that use the same series of intermediate languages but do not necessarily use the same transformations. It is lightweight in the sense that it applies to any standard formulation of logical relations and requires no modification to individual proofs of transformations. I use this framework to show that programs compiled separately through the $\lambda_{\texttt{ANF}}$ pipeline can be linked with each other, regardless of the particular set of $\lambda_{\texttt{ANF}}$ optimizations that they use.

In the last part of the thesis, I showed how to extend the relational proof framework to reason about the resource consumption of programs and I use it to show that CertiCoq's closure conversion transformation is safe for space. This is the first proof that a closure conversion transformation is safe for space.

9.1 Conclusions

I summarize some the take-away messages from the research that I presented in this thesis.

- Keeping transformations small and modular is a desirable design choice for a compiler especially in the context of a proved-correct compiler. Many aspects of compilation and optimization can be expressed as small modular same-language transformations. That leads to a more extensible compiler design that is more amenable to formal verification.

- Logical relations are particularly suitable for proving correctness of modular compiler transformations. The same relation can be used multiple times to verify different transformations. Compositional reasoning is achieved using compatibility lemmas that are proved once and for all and used in every correctness proof.

- Despite their lack of transitivity, logical relations *can* be composed to derive a *compositional* end-to-end correctness theorem. Programs compiled through any pipeline that satisfies such a top-level theorem can be linked at the target level, preserving the behavior of linking in the source level and compiling the program as a whole unit. The limitation of this approach is that it can be used only for pipelines that go through exactly the same series of intermediate representations.

- For a flexible pipeline design, where transformations may be reordered multiple times and composed in different ways, it is important to be able to link modules that have been compiled with different pipeline configurations. The relational proof framework of Chapter 5 provides this flexibility: the individual proofs of transformations can be composed to prove a compositional correctness specification for any combination of the transformations.

- We can use logical relations to reason about divergence preservation of programs. Previous solutions to that problem required adding an extra "tick" instruction with no computational meaning to the intermediate languages of the compiler. The solution presented in this thesis avoids that by allowing to establish arbitrary relations on the steps that the execution of the two programs require.

- Lambda lifting, if implemented carefully, can have a measurable performance improvement in the generated code. In CertiCoq, we have used lambda lifting to eliminate function closures and pass free variables of functions in registers through chains of known function calls, improving both the performance of the generated code.

- The same relational framework that is used to prove functional correctness can be used to prove preservation about the resource consumption of programs.

The main proof can be parametric in the additional resource invariants that are being established, allowing proofs to be easily extended to provide more guarantees about resource preservation.

9.2 Future Work

I outline some potential future directions for the work I have presented.

Additional $\lambda_{\texttt{ANF}}$ optimizations. Because of its modular nature, the $\lambda_{\texttt{ANF}}$ pipeline can serve as a framework for implementing and verifying optimizations for functional languages. The $\lambda_{\texttt{ANF}}$ pipeline can be extended with more optimizations, such as argument specialization (to remove higher-order arguments and therefore reduce closure allocation), unboxing (*e.g.*, of tupled arguments), common subexpression elimination, code-motion transformations, and more sophisticated closure environment representations.

Extension to Different Source Languages. The $\lambda_{\texttt{ANF}}$ intermediate representation and pipeline could provide a general, verified compilation pipeline for functional languages. With modest extensions to the $\lambda_{\texttt{ANF}}$ language, the pipeline could be used as a compilation middle end for other functional languages. In particular, with the addition of primitive operations and primitive data types, the $\lambda_{\texttt{ANF}}$ pipeline could handle compilation of big subsets of languages like OCaml. This is also require to handle Coq's new facility for primitive objects (native integers, floats, and arrays).

Compositional correctness framework. It would be interesting to investigate whether the compositional proof framework can be strengthened by lifting the restriction that compilers should use the same sequence of intermediate representations. This, if possible, would most likely require implementing and proving correct back translations between adjacent intermediate representations.

End-to-end correctness for CertiCoq. The framework presented here could be extended to verify the front-end CertiCoq transformations and the code-generation phase. Defining the framework in a language-generic way could be particularly useful for this purpose.

Gallina/C verification. CertiCoq emits C, allowing Gallina programs to interact with C programs. It would be especially useful to be able to verify the interoperation of Gallina and C code. This could be done in Verified Software Toolchain (VST) [11] that allows Clight programs to be verified in Coq using separation logic. An open question is whether the top-level theorem of CertiCoq suffices for this purpose or its the statement (and the relational framework) must be strengthened.

Bibliography

[1] Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, page 309–322, New York, NY, USA, 2008. Association for Computing Machinery.

[2] Norman Adams, David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. Orbit: An optimizing compiler for scheme. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, page 219–233, New York, NY, USA, 1986. Association for Computing Machinery.

[3] Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In Peter Sestoft, editor, *Programming Languages and Systems*, pages 69–83, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[4] Amal J. Ahmed. *Semantics of Types for Mutable State.* PhD dissertation, Princeton University, 2004.

[5] Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, page 75, USA, 2002. IEEE Computer Society.

[6] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Parametric inference of memory requirements for garbage collected languages. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 121–130, New York, NY, USA, 2010. ACM.

[7] Roberto M. Amadio and Yann Régis-Gianas. Certifying and reasoning on cost annotations of functional programs. In *Proceedings of the Second International Conference on Foundational and Practical Aspects of Resource Analysis*, FOPARA'11, pages 72–89, Berlin, Heidelberg, 2012. Springer-Verlag.

[8] Abhishek Anand, Andrew W. Appel, John Gregory Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Weaver. CertiCoq: A verified compiler for Coq. In *CoqPL'17: The Third International Workshop on Coq for Programming Languages*, 2017.

[9] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards certified meta-programming with typed template-coq. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving*, pages 20–39, Cham, 2018. Springer International Publishing.

[10] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, 1992.

[11] Andrew W. Appel. Verified software toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, page 1–17, Berlin, Heidelberg, 2011. Springer-Verlag.

[12] Andrew W. Appel. Verified functional algorithms, 2020. Version 1.4, http://softwarefoundations.cis.upenn.edu.

[13] Andrew W. Appel and Marcelo J. R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University Department of Computer Science, 1993.

[14] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 293–302, New York, NY, USA, 1989. Association for Computing Machinery.

[15] Andrew W. Appel and Trevor Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 7(5):515–540, September 1997.

[16] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, September 2001.

[17] Andrew W. Appel and Zhong Shao. Callee-save registers in continuation-passing style. *Lisp Symb. Comput.*, 5(3):191–221, September 1992.

[18] Andrew W. Appel and Zhong Shao. Empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47–74, 1996.

[19] Henk Barendregt and Herman Geuvers. *Proof-Assistants Using Dependent Type Systems*, page 1149–1238. Elsevier Science Publishers B. V., NLD, 2001.

[20] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[21] Olivier Savary Bélanger. *Verified Extraction for Coq*. PhD dissertation, Princeton University, 2019.

[22] Olivier Savary Bélanger and Andrew W. Appel. Shrink fast correctly! In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, PPDP '17, page 49–60, New York, NY, USA, 2017. Association for Computing Machinery.

[23] Nick Benton, Andrew Kennedy, Sam Lindley, and Claudio Russo. Shrinking reductions in SML.NET. In Clemens Grelck, Frank Huch, Greg J. Michaelson, and Phil Trinder, editors, *Implementation and Application of Functional Languages*, pages 142–159, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[24] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. CompCertS: A memory-aware verified C compiler using pointer as integer semantics. In *ITP 2017 - 8th International Conference on Interactive Theorem Proving*, volume 10499 of *ITP 2017: Interactive Theorem Proving*, pages 81–97, Brasilia, Brazil, September 2017. Springer.

[25] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

[26] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. End-to-end verification of stack-space bounds for C programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 270–281, New York, NY, USA, 2014. Association for Computing Machinery.

[27] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 467–478, New York, NY, USA, 2015. ACM.

[28] Jason Carr. *Formally Verified Space-Safety for Program Transformations*. Master's thesis, Rochester Institute of Technology, 2019.

[29] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. Relational cost analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 316–329, New York, NY, USA, 2017. ACM.

[30] Ezgi Çiçek, Zoe Paraskevopoulou, and Deepak Garg. A type theory for incremental computational complexity with control flow changes. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 132–145, New York, NY, USA, 2016. ACM.

[31] Ezgi Çiçek, Weihao Qu, Gilles Barthe, Marco Gaboardi, and Deepak Garg. Bidirectional type checking for relational properties. In *Proceedings of the 40th*

ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, page 533–547, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. Register allocation via coloring. *Computer languages*, 6(1):47–57, 1981.

[33] Arthur Charguéraud. Pretty-big-step semantics. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 41–60, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[34] Adam Chlipala. A verified compiler for an impure functional language. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 93–106, New York, NY, USA, 2010. Association for Computing Machinery.

[35] Ezgi Çiçek, Deepak Garg, and Umut Acar. Refinement types for incremental computational complexity. In Jan Vitek, editor, *Programming Languages and Systems*, pages 406–431, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[36] Youyou Cong, Leo Osvald, Grégory M. Essertel, and Tiark Rompf. Compiling with continuations, or without? whatever. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.

[37] The Coq Development Team. *The Coq Reference Manual, version 8.11.2*, August 2020. Available electronically at https://coq.inria.fr/refman.

[38] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2–3):95–120, February 1988.

[39] Patrick Cousot and Radhia Cousot. Inductive definitions, semantics and abstract interpretations. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, page 83–94, New York, NY, USA, 1992. Association for Computing Machinery.

[40] Karl Crary and Stephanie Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 184–198, New York, NY, USA, 2000. ACM.

[41] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '01, page 162–174, New York, NY, USA, 2001. Association for Computing Machinery.

[42] Olivier Danvy and Ulrik P. Schultz. Lambda-dropping: Transforming recursive equations into programs with block structure. *SIGPLAN Not.*, 32(12):90–106, December 1997.

[43] Zaynah Dargaye and Xavier Leroy. A verified framework for higher-order uncurrying optimizations. *Higher-Order and Symbolic Computation*, 22(3):199–231, 2009.

[44] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 273–282, New York, NY, USA, 1992. ACM.

[45] Colin Eberhardt. *What Is WebAssembly?* 2020.

[46] Vyacheslav Egorov. Grokking V8 closures for fun (and profit?), 2012. Blog post. https://mrale.ph/blog/2012/09/23/grokking-v8-closures-for-fun.

[47] Kavon Farvardin and John Reppy. From folklore to fact: Comparing implementations of stacks and continuations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 75–90, New York, NY, USA, 2020. Association for Computing Machinery.

[48] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM.

[49] Matthew Fluet and Stephen Weeks. Contification using dominators. *SIGPLAN Not.*, 36(10):2–13, October 2001.

[50] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, USA, 1989.

[51] David Glasser. An interesting kind of JavaScript memory leak, 2013. Blog post.
https://blog.meteor.com/an-interesting-kind-of-javascript-memory-leak-8b47d2e7f156.

[52] Benjamin Goldberg and Young Gil Park. Higher order escape analysis: Optimizing stack allocation in functional program implementations. In *Proceedings of the 3rd European Symposium on Programming*, ESOP '90, page 152–160, Berlin, Heidelberg, 1990. Springer-Verlag.

[53] Alejandro Gómez-Londoño, Johannes Åman Pohjola, Hira Taqdees Syeda, and Magnus O. Myreen. Do you have space for dessert? A verified space cost semantics for cakeml programs. *To appeat at Proc. ACM Program. Lang.*, 1(OOPSLA).

[54] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep

specifications and certified abstraction layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 595–608, New York, NY, USA, 2015. Association for Computing Machinery.

[55] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent os kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 653–669, USA, 2016. USENIX Association.

[56] Jörgen Gustavsson and David Sands. A foundation for space-safe transformations of call-by-need programs. *Electronic Notes in Theoretical Computer Science*, 26:69 – 86, 1999. HOOTS '99, Higher Order Operational Techniques in Semantics.

[57] Jörgen Gustavsson and David Sands. Possibilities and limitations of call-by-need space improvement. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 265–276, New York, NY, USA, 2001. ACM.

[58] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery.

[59] John Hannan and Patrick Hicks. Higher-order uncurrying. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, page 1–11, New York, NY, USA, 1998. Association for Computing Machinery.

[60] Fergus Henderson. Accurate garbage collection in an uncooperative environment. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, page 150–156, New York, NY, USA, 2002. Association for Computing Machinery.

[61] Jason Hickey, Anil Madhavapeddy, and Yaron Minsky. *Real World OCaml*. 2014.

[62] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 359–373, New York, NY, USA, 2017. Association for Computing Machinery.

[63] Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, pages 103–112, 1996.

[64] Chung-Kil Hur and Derek Dreyer. A kripke logical relation between ml and assembly. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 133–146, New York, NY, USA, 2011. ACM.

[65] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. The marriage of bisimulations and kripke logical relations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 59–72, New York, NY, USA, 2012. Association for Computing Machinery.

[66] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, page 8, USA, 2012. USENIX Association.

[67] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proc. of a Conference on Functional Programming Languages and Computer Architecture*, page 190–203, Berlin, Heidelberg, 1985. Springer-Verlag.

[68] Simon L. Peyton Jones. Compiling haskell by program transformation: A report from the trenches. In Hanne Riis Nielson, editor, *Programming Languages and Systems — ESOP '96*, pages 18–44, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[69] Simon Peyton Jones and André Santos. Compilation by transformation in the glasgow haskell compiler. In Kevin Hammond, David N. Turner, and Patrick M. Sansom, editors, *Functional Programming, Glasgow 1994*, pages 184–204, London, 1995. Springer London.

[70] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 223–236, New York, NY, USA, 2010. ACM.

[71] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating LR(1) parsers. In *ESOP 2012 - Programming Languages and Systems - 21st European Symposium on Programming*, volume 7211 of *Lecture Notes in Computer Science*, pages 397–416, Tallinn, Estonia, March 2012. Springer.

[72] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 637–650, New York, NY, USA, 2015. Association for Computing Machinery.

[73] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Lightweight verification of separate compilation. *SIGPLAN Not.*, 51(1):178–190, January 2016.

[74] Andrew W. Keep, Alex Hearn, and R. Kent Dybvig. Optimizing closures in o(0) time. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*, Scheme '12, pages 30–35, New York, NY, USA, 2012. ACM.

[75] R. Kelsey and P. Hudak. Realistic compilation by program transformation (detailed summary). In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 281–292, New York, NY, USA, 1989. Association for Computing Machinery.

[76] Andrew Kennedy. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 177–190, New York, NY, USA, 2007. ACM.

[77] Chung kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. Parametric bisimulations: A logical step forward. Technical report, 01 2014.

[78] Andrzej Krzemieński. A serious bug in GCC, 2017. Blog post. https://akrzemi1.wordpress.com/2017/04/27/a-serious-bug-in-gcc/.

[79] Ramana Kumar. Self-compilation and self-verification. Technical Report UCAM-CL-TR-879, University of Cambridge, Computer Laboratory, February 2016.

[80] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. *SIGPLAN Not.*, 49(1):179–191, January 2014.

[81] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[82] Xuan Bach Le, Thanh-Toan Nguyen, Wei-Ngan Chin, and Aquinas Hobor. A certified decision procedure for tree shares. In Zhenhua Duan and Luke Ong, editors, *Formal Methods and Software Engineering - 19th International Conference on Formal Engineering Methods, ICFEM 2017, Xi'an, China, November 13-17, 2017, Proceedings*, volume 10610 of *Lecture Notes in Computer Science*, pages 226–242. Springer, 2017.

[83] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, page 42–54, New York, NY, USA, 2006. Association for Computing Machinery.

[84] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. *SIGPLAN Not.*, 41(1):42–54, January 2006.

[85] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[86] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[87] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 4.11*, August 2020. Available electronically at https://coq.inria.fr/refman.

[88] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, February 2009.

[89] Pierre Letouzey. *Programmation fonctionnelle certifiée : l'extraction de programmes dans l'assistant Coq*. PhD thesis, 2004. Thèse de doctorat Informatique Paris 11 2004.

[90] Pierre Letouzey. Extraction in coq: An overview. In *Proceedings of the 4th Conference on Computability in Europe: Logic and Theory of Algorithms*, CiE '08, page 359–369, Berlin, Heidelberg, 2008. Springer-Verlag.

[91] Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, page 3–10, New York, NY, USA, 2007. Association for Computing Machinery.

[92] Luke Maurer, Paul Downen, Zena M. Ariola, and Simon Peyton Jones. Compiling without continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 482–494, New York, NY, USA, 2017. ACM.

[93] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960.

[94] John McCarthy. Computer programs for checking mathematical proofs. In *AMS Symposium on Recursive Function Theory*, 1961.

[95] John Mccarthy and James Painter. Correctness of a compiler for arithmetic expressions. In J.T. Schwartz, editor, *Proceedings of a Symposium in Applied Mathematics, Vol. 19.*

[96] Andrew McCreight, Tim Chevalier, and Andrew Tolmach. A certified framework for compiling and executing garbage-collected languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, page 273–284, New York, NY, USA, 2010. Association for Computing Machinery.

[97] Robin Milner. Implementation and applications of scott's logic for computable functions. In *Proceedings of ACM Conference on Proving Assertions about Programs*, page 1–6, New York, NY, USA, 1972. Association for Computing Machinery.

[98] Robin Milner and R.W. Weyhrauch. Proving compiler correctness in a mechanised logic. *Machine Intelligence*, 7:51–73, 1972.

[99] Yasuhiko Minamide. Space-profiling semantics of the call-by-value lambda calculus and the CPS transformation. *Electr. Notes Theor. Comput. Sci.*, 26:105–120, 1999.

[100] Yasuhiko Minamide. Space-profiling semantics of the call-by-value lambda calculus and the CPS transformation. *Electr. Notes Theor. Comput. Sci.*, 26:105–120, 1999.

[101] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. In *Higher Order Operational Techniques in Semantics, Publications of the Newton Institute*, pages 175–226. Cambridge University Press, 1997.

[102] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. Œuf: Minimizing the Coq extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 172–185, New York, NY, USA, 2018. Association for Computing Machinery.

[103] Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ML from higher-order logic. In *International Conference on Functional Programming (ICFP)*, pages 115–126. ACM Press, September 2012.

[104] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, page 166–178, New York, NY, USA, 2015. Association for Computing Machinery.

[105] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In Peter Thiemann, editor, *Programming Languages and Systems*, pages 589–615, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[106] Scott Owens, Michael Norrish, Ramana Kumar, Magnus O. Myreen, and Yong Kiam Tan. Verifying efficient function calls in CakeML. *Proc. ACM Program. Lang.*, 1(ICFP):18:1–18:27, August 2017.

[107] Zoe Paraskevopoulou and Andrew W. Appel. Closure conversion is safe for space. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.

[108] Daniel Patterson and Amal Ahmed. The next 700 compiler correctness theorems (functional pearl). *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.

[109] James T. Perconti and Amal Ahmed. Verifying an open compiler using multi-language semantics. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*, page 128–148, Berlin, Heidelberg, 2014. Springer-Verlag.

[110] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, pages 209–228, New York, NY, 1990. Springer-Verlag.

[111] Andrew Pitts. *Howe's method for higher-order languages*, page 197–232. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2011.

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

[113] Weihao Qu, Marco Gaboardi, and Deepak Garg. Relational cost analysis for functional-imperative programs. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.

[114] J. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, 1983.

[115] Amr Sabry and Philip Wadler. A reflection on call-by-value. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, ICFP '96, page 13–24, New York, NY, USA, 1996. Association for Computing Machinery.

[116] David Sands. Operational theories of improvement in functional languages. In Rogardt Heldal, Carsten Kehler Holst, and Philip Wadler, editors, *Functional Programming, Glasgow 1991*, pages 298–311, London, 1992. Springer London.

[117] David Sands. Improvement theory and its applications. In Andrew D. Gordon and Andrew M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 275–306, New York, NY, USA, 1998. Cambridge University Press.

[118] Andre Santos. *Compilation by transformation for non-strict functional languages*. PhD thesis, University of Glasgow, July 1995.

[119] Olivier Savary Bélanger, Matthew Z. Weaver, and Andrew W. Appel. Certified code generation from CPS to C. In preparation. https://www.cs.princeton.edu/~appel/papers/CPStoC.pdf, October 2019.

[120] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, LFP '94, page 150–161, New York, NY, USA, 1994. Association for Computing Machinery.

[121] Zhong Shao and Andrew W. Appel. Efficient and safe-for-space closure conversion. *ACM Trans. Program. Lang. Syst.*, 22(1):129–161, January 2000.

[122] Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. CompCertM: CompCert with C-assembly linking and lightweight modular verification. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[123] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, February 2020.

[124] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! Verification of type checking and erasure for Coq, in Coq. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[125] Guy L. Steele. Rabbit: A compiler for scheme. Technical report, USA, 1978.

[126] Gordon Stewart, Lennart Beringer, and Andrew W. Appel. Verified heap theorem prover by paramodulation. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, page 3–14, New York, NY, USA, 2012. Association for Computing Machinery.

[127] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional CompCert. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 275–287, New York, NY, USA, 2015. ACM.

[128] William W. Tait. Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic*, 32(2):198–212, 1967.

[129] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. A new verified compiler backend for CakeML. In *International Conference on Functional Programming (ICFP)*, pages 60–73. ACM Press, September 2016. Invited to special issue of Journal of Functional Programming.

[130] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29, 2019.

[131] Ken Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, August 1984.

[132] Leena Unnikrishnan and Scott D. Stoller. Parametric heap usage analysis for functional programs. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09, pages 139–148, New York, NY, USA, 2009. ACM.

[133] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3), June 2013.

[134] Jean Vuillemin. A data structure for manipulating priority queues. *Commun. ACM*, 21(4):309–315, April 1978.

[135] Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, November 2015.

[136] H. Wang. Toward mechanical mathematics. *IBM Journal of Research and Development*, 4(1):2–22, 1960.

[137] Peng Wang, Di Wang, and Adam Chlipala. Timl: A functional language for practical complexity analysis with invariants. *Proc. ACM Program. Lang.*, 1(OOPSLA):79:1–79:26, October 2017.

[138] Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. Certifying graph-manipulating c programs via localizations within data structures. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[139] Stephen Weeks. Whole-program compilation in MLton. In *Proceedings of the 2006 Workshop on ML*, ML '06, page 1, New York, NY, USA, 2006. Association for Computing Machinery.

[140] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.

[141] Pascal Zimmer. The dangers of being too partial, 2008. Blog post. https://blog.janestreet.com/the-dangers-of-being-too-partial/.