Verified Extraction for Coq

Olivier Savary Bélanger

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: Professor Andrew W. Appel

November 2019

Abstract

Interactive theorem provers allow for the development, in the same environment, of programs and of proofs about them. The programmatic portion of the development can then be extracted to code which is then compiled into an executable. However, unless both the extraction and compilation processes are formally verified, one has no guarantees that the proofs developed still apply to the resulting executable. This thesis describes my work on CertiCoq, a verified extraction pipeline for the Coq theorem prover composing with the CompCert C verified compiler to achieve end-to-end correctness guarantees.

I present a proof framework to prove optimizations over the continuation-passing style (CPS) intermediate representation (IR) used in CertiCoq. This framework has been used by me and others to prove the correctness of nontrivial optimizations. I focus on a novel proof of correctness for a shrink reduction algorithm, a transformation combining in a single pass multiple optimizations which always result in smaller terms.

I also present a verified code generation translating the CPS IR into Clight, a front-end language of CompCert. I show how it interfaces with a verified garbage collector and how its proof composes with the proof of correctness of CompCert.

Taken together, this thesis shows how carefuly crafted intermediate languages facilitate verification effort in the context of an optimizing compiler.

Acknowledgements

I thank:

- My advisor, Andrew Appel, for his continuous support and guidance in becoming a better researcher;
- My other committee members for their detailled comments;
- My undergraduate advisors Laurie Hendren, for sparking my interest in compilers, and Brigitte Pientka, for showing me that the work is not complete until it is elegantly formalized;
- My research collaborators at Princeton and elsewhere: Stefan Monnier, Kaustuv Chaudhuri and the whole CertiCoq team;
- Alyne, Gaëtan, Ann, Daniel, Sophie, Vincent, Mathieu and Diana for their support and encouragement throughout graduate school.

This material is based upon work supported by the National Science Foundation under Grants CCF-1407794 and CCF-1521602. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

To the memory of Professor Laurie Hendren.

Contents

Chapter 1

Introduction

If the program has bugs, why bother proving the compiler correct? If the compiler has bugs, why bother proving the program correct? Verified source programs deserve verified compilers, and vice versa. In this thesis, I describe **CertiCoq**, a verified-correct compiler for Coq—that is, for the functional language that is part of Coq's Gallina specification language.

This work delivers a proved-correct optimizing compiler from a realistic language with a good proof theory.

CertiCoq is part of a multi-year effort by myself and collaborators at Princeton University, INRIA, Cornell University, and the University of Edinburgh [2]. All compiler phases are proved to preserve observable behavior from each intermediate language to the next, with machine-checked proofs in Coq. The user can prove a program correct in Coq, then the verification of **CertiCoq** guarantees that this program compiled to machine-language has the behavior that the user verified at the source level.

Coq is a proof and programming environment. At the center of Coq is a functional programming language, Gallina, which can be used both to write functional programs and, through the Curry-Howard Isomorphism, to write proofs about them. Gallina is

a pure, dependently typed functional programming language with inductive datatypes and mutually recursive functions.

The soundness of Coq is based on the soundness of its evaluation mechanism. Coq provides different trusted evaluation mechanisms, from evaluating the term using call-by-value evaluation rules to a native code just-in-time compiler. The aim of the **CertiCoq** project is to provide both the small trusted computing base of the reduction system and the speed afforded by a native code compiler. This is done using a process called extraction, which strips Gallina terms down to their computationally relevant core, and *verified* optimizing compilation using techniques like those used in ML compilers. **CertiCoq** is an optimizing extraction and compilation pipeline for Coq, verified in Coq.

Extraction mechanisms for proof and programming environments such as Coq have been developed in order to run realistic programs developed inside them. Coq provides an extraction plugin generating programs in general purpose functional languages such as OCaml or Haskell, which can be further compiled and executed. However, neither the extraction nor the compilation has been mechanically verified, making the trusted code base of such developments very large. Compiling the extracted code does not even provide the same guarantees that type safety provides to OCaml and Haskell, as their type systems are not always able to express Coq dependently typed functions – so that the extraction mechanism uses unsafe type-coercion mechanisms. This contrasts with the **CertiCoq** approach, which targets C, an untyped language, but whose proof of correctness composes with the proof of correctness of the CompCert C compiler in order to provide end-to-end correctness guarantees.

In this thesis, I describe my work on the middle and back end of **CertiCoq**, from various optimizations proved correct using a novel framework for modular proofs (Chapter 3), to the code generation phase targeting **Clight** (Chapter 4), itself the source language of the CompCert verified C compiler. Chapter 2 serves as a com-

prehensive overview of the project. Finally, we include in Chapter 5 benchmarks for the extraction time and running time of Coq developments through **CertiCoq**, and drafts a few future exploration path to make it faster.

CertiCoq is available at https://github.com/PrincetonUniversity/certicoq/. In this thesis, we will refer to files and theorems as they appear in commit dfede3e30f, available at https://github.com/PrincetonUniversity/certicoq/tree/dfede3e30f37a63f8671e8d66e5600e10c1e6e9d.

1.1 Contributions

The main technical contributions of this thesis consist of

1. the first proof of correctness of a shrink reduction phase (Section 3.2.2),[1]
2. the development and presentation of a proof framework allowing for modular proofs for optimization phases (Section 3.3),
3. the proof of correctness of a code generation phase from a continuation-passing style (CPS) intermediate representation ($L6$) to **Clight**, the source language of CompCert (Section 4.2),
4. a novel interface between generated code and garbage collector, abstracting away details of the correctness proofs of specific garbage collectors (Section 4.1.6).

We also present, in Chapter 2, the first complete overview of the **CertiCoq** extraction and compilation pipeline. [2]

[1] This was published as "Shrink Fast Correctly", co-authored with Prof. Andrew W. Appel [9]

[2] A brief overview of the **CertiCoq** project was published as "CertiCoq: A verified compiler for Coq" [2]

Chapter 2

CertiCoq

This chapter provides an overview of the **CertiCoq** pipeline, describing the various transformations along the way and the decisions that were made in the design of intermediate languages. The compiler pipeline is joint work with Abhishek Anand, Andrew Appel, John Li, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Matthieu Sozeau, and Katja Vassilev.

Just like CompCert (and CakeML), we design the compiler with many intermediate languages adapted to the transformations and optimizations at each stage of compilation, simplifying the proof effort. We go through continuation-passing style (CPS), making all function calls tail calls, and use a garbage collection mechanism tailored to the heavy heap usage.

Between Coq and CompCert, we have five intermediate languages. In the rest of the thesis, we refer to Gallina as L0, to our intermediate languages as L1 to L6[1], and to **Clight** as L7.

The first three are used for extraction purposes, removing the proof parts of terms, erasing types and producing terms more amenable to compilation.

[1] L5 is unused in the current pipeline

L1: **PCUIC** (see Section 2.2) embeds the Abstract Syntax Trees (AST) of Gallina into Coq.

L2: $\lambda\,\square$ (see Section 2.3) results from erasing Props and Types from **PCUIC** and replacing them with a special constant "$\square$".

L3: η-long $\lambda\,\square$ (see Section 2.4) differs from L2 in that it forces all constructors to be fully applied, and match branches to be η-expanded.

The last two intermediate languages are used for general compiler transformations and preparing for code generation to CompCert Clight.

L4: Globally nameless (see Section 2.5) creates a compilation unit by locally binding the environment of datatypes.

L6: Our CPS intermediate representation (see Section 2.6) adds additional restrictions on top of CPS, such as using globally unique variable names and having all arguments of an application be atoms, in order to make it amenable to code generation to **Clight** (see Section 2.7).

In the remainder of this chapter, we describe each of the intermediate languages in more details. Readers interested in the technical contributions of this thesis found in Chapters 3 and 4 may skip to Section 2.6 as the details of previous phases are not required to understand them.

2.1 L0: Gallina, the specification language of Coq

Gallina is the core language of the Coq theorem prover [35]. It is a pure, dependently typed functional language based on the Calculus of Inductive Constructions [14].

Gallina is used both as the specification and the reasoning language. One can write programs in it, specify them, and prove their correspondences all in the same

language. To make this more practical, it provides separation between two distinct sorts of types: *Prop*, the universe of logical propositions, and *Type*, the universe of program types. The typing judgment of the Calculus of Inductive Constructions ensures that elements of *Prop* do not have computational values – they are logical proofs, and as such only their inhabitance is needed. That is, the observable outcome of a computation cannot depend on a *Prop* or a proof of a *Prop*, even though *Props* and proofs may appear in programs.

Gallina a pure language: it does not have effects. This is important for the soundness of the proof theory, as it ensures that the terms we construct in Gallina adequately model proofs in the Calculus of Inductive Constructions. It also means that all functions written in Gallina have to be terminating.

As a running example for this chapter, we provide, in Figure 2.1, the implementation of a higher-order function applying a function to each element of a list. `map` is a recursive function ("`Fixpoint`") taking in two argument, a function `f`, of type $A \to B$, and `l`, a list of As, and computes a list of Bs. It does so by pattern-matching on `l` and returning `nil` in the case of an empty list. When the list is not empty, it applies `f` to `h`, the head of the list, and recur on `t`, the tail of `l`.

```
Fixpoint map {A B:Type} (f:A -> B) (l:list A):list B :=
  match l with
  | nil => nil
  | cons h t => cons (f h) (map f t)
  end.
```

Figure 2.1: Example: List.Map implemented in Gallina

2.2 L1: PCUIC and MetaCoq

The Coq theorem prover is implemented in OCaml [30], a general-purpose functional programming language. At the core of the implementation is an OCaml inductive datatype representing Gallina terms. The first step of the **CertiCoq** compiler is

to reify the OCaml representation of Gallina as a Coq datatype, effectively deeply-embedding Gallina inside Coq.

The L0-to-L1 translation is part of the **MetaCoq** project [49], and due to Gregory Malecha, Abhishek Anand and Matthieu Sozeau. **MetaCoq**'s intermediate representation, the Polymorphic Calculus of Cumulative Inductive Constructions (or **PCUIC**), is shown in Fig. 2.2. **MetaCoq**, and **PCUIC**, evolved from an earlier reification project, TemplateCoq [33]. **PCUIC** is as close as possible to the OCaml representation of Gallina terms, allowing for ease of reification – generating embedded Gallina from the kernel's representation – and of reflection – recovering the kernel's representation from the embedded encoding. Metatheorems about Coq can be proven in Coq by reasoning about **PCUIC** and connecting the proof with the adequacy of **PCUIC**.

MetaCoq includes a plugin that handles reflection and reification functionalities to go from Gallina to **PCUIC** and back. We use the plugin to transform Gallina program into their representations in **PCUIC**, embedded in Coq.

```
Inductive term : Set :=
| tRel       : nat -> term
| tVar       : ident -> term
| tEvar      : nat -> list term -> term
| tSort      : universe -> term
| tProd      : name -> term (* the type *) -> term -> term
| tLambda    : name -> term (* the type *) -> term -> term
| tLetIn     : name -> term (* the term *) -> term (* the type *) -> term -> term
| tApp       : term -> term -> term
| tConst     : kername -> universe_instance -> term
| tInd       : inductive -> universe_instance -> term
| tConstruct : inductive -> nat -> universe_instance -> term
| tCase      : (inductive * nat) (* # of parameters *) -> term (* type info *)
               -> term (* discriminee *) -> list (nat * term) (* branches *) -> term
| tProj      : projection -> term -> term
| tFix       : mfixpoint term -> nat -> term
| tCoFix     : mfixpoint term -> nat -> term.
```

Figure 2.2: Definition of the **PCUIC** Language

The syntax of **PCUIC** is in Figure 2.2. **PCUIC** uses a locally nameless [12] representation of variables, with `tRel n` referring to a variable occurrence bound by the nth binder, and `tVar a` to named variables `a` introduced in *Sections*.

"`tEvar n l`" represents an existential subterm, and should not be encountered in the complete terms we extract. "`tSort u`" represents a sort u, which could be in *Prop* or in *Type*. "`tProd a t1 t2`" represents a dependent product binding `t1` as `a` in `t2`, and `tProj p t` is a primitive projection `p` from an inductive record `t`. "`tLambda a t1 t2`" binds a variable `a` of type `t1` in `t2`, and `tApp t1 t2` represents a binary application. "`tLetIn a t1 t2 t3`" locally binds a term `t1` of type `t2` as `a` in `t3`. "`tConst n u`", "`tInd i u`" and "`tConstruct i n u`" represent, respectively, a definition, an inductive type, and a constructor of an inductive type in the global environment. "`tCase (i, n) ty t bs`" pattern-matches with predicate `t` on a term `t` of inductive type `ind` with `n` parameters, using a list of branches `bs` of the form `(n, b)` where `n` is the number of variables bound and `b` the body of the branch. Finally, "`tFix t n`" and "`tCofix t n`" create, respectively, fixpoints and cofixpoints of name `n`.

We include in Figure 2.3 our running example, List.Map (see Figure 2.1), encoded in **PCUIC**.

The next step of compilation is to remove the propositional portions of terms, as they are (by the type theory of the calculus of inductive construction) irrelevant to computation. The resulting calculus is called $\lambda\,\square$ (pronounced "lambda box").

2.3 L2: $\lambda\,\square$

The translation of **PCUIC** into $\lambda\,\square$ is part of the **MetaCoq** project, and was developed by Matthieu Sozeau and Yannick Forster, relying on a type inference algorithm for **PCUIC** developed by Simon Boulier, Matthieu Sozeau, Nicolas Tabareau, and Theo Winterhalter ; contributors to the design of lambda-box also include Randy Pollack, Abhishek Anand, and Greg Morrisett. We include it in this thesis for clarity,

```
(tFix
  [{|
    dname := nNamed "map";
      dtype :=
        tProd (nNamed "A") (tSort (Universe.make'' (Level.Level "Top.15", false) []))
          (tProd (nNamed "B") (tSort (Universe.make'' (Level.Level "Top.16", false) []))
            (tProd (nNamed "f") (tProd nAnon (tRel 1) (tRel 1))
              (tProd (nNamed "l")
                (tApp (tInd {| ind_mind := "list"; ind_ind := 0 |} []) [tRel 2])
                (tApp (tInd {| ind_mind := "list"; ind_ind := 0 |} []) [tRel 2])))));
      dbody :=
        tLambda (nNamed "A") (tSort (Universe.make'' (Level.Level "Top.15", false) []))
          (tLambda (nNamed "B") (tSort (Universe.make'' (Level.Level "Top.16", false) []))
            (tLambda (nNamed "f") (tProd nAnon (tRel 1) (tRel 1))
              (tLambda (nNamed "l")
                (tApp (tInd {| ind_mind := "list"; ind_ind := 0 |} []) [tRel 2])
                  (tCase ({| ind_mind := "list"; ind_ind := 0 |}, 1)
                    (tLambda (nNamed "l")
                      (tApp (tInd {| ind_mind := "list"; ind_ind := 0 |} []) [tRel 3])
                       (tApp (tInd {| ind_mind := "list"; ind_ind := 0 |} []) [tRel 3])) (tRel 0)
                   [(0, tApp (tConstruct {| ind_mind := "list"; ind_ind := 0 |} 0 []) [tRel 2]);
                     (2,
                       tLambda (nNamed "h") (tRel 3)
                         (tLambda (nNamed "t")
                           (tApp (tInd {| ind_mind := "list"; ind_ind := 0 |} []) [tRel 4])
                             (tApp (tConstruct {| ind_mind := "list"; ind_ind := 0 |} 1 [])
  [tRel 4; tApp (tRel 3) [tRel 1]; tApp (tRel 6) [tRel 5; tRel 4; tRel 3; tRel 0]])))))));
      rarg := 3 |}] 0)
```

Figure 2.3: Example: List.Map implemented in PCUIC

as this phase is important to understand the application of the verified compilation portion of our compiler to verified extraction.

The $\Box$ is not a modal operator; it is a placeholder for types, proofs, and propositions that have been erased. We include the syntax of $\lambda\,\Box$ in Figure 2.4. The main difference with **PCUIC** is the introduction of a $\Box$ constructor, replacing types and propositions in terms, and the removal of `tSort`, as types are erased when going from **PCUIC** to $\lambda\,\Box$.

We include in Figure 2.5 our running example, List.Map, as compiled to L2. L2 pairs an environment, containing the declaration of the list datatype (not shown) and the map function, together with an entry point `Top.map`. `Top.map` corresponds, in the environment, to a `TFix` named `map` binding `f` and `l` before pattern-matching on index 0 (`l`). The first case, matching `nil`, returns `nil`, the zeroth constructor of the datatype list (held in the environment under the full path `Coq.Init.Datatypes.list`). The second branch, matching `cons`, binds `h` and `t` before returning a new list formed with

```
Inductive term : Set :=
| tBox       : term (* Represents all proofs *)
| tRel       : nat -> term
| tVar       : ident -> term
| tEvar      : nat -> list term -> term
| tLambda    : name -> term -> term
| tLetIn     : name -> term (* the term *) -> term -> term
| tApp       : term -> term -> term
| tConst     : kername -> term
| tConstruct : inductive -> nat -> term
| tCase      : (inductive * nat) (* # of parameters *) ->
               term (* discriminee *) -> list (nat * term) (* branches *) -> term
| tProj      : projection -> term -> term
| tFix       : mfixpoint term -> nat -> term
| tCoFix     : mfixpoint term -> nat -> term.
```

Figure 2.4: Definition of the $\lambda\,\square$ Language

```
(tFix
  [{|
    dname := nNamed "map";
    dbody :=
      tLambda (nNamed "A")
        (tLambda (nNamed "B")
          (tLambda (nNamed "f")
            (tLambda (nNamed "l")
              (tApp (tInd {| ind_mind := "list"; ind_ind := 0 |} []) [tRel 2])
                (tCase ({| ind_mind := "list"; ind_ind := 0 |}, 1)
                  (tLambda (nNamed "l")
                     (tApp (tInd {| ind_mind := "list"; ind_ind := 0 |} []) [tRel 3])) (tRel 0)
                [(0, tApp (tConstruct {| ind_mind := "list"; ind_ind := 0 |} 0 []) [tRel 2]);
                  (2,
                    tLambda (nNamed "h")
                      (tLambda (nNamed "t")
                          (tApp (tConstruct {| ind_mind := "list"; ind_ind := 0 |} 1 [])
  [tRel 4; tApp (tRel 3) [tRel 1]; tApp (tRel 6) [tRel 5; tRel 4; tRel 3; tRel 0]])))])))));
    rarg := 3 |}] 0)
```

Figure 2.5: Example: List.Map implemented in $\lambda\,\square$

the first list constructor (`cons`), with, as first argument, the application of 3 (`f`) to 1 (`h`), and as tail a recursive call to 4 (`map`) with argument 3 (`f`) and 0 (`t`).

2.3.1 Removal of Props

The type system of Coq ensures that the propositional portion of terms is computationally irrelevant. *Prop* is the sort of propositions; *Type* is the sort of datatypes.

The metatheory of the Calculus of Inductive Constructions [14] tells us that the separation of the Prop and Type sorts makes it so that functional programs written in Gallina cannot depend on Prop, any such Prop passed (or constructed) in the program

is irrelevant and as thus can be replaced in order to save space, compilation time, and runtime efficiency of the programs (compared with evaluating the propositional portion at runtime).

It is important that propositional terms are removed early in the compilation pipeline, since

1. erasing propositions and proofs is justified by the type system of CiC, and therefore must be done before (or at the same time as) we erase the types, and

2. propositional terms could be arbitrarily large, and we would prefer not to waste compilation time transforming parts of terms which will be erased before code generation.

While the propositional portion of terms is computationally irrelevant, it is not the case that it is never observed during computation. Letouzey identifies two cases where a proposition may be destructed through pattern-matching in a computational term: `eq_rect` and `False-rec`, included in Figure 2.6, are the most common example of these two cases.

```
Definition False_rec:
  forall (P : Type), False -> P

Definition eq_rect :
  forall (A:Type) (x:A) (P:A -> Type), P x -> forall y:A, y = x -> P y.
```

Figure 2.6: Special cases in the extraction of Gallina terms

The first case results from being able to eliminate a proof of False (or, equivalently, an empty datatype) to create a term of any type. In a logical term, this corresponds to a proof by contradiction, while in computational terms, it corresponds to unreachable code.

The second special case deals with logical singleton types such as equality. In this case, we can observe, through pattern matching, that it is constructed using the

single constructor of the type, even within a computationally relevant portion of a term.

We could identify instances of these or similar examples of these two cases, and replace them at erasure time. However, to keep the transformation simple, and as doing so would not impact the evaluation of the resulting program, we leave them in the code to be reduced later on – both cases are handled by shrink reduction, described in Section 2.6.2.

As is done with the Letouzey's extraction pipeline, **MetaCoq** replaces *Prop* by a new constant, denoted $\square$ (represented as *tBox*). Rule E_BOX, included in Figure 2.7, shows how, according to the semantics of $\lambda\,\square$, *tBox* consumes any number of arguments and evaluates to *tBox* – this is because *tBox* stands in for *Prop* functions of any arity.

$$\frac{}{tApp\ tBox\ M \to tBox}\ \text{E\_BOX}$$

Figure 2.7: Extract from the Evaluation Semantics of $\lambda\,\square$

2.3.2 Type erasure

While the removal of *Prop* considers the removal of irrelevant content at the term level, type erasure removes irrelevant content at the type level.

Types do not have computational values, and by forgetting them, we simplify the rest of the pipeline, while still having the benefits associated with starting with typed, source terms: our pipeline can uses safety and termination assumptions arising from the well-typedness of its source terms. Meanwhile, while we can no longer prove properties such as type preservation, we prove a stronger property, semantics preservation, asserting that behaviors are preserved through compilation.

The translation from **PCUIC** to $\lambda\,\square$ drops the type arguments from the constructs that were holding them (`tProd, tLambda, tLetIn`). Sozeau et al. have verified in Coq the proved correctness of type and proof erasure [50]. This translation involves retypechecking the reflected terms to correctly identify the propositional subterms.

2.4 L3: η-long $\lambda\,\square$

At this point of the pipeline, we have an extracted term in a general call-by-value λ-calculus. However, a few transformations are needed in order to make it more amenable to transformations and optimizations, and closer to our desired intermediate representation. In this phase, we make constructors fully applied before stripping them of their parameters. We also η-expand match branches, and provide an interpretation of cofixpoints using suspended thunks. The translation from $\lambda\,\square$ to η-long $\lambda\,\square$ is due to Randy Pollack.

2.4.1 η-expansion of constructors

We η-expand all constructors, creating, when necessary, anonymous functions to fully applied constructors. For example, the list constructor `cons` could be bound in $\lambda\,\square$ to a variable as:

```
(tConstruct "list" 1 enil)
```

and later applied to arguments. We would η-expand this to

```
tLambda nAnon
  (tLambda nAnon
    (tLambda nAnon
     (tConstruct "list" 1 (tcons (TRel 2) (tcons (TRel 1) (tcons (TRel 0) tnil))))))
```

There are two reasons to do this.

First, this eliminates partially applied constructors. This simplifies code generation, as shown in Section 4.2: all values of a constructor are allocated the same size on the heap, and partially applied constructors in the source become functions from the missing arguments to fully applied constructors.

Second, this allows us to remove the parameters of inductive types, which do not have computational values. While we don't have the full type information, we still remember how many parameters each inductive type has. However, while types have been removed at this point, constructors still hold them. Only after η-expansion can we be certain that the right arguments are removed from constructors.

```
Inductive lambda (n : nat) : Set :=
| var : fin n -> lambda n
| app : lambda n -> lambda n -> lambda n
| lam : lambda (S n) -> lambda n.


Inductive lambda : val :=
| var : val -> val
| app : val -> val -> val
| lam : val -> val
```

Figure 2.8: Example: λ-calculus before and after parameter erasure

2.4.2 η-expansion of branches

Up to this point, every branch reduces to a number of bindings corresponding to the arguments of the matched constructors. We η-expand branches to remove these explicit bindings, and replace them by implicit binders. Later in the pipeline (when translating to L6), we insert explicit projections for the arguments of matched constructors, which are easier to optimize away than general functions.

2.4.3 Cofix elimination

We realize coinductive datatypes by replacing them by inductive datatypes with an additional *unit* argument, creating a suspended thunk on definition. This transfor-

mation has to happen on η-long constructors, as we want it to be inserted as their last argument. Then, a *unit* value is applied to values of coinductive datatypes being matched on, forcing the thunk. This translation is correct, but highly inefficient if a coinductive value is forced more than once.

2.5 L4: Globally nameless

Up to this point, compiled terms live in an environment of datatypes with other top-level constants. We let-bind the relevant content to recover a complete program, on which further optimizations may be applied. We also strip the lambdas of pattern-match branches, which have been η-expanded in the previous phase (see Section 2.4.2) for a more efficient representation of pattern-matching. This transformation was developed by Matthieu Sozeau with advice from Randy Pollack.

```
Inductive exp: Type :=
| Var_e: N -> exp
| Lam_e: name -> exp -> exp
| App_e: exp -> exp -> exp
| Con_e: dcon -> exps -> exp
| Match_e: exp -> branches_e -> exp
| Let_e: name -> exp -> exp -> exp
| Fix_e: efnlst -> N -> exp
| Prf_e : exp
with exps: Type :=
| enil: exps
| econs: exp -> exps -> exps
with efnlst: Type :=
| eflnil: efnlst
| eflcons: name -> exp -> efnlst -> efnlst
with branches_e: Type :=
| brnil_e: branches_e
| brcons_e: dcon -> (N * (* # args *) list name (* arg names *)) -> exp ->
            branches_e -> branches_e.
```

Figure 2.9: Definition of the globally nameless language

We provide the syntax of the globally nameless language in Fig. 2.9. The main difference with $\lambda\,\square$ (see Fig.2.4) is the absence of parameters in the pattern-matching construct `Match_e`. Binders are again represented using De Bruijn indices, with a `name` argument to preserves provenance information about variables (such as their source name, when available).

```
Let_e "map"
  (Fix_e (elcons "map"
    (Lam_e "f"
      (Lam_e "l"
        (Match_e (Var_e 0) 0
          (brcons_e ("list", 0) (0, nil) (Con_e ("list", 0), nil)
          (brcons_e ("list", 1) (2, "h"::"t"::nil)
            (App_e (App_e
                     (Lam_e "h"
                       (Lam_e "t"
                         (Con_e ("list", 1)
                           (econs (App_e (Var_e 5) (Var_e 1))
                           (econs (App_e (App_e (Var_e 6) (Var_e 5)) (Var_e 0)) enil)))))
                     (Var 1))
                   (Var 0))
          brnil_e)))))
    elfnil) 0)
  (Var_e 0)
```

Figure 2.10: Example: List.Map implemented in L4

We include in Figure 2.10 the function `List.Map` translated to L4. `map` is bound as sole function in a mutually recursive bundle `Fix_e`. It binds its arguments using curried, anonymous abstraction `Lam_e` before matching on `l` (as `Var_e 0`). The `cons` case, matched by "`(brcons_e ("list", 1) (2, "h"::"t"::nil)`", is η-expanded by `h` and `t`. Then, a new list is constructed from applying `f` (as `Var_e 5`) to `h` (`Var_e 1`), and `map` (`Var_e 6`) to `f` (`Var_e 5`) and `t`) (`Var_e 0`).

2.6 L6: CPS

Continuation-passing style (CPS) is a restriction over a functional language where all calls are tail calls [51]. This is useful in intermediate languages for functional-language compilers, as it simplifies optimizations over the intermediate language and code generation in the presence of a garbage collector.

Continuation-passing style makes the control flow of programs explicit, making it easier to reason formally about order of execution and to define transformations working over different modes of execution.

All function calls in CPS are tail calls. This is important since functional languages such as Gallina encourage the use of recursion, which could lead to stack overflow

if compiled naively. Finally, CPS allows us to uniformally reason about how the garbage collector can find the live variables. In particular, at a function call, only the arguments to the call are live.

$$
\begin{array}{llll}
\text{(Function Def'n)} & fd & ::= & f\ (\vec{x})\ =\ e \\
\text{(Branch)} & b & ::= & c \Rightarrow e \\
\text{(Expression)} & e & ::= & \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ e \\
 & & & \mid \mathsf{let}\ x = \mathsf{Prim}\ p\ \vec{y}\ \mathsf{in}\ e \\
 & & & \mid \mathsf{let}\ x = \mathsf{Proj}_n\ y\ \mathsf{in}\ e \\
 & & & \mid \mathsf{App}\ x\ \vec{y} \\
 & & & \mid \mathsf{let}\ \vec{fd}\ \mathsf{in}\ e \\
 & & & \mid \mathsf{match}\ x\ \mathsf{with}\ \vec{b} \\
 & & & \mid \mathsf{halt}\ x \\
\text{(Value)} & v & ::= & (c,\ \vec{v}) \\
 & & & \mid (\rho, \vec{fd}, x) \\
\text{(Environment)} & \rho & ::= & \cdot \\
 & & & \mid \rho, x \mapsto v
\end{array}
$$

Figure 2.11: Syntax of the CPS Language (L6)

L6 is a continuation-passing-style functional language with mutually recursive functions and pattern-matching. Figure 2.11 shows its syntax. The term "$\mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ e$" binds the constructor c applied to arguments $\vec{y}$ to variable x in expression e. The term "$\mathsf{let}\ x = \mathsf{Prim}\ p\ \vec{y}\ \mathsf{in}\ e$" binds the result of the primitive operator p on arguments $\vec{y}$ to variable x in expression e. The term "$\mathsf{let}\ x = \mathsf{Proj}_n\ y\ \mathsf{in}\ e$" binds the nth projection of y to variable x in in expression e. The term "$\mathsf{App}\ x\ \vec{y}$" applies function x to arguments $\vec{y}$. The term "$\mathsf{match}\ x\ \mathsf{with}\ \vec{b}$" matches the constructor c of x with the right branch $(c \Rightarrow e) \in \vec{b}$.

The well-formedness property we enforce over L6 terms is that branch patterns do not overlap and that function names within each bundle are distinct. "$\mathsf{halt}\ x$" terminates computation by returning the value bound to x.

An important decision in the design of intermediate language is the representation of binders and variables. We represent variables using globally unique positive binary numbers.

Having globally unique names means that a global map can refer to any binding point in the program, so that we can, for example, tabulate information about the provenance and, when possible, the name of the variable in the original program. It also means that we can perform substitution without fear of variable capture – although global uniqueness is not closed under substitution, as it may duplicate portions of terms containing binders.

Using positive binary numbers as identifiers allows us to implement the lookup tables used in the compilers as *binary tries* with logarithmic access time (Section 3.1 gives examples of the use of efficient lookup tables on variable-names, in an asymptotically efficient algorithm for shrink-reduction).

In Figure 2.12, we include the function List.map compiled to L6. For readability, we pretty-print variables as their name, if available, followed by their unique identifier. Function `map_110` has been closure-converted into `map_code_195` and `env_194`. As `map` was closed, `env_194` is empty. Then, control is given to function `anon_code_196`, created from the conversion of `Match_e`. List `l_114` is pattern-matched on, with the `nil` case resulting in the continuation being called on `nil`. In the `cons` case, `h_123` and `t_124` are projected out of `l_114`. Then, `f`, the argument to `map`, is called on `h` using the new continuation `x166`. This continuation is constructed using `anon_code_196` to handle the recursive call to `map`.

2.6.1 CPS Transformation

We implemented a naive CPS transformation from globally nameless terms to L6. The transformation is adapted from the general, single-pass CPS transformation described

in "Compiling with Continuations" [4], and differs from other CPS transformations in two main ways:

1. We change the representation of binders from De Bruijn indices to globally unique positive numbers in L6.

2. L6 puts many restrictions on what can appear in applied portions of terms. For example, constructors in globally nameless can be applied to terms, where constructors in L6 have to be applied to variables. We use an auxiliary function to convert an expression to an applicative context with a hole containing one more variable in scope, standing for the original expression. This sometime involves β-expanding the term into a new function binding containing the converted expression.

The CPS transformation was developed by Abhishek Anand, Greg Morrisett, and Olivier Savary Bélanger. A new version is under development by Anvay Grover with assistance from Olivier Savary Bélanger and Andrew Appel.

2.6.2 Optimizations over CPS

By making the control flow of the program explicit, CPS makes it easier to apply transformations rearranging this flow. Where the first half of the **CertiCoq** pipeline is concerned with extraction, including type erasure and the elimination of proofs, the optimizations that appear after CPS conversions are transformations that would appear in any optimizing functional-language compilers.

Closure Conversion and Hoisting

The transformation over L6 having the biggest effect on the shape of the program is closure conversion. Closure conversion makes the manipulation of closure objects

explicit. The result is a term whose functions are closed, so that they can be hoisted to the top-level.

The closure conversion phase implemented in **CertiCoq** is described by Paraskevopoulou and Appel [44]. For each function, we compute the free variable of its body, and add a new `env` argument. We then add projections from `env` to the original names. Then, we replace every function application with an application of, in addition to the original argument, a tuple containing the necessary free variables to the closure converted function.

Once functions are closure converted, they are closed and can be hoisted to the top-level. After hoisting, our program has the form "$\mathsf{let}\ \vec{fd}\ \mathsf{in}\ e$", with e containing no function declarations. This simplifies code generation, as shown in Section 4.2.

Shrink Reduction

In a functional language with immutable data structures—such as Gallina, ML, Haskell—several optimizations are particularly important: **function inlining** (β-reduction); **case-folding**, compile-time evaluation of case statements when the discriminant value can be statically determined; **projection-folding**, compile-time fetching of projections of tuple-fields (or generally, fields of inductive data constructors) when the tuple can be statically determined; and **dead variable elimination**. The reason these are more important in functional languages than in traditional imperative languages is that there are far more opportunities: functional languages and their compilers use functions more heavily, and folding of field-projections is possible only when the record-fields cannot have been updated with new values. Also, many compiler transformations introduce β-redexes. For example, simple CPS transformations introduce many so-called *administrative (β-)redexes* which can be safely reduced to recover a more compact program.

It is important to do all these optimizations *together,* because one may produce new opportunities to do another.

For example, in

$$\mathsf{let}\ f\ x := (\mathsf{match}\ x\ \mathsf{with}\ O \Rightarrow M\ ;\ S\ \_ \Rightarrow N)\ \mathsf{in}\ f\ O$$

we can inline f resulting in

$$\mathsf{match}\ O\ \mathsf{with}\ O \Rightarrow M\ ;\ S\ \_ \Rightarrow N$$

at which point the constructor O is exposed and the case-construct can be folded down to M. Then, since the expression N has disappeared, some of the free variables of N (bound in some context external to the entire *let* expression) may now be dead, permitting dead-variable elimination.

Case-folding, projection-folding, and dead-variable elimination are always worth doing, because they make the program smaller and faster. Function inlining usually makes the program faster, but if there are many uses of the function, it may make the program bigger. Inlining a function that has only one applied occurrence will make the program smaller and faster, because the function-definition is now dead and can be deleted. Appel and Jim [8] described this class of optimizations (case-folding, projection-folding, dead-variable elimination, and inlining functions with one applied occurrence) as *shrink reductions.*

A traditional function-inliner or dead-variable optimizer makes one static-analysis pass over the program, counting applied occurrences and learning which functions are worth inlining; then another pass performs the transformations; then (because the transformations may have enabled new optimizations) repeats the analysis pass, then another optimization pass, and so on until the analysis pass yields no new optimizations to perform.

Appel and Jim described an efficient algorithm that performs (almost) all the possible shrink reductions, even cascading ones, in one linear-time pass. This is a *quasilinear time* algorithm: in linear time, it typically reduces to shrink-normal form, but sometimes leaves a very small number of shrink redexes to be reduced in a second pass, or very rarely in a third pass.

We have implemented and proven correct an algorithm inspired by Appel and Jim [8] to perform all shrink reductions (producing a *shrink normal form*) in quasilinear time. Having a fast and effective shrink-reduction phase is useful, because transformations often introduces redexes which, if not eliminated, would clutter and slow down the generated code. We apply shrink reductions after CPS in order to reduce administrative redexes, and again after closure conversion and hoisting.

Uncurrying and General Inlining

In Gallina, our source language, all functions are curried, which is to say they take a single argument. This is impractical for code generation, where we would like to limit the use, whenever possible, of expensive operations such as function calls. The uncurrying phase does so by combining sequences of curried abstractions together and having fully applied sites replaced by calls to uncurried versions of functions.

This transformation happens after CPS in our pipeline, but before any other transformations. This is important because while CPS modifies the structures of sequences of abstractions and applications, it does so in a predictable way, which we can recognize during our uncurrying phase.

Uncurrying happens in two steps. First, we create uncurried version of functions. Then, we replace fully applied calls to the curried functions by calls to the uncurried functions. More details about the uncurry phase of Certicoq are included in Section 3.5.1.

For example, in

$$\mathsf{let}\ add\ x := (\mathsf{let}\ add'\ y := x + y\ \mathsf{in}\ add'\ )\ \mathsf{in}\ add\ 2\ 3$$

we would recognize the declaration and application of the curried add' in the body of add, and create an uncurried version

$$\mathsf{let}\ add_u\ x\ y := x + y\ \mathsf{and}\ add\ x := (\mathsf{let}\ add'\ y := add_u\ x\ y\ \mathsf{in}\ add'\ )\ \mathsf{in}\ add\ 2\ 3$$

We can then replace the curried calls by uncurried body. In our example, the applications of add and add' would be inlined and deleted as dead code by shrink reduction, resulting in

$$\mathsf{let}\ add_u\ x\ y := x + y\ \mathsf{in}\ add_u\ 2\ 3$$

These reductions are not necessarily shrinking, as there might be more than one application of the uncurried function. However, we know that the body of uncurried shell is a single application (to the newly created curried function), so that the size of the program will not increase. As described in Section 3.5.2, we use a general function inliner parameterized over inlining heuristic, and use it to inline uncurried shells and small functions, up to a small inlining bound. The uncurry phase was implemented by Greg Morrisett and Olivier Savary Bélanger, and proved correct by John Li. [32]

2.7 L7: Clight

CertiCoq targets **Clight** [10], the source language of the CompCert compiler. **Clight** is a large subset of the C language, imposing a few syntactic restrictions making for a cleaner formal semantics. In this section, we summarize the formal semantics of **Clight**.

To understand the semantics of **Clight**, we must first show its memory model. A Clight memory M consists of a list of distinct blocks b. Each block b has a size $\delta_{max(b)}$, and content accessible using pointers from $(b, 0)$ to $(b, \delta_{max(b)})$.

A Clight program consists of a list of global-variable declaration and a list of functions (both held in a global environment G) and an entry point b. Functions contain parameters, local variables (held in local environment E) and body (as a Clight statement s).

The main evaluation judgment $G, E \vdash s, M \Rightarrow^t .$, shown in Figure 2.13, is concerned with the evaluation of a statement s in a memory M and environments G and E. Either s diverges (with infinite trace T), or it terminates in return value *out*, a finite trace t and resulting memory M'. In **CertiCoq**, we are only concerned with terminating programs: Gallina is strongly normalizing, and our proof of correctness ensures that programs do not acquire non-terminating behaviors along the way.

Clight statements s include control structures such as sequencing $(s; s)$, for- and while- loops, if-then-else constructors, switch statements, and related controls including break statements and continues. They also include memory operations such as assignments to a memory location or to a local variable, and constructors to call a function and to return from a function call.

A **Clight** expression e evaluates to **Clight** value v according to judgment $G, E \vdash e, M \Rightarrow v$ (see Figure 2.14). Expressions in **Clight** are pure, such that the memory M is not modified during the evaluation of e.

A **Clight** value v can be a pointer `Vptr` (b, δ), an integer `Vint` i, a floating-point value `Vfloat` f or an undefined value `Vundef`. **CertiCoq** does not use floating-point values, and generates code that never handles undefined value, so that we only have to consider the integer and the pointer case.

Our presentation so far has ignored the issue of the size of addresses. Recent versions of CompCert can target 64-bit architecture, and use `Vptr` corresponding to

64-bit integers `Vlong`. Our compiler and the proof of correctness is parameterized over the architecture, handling both 32-bit and 64-bit. For brevity, in the rest of the thesis, we will use `Vint` to refer to adequately sized integers for the address space, which is to say `Vint` in 32-bit mode and `Vlong` in 64-bit mode.

2.7.1 Code Generation and Interface with C Programs

From our CPS IR, we generate stackless closure-passing **Clight** code. The code generation phase of **CertiCoq** was developed by Olivier Savary Bélanger with assistance from Matthew Weaver and advised by Andrew Appel.

Targeting **Clight** rather than machine code has multiple benefits for **CertiCoq**. CompCert is a mature compiler allowing us to target multiple architectures, including x86-32, x86-64, ARM-32, ARM-64, RISC-V, and Power-PC. CompCert is a significant proof effort, and we would have to repeat much of the same work if we were to translate down to machine code. Moreover, by design, L6 is amenable to code generation to Clight, and its constructors correspond almost directly to operations in **Clight**.[2] We provide more details on this translation in Chapter 4.

For users of **CertiCoq** to interact with the generated code, the compiler generates shim functions that construct and destruct C representations of the datatypes that were compiled through **CertiCoq**. A detailed overview of these functions and how to use them is provided in Section 4.4.

2.8 The Overall Proof of Correctness

CertiCoq is proved correct using a program equivalence theorem in Figure 2.15 built on top of an abstractly defined value refinement relation shown in Figure 2.16.

[2]Our L6 CPS language is low-level enough to target assembly language directly, but we would need a register allocator and we would need an instruction-selection phase for each target machine architecture.

Under this relation, a program in L_i that computes to a value v_i is translated to a program in L_{i+1} computing to a refined value v_{i+i}. The semantic framework for composing phases was developed by Zoe Paraskevopoulou and Abhishek Anand with advice from Andrew Appel and Greg Morrisett

Correspondence of values is defined as adherence to observations $\checkmark$, shown in Figure 2.17. v_{1+1} is related (according to $\sqsubseteq$, see Figure 2.16) to value v_i if, for any observation $\checkmark_O$ valid on v_i, the same observation is valid on v_{i+1}. Under `Q-Fun`, $\checkmark_\lambda$ asserts the head of the term is an abstraction. Rule `Q-constr` shows how $\checkmark_{I\,n}$ verifies that the head of the term is a constructor C which is the nth constructor of inductive type I. Finally, Q-subterm allows observations on subvalues held in constructors. In other words, v_{i+1} is said to refine v_i if the heads of subterms of v_{i+1} correspond to those of v_i.

As, at every step of compilation, our value refinement relation is only concerned with the directly upstream and downstream language, our correctness proof is easy to extend vertically, composing seamlessly with new phases and languages.

The overall statement of compiler correctness for **CertiCoq** is in Figure 2.18. It states that for any closed Coq program P evaluating, according to the semantics of Gallina ($\Downarrow$) to value v, **CertiCoq** compiles ($\hookrightarrow$) P to Clight statement $stmt$, which in turn evaluates according to the semantics of Clight ($\rightarrow^*$) in a well-formed memory m properly, resulting in a memory m'. By convention, our code generation will have placed the resulting value in the $args_1$ location of memory, and our proofs ensure that this value refines v. Our proof of correctness composes with the CompCert proof of correctness to prove end-to-end compilation correctness, from Coq to machine code.

Currently, the overall proof of correctness of **CertiCoq** (see Figure 2.18) is only concerned with full program compilation. However, we have explored strengthening the correctness statement to allow for horizontal composition – composing multiple

portions of programs compiled with **CertiCoq**– or even wider notions of horizontal modularity.

Our correctness statement is also too weak for programs that use axioms – only closed (axiom-free) Coq programs are guaranteed to evaluate to a value, our correctness statement can be vacuously true for programs with axioms. We hope to weaken this assumption, in the future, to allow for provable axioms in the computationally irrelevant sort *Prop*, which should follow from a mechanized proof of proof irrelevance as is currently in the work as part of the **MetaCoq** project [49]. We also plan on supporting a few commonly used axioms such as function extensionality which, while not provable in Coq, can be proven not to harm strong normalization as part of a richer, extensional reduction semantics.

2.9 Related Work

The Coq theorem prover currently includes an extraction mechanism to **OCaml** implemented by Pierre Letouzey [31]. In his thesis, Letouzey proves a number of properties, including that proof irrelevance implies that we can replace *Prop* by $\Box$. However, these proofs have never been formalized (in a proof assistant). Moreover, to run the generated **OCaml** program, one would have to use – and trust – the **OCaml** compiler. Instead, our extraction pipeline is verified and targets a verified C compiler, reducing the trusted computing base of Coq.

In addition to extraction facilities to **OCaml** and Haskell, Coq provides a number of evaluation mechanisms. The internal evaluation mechanism is `compute`, an interpreter based on the call-by-value semantics of the Calculus of Inductive Constructions, but supporting other evaluation strategies. While these mechanisms, being so simple, are reasonably trustworthy (and the call-by-value strategy, being the direct implementation of the semantics of the Calculus of Inductive Constructions, is the specification

with respect to which **CertiCoq** is proved correct) , they are not optimized in terms of memory and time consumption, and as such they do not scale well to medium and large development. Coq also provides two evaluation mechanisms based on a process named *reflection*. Using `vm_compute` [25] or `native_compute`, Gallina terms are extracted to OCaml, and compiled with (respectively) the byte-code and native OCaml compiler. The result is then injected back into Coq. These evaluation mechanisms scale reasonably well, but they considerably increase the trusted computing base of the theorem prover, to include not only the reflection facilities, but also the bytecode or native OCaml compiler[3].

CompCert [29] is a verified optimizing compiler for C developed in the Coq theorem prover. Taken together with a Hoare logic for C such as VST [5], this provides a end-to-end, proved correct compilation pipeline from a realistic programming language with a strong proof theory. However, being an impure, imperative programming language, C does not afford as clean a proof theory as Gallina. We see benefits in both projects coexisting, with, in future plans, linking the two proof theories to allow for programs mixing Gallina and C code. Moreover, since CompCert is developed in Coq, we could make it safer by extracting it to C with **CertiCoq** instead of to OCaml using the current extraction pipeline.[4]

The CakeML [53, 54] project includes a verified compiler for a "substantial subset of Standard ML" to assembly, which has been used as an extraction mechanism for HOL4 [36]. The backend generates machine code for multiple architectures (as does CompCert). CakeML is the most mature verified extraction pipeline (for a functional language in a proof assistant), and includes many backend optimizations which have not yet been implemented in **CertiCoq**. Meanwhile, the source language of CakeML

[3]While Coq is implemented in OCaml, and as such, depends on its compiler and runtime system, Gallina could be type-checked by a foundational checker in the style of Flit [62]

[4]This is not done at the moment since a few passes of CompCert are developed in OCaml and proved correct using translation validation. Work would need to be performed to implement and prove correct these phases in Gallina or in C.

is not dependently typed, so it does not share the same concerns with respect to erasure in its front end. Another big difference between the two projects is that CakeML does not use continuation-passing style (CPS). CPS makes all function calls tail calls, and we argue in Chapter 3 that it facilitates the compilation process.

PILSNER [42] is another verified compilation pipeline from ML to assembly developed and extracted from Coq, and built to showcase a novel proof method for simulation relation between language (Parametric Inter-Language Simulations). Like us, it compiles through a CPS-based IR. It however assumes unbounded memory, and includes very few optimizations.

TIL [55] is a type-directed optimizing compiler from Standard ML to machine code. In his thesis, Tarditi identifies two cases where types provides optimization with information that would be difficult to recover with simple static analysis. First, types make debugging the compiler easier. While this is useful during development, **CertiCoq**'s transformations are proven correct, making testing superfluous. Second, Standard ML has a notion of constructor-level and term-level computation, with constructor-level computation being side-effect free, and as such easier to optimize. As Gallina, our source language, is side-effect free, we would gain no advantage from this in keeping the types around. TIL also uses type information to achieve tag-free representation of most data structures. However, as noted by Tolmach[57], only a simplified digest of type information is needed for tag-free garbage collection, consisting of the arity of each constructors and the digested type of each of their fields. While we only keep track of the former in the current **CertiCoq** pipeline, it would be simple to extend it to keep track of the latter. In the meantime, we use a lightweight tagged representation inspired by **OCaml** [30] where the last bit of values distinguish pointers from unboxed values.

TAL [40] is a typed assembly language together with a type-preserving compiler from System F to TAL, pushing the typing information all the way down to assembly.

In both TIL and TAL, the properties enforced are limited to type safety, and their source language does not afford the proof theory provided by the Coq theorem prover.

Type information has also been used to achieve more precise deforestation[60], a transformation which removes intermediate data structures by composing, when possible, operations performed on the source data structure and on the intermediate one. Voigtländer [58] uses parametricity to manipulate operations on tree-like data structures, while Chevalier [13] uses type information to selectively inline functions to enable deforestation. Both improve on previous syntactic deforestation algorithms. As an alternative to performing type-based deforestation directly on **PCUIC**, we could keep track of functions whose inlining would enable deforestation to perform the optimization after type erasure.

Verisoft [1] is a verified compilation stack aimed at systems software verification, from C0, a small subject of the C programming language, down to machine code, verified in Isabelle/HOL. The project include a Hoare logic for C0 providing facilities to reason about the software being compiled. Verisoft does not fit our desiderata in that C0 is not a realistic language, and the compiler to machine-code, while being verified, is not optimizing.

Why3 [23] is a tool for deductive program verification including WhyML, a programming and specification language which can be extracted in a proved correct way to OCaml. This extraction procedure generates first-order verification condition which are discharged by an SMT solver.

2.10 Conclusion

In this chapter, we have presented a verified extraction and compilation pipeline for the Coq theorem prover. Extraction is a crucial component of a proof and programming environment: a proof about a program is only useful if what is being run

corresponds to that program. Special care has been taken to ensure the efficiency of our pipeline, both in terms of optimization phases and in the code generation to C.

```
letrec [
  fun map_code_195(map_env_197,k_112,kapArg_191) :=
    let anon_clo_199 := con_15(anon_code_196,map_env_197) in
    let env_198 := con_103(kapArg_191,anon_clo_199) in
    let x188 := con_15(anon_code_200,env_198) in
    let k_code_201 := proj_0 15 k_112 in
    let k_env_202 := proj_1 15 k_112 in
    k_code_201(k_env_202,x188)
  fun anon_code_200(anon_env_203,x189,x190) :=
    let anon_proj_204 := proj_1 103 anon_env_203 in
    let kapArg_proj_205 := proj_0 103 anon_env_203 in
    let anon_code_206 := proj_0 15 anon_proj_204 in
    let anon_env_207 := proj_1 15 anon_proj_204 in
    anon_code_206(anon_env_207,x189,x190,kapArg_proj_205)
  fun anon_code_196(anon_env_208,k_115,l_114,f_111) :=
    case l_114 of {
    | cons =>
      let x124 := proj_1 101 l_114 in
      let x123 := proj_0 101 l_114 in
      let anon_clo_210 := con_15(anon_code_196,anon_env_208) in
      let env_209 := con_104(f_111,k_115,x124,anon_clo_210) in
      let x166 := con_15(anon_code_211,env_209) in
      let f_code_212 := proj_0 15 f_111 in
      let f_env_213 := proj_1 15 f_111 in
      f_code_212(f_env_213,x166,x123)
    | nil =>
      let x121 := nil() in
      let k_code_228 := proj_0 15 k_115 in
      let k_env_229 := proj_1 15 k_115 in
      k_code_228(k_env_229,x121)
    }
  fun anon_code_211(anon_env_214,x0kdcon_144) :=
    let k_proj_216 := proj_1 104 anon_env_214 in
    let env_215 := con_105(k_proj_216,x0kdcon_144) in
    let x165 := con_15(anon_code_217,env_215) in
    let anon_proj_218 := proj_3 104 anon_env_214 in
    let anon_proj_219 := proj_2 104 anon_env_214 in
    let f_proj_220 := proj_0 104 anon_env_214 in
    let anon_code_221 := proj_0 15 anon_proj_218 in
    let anon_env_222 := proj_1 15 anon_proj_218 in
    anon_code_221(anon_env_222,x165,anon_proj_219,f_proj_220)
  fun anon_code_217(anon_env_223,x1kdcon_163) :=
    let x0kdcon_proj_224 := proj_1 105 anon_env_223 in
    let x164 := cons(x0kdcon_proj_224,x1kdcon_163) in
    let k_proj_225 := proj_0 105 anon_env_223 in
    let k_code_226 := proj_0 15 k_proj_225 in
    let k_env_227 := proj_1 15 k_proj_225 in
    k_code_226(k_env_227,x164)
] in
let env_194 := con_102() in
let map_110 := con_15(map_code_195,env_194) in
halt map_110
```

Figure 2.12: Example: List.Map implemented in L6

$$G, E \vdash s, M \Rightarrow^{T} \infty$$
$$G, E \vdash s, M \Rightarrow^{t} out, M'$$

$$\frac{G, E \vdash s_1, M \Rightarrow^{t_1} \mathsf{Normal}, M_1 \quad G, E \vdash s_2, M_1 \Rightarrow^{t_2} out, M_2}{G, E \vdash (s_1; s_2) \Rightarrow^{t_1, t_2} out, M_2}$$

Figure 2.13: Some of the evaluation judgments for **Clight** statements [10]

$$G, E \vdash e, M \Rightarrow v$$

$$\frac{G, E \vdash a_1, M \Rightarrow v_1 \quad G, E \vdash a_2, M \Rightarrow v_2 \quad \mathsf{evalbinop}(op, v_1, v_2) = v}{G, E \vdash a_1\, op\, a_2, M \Rightarrow v}$$

Figure 2.14: Some of the evaluation judgments for **Clight** (r)-expressions [10]

$$p_i : L_i \rightsquigarrow p_{i+1} : L_{i+1} \wedge p_i \Downarrow v_i \qquad \Rightarrow \qquad p_{i+1} \Downarrow v_{i+1} \wedge v_i \sqsubseteq v_{i+1}$$

Figure 2.15: The abstractly defined program equivalence theorem

$$v \sqsubseteq v' := \forall O.\ v \checkmark_O \Rightarrow v' \checkmark_O$$

Figure 2.16: The abstractly defined value refinement relation

$$\frac{}{\lambda.v \checkmark_{lam}}\ Q - fun$$
$$\frac{nth\ n\ I = C}{C\vec{v} \checkmark_{ind\ I\ n}}\ Q - constr$$
$$\frac{nth\ n\ \vec{v} = v \quad v \checkmark_O}{C\vec{v} \checkmark_{sub\ O\ n}}\ Q - subterm$$

Figure 2.17: Definition of the observations predicate ✓

$$P \Downarrow v \wedge \mathsf{CLO}(P) \Rightarrow P \hookrightarrow stmt \wedge (m, stmt) \rightarrow^{*} (m', skip) \wedge v \sqsubseteq m'_{args_1}$$

Figure 2.18: Statement of the Overall Correctness Theorem

Chapter 3

The CPS Intermediate Representation and the Proof Framework

The last intermediate language of **CertiCoq** is L6. Its syntax can be found in Figure 2.11, duplicated here for convenience, and is described in Section 2.6.

L6 is the language on which many optimizations and transformations are applied, and as such we have spent a considerable amount of time making sure its syntax and semantics make it easier to prove things about it. In this chapter, we show its semantics, before presenting the proof framework, a series of rewrite systems aimed at facilitating proofs about transformations over L6. Portions of this chapter are adapted from Savary Bélanger and Appel [9], which presented the the proof of correctness of the shrink reduction phase of **CertiCoq** and the proof framework in which it was developed.

$$
\begin{array}{llll}
\text{(Function Def'n)} & fd & ::= & f\ (\vec{x})\ =\ e \\
\text{(Branch)} & b & ::= & c \Rightarrow e \\
\text{(Expression)} & e & ::= & \quad \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ e \\
 & & & \mid \mathsf{let}\ x = \mathsf{Prim}\ p\ \vec{y}\ \mathsf{in}\ e \\
 & & & \mid \mathsf{let}\ x = \mathsf{Proj}_n\ y\ \mathsf{in}\ e \\
 & & & \mid \mathsf{App}\ x\ \vec{y} \\
 & & & \mid \mathsf{let}\ \vec{fd}\ \mathsf{in}\ e \\
 & & & \mid \mathsf{match}\ x\ \mathsf{with}\ \vec{b} \\
 & & & \mid \mathsf{halt}\ x \\
\text{(Value)} & v & ::= & (c,\ \vec{v}) \\
 & & & \mid (\rho, \vec{fd}, x) \\
\text{(Environment)} & \rho & ::= & \cdot \\
 & & & \mid \rho, x \mapsto v
\end{array}
$$

Figure 2.11: Syntax of the CPS Language (L6)

3.1 The Semantics of our CPS IR

The semantics of our object language is given through a big-step, environment-based judgment $\rho \vdash e \Downarrow_k v$[1] evaluating expressions e in environment ρ into value v in at most k reductions. We will sometimes omit the argument k and just write $\rho \vdash e \Downarrow v$ when the cost is inconsequential. The environment maps variables to values. A value is either a constructor c with its arguments $\vec{v}$ or a closure including a function's body e with its parameters $\vec{x}$ and an environment ρ providing values for the function's free variables. Figure 3.1 shows the evaluation rules.

In our object language, pattern-matching is broken into two operations. First, our case constructor "match x with $(c_1 \Rightarrow e_1, ..., c_n \Rightarrow e_n)$" determines which pattern c_i the construction bound to variable x matches, and proceeds to evaluate e_i, as seen in

[1] bstep_e in theories/L6_PCPS/eval.v

$$\frac{\rho(x) = c\ \vec{w} \quad (c \Rightarrow e) \in \vec{b} \quad \rho \vdash e \Downarrow_k v}{\rho \vdash \mathsf{match}\ x\ \mathsf{with}\ \vec{b} \Downarrow_k v}\ \text{E\_MATCH}$$

$$\frac{\rho(y) = c\ \vec{w} \quad \rho; x \mapsto w_n \vdash e \Downarrow_k v}{\rho \vdash \mathsf{let}\ x = \mathsf{Proj}_n\ y\ \mathsf{in}\ e \Downarrow_k v}\ \text{E\_PROJ}$$

$$\frac{\rho(f) = (\rho',\ \vec{fd},\ f) \quad (f\ (\vec{x})\ =\ e) \in \vec{fd} \quad \rho'; f_i \mapsto (\rho', \vec{fd}, f_i); \vec{x} \mapsto \rho(\vec{y}) \vdash e \Downarrow_k v}{\rho \vdash \mathsf{App}\ f\ \vec{y} \Downarrow_{k+1} v}\ \text{E\_APP}$$

$$\frac{\forall_{y_i \in \vec{y}}, \rho(y_i) = w_i \quad \rho; x \mapsto (c, \vec{w}) \vdash e \Downarrow_k v}{\rho \vdash \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ e \Downarrow_k v}\ \text{E\_CONSTR}$$

$$\frac{\forall_{y_i \in \vec{y}}, \rho(y_i) = w_i \quad f\ \vec{w} = w \quad \rho; x \mapsto w \vdash e \Downarrow_k v}{\rho \vdash \mathsf{let}\ x = \mathsf{Prim}\ f\ \vec{y}\ \mathsf{in}\ e \Downarrow_k v}\ \text{E\_PRIM}$$

$$\frac{\rho; f_1 \mapsto (\rho, \vec{fd}, f_1); ...; f_n \mapsto (\rho, \vec{fd}, f_n) \vdash e \Downarrow_k v \quad \mathsf{where\ names}(\vec{fd}) = \{f_1, ..., f_n\}}{\rho \vdash \mathsf{let}\ \vec{fd}\ \mathsf{in}\ e \Downarrow_k v}\ \text{E\_FUN}$$

$$\frac{\rho(x) = v}{\rho \vdash \mathsf{halt}\ x \Downarrow_k v}\ \text{E\_HALT}$$

Figure 3.1: Evaluation rules of the object language

E_MATCH.[2] Then, projection constructs "let $x_1 = \mathsf{Proj}_1\ x$ in...let $x_m = \mathsf{Proj}_m\ x$ in" are used to bind variables to the m arguments of c_i which will be replaced by the right values when evaluated as shown in rule E_PROJ.

ML's and Haskell's syntax and type systems connect case matching with projection, so that the programmer cannot mistakenly project a field from the wrong constructor. We separate projections from cases because it makes the operational semantics simpler, the optimizer simpler, and the proof simpler: our language is an *untyped intermediate language,* not a typed source language. CertiCoq is meant to be used only to compile source programs type-checked in Coq; the Coq type system guarantees that they will not get stuck. Therefore, as the front end phases are proved correct, the program translated to lower-level intermediate languages (such as the CPS presented here) will not get stuck.

[2]The well-formedness property ensures that each constructors appear once and as such that patterns are non-overlapping.

Rule E_APP shows how applications are evaluated. When a function f is applied to arguments $\vec{y}$, we look up f in the environment ρ to retrieve the function closure $(\rho', \vec{fd}, f)$. Next, we find function f in $\vec{fd}$ with arguments $\vec{x}$ and function body e. We then evaluate the function body e in saved environment ρ' extended with bindings for each mutually recursive function in $\vec{fd}$ and by associating each y_i in $\vec{y}$ to their respective x_i in $\vec{x}$.

We define $\mathsf{FV}(e)$ and $\mathsf{BV}(e)$ to be respectively the set of free and bound variables of a term e or of a bundle of function definitions $\vec{fd}$. We also define $\mathsf{names}(\vec{fd})$ to be all the names of functions from the bundle:

$$\mathsf{names}(\vec{fd}) := \{f | f\ (\vec{x})\ =\ e \in \vec{fd}\}$$

An important property that is not enforced by the syntax presented in Fig. 2.11 is that bound names are globally unique. This property is easy to achieve and maintain; the translation from the previous intermediate language uses a state monad to assign unique variable names. We also make sure that the free variables of the top-level program are disjoint from its bound variables. This allow us, for example, to perform function inlining without worrying about variable capture. We define the proposition $\mathsf{UB}(e)$ to assert that e has the unique binding property.

Applicative context

We define a notion of applicative context, intuitively a term with a hole, which will be used in the statement of the rewriting rules and in the proof of correctness of our function inliner.

An applicative context[3] is either a hole, a let-binder over an applicative context, a case construct where one of the branches is an applicative context or a function

[3] exp_ctx and fundefs_ctx in theories/L6_PCPS/ctx.v

$$
\begin{array}{llll}
\text{(Function Context)} & fc & ::= & f\ (\vec{x})\ =\ C \\
\text{(Expression Context)} & C & ::= & [\![\ ]\!] \\
 & & & \mid \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ C \\
 & & & \mid \mathsf{let}\ x = \mathsf{Prim}\ p\ \vec{y}\ \mathsf{in}\ C \\
 & & & \mid \mathsf{let}\ x = \mathsf{Proj}_n\ y\ \mathsf{in}\ C \\
 & & & \mid \mathsf{let}\ \vec{fd}\ \mathsf{in}\ C \\
 & & & \mid \mathsf{let}\ \vec{fd} +\!\!+ fc :: \vec{fd}\ \mathsf{in}\ e \\
 & & & \mid \mathsf{match}\ x\ \mathsf{with}\ \vec{b} +\!\!+ (c \Rightarrow C) :: \vec{b}
\end{array}
$$

Figure 3.2: Applicative Context

bundle where exactly one of the function bindings has an expression context as body. An expression e can be placed in the hole of a context C to form expression $C[\![e]\!]$. Similarly, a context C_2 can be placed in the hole of a context C_1 to form a composed context $C_1 \cdot C_2$.

We define $\mathsf{BV}_{\mathrm{stem}}(C)$[4] to be the variables bound on the stem of C, the variables in scope at the hole in the applicative context C:

$$
\mathsf{FV}(C[\![e]\!]) \quad = \quad \mathsf{FV}(C) \cup (\mathsf{FV}(e) \setminus \mathsf{BV}_{\mathrm{stem}}(C))
$$

For example,

$$
\begin{aligned}
&\mathsf{BV}_{\mathrm{stem}}\big(\mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ \mathsf{let}\ \vec{fd} +\!\!+ (f\ (\vec{z})\ =\ [\![\ ]\!]) :: \vec{fd'}\ \mathsf{in}\ e\big) \\
&\quad = \{x,\ f\} \cup \vec{z} \cup \mathsf{names}(\vec{fd} +\!\!+ \vec{fd'})
\end{aligned}
$$

Logical relation

Our notion of equivalence reuses a step-indexed logical relation developed by Paraskevopoulou for the proof of correctness of CertiCoq's closure-conversion phase [44]. The main idea is that terms e_1 and e_2 are related at index k ($e_1 \cong_k^{\mathrm{val}} e_2$) whenever they are observationally equal for up to k β-reductions ($e_1 \approx_k e_2$).

[4] bound_stem_ctx in theories/L6_PCPS/stem_ctx.v

Two values v and w are related ($v \cong_k^{\text{val}} w$[5]) if $k = 0$ or if:

- both are constructors with k-equivalent arguments: $v = c\ v_1\ ...\ v_n$, $w = c\ w_1\ ...\ w_n$ and $\forall_{i=1}^{n}, v_i \cong_k^{\text{val}} w_i$
- both are functions, and for any related list of arguments, they evaluate to related values, which is to say: $v = (\rho_1',\ \vec{fd_1},\ f_1)$, $w = (\rho_2',\ \vec{fd_2},\ f_2)$ with $(f_1\ (\vec{x})\ =\ e_1) \in \vec{fd_1}$, $(f_2\ (\vec{y})\ =\ e_2) \in \vec{fd_2}$ and, given two lists $v_1, ..., v_n$ and $w_1, ..., w_n$ of $k-1$-related values ($v_i \cong_{k-1}^{\text{val}} w_i$), evaluating the functions' body after extending the functions' environments with these related mappings (for all $f_i \in \vec{fd_1}$, $g_i \in \vec{fd_2}$, $x_i \in \vec{x}$, $y_i \in \vec{y}$, $v_i \in \vec{v}$ and $w_i \in \vec{w}$) produces related values:

$$(\rho_1'; f_i \mapsto (\rho_1', \vec{fd_1}, f_i); x_i \mapsto v_i, e_1) \quad \cong_{k-1}^{\text{exp}}$$
$$(\rho_2'; g_i \mapsto (\rho_2', \vec{fd_2}, g_i); y_i \mapsto w_i, e_2)$$

Two environments ρ_1 and ρ_2 are related ($\rho_1 \cong_k^{\text{env}} \rho_2$) if, for every variable x, either x is not present in either, or $\rho_1\ x = v_1$ and $\rho_2\ x = v_2$ and $v_1 \cong_k^{\text{val}} v_2$.

Two terms e_1 and e_2 are related under environments ρ_1 and ρ_2 (written $(\rho_1, e_1) \cong_k^{\text{exp}} (\rho_2, e_2)$) if they evaluate to related values. More precisely, they are related at index k if, whenever $\rho_1 \vdash e_1 \Downarrow_j v_1$ (with $j \leq k$), then there exists some j' and v_2 such that $\rho_2 \vdash e_2 \Downarrow_{j'} v_2$ and $v_1 \cong_{k-j}^{\text{val}} v_2$.

If, for all i and for all environments ρ_1 and ρ_2 such that $\rho_1 \cong_i^{\text{env}} \rho_2$, two terms e_1 and e_2 are related according to $(\rho_1, e_1) \cong_i^{\text{exp}} (\rho_2, e_2)$, then e_1 and e_2 are contextually equivalent ($e_1 \approx e_2$).

[5] preord_val in theories/L6_PCPS/logical_relations.v

3.2 The Proof Framework

For compiler optimization phases, correctness means that the input and the output of the transformation compute to equivalent values. For reasoning purpose, this is often done in two steps:

1. First, an inductive relation relating the input and the output of the transformation is defined, and this relation is proven to correspond to the optimization phase

2. Then, it is proven that the relation only contains semantically related terms – this is the hard part

The first one is usually straightforward – in most cases, the induction principle arising from the definition of the inductive relation corresponds directly to the recursive form of the function. The second part is much harder, because it deals with relating a specialized, optimized transformation with very general semantics notion.

What we propose instead is to go through a series of rewrites relation which specializes towards different optimization phases and admits fewer and fewer programs. The first rewrite system, proven to be semantically correct, is similar to a general evaluation system for the system. Then, each rewrite system restricts the application of the rewrites rule down to describing the operations of the implemented code transformation. This method has two main benefits:

1. By reducing the gap between the first relation and the semantics equivalence, we simplify that proof, which is often the most complex portion of proofs of semantics correctness

2. By layering the proof, we allow for reusing portions of the correctness proofs for other optimization phases which uses similar operations

In the remainder of this chapter, we show two rewrite systems: a general rewrite system corresponding to general reductions rule over L6, and a shrinking rewrite system specialized to shrinking reductions. These systems are presented as rounded box in Figure 3.3. We then show applications of these system to the proofs of various optimizations, presented as squared box in Figure 3.3.

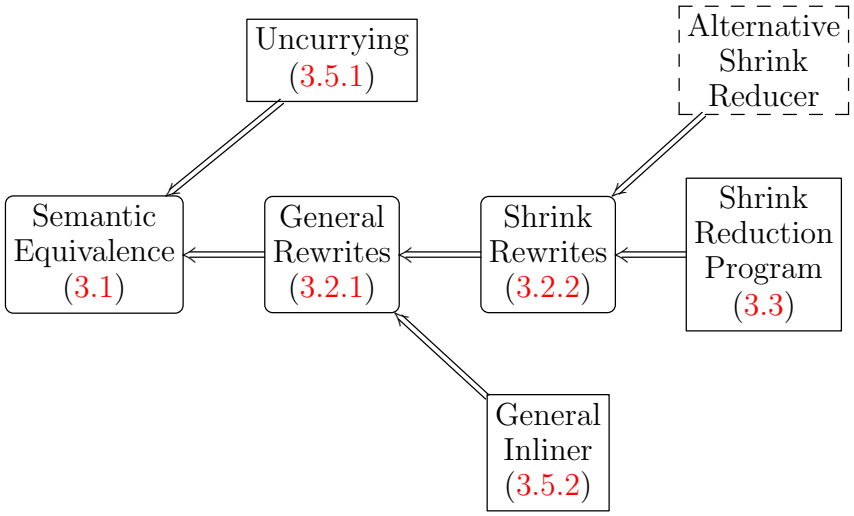


Figure 3.3: The Proof Framework

3.2.1 General Rewrites

Figure 3.4 shows the rules of our general rewriting system[6], which we then prove correct using the above logical relation.

Dead variables

When a variable does not occur statically within its scope, we can remove its binding without affecting evaluation. Dead variable rewriting rules have the form $\mathsf{let}\ x = \_\ \mathsf{in}\ e \rightsquigarrow e$, where $\_$ is any let-binding construct in our object language, for example $\mathsf{Con}\ c\ \vec{y}$, whenever x is not used in e.

[6] rw in theories/L6_PCPS/shrink_cps_correct.v

$$\frac{f \notin \mathsf{FV}(e_2) \quad \forall_{(f'\ (\vec{y}) = e_3) \in f\vec{d}_1 +\!\!+ f\vec{d}_2}, f \notin \mathsf{FV}(e_3)}{\mathsf{let}\ f\vec{d}_1 +\!\!+ (f\ (\vec{x})\ =\ e_1) :: f\vec{d}_2\ \mathsf{in}\ e_2 \rightsquigarrow \mathsf{let}\ f\vec{d}_1 +\!\!+ f\vec{d}_2\ \mathsf{in}\ e_2}\ \textsc{Dead\_fun}$$

$$\frac{\forall\ f \in \mathsf{names}(\vec{fd}), f \notin \mathsf{FV}(e)}{\mathsf{let}\ \vec{fd}\ \mathsf{in}\ e \rightsquigarrow e}\ \textsc{Dead\_bundle}$$

$$\frac{x \notin \mathsf{FV}(e)}{\mathsf{let}\ x = \_\ \mathsf{in}\ e \rightsquigarrow e}\ \textsc{Dead\_var}$$

$$\frac{(f\ (\vec{x})\ =\ e) \in \vec{fd} \quad (\mathsf{FV}(e) \bigcup \mathsf{names}(\vec{fd})) \bigcap \mathsf{BV}_{\mathrm{stem}}(C) = \emptyset \quad (\mathsf{BV}(e_\alpha) \bigcup \vec{x}) \bigcap \vec{y} = \emptyset}{\mathsf{let}\ \vec{fd}\ \mathsf{in}\ C[\![\mathsf{App}\ f\ \vec{y}]\!] \rightsquigarrow \mathsf{let}\ \vec{fd}\ \mathsf{in}\ C[\![(\vec{x} \mapsto \vec{y})e_\alpha]\!]}\ \textsc{Inl\_fun}$$

$$\frac{x \notin \mathsf{BV}_{\mathrm{stem}}(C) \quad (c \Rightarrow e) \in \vec{b}}{\mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ C[\![\mathsf{match}\ x\ \mathsf{with}\ \vec{b}]\!] \rightsquigarrow \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ C[\![e]\!]}\ \textsc{Fold\_case}$$

$$\frac{x \notin \mathsf{BV}_{\mathrm{stem}}(C) \quad z_n \notin \mathsf{BV}_{\mathrm{stem}}(C) \bigcup \mathsf{BV}(e)}{\mathsf{let}\ x = \mathsf{Con}\ c\ \vec{z}\ \mathsf{in}\ C[\![\mathsf{let}\ y = \mathsf{Proj}_n\ x\ \mathsf{in}\ e]\!] \rightsquigarrow \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{z}\ \mathsf{in}\ C[\![(y \mapsto z_n)e]\!]}\ \textsc{Fold\_proj}$$

Figure 3.4: General Rewrite Rules

To handle removal of dead mutually recursive functions, things are a bit more complicated; we use two rules to handle different scenarios under which it is safe to remove function bindings. Dead_bundle, removes a bundle of mutually recursive functions if none of them occurs in the rest of the term. However, this is too coarse-grain to handle the case where only some of the functions in the bundle are dead. For this situation, Dead_fun removes a function definition if it has no applied occurrences outside its own body.

Folding and inlining

Folding and inlining rules perform general reduction steps at compile time. One such folding rule is Fold_case, which performs ι-reduction whenever the correct branch can be statically predicted. Fold_case would be used to perform this reduction:

$$\begin{aligned}
&\mathsf{let}\ x = \mathsf{Con}\ S\ y\ \mathsf{in} \\
&\quad \mathsf{match}\ x\ \mathsf{with}\ (O \Rightarrow M); (S \Rightarrow \mathsf{let}\ z = \mathsf{Proj}_1\ x\ \mathsf{in}\ N) \\
&\rightsquigarrow \mathsf{let}\ x = \mathsf{Con}\ S\ y\ \mathsf{in}\ (\mathsf{let}\ z = \mathsf{Proj}_1\ x\ \mathsf{in}\ N)
\end{aligned}$$

As our language separates pattern-matching into the matching and the binding of projections, we can simply (using firstMatch) return the body of the first branch in $\vec{b}$ matching c in place of the match and let other reductions handle the projections, if any. These projections can then be folded using rule FOLD_PROJ. It lets us eliminate making a binding y for the nth projection of the value bound to x if x is bound in the context to $c\ \vec{z}$ and the nth variable of $\vec{z}$ is not rebound in the term. For example, we could further reduce the previous example:

$$\begin{aligned} &\mathsf{let}\ x = \mathsf{Con}\ S\ y\ \mathsf{in}\ (\mathsf{let}\ z = \mathsf{Proj}_0\ x\ \mathsf{in}\ N) \\ &\rightsquigarrow \mathsf{let}\ x = \mathsf{Con}\ S\ y\ \mathsf{in}\ (z \mapsto y)N \end{aligned}$$

Function inlining replaces a call to a function by the body of the function. If this was the only call to the function, the function definition can then be eliminated using the DEAD_FUN rule. It allows for α-renaming (changing the name of bound variables) of the function body e into e_α, This rule is only valid if $\mathsf{FV}(e)$ and the functions' name from $\vec{fd}$, including f, are disjoint from the variables bound on the stem of C, and if the function's parameters $\vec{x}$ and bound variables of e_α are disjoint from arguments $\vec{y}$, so that there is no variable capture occurring.

General rewrite system

From the rewrite rules shown so far, we create a rewrite system which will describe the transformations that our optimizations apply to a program.

We first take the contextual closure of $\rightsquigarrow$, denoted $\rightsquigarrow_C$[7], defined as:

$$\frac{e_1 = C[\![e_1']\!] \quad e_2 = C[\![e_2']\!] \quad e_1' \rightsquigarrow e_2'}{e_1 \rightsquigarrow_C e_2}$$

[7] gr_clos in theories/L6_PCPS/shrink_cps_correct.v

This allows reductions to happen anywhere in the term, following the usual notion of general reductions.

We then take the reflexive transitive closure of the contextual closure of the general rewrite rules to form a system of *General Reduction*, denoted $e \rightsquigarrow_C^* e'$.

Proof of correctness

Our correctness theorem has the following form: Any two terms related by general rewrites evaluate, under equivalent environments, to equivalent values.

Theorem 3.2.1 (Correctness of GR[8])**.**

$$\forall e_1\ e_2,\ e_1 \rightsquigarrow_C^* e_2 \implies \\ \forall \rho_1\ \rho_2\ k,\ \rho_1 \cong_k^{\text{env}} \rho_2 \implies (\rho_1, e_1) \cong_k^{\text{exp}} (\rho_2, e_2)$$

We prove a generalization of contextual compatibility that allows us to prove non-local rewrite rules. Contextual compatibility states that two expressions e_1 and e_2 are related (at k) under a given applicative context C in related evaluation environments ρ_1 and ρ_2 if, for any related ρ_3 and ρ_4, e_1 and e_2 are related (at k). This is because C will affect related ρ_1 and ρ_2 in the same way, resulting in related ρ_3 and ρ_4.

The function inlining case of this proof reuses the proof of equivalence of α-equivalent terms developed by Zoe Paraskevopoulou

Remark 3.2.2 (Contextual Compatibility[9])**.**

$$\forall e_1\ e_2\ C\ \rho_1\ \rho_2\ k, \\ \left(\forall \rho_3\ \rho_4,\ \rho_3 \cong_k^{\text{env}} \rho_4 \implies (\rho_3, e_1) \cong_k^{\text{exp}} (\rho_4, e_2)\right) \implies \\ \rho_1 \cong_k^{\text{env}} \rho_2 \implies \\ (\rho_1, C[\![e_1]\!]) \cong_k^{\text{exp}} (\rho_2, C[\![e_2]\!])$$

[8] rw_correct in theories/L6_PCPS/shrink_cps_correct.v
[9] preord_exp_compat in theories/L6_PCPS/shrink_cps_correct.v

$$\frac{|e|_x = 0}{\mathsf{let}\ x = \_\ \mathsf{in}\ e \rightharpoonup e}\ \textsc{S\_Dead\_var} \qquad \frac{\forall\ f \in \mathsf{names}(\vec{fd}), |e|_f = 0}{\mathsf{let}\ \vec{fd}\ \mathsf{in}\ e \rightharpoonup e}\ \textsc{S\_Dead\_bundle}$$

$$\frac{|\mathsf{let}\ f\vec{d}_1 +\!\!+ f\vec{d}_2\ \mathsf{in}\ Q|_f = 0}{\mathsf{let}\ f\vec{d}_1 +\!\!+ (f\ (\vec{x})\ =\ e) :: f\vec{d}_2\ \mathsf{in}\ Q \rightharpoonup \mathsf{let}\ f\vec{d}_1 +\!\!+ f\vec{d}_2\ \mathsf{in}\ Q}\ \textsc{S\_Dead\_fun}$$

$$\frac{(c \Rightarrow e) \in \vec{b}}{\mathsf{let}\ x = \mathsf{Con}\ c\ \vec{x}\ \mathsf{in}\ C[\![\mathsf{match}\ x\ \mathsf{with}\ \vec{b}]\!] \rightharpoonup \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{x}\ \mathsf{in}\ C[\![e]\!]}\ \textsc{S\_Fold\_case}$$

$$\frac{}{\mathsf{let}\ x = \mathsf{Con}\ c\ \vec{z}\ \mathsf{in}\ C[\![\mathsf{let}\ y = \mathsf{Proj}_n\ x\ \mathsf{in}\ e]\!] \rightharpoonup \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{z}\ \mathsf{in}\ C[\![(y \mapsto z_n)e]\!]}\ \textsc{S\_Fold\_proj}$$

$$\frac{|\mathsf{let}\ f\vec{d}_1 +\!\!+ (f\ (\vec{x})\ =\ e) :: f\vec{d}_2\ \mathsf{in}\ C[\![\mathsf{App}\ f\ \vec{y}]\!]|_f = 1}{\mathsf{let}\ f\vec{d}_1 +\!\!+ (f\ (\vec{x})\ =\ e) :: f\vec{d}_2\ \mathsf{in}\ C[\![\mathsf{App}\ f\ \vec{y}]\!] \rightharpoonup \mathsf{let}\ f\vec{d}_1 +\!\!+ f\vec{d}_2\ \mathsf{in}\ C[\![(\vec{x} \mapsto \vec{y})e]\!]}\ \textsc{S\_Shrink\_fun}$$

Figure 3.5: Shrink Rewrite Rules

However, this is too weak to prove the correctness of nonlocal rules such as Fold_proj, where, for the term that binds the projection to be related when the binding is substituted with the right projection, we need to ensure ρ_3 and ρ_4 still contain the binding of the constructor.

$$(\rho_1, \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{z}\ \mathsf{in}\ C[\![\mathsf{let}\ y = \mathsf{Proj}_n\ x\ \mathsf{in}\ e]\!])$$
$$\cong_k^{\exp}$$
$$(\rho_2, \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{z}\ \mathsf{in}\ C[\![(y \mapsto z_n)e]\!])$$

In order to prove this, we bind x in the context:

$$(\rho_1[x \mapsto (c, \vec{v})], C[\![\mathsf{let}\ y = \mathsf{Proj}_n\ x\ \mathsf{in}\ e]\!])$$
$$\cong_k^{\exp}$$
$$(\rho_2[x \mapsto (c, \vec{v})], C[\![(y \mapsto z_n)e]\!])$$

We cannot apply Contextual Compatibility here, because "$\mathsf{let}\ y = \mathsf{Proj}_n\ x\ \mathsf{in}\ e$" and "$(y \mapsto z_n)e$" are only related in contexts that map x to $(c, \vec{v})$ and z_n to v_N, even though x and z_n cannot appear in C due the premise of Fold_proj. So we must be more precise and state that C will only affect the mapping of variables of

$\rho_1[x \mapsto (c, \vec{v})]$ and $\rho_2[x \mapsto (c, \vec{v})]$ which are bound on the stem of C. Thus, we can select a set of variables S not bound in C and only consider ρ_3 and ρ_4 that agree with ρ_1 and ρ_2 on variables from S (this is written $\rho \doteq_S \rho'$). In our previous example, we could select $S = \{x, z_n\}$

Theorem 3.2.3 (Extended Contextual Compatibility[10])**.**

$$\begin{aligned}
&\forall e_1\ e_2\ C\ \rho_1\ \rho_2\ S\ k,\ \mathsf{BV}_{\text{stem}}(C) \bigcap S = \emptyset \implies \\
&\Big(\forall \rho_3\ \rho_4,\ \rho_1 \doteq_S \rho_3 \implies \rho_2 \doteq_S \rho_4 \implies \\
&\qquad \rho_3 \cong_k^{\text{env}} \rho_4 \implies (\rho_3, e_1) \cong_k^{\text{exp}} (\rho_4, e_2)\Big) \implies \\
&\rho_1 \cong_k^{\text{env}} \rho_2 \implies (\rho_1, C[\![e_1]\!]) \cong_k^{\text{exp}} (\rho_2, C[\![e_2]\!])
\end{aligned}$$

3.2.2 Shrink Reduction

Shrink Rewrites Most of the rules in Fig. 3.5 are very similar to the general rewrite rules given earlier. We write $|e|_x$ for the number of applied occurrences of variable x in expression e. The main difference is that their assumptions are computational, relying on the number of occurrences and (globally) on the unique binder property rather than on sets such as FV and BV. This is an important distinction which will make our life easier in the proof of correspondence to the algorithm. Consider for example S_Fold_proj. Due to the unique binding property, we can drop the assumption that $x \notin \mathsf{BV}_{\text{stem}}(C)$.

Other than the assumptions, the main difference between the two rewrite systems is the use of S_Shrink_fun in place of Inl_fun. Indeed, the latter does not qualify as a shrink reduction as the overall size of the program grows when we inline a function and keep its definition. S_Shrink_fun is only applicable when the inlined function has a single applied occurrence. It is admissible (assuming the unique binding property) from Inl_fun followed by Dead_fun.

[10] preord_exp_compat_stem_vals in theories/L6_PCPS/shrink_cps_correct.v

The shrink rewrite system and its correctness

We take the reflexive transitive closure of the contextual closure of the shrink rewrite rules presented in Fig. 3.5 to form a system of *shrink rewrites* denoted $\rightharpoonup_C^*$ [11].

We then prove that terms related by shrink reduction are also related by general reduction:

Theorem 3.2.4 (GR includes SR[12])**.**

$$\forall e_1\ e_2, e_1 \rightharpoonup_C^*\ e_2 \implies e_1 \rightsquigarrow_C^*\ e_2$$

Moreover, we use the fact that $\rightharpoonup$ is more restrictive than $\rightsquigarrow$ to prove certain properties for any term related by it. For example, shrink reduction preserves the unique binding property (this includes the disjointness of the bound and free variables of the term):

Theorem 3.2.5 (SR preserves UB[13])**.**

$$\forall e_1 e_2, e_1 \rightharpoonup_C^*\ e_2 \wedge \mathsf{UB}(e_1) \implies \mathsf{UB}(e_2)$$

The set of bound variables does not increase as we shrink a term:

Theorem 3.2.6 (SR reduces BV)**.**

$$\forall e_1 e_2, e_1 \rightharpoonup_C^*\ e_2 \implies \mathsf{BV}(e_2) \subseteq \mathsf{BV}(e_1)$$

It does not introduce free variables, for example at the top level:

Theorem 3.2.7 (SR reduces FV)**.**

$$\forall e_1 e_2, e_1 \rightharpoonup_C^*\ e_2 \implies \mathsf{FV}(e_2) \subseteq \mathsf{FV}(e_1)$$

[11] gsr_clos in theories/L6_PCPS/shrink_cps_correct.v
[12] grs_in_gr in theories/L6_PCPS/shrink_cps_correct.v
[13] gsr_preserves_clos in theories/L6_PCPS/shrink_cps_correct.v

Therefore, closed terms remain closed under shrink reductions[14].

3.3 Shrink Inliner

Function contract, shown in Figure 3.6, performs shrink reductions in a single pass down and up a program (or top-level expression) P. As contract proceeds down the term e (initially P, then some e such that $\exists C, P = C[\![e]\!]$), we collect in table ρ the functions and constructors which could respectively be inlined and folded.

In addition to the term e currently being transformed, contract maintains four tables:

- $\sigma : var \to var$, is a delayed renaming substitution (mapping variables to variables) under which e is being considered.
- $\delta : var \to nat$, tallies the number of occurrences of each variable in the whole program.
- $\rho : var \to \mathsf{Value}'$, maps function and constructor variables encountered so far (on the stem of C) to their definitions.

$$(\mathsf{Value}') \quad V \quad ::= \quad (c,\ \vec{x}) \ \mid \ (\vec{x}, e)$$

- $\theta : var \to bool$, indicates which functions have been inlined.

δ is updated using functions decreaseOcc $\sigma\ \delta\ \vec{x}$ which decreases by one the count of each variables in $\sigma\vec{x}$ and decreaseCount $\sigma\ \delta\ e$ which decreases $\delta(x)$ by $|\sigma e|_x$, the number of applied occurrences of x in σe.

In the SML/NJ implementation, these maps are implemented using imperative arrays with constant access time. As our compiler is implemented in Gallina, a pure functional language, we instead represent our variables as positive binary numbers

[14] gsr_preserves_clos in theories/L6_PCPS/shrink_cps_correct.v

and implement maps by *binary tries*, resulting in logarithmic access time. As shown in Fig. 3.1, this is still quite fast. Moreover, if one wanted to use constant-access-time impure arrays (a monadic extension to Coq)—thus recovering the original constant access time—our proof of correctness could easily be adapted.

At the top-level, function contract_top calls contract after initializing the maps: $\sigma = \mathsf{id}$, δ is initialized to have $\delta(x) = |e|_x$ for each variable x appearing in e, ρ is empty and θ maps all variables to $\bot$.

The function contract calls helper functions to process the branches of a pattern-match (contract_branches, see Figure 3.9) and blocks of recursive functions (preFun and postFun, see Figures 3.7 and 3.8).

When encountering a let-bound constructor "$\mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ e$", we first check, by looking up x in δ, if x does not occur in the whole program, in which case we can remove the binding of x and decrease the occurrence count for each variable in $\vec{y}$ under σ. Otherwise, we recursively shrink-reduce e after updating the environment map with the binding $x \mapsto (c, \vec{y})$. On return, we check again if x is dead in the updated counts (i.e., δ), as shrink reductions performed in e may have decreased the occurrence count of x. When encountering a let-bound projection "$\mathsf{let}\ x = \mathsf{Proj}_n\ y\ \mathsf{in}\ e$", if x is not dead, we look up σy in our environment map ρ to see if we statically know the construct $(c, \vec{y})$ bound to it. If it is, we can remove the binding of x and replace in the rest of e (by extending the renaming σ) all occurrences of x by the nth projection of $\sigma\vec{y}$ and update the count using "$\mathsf{foldCount}\ \delta\ \sigma\ x\ \vec{y}_n\ y$", setting x to 0, increasing the occurrence count of $\sigma\vec{y}_n$ by $\delta(x)$ and decreasing $\sigma\vec{y}$ by one.

When converting pattern-matching construct "$\mathsf{match}\ v\ \mathsf{with}\ \vec{b}$", we first look up v in ρ to see if we know enough about what is bound to it to select the correct branch, which is to say that $\rho(\sigma v) = (c, \vec{y})$ and $(c \Rightarrow e) \in \vec{b}$, and we proceed to shrink-reduce e after adjusting δ to account for the removed occurrence of σv and (using caseCount)

for the deletion of all other branches. If σv is not known or if no branches match, we recursively shrink-reduce each of the branches using contractCase (see Fig. 3.9)

When we get to an application "App f $\vec{y}$", we first look up σf in the environment map ρ to see if it is a known function $(\vec{x}, e)$ and if this is the only occurrence of σf. In that case, we inline the function and proceed with shrink inlining within its body e after updating the renaming substitution with mappings $x_i \mapsto (\sigma y_i)$ for each x_i, y_i in $\vec{x}, \vec{y}$ and updating the occurrence count with inlineCount σ δ f $\vec{x}$ $\vec{y}$, decreasing to 0 all $x_i \in \vec{x}$ and σf and adding $\delta(\vec{x_i}) - 1$ to each $\sigma\vec{y_i}$.

We process a block of mutually recursive functions "let $\vec{fd}$ in e" by first (using function preFun) adding live functions in $\vec{fd}$ to the environment map ρ. We then apply the contract function to e, the rest of the program. We then traverse $\vec{fd}$ a second time with function postFun, this time converting the body of live, non-inlined functions. The second traversal uses the initial ρ rather than the one augmented by preFun, such that we don't inline functions within their mutually recursive bundle. The algorithms of each of those pass are given in Figures 3.7 and 3.8.

preFun σ δ ρ $\vec{fd}$ is used on the downwards pass through the term, removing dead functions (and adjusting the occurrence count map δ accordingly) and adding the live ones to the environment ρ.

postFun σ δ ρ θ $\vec{fd}$ processes a block of mutually recursive function $\vec{fd}$ on the upward pass of contract. For each function f $(\vec{x})$ $=$ e, we first check if it has been inlined ($\theta(f) = \top$), in which case we simply remove the binding of the function and continue processing the rest of the block (as the count δ has already been adjusted at the inlining points, as shown in Fig. 3.6). If the function isn't inlined, we check if it is dead ($\delta(f) = 0$), in which case we delete the binding of f, decrease the count of variables occurring in the body of the function e under the renaming substitution σ and continue processing the rest of the block. Finally, if the function is neither dead

nor inlined, we apply the shrink inlining algorithm (see Fig. 3.6) to the body of the function before processing the rest of the block.

Proof of termination

contract is not structurally recursive. While most recursive calls are done on a strictly smaller subterm of its term input e, the inlining case receives a one-AST-node program (App f $\vec{y}$) and calls contract on the body of f as found in map ρ. However, if we believe our algorithm is indeed applying shrink reduction to the term, as we are going to prove next, we know that the size of the overall program is decreasing. We can use the other inputs of contract to approximate the size of the whole program. At any point in the algorithm contract e, while converting program P, there exists some applicative context C such that $P = C[\![e]\!]$. This context C consists of all of the bindings encountered on the way to e, some of which (those eligible to be inlined or folded) are reflected in ρ, minus all of the functions which have already been inlined. Our termination measure for contract σ δ ρ θ e is $|e| + |\rho|_\theta$ where $|e|$ is the number of AST nodes in e and $|\rho|_\theta$ the environment map size, defined as:

$$|\rho|_\theta = \sum_{x \in \mathbb{D}(\rho)} \mathsf{if}\ \big(\rho(x) = (\vec{x}, e) \wedge \theta(x) = \bot\big)\ \mathsf{then}\ |e|\ \mathsf{else}\ 0$$

Which is to say that we add up to the size of e the size of each body of non-inlined functions (according to θ) in ρ.

This approximation of the size of P is enough to show termination. For example, in the function inlining case where $\rho(f) = (\vec{x}, e)$ and $\theta(f) = \bot$, we start we size $|\mathsf{App}\ f\ \vec{y}| + |\rho|_\theta$ and the recursive call has measure $|e| + |\rho|_{\theta[f \mapsto \top]}$ which can easily be shown to be smaller:

$$\begin{aligned} |\mathsf{App}\ f\ \vec{y}| + |\rho|_\theta &= 1 + |\rho_{\setminus f}|_\theta + |e| \\ &= 1 + |\rho|_{\theta[f \mapsto \top]} + |e| \\ &< |e| + |\rho|_{\theta[f \mapsto \top]} \end{aligned}$$

Termination of the helper functions is proven in a similar manner. For contractCase, we keep track of the fact that the current $\vec{b}$ is a suffix of the original one $\vec{b'}$, and as such for any $(c \Rightarrow e) \in \vec{b}$, $|e| < |\mathsf{match}\ y\ \mathsf{with}\ \vec{b'}|$, and similarly for postFun with the list of function declaration $\vec{fd}$.

Proof of correspondence

Our proof of correspondence relies on top-level programs being closed. The main theorem for the correspondence of contractTop with our shrink-rewrite system is stated as:

Theorem 3.3.1 (contractTop on closed program is in SR[15])**.**

$$\mathsf{UB}(P) \wedge \mathsf{CLO}(P) \implies (P \rightarrow_C^* \mathsf{contractTop}\ P)$$

where $\mathsf{CLO}(e)$ is defined as $\mathsf{FV}(e) = \{\}$.

This composes with Theorem 3.2.4 and further with Theorem 3.2.1 to have:

Theorem 3.3.2 (Correctness of contractTop)**.**

$$\mathsf{UB}(P) \wedge \mathsf{CLO}(P) \implies P \approx \mathsf{contractTop}\ P$$

We might like to apply the shrink-reducer to open terms as well. For any term P with the unique-binding property $\mathsf{UB}(P)$, there exists a context C such that $\mathsf{CLO}(C[\![P]\!])$ and $\mathsf{UB}(C[\![P]\!])$; where C is constructed such that for any sequence of

[15] shrink_corresp_top in theories/L6_PCPS/shrink_cps_corresp.v

rewrites $C[\![P]\!] \rightharpoonup_C^* e'$, there exists some e'' such that $e' = C[\![e'']\!]$ and $P \rightharpoonup_C^* e''$. Thus, we can use an alternative function $\mathsf{contractTop}'$ which closes term P with C, performs shrink reductions and then returns the unpacked term. For this function, we have:

Theorem 3.3.3 (Correctness of $\mathsf{contractTop}'$)**.**

$$\mathsf{UB}(P) \implies P \approx \mathsf{contractTop}'\ P$$

As we recur down the program P and populate the different maps carried by $\mathsf{contract}$, we need a generalization of this theorem where the current term e being converted is related to the state of the top level program P. Every time the algorithm modifies the term, we have to justify it through our shrink-rewrite rules, which may depend on global properties about the program being transformed. For example, removing the definition of a dead variable involves invoking the DEAD_VAR rule which assume that the variable does not occur in the rest of the program, which would be inconvenient to calculate every time we want to use it. For that reason, a big part of the correspondence proof is to show that the maps that are maintained in the algorithm correctly represent the state of the whole program. Intuitively, while converting program P, at any point in the algorithm where we call $\mathsf{contract}\ \sigma\ \delta\ \rho\ \theta\ e$, there exists some applicative context C such that $P = \sigma(\mathsf{inline}\ C\ \theta)[\![\sigma e]\!]$, where inline is a function that removes the definition of any function f in C such that $\theta(f) = \top$. P is the state of the program, and each of the maps σ, ρ, δ and θ are correct (according to their invariants) for it. The reductions applied as we process term e affect P and the maps are adjusted accordingly.

The generalized theorem is:

Theorem 3.3.4 (contract is in SR[16])**.**

$$
\begin{aligned}
&\mathit{let}\ P := \sigma(\mathsf{inline}\ C\ \theta)[\![\sigma e]\!],\\
&\mathsf{UB}(P) \wedge\\
&\mathsf{CLO}(P) \wedge\\
&\mathsf{INV}_P(\delta) \wedge\\
&\mathsf{INV}_C(\rho) \wedge\\
&\mathsf{INV}_{\rho,P}(\theta) \wedge\\
&\mathsf{INV}_{\sigma(\mathsf{inline}\ C\ \theta),(\sigma e)}(\sigma)\ \Longrightarrow\\
&\qquad\qquad \exists e'\ \delta_r\ \theta_r,\\
&\qquad\qquad \mathit{let}\ P' := \sigma(\mathsf{inline}\ C\ \theta')[\![\sigma e']\!],\\
&\qquad\qquad (e', \delta_r, \theta_r) = \mathsf{contract}\ \sigma\ \delta\ \rho\ \theta\ e \wedge\\
&\qquad\qquad P \rightharpoonup_C^* P' \wedge\\
&\qquad\qquad \mathsf{INV}_{P'}(\delta_r) \wedge\\
&\qquad\qquad \mathsf{INV}_{\rho,P'}(\theta')\\
&\qquad\qquad \mathsf{INV}_{\sigma(\mathsf{inline}\ C\ \theta_r),(\sigma e')}(\sigma).
\end{aligned}
$$

which is to say that when running contract e with maps respecting their invariants and corresponding to a program P, contract returns a term e' and modified maps δ_r and θ_r describing the updated program P', and proofs that P shrink rewrites to P' and that the invariants still hold on current maps on the new state. The proof goes by induction on the size of the approximation of P given by $|e| + |\rho|_\theta$, just like the proof of termination.

We now detail the invariant on each of the maps and give a sketch of their importance in the proof of correspondence, before describing the auxiliary lemmas to handle case and functions.

[16] shrink_corresp in theories/L6_PCPS/shrink_cps_correct.v

$\mathsf{INV}_{C,e}(\sigma)$

σ is a renaming substitution under which the program is being considered. Its invariant states that any variable in its domain is not bound in P, and that variables in its range are either dead or bound on the stem of C:

$$\begin{aligned}\mathsf{INV}_{C,e}(\sigma) := \quad & \forall x\ y, \\ & (x \mapsto y) \in \sigma \implies x \notin \mathsf{BV}(P) \\ & \qquad \wedge |P|_y = 0 \vee y \in \mathsf{BV}_{\text{stem}}(C)\end{aligned}$$

σ is applied everywhere in P, both in e and in C. Due to the unique binding property, adjustment to σ due to a variable bound in e will not affect C, because the variable could not occur free (or otherwise) in C ($\sigma\mathsf{weaken}$). Moreover, the domain of σ is disjoint from its codomain. Combined with the fact that we only add to σ mapping from binding we remove (inlined functions arguments, folded projections, etc.), we can freely fuse multiple delayed substitutions together ($\sigma\mathsf{fuse}$) or stage them as needed, as shown in Figure 3.10.

$\mathsf{INV}_{\rho,P}(\theta)$

θ keeps track of which functions have been inlined by the algorithm. θ is threaded through the algorithm, and it is shown monotonic, which is to say that for any variable f, if $\theta(f) = \top$ for input θ then output θ_r will have $\theta_r(f) = \top$, which is important to prove termination of $\mathsf{contract}$. For the proof of correctness of $\mathsf{contract}$, θ's invariant states that inlined functions and their arguments do not appear bound in P.

$$\begin{aligned}\mathsf{INV}_{\rho,P}(\theta) := \quad & \forall f,\ \theta(f) = \top \implies \\ & f \notin \mathsf{BV}(P) \wedge \rho(f) = (\vec{x}, e) \implies \vec{x} \notin \mathsf{BV}(P)\end{aligned}$$

$\mathsf{INV}_P(\delta)$

δ accounts for the number of occurrences of each variable in P. Its invariant is stated as:

$$\mathsf{INV}_P(\delta) := \quad \forall x, |P|_x = \delta(x)$$

We say δ is a correct count for P if for all variables x, $\delta(x)$ is exactly the number of times x occurs in P. In the statement of the theorem, this accounts for the delayed substitution σ and for the bodies of inlined functions according to θ.

The unique binding property is important here again to ensure the algorithm updates the count correctly. For example, on "$\mathsf{contract\ let}\ x = \mathsf{Proj}_n\ y\ \mathsf{in}\ e$", we know δ is correct for "$(\sigma C_\theta)[\![\sigma(\mathsf{let}\ x = \mathsf{Proj}_n\ y\ \mathsf{in}\ e)]\!]$" for some C which respects the provided maps. In the case where we fold the projection, we need to prove that δ after "$\mathsf{foldCount}\ \delta\ \sigma\ x\ \vec{y}_n\ y$" is correct for "$(\sigma_{x \mapsto (\sigma \vec{y}_n)}\ C_\theta)[\![\sigma[x \mapsto \sigma\vec{y}_n]e]\!]$". We can first observe that x cannot occur in C_θ due to the unique binding property, so this is equivalent to "$(\sigma C_\theta)[\![\sigma[x \mapsto (\sigma\vec{y}_n)]e]\!]$". By the invariant on σ, we know that x is neither in the domain or the range of σ and as such "$\sigma[x \mapsto (\sigma\vec{y}_n)]e$" is the same as "$(x \mapsto (\sigma\vec{y}_n))(\sigma e)$", which is to say we can first apply σ before substituting $\sigma\vec{y}_n$ for x. Finally, by the unique binding property, we know that x will not be bound in e such that all of its occurrences will be replaced by $\sigma\vec{y}_n$, which brings us to the correct count.

$\mathsf{INV}_C(\rho)$

ρ is a view of the current context. The invariant for ρ asserts that it contains every function and constructor on the stem of C and nothing more.

$$
\begin{aligned}
\mathsf{INV}_C(\rho) := \quad & \forall x, \\
& \rho(x) = (c, \vec{y}) \leftrightarrow \exists C_1 C_2, C = C_1 \cdot (\mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ C_2) \\
& \wedge \\
& \rho(x) = (\vec{y}, e) \leftrightarrow \exists C_1 C_2\ \tilde{fd}, C = C_1 \cdot (\mathsf{let}\ \vec{fd}\ \mathsf{in}\ \ C_2) \\
& \qquad\qquad \wedge (x\ (\vec{y})\ =\ e) \in \vec{fd}
\end{aligned}
$$

Some of the functions in ρ may have been inlined (such that they are not in $\mathsf{inline}\ C\ \theta$) and are thus not eligible to be inlined. However, this means they do not occur in P, so we will will never look them up in ρ again.

Auxiliary proofs

When converting case and bundles of functions, we call the auxiliary functions shown in figure 3.9, 3.7 and 3.8. Just like for contract, we need to carefully select a P that best represents the current state of the program; it is important to be aware of which portions of the term have already been converted as they no longer need to be considered under delayed σ and what is available to be folded or inlined.

Case When contracting term "$\mathsf{match}\ x\ \mathsf{with}\ \vec{b}$", we first verify if we can fold the statement. If this is not possible, we contract each of the branches in $\vec{b}$ using function contractCase. As we progress through the lists of branches, $\vec{b}$ is split into the contracted branches $\vec{b_1}$ (initially empty) and its remaining suffix $\vec{b_2}$ (empty when the contractCase returns to contract) . When calling "$\mathsf{contractCase}\ \sigma\ \delta\ \rho\ \theta\ \vec{b_2}$", the current state P is

$$\sigma(\mathsf{inline}\ C\ \theta)[\![\mathsf{match}\ x\ \mathsf{with}\ (\vec{b_1} +\!\!+ \sigma\vec{b_2})]\!]$$

The invariant on σ allows us to prove that $x = \sigma x$ and $\vec{b_1} = \sigma\vec{b_1}$ such that

$$\mathsf{match}\ x\ \mathsf{with}\ (\vec{b_1} +\!\!+ \sigma\vec{b_2}) = \sigma\big(\mathsf{match}\ x\ \mathsf{with}\ (\vec{b_1} +\!\!+ \vec{b_2})\big)$$

On $b_2 = (c \Rightarrow e_b) +\!\!+ b_3$, we can rewrite the state as

$$\sigma\Big(\mathsf{inline}\ \big(C \cdot \mathsf{match}\ x\ \mathsf{with}\ \vec{b_1} +\!\!+ (c \Rightarrow [\![\ ]\!]) :: \vec{b_3}\big)\ \theta\Big)[\![\sigma e_b]\!]$$

to recur on e_b with contract. On return b_3' with updated δ_r and θ_r, the state is

$$P' = \sigma(\mathsf{inline}\ C\ \theta_r)[\![\mathsf{match}\ x\ \mathsf{with}\ \vec{b_1} +\!\!+ (c \Rightarrow e_b') :: \vec{b_3}]\!]$$

We also return proofs that $P \rightharpoonup_C P'$, that δ_r is a correct count for P', that the invariant for θ_r holds for σ and P' and that the invariant for σ holds for P'.

Functions When contract is called on a bundle of functions "$\mathsf{let}\ \vec{fd}\ \mathsf{in}\ e$", we first call "$\mathsf{preFun}\ \sigma\ \delta\ \rho\ \vec{fd}$", before converting e and calling "$\mathsf{postFun}\ \sigma\ \delta\ \rho\ \theta\ \vec{fd}$". The carried maps already account for some prefix $\vec{fd_1}$ for which $\vec{fd_1} +\!\!+ \vec{fd_2} = \vec{fd}$, with $\vec{fd_1} = [\ ]$ at first.

For "$\vec{fd_2'} \leftarrow \mathsf{preFun}\ \sigma\ \delta\ \rho\ \vec{fd_2}$", program P, originally

$$\sigma(\mathsf{inline}\ C\ \theta)[\![\sigma(\mathsf{let}\ \vec{fd_1} +\!\!+ \vec{fd_2}\ \mathsf{in}\ e)]\!]$$

is updated to

$$P' = \sigma(\mathsf{inline}\ C\ \theta)[\![\sigma(\mathsf{let}\ \vec{fd_1} +\!\!+ \vec{fd_2'}\ \mathsf{in}\ e)]\!]$$

with $P \rightharpoonup_C P'$. Functions in $\vec{fd_2}$ which are already dead have their bindings removed from $\vec{fd_2'}$. δ_r is updated accordingly, and is correct from P'. The updated environment ρ_r adds to ρ all the functions bound by $\vec{fd_2'}$. Because the names in $\vec{fd}$ are disjoint from the inlined functions as tallied by θ, the resulting P' (where $\vec{fd_1}$ is empty) can be rewritten as

$$\sigma\big(\mathsf{inline}\ (C \cdot \mathsf{let}\ \vec{fd_2'}\ \mathsf{in}\ [\![\ ]\!])\ \theta\big)[\![\sigma e]\!]$$

which is in the right form to contract e.

When we call "postFun σ δ ρ θ $\vec{fd}$" from contract, e has already been converted by the main function, and we turn on to processing the bodies of live, noninlined functions in $\vec{fd}$. After converting e, the program state P is

$$\sigma\big(\mathsf{inline}\ (C \cdot \mathsf{let}\ \vec{fd_1} +\!\!+ \vec{fd_2}\ \mathsf{in}\ [\![\ ]\!])\ \theta\big)[\![e_r]\!]$$

When $\vec{fd_2} = (f\ (\vec{x})\ =\ e_b; \vec{fd_3})$ for some live f, we need to show that we can rewrite P to be of the right form for its body e_b to be translated (into e_b') using convert:

$$\sigma\Big(\mathsf{inline}\ \big(C \cdot \mathsf{let}\ \vec{fd_1} +\!\!+ (f\ (\vec{x})\ =\ e_b) :: \vec{fd_3}\ \mathsf{in}\ [\![\ ]\!]\big)\ \theta\Big)[\![e_r]\!]$$
$$=$$
$$\sigma\Big(\mathsf{inline}\ \big(C \cdot \mathsf{let}\ \vec{fd_1} +\!\!+ (f\ (\vec{x})\ =\ [\![\ ]\!]) :: \vec{fd_3})\ \mathsf{in}\ e_r\big)\ \theta\Big)[\![e_b]\!]$$

By the invariant on σ and θ, we know e_r is equivalent to σe_r and that inline with θ has no effect on e_r. The proof of correctness for postcontract carries this fact along to be able to move e_r in and out of the context as we recur on functions' bodies. postFun updates δ_r and θ_r and returns $\vec{fd_3'}$ which can form $\vec{fd_2'} = (f\ (\vec{x})\ =\ e_b'; \vec{fd_3'})$, with the resulting state being

$$P' = \sigma\big(\mathsf{inline}\ (C \cdot \mathsf{let}\ \vec{fd_1} +\!\!+ \vec{fd_2'}\ \mathsf{in}\ [\![\ ]\!])\ \theta_r\big)[\![e_r]\!]$$

which can be rewritten as

$$\sigma(\mathsf{inline}\ C\ \theta_r)[\![\mathsf{let}\ \vec{fd_1} +\!\!+ \vec{fd_2'}\ \mathsf{in}\ e_r]\!]$$

We also return proofs that $P \rightarrow_C P'$ and that the maps properly characterize P'.

3.3.1 Reduction of Administrative redexes

Administrative redexes are β-redexes introduced by the CPS transformation and that can safely be reduced without affecting the original term. For example, an early CPS transformation [45] converts the term "$(\lambda x.x)\ y$" as

$$\lambda k_1.(\lambda k_2.k_2(\lambda x.\lambda k_3.k_3\ x))(\lambda m.(\lambda k_4.k_4\ y)(\lambda n.(m\ n)\ k_1))$$

Implementations of the CPS transformation in several compilers, in order to generate smaller terms that leave less work for later optimization phases to do, cleverly avoid producing so many administrative redexes [16, 46]. Danvy and Nielsen [17] give a comprehensive account of different CPS transformations and on the administrative redexes they introduce.

But these clever CPS transformations that avoid producing administrative redexes are more difficult to prove correct [19]. Furthermore, some administrative redexes *should not* be reduced! They represent join points of the control flow; reducing them duplicates the instructions following the join point [47]. This duplication occurs in many optimizing CPS transformations over languages with pattern-matching [47, 19].

We recommend: use a simple CPS transformation that makes no effort to reduce administrative redexes; then use shrink-reduction. This is approximately as efficient as the more clever CPS transformation, and it reduces just the right set of redexes, including all the administrative that are not join points.

Theorem 3.3.5. *All administrative redexes with a single applied occurrence will be reduced in a single pass of the shrink inliner.*

The proof is a corollary of our proof of correspondence of the shrink inliner (Theorem 3.3.4), where we prove that the algorithm correctly tabulates the number of occurrences for every variables in the program, such that administrative redexes with a single applied occurrence will be eligible for inlining when we get to them during

the first shrink inlining pass. Since all administrative redexes introduced by the CPS translation have a single applied occurrence[17], which is to say that there is no dead applied occurrences that would require clever ordering of reductions to eliminate in a single pass, all of them are reduced in a single pass.

3.4 Performance

Benchmark	Binom	Color	Veristar
Size without Shrink Inlining (AST nodes)	3156	76.6k	82.0k
Size with S.I. (AST nodes)	616	28.5k	14.8k
# of evaluation steps without S.I.	4560	120.3M	348.3M
# of evaluation steps with S.I.	1132	26.9M	82.9M
Time for one S.I. pass (sec.)	0.0069	0.34	0.27
# inlined functions in one S.I. pass	620	9240	14305
# of cases folded by one S.I. pass	2	1	8
# of projections folded by one S.I. pass	2	2	14
# of dead constructors removed by one S.I. pass	41	52	486
# of dead functions removed by one S.I. pass	0	87	51
# of shrink reductions performed by second S.I. pass	0	24	16
# of shrink reductions performed by third S.I. pass	0	3	0
Size after closure conversion without S.I. (AST nodes)	6390	188.5k	255.8k
Size after C.C. with S.I. (AST nodes)	1163	34.9k	32.3k
Time for C.C. without S.I. (s.)	0.30	1039.68	481.43
Time for C.C. with S.I. (s.)	0.0080	3.28	1.41
# of functions inlined by S.I. after C.C.	0	0	0
# of cases folded by S.I. after C.C.	0	0	0
# of projections folded by S.I. after C.C.	6	136	250
# of dead constructors by S.I. removed after C.C.	4	1261	796
# of dead functions by S.I. removed after C.C.	0	4	0

Table 3.1: Shrink reduction performance measurements

[17]Case statements are often converted with the continuation hoisted outside of the branches, resulting in sharing a single continuation between all branches (and in multiple applied occurrences). This conflates common-subexpression evaluation with the conversion – we instead bind the continuation in each branch to ensure reduction of all administrative redexes. Case-folding from shrink reducing, or an eventual common-subexpression phase would then be able to eliminate the leftover, duplicated continuations

We have tested the effect of the shrink inliner on a few programs when evaluated in the intermediate language on which the transformation is performed. The results are included in Figure 3.1. *Binom* is an implementation of binomial queues [59] (priority queues with log-time insert, delete-min, and merge). *Color* runs a verified implementation of the Kempe/Chaitin algorithm for graph coloring [11] on a large graph. *Veristar* [52] is a verified theorem prover (resolution theorem proving with paramodulation) for a subset of separation logic, run over a large entailment.

We see a significant number of functions inlined in a single shrink inlining (S.I.) pass, resulting in substantially smaller programs that run 5x faster. Most of the inlined functions are administrative redexes. Although one pass does not always reduce to shrink-normal form, very few redexes remain for the second and third passes; this justifies the *quasilinear time*[18] designation [8].

Shrink-inlining is fast: even on a large program such as *Veristar*, it takes a fraction of a second. The table shows that it's important to shrink-inline both before and after closure-conversion; if not run before, closure-conversion takes too long; if not after, the compiled program will run slower.

3.5 Other Optimizations over L6

3.5.1 Uncurrying

By default, all functions in Gallina are curried, which is to say that they take a single argument. L6 functions, on the other hand, explicitly take multiple arguments, and each L6 function call must supply all arguments to the function. Multiple-argument functions are represented in the Coq kernel using series of function declarations, and calling a function with n arguments is represented as n function calls. Uncurrying

[18] Our implementation runs in *quasi-*$N \log N$ time due to using functional datastructures with $\log N$ access time.

eliminates such calls, creating fully applied version of the functions which can be called directly by fully applied application sites. An example of the effect of uncurrying is given in Section 2.6.2. We include in figure 3.11 the rule for creating uncurried shells for curried functions, taken from John Li's report on the proof of correctness of the uncurry phase [32]. In a case of a function of n arguments, BUNDLE-CURRIED is applied $n-1$ times.

The proof of correctness of the uncurrying phase is described in a report by John Li [32]. Uncurrying, like closure-conversion, does not correspond to operations of general reductions, and thus the proof is made directly over the logical relation at the base of the proof framework. We note that if parts of the transformation were parts of rewrite systems in the framework, the proof of correctness of that rewrite systems could be reused directly in the context of the proof of correctness of the transformation with respect to the logical relation. This happens, for example, whenever α-conversion is used as part of an optimization phase.

3.5.2 Function Inlining

While shrink inlining is an efficient way to clean up code after transformations introducing shrink redexes, other transformations introduce redexes which are not shrink redexes. For this, we implemented a general β-reducer. We proved this phase correct using the proof framework described in Section 3.2 – function inlining corresponds directly to the general β-reduction rule included in the general rewrite system (Section 3.2.1).

Function inlining β-reduces some of the redexes in a program, without the restriction of the shrink reducer, which only reduces β-redexes if such a reduction would result in a dead function. Of course, not every β-redex should be reduced – this could be non terminating, in the case where unreachable code is non terminating, or could

lead to an explosion in the size of the program. For this, we want to be able to select which β-redex is reduced.

To avoid specializing the inlining phase to a specific use-case, we separate the transformation into two phases:

1. First, a phase computes an inlining heuristic which can statefully determine which β-redexes should be reduced

2. Then, this inlining heuristic is provided to a general inlining phase as a parameter.

We see benefits in separating analysis and rewriting phases – the analysis may do complicated reasoning, and produce potentially unsafe optimization heuristic, but it doesn't have to be proven correct. Meanwhile, the rewriting phase only follow through with safe rewrites, and is proven correct for any heuristic. This separation simplifies the rewriting phase, making it easier to prove it correct. It also makes it reusable for different optimizations, for example, in this case, to inline uncurried shells and to inline small functions.

Inlining Heuristic The inlining heuristic is described as a record holding three functions:

- updateFunDef, a function updating the inlining decision at the declaration point of bundles of mutually recursive functions

- updateInFun, a function updating inlining decision when converting a particular function within a bundle

- updateApp, a function updating and returning inlining decision on applications

Examples of Inlining Heuristics In **CertiCoq**, we have two uses for function inlining:

1. The first is to inline function definitions when these definitions are small. Functions under a small size b can be inlined (up to a maximum inlining depth n) without worrying about the size of the resulting program.

   The inlining heuristic for small functions has as state s a map from function name to inlining decision, initialized with everything mapping to *false*. For a fixed bound b, we would have:

   - updateFunDef for every function f in the bundle, set f to *true* if the size of its body is under the bound b
   - updateInFun on f sets f to *false*, to avoid unfolding a function body within itself
   - updateApp on App f $\vec{y}$ returns as inlining decision $s(f)$ without updating the heuristic

2. The second arises from the uncurry optimization which is performed on L6, described in section 3.5.1. This optimization creates shell functions currying the application of newly created uncurried functions. Whenever possible, we would like to inline these shells, and, if the original curried function does not escape, we would like to completely remove all occurrences of it, replacing it by the uncurried version.

   For post-uncurrying contraction, our inlining heuristic has as state s a map from function name to a natural number, initialized with all functions shell mapping to 1, and updated to have the continuation of shells mapping to 2:

   - updateFunDef does not update the heuristic
   - updateInFun does not update the heuristic
   - updateApp on App f $k :: \vec{y}$ checks if f is a shell (1), in which case we set k to 2 and return *true* as inlining decision, on App k $\vec{y}$ checks if k is a

shell's continuation (2), in which case we return *true* as inlining decision, otherwise we return *false*.

The inlining decisions can be composed, with their state becoming a product of the states, and their inlining decision becoming a boolean *or* of the inlining decisions of the underlying heuristics:

- $\theta = (\theta_1, \theta_2)$
- $\mathsf{updateFunDef}\ \theta\ e = (\mathsf{updateFunDef}_1\ (\pi_1\ \theta)\ e, \mathsf{updateFunDef}_2\ (\pi_2\ \theta)\ e)$
- $\mathsf{updateInFun}\ \theta\ e = (\mathsf{updateInFun}_1\ (\pi_1\ \theta)\ e, \mathsf{updateInFun}_2\ (\pi_2\ \theta)\ e)$
- $\mathsf{updateApp}\ \theta\ e = \mathsf{updateApp}_1\ (\pi_1\ \theta)\ e \vee \mathsf{updateApp}_2\ (\pi_2\ \theta)\ e$

General function inlining in Coq We parameterize our function inlining phase with an inlining heuristic which determines, at every function application, if a function should be inlined. For termination purpose, we provide a maximum inlining depth which we decrease when converting the body of an inlined function.

We show in Figure 3.12 the function inlining algorithm. Function inline traverses an expression e while querying and updating θ, the state of the inlining heuristic. In addition to e, the term e being transformed, θ, the state of the inlining heuristic, and n, the maximum inlining depth, inline maintains two tables:

- $\sigma : var \to var$, is a delayed renaming substitution (mapping variables to variables) under which e is being considered.
- $\rho : var \to \mathsf{Value}'$, maps function variables encountered so far to their definitions.

inline traverses e, propagating downward the state of the inlining heuristic. There are two interesting cases in the traversal of e: function declaration, and function application.

When reaching the declaration of a mutually recursive bundle of functions "let $\vec{fd}$ in e", we first extend the function environment ρ with the function bindings $\vec{fd}$. We then query the inlining heuristic updateFunDef, providing us with states θ_1 and θ_2 which are used, respectively, to convert the functions and the rest of the program. We recursively transform expression e in the updated function environment and the updated heuristic state θ_1, before transforming all of the function bodies in bundle $\vec{fd}$ with updated heuristic state θ_2.

Function inlineFun$_n$ ρ σ θ $\vec{fd}$ is defined as a mutually recursive function together with inline. For every function declaration $(f\ (\vec{x})\ =\ e_b) \in \vec{fd}$, we update the heuristic state θ (θ_2 from the last paragraph) using the heuristic at function declaration updateInFun, before converting the body e_b.

The second important case is on function application "App f $\vec{y}$". There, after applying the renaming substitution σ, we query updateAppto know if σ f should or should not be inlined. If it is to be inlined, and ρ contains the declaration of σ f as $(\vec{x}, e)$, we extend the renaming substitution σ with parameters $\vec{x}$ mapping to arguments σ $\vec{y}$, and recursively call inline while decreasing the maximum inlining depth by one.

Preserving globally unique names In order to preserve globally unique names when performing general inlining, we have to freshen the name of variables bound in the term being substituted. To do so, we have a global counter keeping track of the next available name. We can then freshen the variables bound in a term by replacing each of them with sequential names starting from the next available name, and replacing the bound occurrences with the new names – substituting new names for the originally bound one is simple since the original term was uniquely bound and as such no capture can occur. The resulting term is α-equivalent to the original one, but its variable bindings are globally unique.

Proof of Correctness The proof of correctness of inline goes by lexicographic induction on (e, n). The statement of correctness is:

Theorem 3.5.1 (inline in GR)**.**

$$\mathsf{UB}(P) \implies (P \rightharpoonup_C^* \mathsf{inline}_n \ \cdot \ \cdot \ \theta \ P)$$

Which is to say that P can be rewritten using the general rewrite system (see Section 3.2.1), under any starting state θ and maximum inlining depth n, to the output of inline. The proof follows directly from showing that the rewriting done by the inlinecorresponds to Rule INL_FUN (see Figure 3.4). This proof composes with the proof of correctness of the general rewrite system (Theorem 3.2.1) to achieve a proof that the input and the output of inlineare semantically related under our logical relation (Section 3.1).

3.6 Related Work

The shrink inliner we present in Figure 3.6 is taken almost directly from Appel and Jim [8], who describe the algorithm implemented in the SML/NJ compiler. They present a set of rewriting rules which was the main source of inspiration for our shrink-rewrite system, and prove its confluence. The main difference is that their algorithm allowed occurrence-counts to be over-approximations, and they split the occurrence-counts into applied and escaping in order to tolerate this approximation. However, with a few changes to the occurrence updates, we can get the exact number of occurrences in our map δ, and as such have no reason to split it into the two types of occurrences.

In addition to the algorithm we implemented, Appel and Jim [8] presented fully linear-time algorithm that heavily uses imperative graph-update, which is less convenient to implement in a functional programming language. Kennedy [28] improved,

implemented, and measured the fully linear-time algorithm, and reported excellent performance. Neither Appel and Jim nor Kennedy formally proved the correctness of their their algorithms.

The CakeML project [53] includes a verified compiler for a "substantial subset of Standard ML". It contains a two-pass optimization inlining small, non-recursive functions. Inlining decisions are not updated as inlining is performed, making it similar to the naive algorithm that Appel and Jim show performs much worse than linear time. The optimization pipeline also includes a constant propagation and folding phase which, for example, folds *if*-statements if their guard can be computed statically. However, doing these optimizations in different phases misses cascading reductions where further optimizations are enabled by each reduction.

Pilsner [42] is a verified compiler with an ML-like source language and CPS-based intermediate language. It includes a simple function-inlining optimization which does not update its inlining decisions during the inlining pass, nor does it inline within the body of inlined functions. It has a dead-variable-elimination phase deleting dead definitions in a single pass up and down a program. In addition to missing cascading reductions, it misses optimization opportunities from the interaction of dead definitions arising from projection-folding, which this optimizer does not do.

CompCert [29] is a verified optimizing compiler for C. It includes a function inlining pass. However, the decisions to inline are taken in a different pass and are not updated as inlining is done. There is no attempt (and in a C compiler, less need) to combine inlining, constant folding, and dead-variable elimination into a single efficient pass.

Administrative redexes in the context of CPS transformations have been the subject of many papers since being introduced by Plotkin [45]. Our pipeline which consists of a simple CPS transformation followed by a pass which reduces administrative redexes is similar to the two-pass CPS transformation presented by Sabry and

Felleisen [46]. However, our shrink inlining pass is not limited to reducing administrative redexes; it also performs case-folding, dead-variable elimination, and reduction of many non-administrative redexes.

Administrative normal form (ANF)[24] is a representation aimed at providing the benefits of CPS while still being in direct style. While some transformations such as contification are difficult to represent in ANF[28], recent work[34] has shown how ANF can be augmented with explicit *join points* to allow these transformations to be performed. In addition, using ANF with *join points* instead of CPS may make the identification (and subsequent elimination) of common-subexpressions easier. One difference between our language and ANF is that the latter does not restrict arguments of function applications to be atoms. This mean that a shrink inlining rewrite (S_Shrink_fun from Figure 3.5) could increase the size of the term. We believe ANF could be made to enforce that restriction. Alternatively, we could change the shrink inlining rule to only apply when, for each of the parameters of the function being applied, either they occur a single time in the body, or the applied argument is an atom. In order to limit test to the garbage collection trigger to the start of functions (see Section 4.1.6), the code generated by our back end uses the property that all calls are in tail position, and that we can statically compute the maximum number of values allocated by functions. ANF does not ensure that all calls are in tail position, making the computation of roots and of the maximum allocation between function calls more complicated, so that we would have to use the more conservative approach of testing for the garbage collection trigger at every allocation.

3.7 Conclusion

In this chapter, we presented a proof of correctness for a shrink inliner compilation phase combining constant folding, function inlining and dead-variable elimination.

The full proof composes multiple correspondence proofs, step-by-step refining a semantic notion of equivalence into our syntax-driven algorithm.

We also showed how other transformations could reuse the proof of correctness of the general rewriting system, either directly or through a refined system such as shrink rewrites.

Proving correspondence of the algorithm to the shrink-rewrite system rather than the general one or the logical relation significantly simplifies the reasoning. As previously stated, some of the invariants on the terms and maps, such as closedness, are preserved by shrink reductions, and as such do not have to be threaded through the proof. Moreover, the shrink-rewrite system already incorporates some optimizations that make it easier to prove the algorithm correspondence. For example, substitution is performed in a global way since the unique binding property prevents any shadowing and capture of variables. Meanwhile, the notion of substitution used for the rewrite system is the more usual one which corresponds closely with the semantics of our language which is defined for nonuniquely bound terms.

$\mathsf{contract}\ \sigma\ \delta\ \rho\ \theta\ e = \text{match}\ e\ \text{with}$

$$
\begin{array}{ll}
\mid \mathsf{halt}\ x & \Rightarrow (\mathsf{halt}\ (\sigma x), \delta, \theta) \\
\mid \mathsf{let}\ x = \mathsf{Prim}\ p\ \vec{y}\ \mathsf{in}\ e & \Rightarrow \text{if}\ \delta(x) = 0 \\
 & \quad \text{then}\ \delta \leftarrow \mathsf{decreaseOcc}\ \delta\ \sigma\ \vec{y} \\
 & \quad\quad \mathsf{contract}\ \sigma\ \delta\ \rho\ \theta\ e \\
 & \quad \text{else}\ (e', \delta, \theta) \leftarrow \mathsf{contract}\ \sigma\ \delta\ \rho\ \theta\ e \\
 & \quad\quad \text{if}\ \delta(x) = 0 \\
 & \quad\quad \text{then}\ \delta \leftarrow \mathsf{decreaseOcc}\ \delta\ \sigma\ \vec{y} \\
 & \quad\quad\quad (e', \delta, \theta) \\
 & \quad\quad \text{else}\ (\mathsf{let}\ x = \mathsf{Prim}\ p\ (\sigma\vec{y})\ \mathsf{in}\ e', \delta, \theta) \\
\mid \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ e & \Rightarrow \text{if}\ \delta(x) = 0 \\
 & \quad \text{then}\ \delta \leftarrow \mathsf{decreaseOcc}\ \delta\ \sigma\ \vec{y} \\
 & \quad\quad \mathsf{contract}\ \sigma\ \delta\ \rho\ \theta\ e \\
 & \quad \text{else}\ \rho := \rho[x \mapsto (c, \vec{y})] \\
 & \quad\quad (e', \delta, \theta) \leftarrow \mathsf{contract}\ \sigma\ \delta\ \rho\ \theta\ e \\
 & \quad\quad \text{if}\ \delta(x) = 0 \\
 & \quad\quad \text{then}\ \delta \leftarrow \mathsf{decreaseOcc}\ \delta\ \sigma\ \vec{y} \\
 & \quad\quad\quad (e', \delta, \theta) \\
 & \quad\quad \text{else}\ (\mathsf{let}\ x = \mathsf{Con}\ c\ (\sigma\vec{y})\ \mathsf{in}\ e', \delta, \theta) \\
\mid \mathsf{App}\ f\ \vec{y} & \Rightarrow \text{if}\ \theta(\sigma f) = 1 \wedge \rho\ (\sigma f) = (\vec{x},\ e) \\
 & \quad \text{then}\ \delta \leftarrow \mathsf{inlineCount}\ \delta\ \sigma\ f\ \vec{x}\ \vec{y} \\
 & \quad\quad \sigma := \sigma[\vec{x} \mapsto (\sigma\vec{y})] \\
 & \quad\quad \theta := \theta[(\sigma f) \mapsto \top] \\
 & \quad\quad \mathsf{contract}\ \sigma\ \delta\ \rho\ \theta\ e \\
 & \quad \text{else}\ (\mathsf{App}\ (\sigma f)\ (\sigma\vec{y}), \delta, \theta) \\
\mid \mathsf{let}\ x = \mathsf{Proj}_n\ y\ \mathsf{in}\ e & \Rightarrow \text{if}\ \delta(x) = 0 \\
 & \quad \text{then}\ \delta \leftarrow \mathsf{decreaseOcc}\ \delta\ \sigma\ y \\
 & \quad\quad \mathsf{contract}\ \sigma\ \delta\ \rho\ \theta\ e \\
 & \quad \text{else if}\ \rho(\sigma y) = (c, \vec{y}) \\
 & \quad\quad \text{then}\ \delta \leftarrow \mathsf{foldCount}\ \delta\ \sigma\ x\ \vec{y}_n\ y \\
 & \quad\quad\quad \sigma := \sigma[x \mapsto (\sigma\vec{y}_n)] \\
 & \quad\quad\quad \mathsf{contract}\ \sigma\ \delta\ \rho\ \theta\ e \\
 & \quad\quad \text{else}\ (e', \delta, \theta) \leftarrow \mathsf{contract}\ \sigma\ \delta\ \rho\ \theta\ e \\
 & \quad\quad\quad \text{if}\ \delta'(x) = 0 \\
 & \quad\quad\quad \text{then}\ \delta \leftarrow \mathsf{decreaseOcc}\ \delta'\ \sigma\ y \\
 & \quad\quad\quad\quad e' \\
 & \quad\quad\quad \text{else}\ (\mathsf{let}\ x = \mathsf{Proj}_n\ (\sigma y)\ \mathsf{in}\ e', \delta, \theta) \\
\mid \mathsf{match}\ v\ \mathsf{with}\ \vec{b} & \Rightarrow \text{if}\ \rho(\sigma v) = (c, \vec{y}) \wedge (c \Rightarrow e) \in \vec{b} \\
 & \quad \text{then}\ \delta \leftarrow \mathsf{caseCount}\ \delta\ \sigma\ \vec{b} \\
 & \quad\quad \delta \leftarrow \mathsf{decreaseOcc}\ \delta\ \sigma\ v \\
 & \quad\quad \mathsf{contract}\ \sigma\ \delta\ \rho\ \theta\ e \\
 & \quad \text{else}\ (\vec{b'}, \delta, \theta) \leftarrow \mathsf{contractCase}\ \sigma\ \delta\ \rho\ \theta\ \vec{b} \\
 & \quad\quad (\mathsf{match}\ (\sigma v)\ \mathsf{with}\ \vec{b'}, \delta, \theta) \\
\mid \mathsf{let}\ \vec{fd}\ \mathsf{in}\ e & \Rightarrow (\vec{fd'}, \delta, \rho') \leftarrow \mathsf{preFun}\ \sigma\ \delta\ \rho\ \vec{fd} \\
 & \quad (e', \delta, \theta) \leftarrow \mathsf{contract}\ \sigma\ \delta\ \rho'\ \theta\ e \\
 & \quad (\vec{fd''}, \delta, \theta) \leftarrow \mathsf{postFun}\ \sigma\ \delta\ \rho\ \theta\ \vec{fd'} \\
 & \quad (\mathsf{let}\ \vec{fd''}\ \mathsf{in}\ e', \delta, \theta)
\end{array}
$$

Figure 3.6: Shrink Inliner Algorithm

$\mathsf{preFun}\ \sigma\ \delta\ \rho\ \vec{fd_2} = \text{match}\ \vec{fd_2}\ \text{with}$

$$
\begin{array}{ll}
\mid [\,] & \Rightarrow ([\,], \delta, \rho) \\
\mid (f\ (\vec{x})\ =\ e_b) :: \vec{fd_3} & \Rightarrow \text{if } \delta(f) = 0 \\
& \quad \text{then } \delta \leftarrow \mathsf{decreaseCount}\ \delta\ \sigma\ e_b \\
& \qquad \mathsf{preFun}\ \sigma\ \delta\ \rho\ \vec{fd_3} \\
& \quad \text{else } (\vec{fd'_3}, \delta, \rho) \leftarrow \mathsf{preFun}\ \sigma\ \delta\ \rho\ \vec{fd_3} \\
& \qquad \rho := \rho[f \mapsto (\vec{x}, e_b)] \\
& \qquad ((f\ (\vec{x})\ =\ e_b) :: \vec{fd'_3}, \delta, \rho)
\end{array}
$$

Figure 3.7: Pre Function Inlining Algorithm

$\mathsf{postFun}\ \sigma\ \delta\ \rho\ \theta\ \vec{fd_2} = \text{match}\ \vec{fd_2}\ \text{with}$

$$
\begin{array}{ll}
\mid [\,] & \Rightarrow ([\,], \delta, \theta) \\
\mid (f\ (\vec{x})\ =\ e_b) :: \vec{fd_3} & \Rightarrow \text{if } \theta(f) \\
& \quad \text{then } \mathsf{postFun}\ \sigma\ \delta\ \rho\ \theta\ \vec{fd_3} \\
& \quad \text{else if } \delta(f) = 0 \\
& \qquad \text{then } \delta \leftarrow \mathsf{decreaseCount}\ \delta\ \sigma\ e_b \\
& \qquad\quad \mathsf{postFun}\ \sigma\ \delta\ \rho\ \theta\ \vec{fd_3} \\
& \qquad \text{else } (e'_b, \delta, \theta) \leftarrow \mathsf{contract}\ \sigma\ \delta\ \rho\ \theta\ e_b \\
& \qquad\quad (\vec{fd'_3}, \delta, \theta) \leftarrow \mathsf{postFun}\ \sigma\ \delta\ \rho\ \theta\ \vec{fd_3} \\
& \qquad\quad ((f\ (\vec{x})\ =\ e'_b) :: \vec{fd'_3}, \delta, \theta)
\end{array}
$$

Figure 3.8: Post Function Inlining Algorithm

$\mathsf{contractCase}\ \sigma\ \delta\ \rho\ \theta\ \vec{b_2} = \text{match}\ \vec{b_2}\ \text{with}$

$$
\begin{array}{ll}
\mid [\,] & \Rightarrow ([\,], \delta, \theta) \\
\mid (c \Rightarrow e_b) :: \vec{b_3} & \Rightarrow (e'_b, \delta, \theta) \leftarrow \mathsf{contract}\ \sigma\ \delta\ \rho\ \theta\ e_b \\
& \quad (\vec{b'_3}, \delta, \theta) \leftarrow \mathsf{contractCase}\ \sigma\ \delta\ \rho\ \theta\ \vec{b_3} \\
& \quad ((c \Rightarrow e'_b) :: \vec{b'_3}, \delta, \theta)
\end{array}
$$

$\mathsf{caseCount}\ \delta\ \sigma\ \vec{b} = \text{match}\ \vec{b}\ \text{with}$

$$
\begin{array}{ll}
\mid [\,] & \Rightarrow \delta \\
\mid (c' \Rightarrow e) :: \vec{b} & \Rightarrow \text{if } c = c' \\
& \quad \text{then } \mathsf{decreaseCount}\ \delta\ \sigma\ (\mathsf{snd}\ \vec{b}) \\
& \quad \text{else } \delta \leftarrow \mathsf{decreaseCount}\ \delta\ \sigma\ e \\
& \qquad \mathsf{caseCount}\ \delta\ \sigma\ \vec{b}
\end{array}
$$

Figure 3.9: Case Algorithm

$$
\begin{aligned}
(\sigma C)[\![\sigma(\mathsf{let}\ x = \mathsf{Proj}_2\ y\ \mathsf{in}\ e')]\!] &= (\sigma C)[\![\mathsf{let}\ x = \mathsf{Proj}_2\ (\sigma y)\ \mathsf{in}\ (\sigma e')]\!] && \text{by definition} \\
&= (\sigma C)[\![(x \mapsto (\sigma y_2))\ (\sigma e')]\!] && \text{by } \mathsf{fold\_proj}(\rightharpoonup) \\
&= (\sigma C)[\![(\sigma[x \mapsto (\sigma y_2)])\ e']\!] && \text{by } \sigma\mathsf{fuse} \\
&= (\sigma[x \mapsto (\sigma y_2)])\ C[\![(\sigma[x \mapsto (\sigma y_2)])\ e']\!] && \text{by } \sigma\mathsf{weaken}
\end{aligned}
$$

Figure 3.10: Example of substitution fusion

$$
\frac{\begin{array}{c} g \notin m \qquad g \notin \mathrm{vars}(e) \\ k \notin \mathrm{vars}(e) \qquad \mathrm{length}(\vec{x}) = \mathrm{length}(\vec{x}') \qquad \mathrm{length}(\vec{y}) = \mathrm{length}(\vec{y}') \\ \mathrm{vars}(\vec{x}') \cap s = \mathrm{vars}(\vec{y}') \cap s = \mathrm{vars}(\vec{x}') \cap \mathrm{vars}(\vec{y}') = \varnothing \\ \mathrm{NoDup}(\vec{x}') \qquad \mathrm{NoDup}(\vec{y}') \qquad f' \notin s \cup \mathrm{vars}(\vec{x}') \cup \mathrm{vars}(\vec{y}') \end{array}}{\begin{array}{c} ((f(k :: \vec{x}) = \mathrm{let}\ g(\vec{y}) = e\ \mathrm{in\ App}\ k\ [g]) :: \vec{b},\ s,\ m) \\ \rightsquigarrow_{\mathrm{fun}} (f(k :: \vec{x}') = \mathrm{let}\ g(\vec{y}') = \mathrm{App}\ f'\ (\vec{y}' +\!\!+ \vec{x}')\ \mathrm{in\ App}\ k\ [g]\ \mathrm{and}\ f'(\vec{y} +\!\!+ \vec{x}) = e :: \vec{b}, \\ s \cup \mathrm{vars}(\vec{y}') \cup \mathrm{vars}(\vec{x}') \cup \{f'\},\ m \cup \{g\}) \end{array}}\ \textsc{bundle-curried}
$$

Figure 3.11: Rewrite rule for one step of uncurrying, from Li [32]

$\mathsf{inline}_n\ \rho\ \sigma\ \theta\ e = \text{if } n = 0 \text{ then } (\sigma\ e, \theta) \text{ else match } e \text{ with}$

$$
\begin{array}{ll}
\mid \mathsf{halt}\ x & \Rightarrow \mathsf{halt}\ (\sigma x) \\
\mid \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ e & \Rightarrow e' \leftarrow \mathsf{inline}_n\ \rho\ \sigma\ \theta\ e \\
 & \quad \mathsf{let}\ x = \mathsf{Con}\ c\ (\sigma\vec{y})\ \mathsf{in}\ e' \\
\mid \mathsf{App}\ f\ \vec{y} & \Rightarrow (inl, \theta) \leftarrow \mathsf{updateApp}\ \theta\ (\mathsf{App}\ (\sigma f)\ (\sigma\vec{y})) \\
 & \quad \text{if } inl \\
 & \quad \text{then } (\vec{x}, e) \leftarrow \rho(\rho\ f) \\
 & \qquad e' \leftarrow \mathsf{freshen}(e) \\
 & \qquad \sigma := \sigma[\vec{x} \mapsto (\sigma\vec{y})] \\
 & \qquad \mathsf{inline}_{n-1}\ \rho\ \sigma\ \theta\ e' \\
 & \quad \text{else } \mathsf{App}\ (\sigma f)\ (\sigma\vec{y}) \\
\mid \mathsf{let}\ x = \mathsf{Prim}\ p\ \vec{y}\ \mathsf{in}\ e & \Rightarrow e' \leftarrow \mathsf{inline}_n\ \rho\ \sigma\ \theta\ e \\
 & \quad \mathsf{let}\ x = \mathsf{Prim}\ p\ (\sigma\vec{y})\ \mathsf{in}\ e' \\
\mid \mathsf{let}\ x = \mathsf{Proj}_n\ y\ \mathsf{in}\ e & \Rightarrow e' \leftarrow \mathsf{inline}_n\ \rho\ \sigma\ \theta\ e \\
 & \quad \mathsf{let}\ x = \mathsf{Proj}_n\ (\sigma y)\ \mathsf{in}\ e' \\
\mid \mathsf{match}\ v\ \mathsf{with}\ \vec{b} & \Rightarrow \vec{b'} \leftarrow \mathsf{inlineCase}_n\ \rho\ \sigma\ \theta\ \vec{b} \\
 & \quad \mathsf{match}\ (\sigma v)\ \mathsf{with}\ \vec{b'} \\
\mid \mathsf{let}\ \vec{fd}\ \mathsf{in}\ e & \Rightarrow \rho' \leftarrow \rho, \vec{fd} \\
 & \quad (\theta_1, \theta_2) \leftarrow \mathsf{updateFunDef}\ \theta\ (\mathsf{let}\ \vec{fd}\ \mathsf{in}\ e, \sigma) \\
 & \quad e' \leftarrow \mathsf{inline}_n\ \rho'\ \sigma\ \theta_1\ e \\
 & \quad \vec{fd'} \leftarrow \mathsf{inlineFun}_n\ \rho'\ \sigma\ \theta_2\ \vec{fd} \\
 & \quad \mathsf{let}\ \vec{fd'}\ \mathsf{in}\ e'
\end{array}
$$

$\mathsf{inlineFun}_n\ \rho\ \sigma\ \theta\ \vec{fd} = \text{match } \vec{fd} \text{ with}$

$$
\begin{array}{ll}
\mid [\,] & \Rightarrow [\,] \\
\mid (f\ (\vec{x})\ =\ e_b) :: \vec{fd_3} & \Rightarrow \vec{fd'_3} \leftarrow \mathsf{inlineFun}_n\ \rho\ \sigma\ \theta\ \vec{fd_3} \\
 & \quad \theta \leftarrow \mathsf{updateInFun}\ \theta\ (f\ (\vec{x})\ =\ e_b) \\
 & \quad e'_b \leftarrow \mathsf{inline}_n\ \rho\ \sigma\ \theta\ e_b \\
 & \quad (f\ (\vec{x})\ =\ e'_b) :: \vec{fd'_3}
\end{array}
$$

Figure 3.12: Function Inlining Algorithm

Chapter 4

Code Generation

The last step of compilation in **CertiCoq** generates CompCert **Clight** from the CPS intermediate representation discussed in Chapter 3. While the two languages follow different paradigms, with L6 being a pure, functional language and **Clight** an impure, imperative one, restrictions over L6, both syntactic and in terms of additional properties enforced, facilitate the generation of equivalent **Clight** programs. In this chapter, we discuss our approach to code generation, describing how we represent each component of L6 in **Clight**, how the runtime system interacts with our generated code, and finally how we prove correct the code generation phase of **CertiCoq** with a clean interface to garbage collection.

4.1 Generating C from a functional language

At this point of compilation, branches of case-constructs have been bound as functions, and every L6 function is in continuation-passing style, closed and lambda-lifted as part of the same mutually recursive function bundle.

As discussed in Kelsey [27] and in Appel [3], functional programming, and in particular functional languages in continuation-passing style, is directly related to static single-assignment (SSA), an intermediate representation for imperative language. We

use this correspondence to generate **Clight** as an intermediate representation between **CertiCoq** and CompCert.

Extended basic blocks of the original program are represented as functions in $L6$. Because these functions have been closure-converted and lambda-lifted, they correspond to first-class, global functions in **Clight**. Closure-conversion and continuation-passing style simplify memory management, allowing us to easily identify the live variables at function entry.

Finally, continuation-passing style coupled with tail call optimization in C compilers allows us to directly generate C functions from L6 functions. Previous functional compilers targeting C (such as sml2c [56]) relied on analysis to determine the right way to handle calls. Generating C functions from L6 functions results in a simpler code generator and exposes the structure of programs to optimizations in the C compiler.

4.1.1 Representing datatypes in C

We represent values on the heap using a representation used by many ML compilers including **OCaml** [30] and **SML/NJ**[37], leaving nullary constructors unboxed and boxing non-nullary ones. To differentiate unboxed values from pointers to a boxed values, we use the last bit of unboxed values as a flag set to 1 – meanwhile, the pointers used for boxed values are word aligned, always ending in 0.[1]

For an inductive type T, defined as shown in Figure 4.1, we assign (unboxed) ordinal 0 to A and 1 to C, and (boxed) ordinal 0 to B and 1 to D.

Unboxed values are kept in local memory as the integer $2 \times \mathsf{ordinal} + 1$.

Boxed values of arity n are represented as a pointer to the second of $n+1$ contiguous memory location, each of the size of a value, providing access to the representation

[1] In this chapter, we assume a 64-bit architecture, with 8-byte pointers. However, **CertiCoq** works in 32 and in 64-bit mode, and its proof is parameterized over the size of pointers.

```
datatype T : Type
| A : T
| B : R -> T
| C : T
| D : R -> S -> T
```

Figure 4.1: Example of a Coq datatype

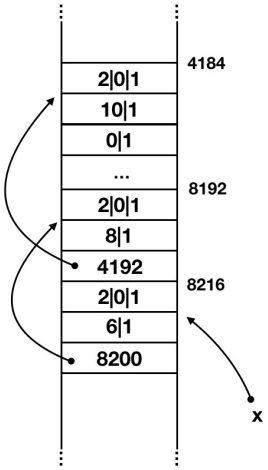
of each of its fields, while the first location holds a header containing the ordinal and the arity of the constructor.

```
Cons 6 (Cons 8 (Cons 10 Nil)))
```

(a) A list in Coq

```
(* Where 5 -> Nil and 6 -> Cons *)
Vconstr 6 [Vint 6;
  Vconstr 6 [Vint 8;
    Vconstr 6 [Vint 10;
      Vconstr 5 []]]]
```

(b) A list in L6



(c) A list in L7

Figure 4.2: Example of an Inductive Value throughout the compilation process

We include in Figure 4.2 an example of the same inductive value represented in different languages during compilation. First, as shown in Figure 4.2a, the list `[6;8;10]` can be represented in an inductive datatype `list` with constructor `Nil: list` and `Cons:int -> list -> list`. We show how this value would be represented in L6 in Figure 4.2b. In L6, information about the constructors is kept in a global map – we assume here that 5 refers to `Nil` and 6 to `Cons`. Finally, as shown in

Figure 4.2c, in **Clight**, non-nullary constructors are represented in the heap as boxed values. We use the notation $n|m$ to represent concatenating the bit representation of n with the one of m. Here, the list is represented as three blocks of three values each. The head of the list is pointed to by x at address 8224. The value before that, $2|0|1$, is the header for `Cons`, which is the first boxed constructor of `list` and has two arguments (the zero in the middle represents bits reserved for garbage compilation). The first field of the head contains the unboxed value 6, while the second contains a pointer to the next element of the list, at memory location 8200. This value has the same header, also representing `Cons`, and hold unboxed value 8 in its first field, and a pointer to memory location 4192 in its second. 4192 holds the last link in our list, with a representation of Nil (first unboxed constructor of `list`) in its second argument.

4.1.2 Garbage Collection

Because we target **Clight**, which has manual memory management, we need to provide a runtime memory management system to reclaim unused memory. Inspired by the SML-NJ compiler, **CertiCoq** uses heap-allocated data structures and closures, and garbage collection to efficiently collect dead portions of the heap. Garbage collection is a runtime mechanism to identify data in memory that are no longer needed by the code, and reuse (or deallocate) their memory footprint. Different garbage collection techniques exist, with varying complexity and cost. **CertiCoq** provides a general interface that allows using a wide variety of collectors.

Garbage collection is a hard problem that has been the subject of numerous publications. Diwan et al. [20] presented a list of challenges encountered when garbage collecting statically typed languages:

1. Garbage collection "must be able to determine if an object is reachable from other live objects or from the roots".

2. Garbage collection "must be able to find all pointers to a given object so that they may be updated when the object is moved".

McCreight et al. [39] and Dargaye [18] concurrently presented a way to handle those challenges, making use of a "shadow stack" (as introduced by Henderson [26]) to store values across calls. Their intermediate language includes primitives to explicitly keep track of roots, guiding the translation to a Clight program interfacing with a garbage collector. However, in both cases, measured overhead for the generated code was high due to the expensive operations that had to be performed over any local variables that could contain a live pointer.

In our case, garbage collection is facilitated by the design of L6, the source of our code generation phase:

1. Since L6 is in CPS and closure-converted, and we perform garbage collection on function entry, live roots are exactly the arguments to the function.

2. Since Gallina, and all our intermediate languages including L6, are pure, newly allocated objects can only refer to old ones. This is because objects, in pure languages, are immutable. This greatly simplifies the implementation of generational garbage collection – when tracing the live objects, we don't have to traverse older generations.

In the rest of this section, we present the interface we have formalized between code generation and garbage collection, and the generational garbage collector provided with **CertiCoq** which has been verified and respects the provided interface.

4.1.3 Abstract state for the generated code

We define an abstract state that we target when translating from L6, and later instantiate it as concrete **Clight** memory and local environment.

The abstract state, shown in Figure 4.3, has three components:

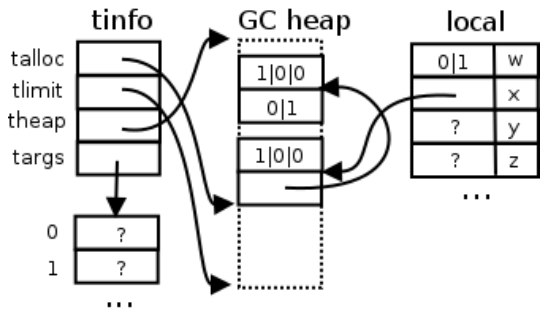


Figure 4.3: Abstract state for the generated code

1. tinfo, a structure containing three pointers representing the state of `GC heap` and a fourth pointer to an array containing either pointers to `GC heap` or unboxed values.

2. `GC heap`, an abstract representation of a heap containing boxed values at non-overlapping virtual addresses.

3. `local`, a table containing mappings from variables to either a pointer to the `GC heap` or an unboxed value.

The example in Figure 4.3 shows a local environment containing the mapping (w, O), with O represented as the first unboxed constructor of nat, and (x, SSO), with SSO represented as a pointer to a pair of words containing the header (S has arity 1 and is the first boxed constructor), and a pointer to a second pair of words containing the same header, and the representation of O, as discussed in Section 4.1.1.

tinfo contains the state of the `GC heap` and of the argument array using 4 cells:

1. talloc, showing where in the `GC heap` to allocate the next boxed value.

2. tlimit, showing where the `GC heap` ends.

3. theap, showing where the `GC heap` starts.

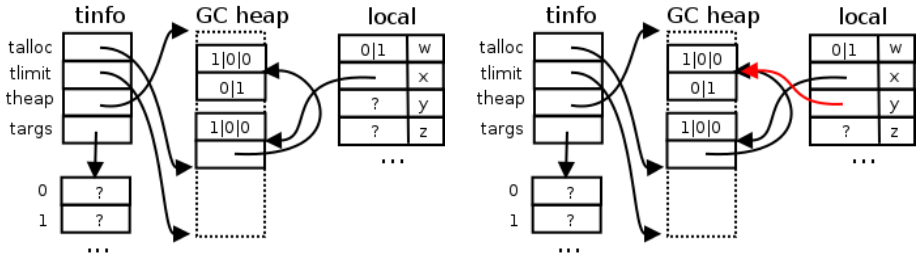
4. targs, an array of values containing the arguments to the current function

Throughout the next section, we would show the effect of each L6 operations on the abstract state, before instantiating the state in **Clight** and generating corresponding **Clight** statements.

4.1.4 Simulating L6 in the Abstract State

Before describing our code generation algorithm, we give a sketch of the effect of the L6 operations on the abstract state presented in Section 4.1.3.

`Eproj` "let $y = \mathsf{Proj}_0\ x$ in e" executes, according to rule E_PROJ, by setting y to the first projection of x (which has been constructed using S) before proceeding with executing e. We show in Figure 4.4 the effect of executing this expression starting from the abstract state from Figure 4.3. In this case, where x represents SSO, y ends up pointing to the representation of SO.

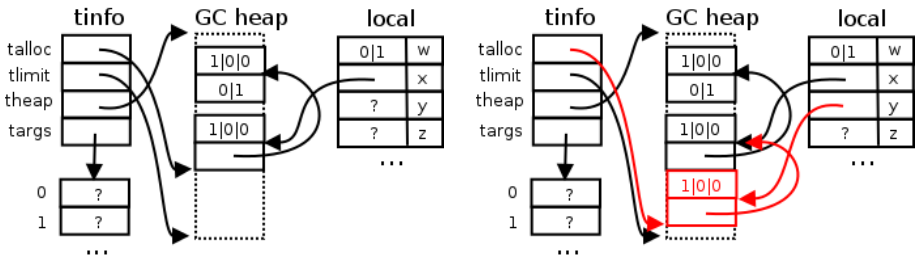


(a) Before (b) After

Figure 4.4: The effect of "let $y = \mathsf{Proj}_0\ x$ in e" on the abstract state

`Econstr` "let $y = \mathsf{Con}\ S\ x$ in e", according to Rule E_CONSTR, has y set to be a value constructed by applying S to the value of x. Figure 4.5 shows the effect of this on the abstract state. In Figure 4.5b, y points to a representation of $SSSO$.

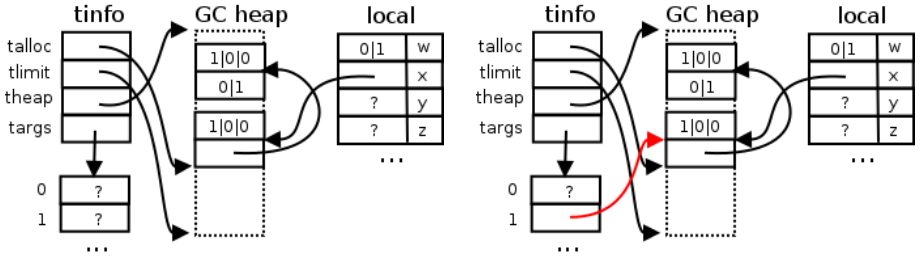
`Ecase` "match x with $\vec{b}$" has no effect on the abstract state.



(a) Before (b) After

Figure 4.5: The effect of "let $y = \mathsf{Con}\ S\ x\ \mathsf{in}\ e$" on the abstract state

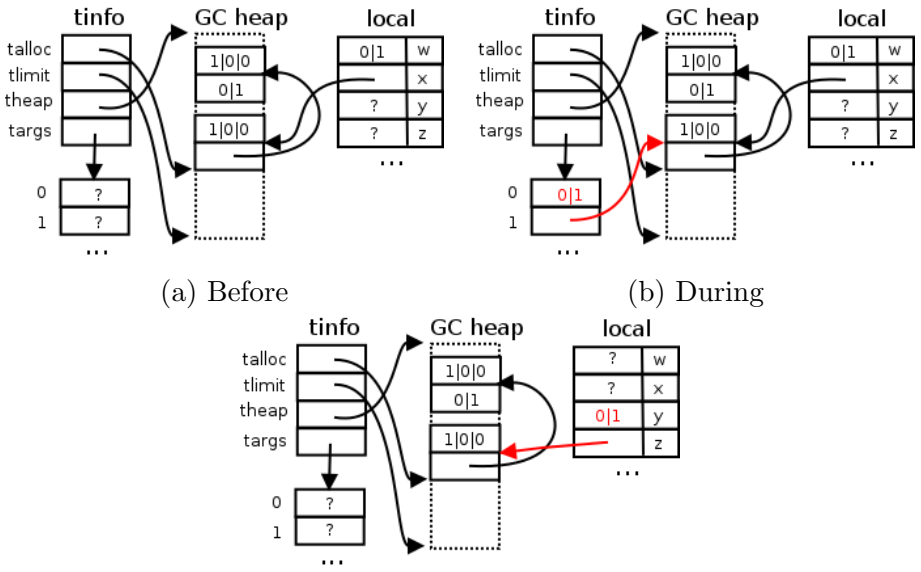
`Ehalt` "halt x", according to E_HALT, evaluates to the value of x. By convention, we take the second field of the argument array to hold the return value. As shown in Figure 4.6, this results in targs_1 representing SSO.



(a) Before (b) After

Figure 4.6: The effect of "halt x" on the abstract state

`Eapp` "App $f\ w\ x$" calls function $f\ y\ z = e$ on arguments w and x using the calling convention shared by all functions with tag t (see E_APP). Figure 4.7 shows the effect of evaluating this expression on the abstract state, assuming that t corresponds to a calling convention where the first argument is kept in cell 0 and the second in cell 1 of the argument array. As shown in Figure 4.7, this is done in two steps: First, the arguments w and x are copied to targs according to t, which results in targs_0 representing O and targs_1 representing SSO (see Figure 4.7b). Then, as shown in Figure 4.7c, we restore targs_0 and targs_1 to the function parameters y and z before proceeding with the execution of e.



(a) Before (b) During

(c) After

Figure 4.7: The effect of "App f w x" on the abstract state

4.1.5 From abstract state to Clight memory

In this section, we present how we realize the abstract state described in Section 4.1.3 in **Clight**, using a **Clight** memory and a local environment.

Figure 4.8 shows how we map different portion of the abstract state and of the program to disjoint portions of memory.

First, an area of the heap holds, for every function f in the source program, function information finfo_f and function code fcode_f.

Then, a disjoint area holds tinfo and the targs array.

Finally, a third area holds the boxed values which were contained in the `GC heap` from Figure 4.3.

4.1.6 The Interface with Garbage Collection

Rather than integrating a specific garbage collector with our generated code, and proving our code generation phase correct with respect to that particular garbage

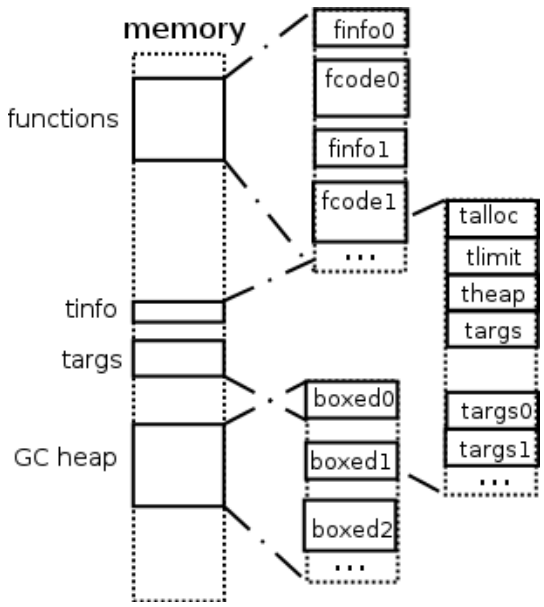


Figure 4.8: Mapping of the abstract state in a **Clight** memory

collector, we provide a general interface for garbage collection, and prove our code generation phase correct with respect to a more general notion of garbage collection.

As shown in Figure 4.8, tinfo is a **Clight** structure containing, in addition to the argument array, three pointers describing the state of the garbage-collected portion of memory:

1. talloc, a pointer to the next allocatable word of memory in the current block
2. tlimit, a pointer to the end of the allocatable portion of the current block
3. theap, a pointer to the garbage collector's own description of its memory regions; the format of this data is left abstract to the **CertiCoq** compiler

Our interface works by using a global [2] array targs containing all live roots at the point garbage collection is called. In addition to pointers to the garbage-collected area, this array may contained unboxed values and pointers to non-garbage-collected area, both of which are to be ignored by garbage collection.

[2]Actually, one such array per thread; tinfo stands for "thread info". The current **CertiCoq** runtime system is not multithreaded, but the interface between code generator and runtime system (that we describe in this section) permits multithreading.

Our interface makes the following assumptions (i.e., a garbage collector can expect these things to hold):

1. Any live portions of `GC heap` are reachable from the live roots in targs– which is to say, since the live roots correspond to the environment computed by closure conversion, functions are fully closed after closure conversion.

2. No request for allocation is made for more space than what garbage collection can provide. In our case, this means that no function allocates more than what garbage collection can provide before calling its continuation. In principle, we can represent functions allocating up to $2^{64} - 1$ in finfo. However, the garbage collector may further limit the maximum number of blocks allocated between collection – the generational collector included in **CertiCoq** limits it to the size of the nursery, which is 65536 bytes by default. We parameterize **CertiCoq** with the same constant, and refuse to compile functions which would allocate more than this amount.[3]

We note that while both of these assumptions hold on calls to the garbage collection at every function entry due to all calls being tail calls to closed functions, resulting in the roots and maximum number of blocks allocated being easily computable at compile time, the interface would work in a more general setting by, for example, testing if garbage collection is needed before every allocation.

On return, the code generator knows that

1. The argument array contains **Clight** values representing the same L6 values as before.

2. At least as much allocatable memory as requested is available between talloc and tlimit.

[3]Of course, it is still possible to run out of memory if the program's live data exceed the available space. Our interface with garbage collection assumes that this does not happen.

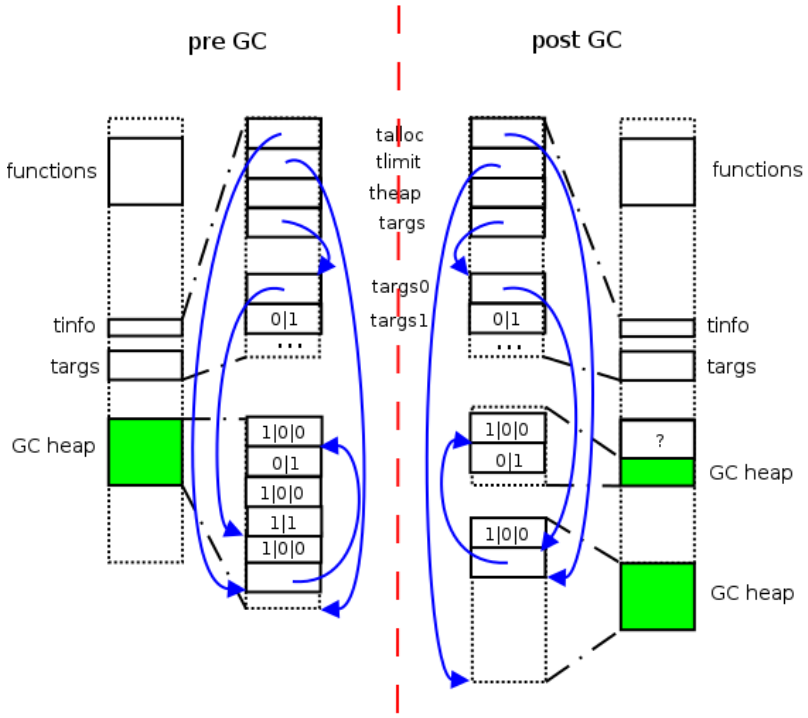


Figure 4.9: The state of the heap before and after garbage collection

We include in Figure 4.9 a representation of the heap before and after garbage collection.

On the left, the heap contains an area with function declarations and finfo, followed by tinfo and the argument array targs and finally the GC heap, whose area is represented in green. In the blown up view of tinfo, talloc points to the start of the free area in the GC heap, tlimit to the end. We do not represent the value of theap in the picture – it is used by the garbage collector to keep track of its memory regions, and left abstract to our compiler. Finally, targs points to the start of the argument array. In the argument array, the first slot is taken by a pointer to the representation of SSO in the GC heap. The second slot is taken by a representation of O.

On the right, representing the state of the heap after garbage collection (assuming the roots were 0 and 1), the areas are unchanged, except for the GC heap – the new GC heap may only contain part of the old GC heap or newly allocated portions of

memory. The pointers in talloc and targs have been updated to reflect the new GC heap.

4.2 Code Generation

We will now describe our code generation algorithm, generating **Clight** statements from a program in L6. An earlier version of this algorithm was implemented by Matthew Weaver and Andrew Appel.

After closure conversion and lambda-lifting, all functions are part of the same mutually recursive bundle $\vec{fd}$, and we know that the body of the program `e` does not contain function declarations. Moreover, we know that for any function `(f,` $\vec{y}$`, e')` in $\vec{fd}$, `e'` does not contain function declarations.

The back end processes let $\vec{fd}$ in e by

1. computing the arity (number of function-parameters) and the maximum number of words allocated by every function in the program, and generating a map $\theta : var \to \mathbb{N} \times \mathbb{N}$ with this information.

2. generating forward declarations for all functions in $\vec{fd}$, as they may represent mutually recursive functions in Coq (and in C) and as such need to call each other.

3. For each function in $\vec{fd}$, `(f,` $\vec{y}$`, e')`, creating a **Clight** function `f` with `tinfo` as argument and codegen`(e')` as function body

4. generating a **Clight** function `body`, converting e using codegen.

4.2.1 Code generation for L6 functions

We represent L6 functions as **Clight** functions, taking as single parameter a structure tinfo holding the information needed to execute the function.

tinfo is a **Clight** structure containing the information necessary to interpret the memory as the current state of evaluation. It includes the information needed to allocate new values on the heap and a pointer to an array containing the arguments to the function. As shown in Figure 4.8, tinfo contains four pointers. The first two, talloc and tlimit, point to, respectively, the next available block of available memory, and the last available block, and they are used both to allocate new values on the heap and to determine when is time to garbage collect. The third one points to a representation of the regions of memory containing the GC heap. The fourth one points to an array of values used for arguments during function calls.

Every function f is associated with a structure $\mathsf{finfo}_\mathsf{f}$ containing the maximum number of words the function could allocate, followed by its arity, and, for each of its arguments, the slots used in the argument array. Figure 4.10 provides an example of such structure for a function with three arguments held in position 0, 1 and 2 of the arguments array and allocating at most ten words:

```
value const f_info_1000000100[5] = { 10, 3, 0, 1, 2, };
```

Figure 4.10: Example of the structure containing information about a function

On function entry, we verify if the maximum number of memory words that could be allocated by the function is more than the difference between tlimit and talloc. If this is the case, we need to run the garbage collector (described in 4.1.2) before restoring the arguments to local memory, and proceeding with the body of the function.

4.2.2 The code generation algorithm

codegen, shown in Figure 4.12, is a function constructing a **Clight** statement from a L6 expression. By design of L6, most of the mapping between L6 expressions and statement is direct with, for example, projections mapping directly to field access.

```
void f(struct thread_info *tinfo){

// declarations
...

alloc = ;
limit = (*tinfo).limit;
args = (*tinfo).args;

if (!(*f_code_info <= tinfo->limit - tinfo->alloc)) {
  (garbage_collect)(f_info, tinfo);
}
alloc = (*tinfo).alloc;
limit = (*tinfo).limit;
args = (*tinfo).args;

x = *(args + 0);
y = *(args + 1);
z = *(args + 2);

// function body
...
}
```

Figure 4.11: Example of the start of a function generated by our back end

Others, such as function application, are a bit more involved. In the rest of this section, we describe how we represent L6 expressions as **Clight** statements.

In addition to e, the L6 term being converted, function codegen takes in:

1. Δ, mapping constructor tags to the name, arity and ordinal of the constructor and to the name and tag of its datatype.

2. θ, mapping the name of each function to its arity, calling-convention, and the maximum number of values its body could allocate. Knowing which variable represents a function is also important to determine if a variable represents a value in local memory, or if it represents the address of a function in the heap.

We will now go through each cases for L6 expression e, describing the correspond, through the abstract machine described in Section 4.1.4, with the **Clight** statements generated by codegen.

`Ehalt` Every trace of L6 ends with an application of the halt evaluation rule due to the occurrence of `Ehalt x`. The **Clight** equivalent is to place the value referred to as x

$\mathsf{codegen}_{\Delta,\theta}\ e = \text{match } e \text{ with}$

$$
\begin{array}{ll}
\mid \mathsf{halt}\ x & \Rightarrow \mathsf{Sassign}\ (\mathsf{args}[1])\ (\mathsf{VarOrFunVar}_\theta x);\ \mathsf{Sreturn}\ 1 \\
\mid \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ e & \Rightarrow \text{if } \mathsf{arr}_\Delta\ (c) = 0 \\
 & \quad \text{then } \mathsf{Sset}\ x\ ((\mathsf{ord}_\Delta\ c \ll 1) + 1);\ \mathsf{codegen}\ e \\
 & \quad \text{else } s_1 \leftarrow \mathsf{Sset}\ x\ \mathsf{talloc};\ \mathsf{Sassign}\ \mathsf{talloc}\ ((\mathsf{arr}_\Delta\ c \ll 8) + \mathsf{ord}_\Delta\ c) \\
 & \qquad s_2 \leftarrow \mathsf{Sassign}\ x[0]\ (\mathsf{VarOrFunVar}_\theta(y_0));\ \ldots\ ; \\
 & \qquad\quad \mathsf{Sassign}\ x[\mathsf{ord}_\Delta\ (c) - 1]\ (\mathsf{VarOrFunVar}_\theta(y_{\mathsf{ord}_\Delta\ (c)-1})) \\
 & \qquad s_1;\ s_2;\ \mathsf{Sset}\ \mathsf{talloc}\ (\mathsf{talloc} + \mathsf{ord}_\Delta\ c + 1);\ \mathsf{codegen}\ e \\
\mid \mathsf{let}\ x = \mathsf{Proj}_n\ y\ \mathsf{in}\ e & \Rightarrow \mathsf{Sset}\ x\ (y[n]);\ \mathsf{codegen}\ e \\
\mid \mathsf{match}\ x\ \mathsf{with}\ \vec{b} & \Rightarrow (\vec{b_1}, \vec{b_2}) \leftarrow \mathsf{split\ boxed?}\ \vec{b} \\
 & \quad \vec{l_1} \leftarrow \mathsf{codegenBranches}_{(\Delta,\theta)} \vec{b_1} \\
 & \quad s_1 \leftarrow \mathsf{Sswitch}\ (x[-1]\ \&\ 255)\ \vec{l_1} \\
 & \quad \vec{l_2} \leftarrow \mathsf{codegenBranches}_{(\Delta,\theta)} \vec{b_2} \\
 & \quad s_2 \leftarrow \mathsf{Sswitch}\ (x \gg 1)\ \vec{l_2} \\
 & \quad \mathsf{Sifthenelse}\ (\mathsf{isptr}\ x)\ s_1\ s_2 \\
\mid \mathsf{App}\ f\ \vec{y} & \Rightarrow s \leftarrow \mathsf{Sassign}\ \mathsf{targs}[\mathsf{finfo}_f[2]]\ y_0;\ \ldots\ ; \\
 & \qquad\quad \mathsf{Sassign}\ \mathsf{targs}[\mathsf{finfo}_f[\mathsf{arr}_\theta f + 2]]\ y_{\mathsf{arr}_\theta f} \\
 & \quad s;\ \mathsf{Scall}\ f\ \mathsf{tinfo}
\end{array}
$$

Figure 4.12: Code Generation Algorithm

$\mathsf{codegenBranches}_{(\Delta,\theta)} \vec{b} = \text{match } \vec{b} \text{ with}$

$$
\begin{array}{ll}
\mid [\,] & \Rightarrow \mathsf{LSnil} \\
\mid (c \Rightarrow e) :: \vec{b} & \Rightarrow \mathsf{LScons}\ (\mathsf{ord}_\Delta\ c)\ (\mathsf{codegen}_{\Delta,\theta}\ e;\ \mathsf{Sbreak}\ )\ (\mathsf{codegenBranches}\ \vec{b})
\end{array}
$$

Figure 4.13: Case Algorithm for Code Generation

in a fixed place in memory to make it available to the caller of the procedure. By convention, we place x in the second field of the argument array. Function $\mathsf{VarOrFunVar}_\theta$ returns x if x is a pointer to a function (in the functions area of memory), and $\mathsf{local}(x)$ otherwise.

`Econstr` The restriction that the arguments of constructor bindings be variables rather than expressions greatly simplifies the code generation for `Econstr`. Each of these variables represents a **Clight** value, either boxed (represented by the **Clight** value Vptr) or unboxed (either Vint or Vlong depending on the architecture).

In the case of nullary constructors, variable x is set to be the representation of c. For the example in Figure 4.14, c is the first unboxed constructor of its datatype, so its representation is $O|1$, and we set x to be 1.

```
...
x = 1;
...
```

Figure 4.14: Example of the code generated for unboxed constructors

In all other cases, we place the value on the heap, at the next available space in the GC heap, as indicated by the talloc pointer. Our boxed representation of constructors consists of a header field followed by value fields containing either an unboxed value, a pointer to a function or a pointer to another boxed value, corresponding to $\vec{y}$. We then set x to point to this assigned representation before updating the tallocpointer by $(1 + |\vec{y}|) * 8$.

```
  ...
x = alloc;
alloc = alloc + 3;
*((value *) x - 1) = 2048;
*((value *) x + 0) = y;
*((value *) x + 1) = z;
...
```

Figure 4.15: Example of the code generated for the allocation of boxed constructors

`Eproj` Projections in L6 directly correspond to fields in **Clight**. Since projections could only be taken of non-nullary constructors, then y stands in for a boxed value, represented in **Clight** as a Vptr. Since each of the fields of this constructor is 8 bytes wide, we can access the nth field by adding $n \times 8$ to the address of the pointer y, which is precisely $y[n]$ – this is shown, with $n = 1$, in Figure 4.18.

`Ecase` In L6, pattern-matching is separated into case-switching and projections (see Section 2.6). As shown in the code generation algorithm in Figure 4.12 case-switching

```
...
x = *((value *) y + 1)
...
```

Figure 4.16: Example of the code generated for a projection

match x with $\vec{b}$ corresponds closely to a switch statement in **Clight**. As nullary and non-nullary constructors are encoded differently (as boxed or unboxed **Clight** values), the encoding of case-switching in **Clight** first splits the branches into those targeted by unboxed constructors $\vec{b_1}$ and those targeted by boxed ones $\vec{b_2}$.

We use external function VptrOrVint to determine if v is a pointer or an integer value. We include the implementation of VptrOrVint in Figure 4.17. VptrOrVint returns true if the last bit of the value is 0, and false otherwise (as pointers are word aligned, while we represent integers and unboxed values x are as $|2 \times x + 1|$). [4]

We then generate branches for boxed and unboxed constructors using auxiliary function codegenBranches $\vec{b_i}$, included in Figure 4.13. Each branch pairs the ordinal of the constructor with the body of the branch as converted by our code generation algorithm. The resulting cases are embedded in two switch statement observing the ordinal of x depending on if it is boxed or not.

In Figure 4.18, we include an example of code generated from a case-construct on a value `n` of inductive type `T` shown in Figure 4.1. We recover the ordinal associated with the constructor, in the header or in the payload for boxed and unboxed values respectively, before using a **Clight** switch on that ordinal.

As explained in Section 4.1.1, `T` has two boxed constructors, `B` with ordinal 0, and `D` with ordinal 1. We can expect the first two statements in the `B` branch to be of the form `x = *((value *) n + 0); y = *((value *) n + 1)` as described in the translation of projections. We use default to catch the last ordinal of each

[4] Testing a pointer value for whether it is an odd number is technically undefined in the C11 standard, and in the current CompCert operational semantics. We rely on a refinement of the CompCert semantics, axiomatized to handle such a test.

switch; we generate a switch with a single default statement when there is a single boxed or unboxed constructor, but expect the C compiler to subsequently optimize the comparison away.

```
#define Is_ptr(x)  (((x) & 1) == 0)
```

Figure 4.17: The implementation of `Is_ptr`

```
...
if ((Is_ptr)((value) n)) {
  switch (*((value *) n - 1) & 255) {
    0:       // B
    ...
    break;

    default: // D
    x = *((value *) n + 0);
    y = *((value *) n + 1);
    ...

  }
} else {
  switch (n >> 1) {
    0:       // A
    ...
    break;

    default: // C
    ...
  }
}
```

Figure 4.18: Example of the code generated for a case statement

`Efun` Since our initial code is fully closure-converted and lambda-lifted, we cannot encounter `Efun` during the computation of `translate_body`. We describe in Section 4.2.1 how closure-converted and fully hoisted functions are represented as **Clight** top-level functions.

`Eapp` As shown in Figure 4.7, the code generated for "App f $\vec{y}$" needs to store according to the calling convention represented by t the arguments $\vec{y}$ passed to the function in the argument array targs, before calling f. Then, the code generated for

the function, described in Section 4.2.1, potentially calling the garbage collector if executing the body of f allocated more than what is available in the `GC heap` before restoring the arguments to f's local environment.

4.3 The Proof of Correctness of Code Generation

We prove the correctness of the **CertiCoq** backend using a forward simulation proof over the operational semantics of L6 and of CompCert **Clight**.

The main theorem , included in Figure 4.19, states that if program P compiles to statements stmt, and P evaluates to value v, then a starting state with stmt multisteps to a skip state where the value held in memory at location targs_1 is related to v.

Theorem 4.3.1.

$$\begin{array}{l} \cdot \vdash P \Downarrow v \wedge \\ \mathsf{codegen}_{\Delta,\theta}\ P = s_P \wedge \\ \mathsf{init}(m) \implies \end{array} \quad \begin{array}{l} \exists m', \\ \cdot, \cdot \vdash s_P, m \Rightarrow^{\epsilon} \cdot, m' \wedge \\ (\cdot, v) \simeq^{\mathrm{val}}_{\theta,m'} \mathit{targs}_1. \end{array}$$

Figure 4.19: Statement of correctness for code generation

In order to prove this theorem, we need to prove a more general statement where e is a subterm of P being evaluated in evaluation environment ρ. The generalized statement of correctness[5] is given in Figure 4.20. The proof goes over the derivation of big-step evaluation for L6. For every rules that could be used to evaluate a term, we show that there exist stepping rules in the **Clight** evaluation semantics to evaluate, in corresponding environments, the **Clight** statements generated resulting in a corresponding value.

In the rest of this section, we show the different assumptions and invariants used in the proof of correctness, before presenting an overview of the proof.

[5] repr_bs_L6_L7_related in theories/L7/shrink_cps_correct.v

Theorem 4.3.2.

$$
\begin{array}{l}
\mathsf{UB}(\rho, e) \wedge \\
\mathsf{INV}_{m,lenv}(\mathit{tinfo}) \wedge \\
\mathsf{INV}_{e}(\Delta) \wedge \\
\rho \vdash e \Downarrow v \wedge \\
\mathsf{codegen}_{\Delta,\theta}\ e = s_e \wedge \\
\rho \simeq^{\mathrm{env}}_{\theta,e} (G, m, lenv) \implies
\end{array}
\begin{array}{l}
\exists m', \\
G, lenv \vdash s_e, m \Rightarrow^{\epsilon} \cdot, m' \wedge \\
(\rho, v) \simeq^{\mathrm{val}}_{\theta,m'} \mathit{targs}_1.
\end{array}
$$

Figure 4.20: Generalized statement of correctness for code generation

4.3.1 The Memory Relation

Central to the proof of correctness is the relation between the L6 evaluation environment ρ and the **Clight** evaluation environment comprised of a global environment G, a memory m and a local environment $lenv$.[6]

$$\rho \simeq^{\mathrm{env}}_{\theta,e} (G, m, lenv)$$

For any $x \in \rho$, if $\rho(x)$ is a function, or if x occurs free in e, then $(m, lenv)$ holds a related value for x according to the value relation shown in Section 4.3.2.

If x is the name of a function (which is to say, if $\rho(x) = (\rho', \vec{fd}, x)$, then global environment G provides us with the location $\mathsf{Vptr}\ G(x)\ 0$ of a **Clight** function related with $\rho(x) \simeq^{\mathrm{val}}_{\theta,m} \mathsf{Vptr}\ G(x)\ 0$.

Otherwise, the **Clight** value is held in the local environment at $lenv(x)$, and the value relation $\rho(x) \simeq^{\mathrm{val}}_{\theta,m} lenv(x)$ holds.

[6] Clight has two local environments, the "var env" for addressable local variables, and the "temp env" for nonaddressable local variables. Our code generator never generates code that uses addressable locals, so the var env is always empty; the "lenv" here is Clight's temp env.

$$\frac{fd(x) = (\vec{y}, e) \quad g[b, 0] \mapsto \mathsf{Internal}\ F \quad (\vec{y}, e) \sim_m F}{(\rho, \vec{fd}, x) \simeq^{\mathrm{val}}_{\theta,m} \mathsf{Vptr}\ b\ 0}\ \text{VR\_FUN}$$

$$\frac{}{(c,\ \cdot) \simeq^{\mathrm{val}}_{\theta,m} \mathsf{Vint}\ (\mathsf{hdr}_\Delta\ c)}\ \text{VR\_UCON}$$

$$\frac{m[b, o-8] = \mathsf{Vint}(\mathsf{hdr}_\Delta\ c) \quad (\forall v_i \in \vec{v}.\ m[b, o + (i \times 8)] = v_i^7 \wedge v_i \simeq^{\mathrm{val}}_{\theta,m} v_i^7)}{(c,\ \vec{v}) \simeq^{\mathrm{val}}_{\theta,m} \mathsf{Vptr}\ b\ o}\ \text{VR\_BCON}$$

Figure 4.21: Value Relation between L6 and **Clight**

4.3.2 The Value Relation

We include in Figure 4.21 the relation $v^6 \simeq^{\mathrm{val}}_{\theta,m} v^7$ between a L6 value v^6 and a **Clight** value v^7 as represented in memory m by **CertiCoq**.

Rule VR_FUN states that a function value $(\rho, \vec{fd}, x)$ is related to a pointer $\mathsf{Vptr}\ b\ 0$ in memory m if x corresponds to an entry $(\vec{y}.e)$ in $\vec{fd}$, b maps, in the global environment G to a function F [7], and $(\vec{y}, e)$ is related to F according to $\sim_m$ as described in Figure 4.22. F is a function taking in tinfo and returning *void*. Its body starts with the garbage collection test s_{gc} described in Section 4.1.6, followed by assignments setting the variables $\vec{y}$ to the arguments passed to the function in targs (governed by the calling-convention represented in finfo). Finally, the rest of the body, s_e, corresponds to expression e according to the code generation relation $\mathsf{codegen}(e) = s_e$.

Rule VR_UCON shows how an unboxed constructor $(c,\ \cdot)$ is related to a **Clight** $\mathsf{Vint}\ \ \mathsf{hdr}_\Delta\ c$, if c has arity 0, and $\mathsf{hdr}_\Delta\ c$ is the proper header for c, as described in Section 4.1.1.

Finally, rule VR_BCON relates a boxed constructor $(c, \vec{v})$ with a **Clight** pointer $\mathsf{Vptr}\ b\ o$ if, accessing the pointer to a value array at element -1, we find $\mathsf{hdr}_\Delta\ c$, and if, for every value v_i in $\vec{v}$, v_i is related to the **Clight** value held at the ith value-sized location after $\mathsf{Vptr}\ b\ o$.

[7] In **Clight**, functions are always block-aligned, that is, are at addresses (Vptr b offset) whose offset is 0

$$
\begin{aligned}
(\vec{y}, e) \sim_m F := \\
&\mathsf{return}\ F = \mathit{void} \wedge \\
&\mathsf{param}\ F = [\mathsf{Tpointer}(\mathsf{Tstructtinfo})\mathit{noattr}]; \wedge \\
&\mathsf{temps}\ F = \vec{y} \wedge \\
&\mathsf{body}\ F = s_{\mathsf{gc}}; s_{\vec{y}}; s_e \wedge \\
&s_{\vec{y}} = \mathsf{Sset}\ y_1\ \mathsf{targs}_{\mathsf{finfo}_1}; ...; \mathsf{Sset}\ y_n\ \mathsf{targs}_{\mathsf{finfo}_n} \wedge \\
&\mathsf{codegen}(e) = s_e
\end{aligned}
$$

Figure 4.22: Representation relation between L6 and **Clight** functions

4.3.3 Assumptions in the proof of correctness

In order to represent L6 programs in **Clight**, a few things need to be true of the original program. For some of these assumptions, we are able to transform an incompatible program to a compatible one. For others, we refuse to generate code. In this section, we present the different assumptions made and how they are handled by our code generation phase.

All functions are closure-converted and hoisted. When the back end receives the program under the form of an L6 term, we expect it to be hoisted, which is to say of the form $\mathsf{let}\ \vec{fd}\ \mathsf{in}\ e$, with e containing no function declarations.

Moreover, for every function `(f, `$\vec{y}$`, e')` $\in fd$, e' should be closed under $\vec{y} \cup fd$, as is the case with closure-converted functions. This is important for the proof of the code generation phase, as $\vec{y}$ then corresponds to the live roots at the entry of e', which are needed if we are to safely garbage collect memory before executing e'.

Maximum allocation per function. Our compiler assumes that no function allocates more memory than what the garbage collector can provide. This is because, in the back end, calls to the garbage collector are only inserted at the beginning of functions, and executed if there is enough available space, in the worse case, to run the function.

To insert calls to the garbage collectors in the middle of functions, or indeed to break functions at that point, we would have to compute the live variables and then closure-convert and hoist the remainder of the function. Instead, we parameterize the proof of the back end by a constant assumed to be bigger than the number of blocks allocated by any function in the program being compiled, and refuse to generate code if a function allocating more is found.[8]

Maximum number of arguments per function. Under the current calling convention, we assume that no function has more arguments than a fixed number (for example, 1024). However, compared to the allocation limit, it is easier to deal with:

1. As functions in Gallina are curried, multiple-argument functions are created by us, either through CPS, uncurrying, or lambda-lifting. We can limit the maximum number of arguments uncurried and lambda-lifted to a fixed bound (and CPS conversion merely translates 1-argument functions to 2-argument functions).

2. Under the current calling convention, arguments are passed in a heap-allocated array (allocated once, at the beginning of program (or thread) execution). This array can be declared to be bigger or smaller depending on the maximum number of arguments allocated by a function in the compiled program.

Under the current calling convention, and considering we are for now interested in whole program compilation, the second option is what we adopt. We think the first solution would be an easy addition and would make sense with other convention, for example, when passing the arguments through callee-saved registers, by limiting the number of arguments to the number of available registers.

[8]We remind the reader that an L6 "function" is a tree of control flow, in which there are a bounded, statically determinable number of allocations of fixed, statically known size.

Maximum number of arguments per constructor. Our header representation (see Section 4.1.1) allows for 54 bits for the arity of constructors. **CertiCoq** will refuse to compile a program if it refers to a computational datatype containing a constructor with more than 2^{54} arguments.

Maximum number of constructors per inductive datatype. Our header representation (see Section 4.1.1) allows for 8 bits for the ordinal of a boxed constructor, and 63 for unboxed constructors. **CertiCoq** will refuse to compile a program if it refers to a datatype with more than $2^8 - 1$ non-nullary constructors or more than $2^{63} - 1$ nullary constructors.

Full program compilation. Our proof of correctness assumes we are compiling the whole program.

This is in line with our expected use of developing safety-critical portions of the code in Coq, and interacting with that safe core through the generated shim.

We envision being able to relax this assumptions, at the level of the code generation proof, by including a global environment of safe import and export procedures, effectively proving the correctness of our generated shim.

Axiomatized (external) function. As detailed previously, we make use of the last bits of **Clight** values to differentiate between integers and pointers. Unfortunately, the **Clight** semantics does not allow for this distinction. We thus have to axiomatize the semantics of the function isptr to return *true* on aligned (divisible by word size) addresses.[9]

The garbage collector is also abstracted as an external function in the proof of correctness, with the semantics described in Section 4.1.2. We then prove on the side

[9]To realize this axiom as a theorem would require an extension to the CompCert correctness proof (but not a change to the behavior of the CompCert compiler).

that this specification is compatible with the specification under which the implementation of the garbage collector was proved correct [61], in **VST**.

Correctness of the conversion environments. When starting the code generation, we receive global environments describing the name and components of inductive datatypes found in the program. By this point of the compiler, we made sure that

- every constructor found in the program is represented in Δ.
- every constructor is applied to the number of arguments corresponding to its arity as recorded in Δ.

4.3.4 Invariants in the proof of correctness

To generalize the statement of correctness (see Figure 4.20), we need invariants asserting the correspondence between the L6 structures, the conversion environments and the **Clight** state. We now detail the invariant used in the proof.

Allocatable space. In the expression simulation relation, we assert that there is enough writable space between the allocation pointer and the limit pointer to allocate all the values assigned in the expression:

Theorem 4.3.3 (Sufficient allocatable space)**.**

$$(8 \times m \leq \mathit{tlimit} - \mathit{talloc})$$

This fact is reasserted at each function entry using, when needed, the proof that running the garbage collector results in enough space. This stays true throughout evaluation – since we provision memory for the heaviest path of the function, any step preserves or reduces the sum of the space used currently and the space needed until the end of the function.

Separation of spaces in memory. In a disjoint area of the heap, we keep the structure describing the current state of the allocatable space and information about the running program (as described in Section 4.1.6). It is important that this space is separated from the allocatable space, and that both of these are separated from the portion of memory holding the code portion of functions.

4.3.5 Specification of the interface with garbage collection

At the proof level, we axiomatize the effect of garbage collection on the provided interface (described in Section 4.1.6). We separate the logical portion of the proof from the spatial component:

Before garbage collection, we have a list of roots $\vec{v_7}$ held in the arguments array of tinfo at position described in finfo pointing to the garbage-collected area L (which we referred to as "GC heap" previously in this chapter) of a memory m and representing a list of L6 values $\vec{v_6}$.

After garbage collection, in the same arguments array of tinfo, and at the same position described in finfo, we have a list of roots $\vec{v_7}'$ pointing to a modified garbage-collected area L' of a memory m' representing the same list of L6 values $\vec{v_6}$. We also know that the space between the new talloc and tlimit pointer of the updated tinfo is writable, and at least the required size as described in finfo.

Theorem 4.3.4 (Assumptions w.r.t. correctness of a garbage collector[10])**.**

$$
\begin{aligned}
GC_{\mathit{finfo}_f}\,\mathit{tinfo}\,m = (m', \mathit{tinfo}') \;\Rightarrow\; & \\
& (\forall i \in \mathit{finfo}_f, v_6 \simeq^{\mathrm{val}}_{\theta,m} \mathit{targs}[i] \Rightarrow v_6 \simeq^{\mathrm{val}}_{\theta,m'} \mathit{targs}[i]) \wedge \\
& \mathit{tlimit}' - \mathit{talloc}' \geq \mathit{finfo}_f[0] \wedge \\
& (\forall \mathit{talloc}'_o \leq o < \mathit{tlimit}'_o, \mathsf{Writable}(m\ \mathit{talloc}'_b\ o))
\end{aligned}
$$

[10] program_gc_inv in theories/L7/shrink_cps_correct.v

On the spatial side, before garbage collection, L contains all the pointers reachable from $\vec{v_7}$, and is disjoint from tinfo and the area in which functions are allocated. After garbage collection, any location in m not in L and tinfo is unchanged. tinfo is still allocated at the same location in m', but the values it holds may have changed. Finally, all pointers reachable from $\vec{v_7}'$ are contained in L', which is disjoint from tinfo and the area in which functions are allocated.

Theorem 4.3.5 (Spatial Assumptions w.r.t. correctness of a garbage collector[11])**.**

$$GC_{finfo_f}\ tinfo\ m = (m', tinfo') \wedge$$
$$(\forall talloc_o \leq o < tlimit_o, \neg L\ talloc_b\ o) \Rightarrow$$
$$\exists L', (\forall talloc'_o \leq o < tlimit'_o, \neg L'\ talloc'_b\ o) \wedge$$
$$(\forall i \in finfo_f, \forall b\ o, \mathsf{reachable}_{m'}\ targs[i]\ b\ o \Rightarrow L'\ b\ o)$$

4.3.6 A correct generational garbage collector

The generational garbage collector developed for **CertiCoq** has been proved correct by Wang et al. [61]. We showed that their representation of garbage collection is compatible with our interface. However, we have not yet proven the spatial portion of the interface. This is because the garbage collector has been proved correct using the VST program logic [5], while the code generator is proved correct directly over the semantics of **Clight**. The proof could be completed by unfolding the definition of VST's Hoare triple (semax), as described in *Program Logics for Certified Compilers* [7]. Doing so would allow us to show that only the portions concerned with garbage compilation (which is to say, tinfo, targs and the GC heap) have been affected by garbage collection.

[11] program_gc_inv in theories/L7/shrink_cps_correct.v

4.3.7 Forward simulation between L6 and Clight

In this section, we give an overview of the simulation proof between L6 and **Clight**. The proof goes by induction on the L6 big-step evaluation derivation, in well-formed L6 and **Clight** environments.

The top-level statement of correctness is given in Figure 4.19. The generalized statement of correctness is given in Figure 4.20.

We provide here the details of each case of the proof, corresponding to the evaluation rule of the semantics of L6 included in Figure 3.1:

$\rho \vdash \mathsf{halt}\ x \Downarrow v$ By the evaluation derivation (E_HALT), we know that $\rho(x) = v$. Since x is free in $\mathsf{halt}\ x$, by the memory relation, we know that $v \simeq^{\text{val}-\text{id}}_{m,l} x$. By $\rightsquigarrow$, the value corresponding to v will be placed in the first slot of the argument array, as required.

$\rho \vdash \mathsf{let}\ x = \mathsf{Proj}_n\ y\ \mathsf{in}\ e \Downarrow v$ By the evaluation derivation (E_PROJ), we know that $\rho(y) = c\ \vec{w}$, and that $\rho; x \mapsto w_n \vdash e \Downarrow v$. Since y is free in $\mathsf{let}\ x = \mathsf{Proj}_n\ y\ \mathsf{in}\ e$, by the memory relation, we know that $v \simeq^{\text{val}-\text{id}}_{m,l} x$. Since v is a boxed constructor, we have $l\ y = \mathsf{Vptr}\ b\ o$ and $(c,\ \vec{w}) \simeq^{\text{val}}_{\theta,m} \mathsf{Vptr}\ b\ o$. By inversion on the value relation, we could only be in the boxed constructor case VR_BCON, and $m[b, o + (n \times 8)] = v^7_n \wedge w_n \simeq^{\text{val}}_{\theta,m} v^7_n)$. Extending l with $x \mapsto v^7_n$ by stepping through the assignment statements provides a local environment and memory related to $\rho; x \mapsto w_n$, and we can apply the induction hypothesis on $\rho; x \mapsto w_n \vdash e \Downarrow v$ and $\mathsf{codegen}(e) = s$.

$\rho \vdash \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ e \Downarrow v$ The constructor case of code generation relies on assumptions we are holding about allocatable space in the **Clight** memory. By the evaluation derivation (E_CONSTR), we know that $\forall_{y_i \in \vec{y}}, \rho(y_i) = w_i$ and $\rho; x \mapsto (c, \vec{w}) \vdash e \Downarrow v$. Here, we need to consider two different cases, corresponding to our value representation described in Section 4.1.1:

If $\mathsf{ord}_\Delta\ c = 0$, then we are generating **Clight** code "$\mathsf{Sset}\ x\ ((\mathsf{ord}_\Delta\ c \ll 1) + 1);\ s$" where $\mathsf{codegen}(e) = s$. Stepping through the assignment statement updated the local environment to $l, x \mapsto \mathsf{hdr}_\Delta\ (c)$, with $\rho; x \mapsto (c, \vec{w}) \simeq^{\text{env}}_{\theta,e} (G, m, l, x \mapsto \mathsf{hdr}_\Delta\ (c))$. Correctness follows by induction hypothesis on $\mathsf{codegen}(e) = s$ and $\rho; x \mapsto (c, \vec{w}) \vdash e \Downarrow v$.

If $\mathsf{ord}_\Delta\ c \neq 0$, we are generating code to allocate a boxed value: "$\mathsf{codegen}(\mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ e) = \mathsf{Sset}\ x - 1\ ((\mathsf{arr}_\Delta\ c \ll 8) + \mathsf{ord}_\Delta\ c);\ \mathsf{Sset}\ y_i\ v_{i+1};\ s$". In this case, because of our assumption about the memory, we know that there is enough allocatable space in m after the allocation pointer for $|\vec{y}| + 1$ value-sized blocks. First, we place the header of c in $m[alloc]$. Then, for each $y_i \in \vec{y}$, we have $w_i \simeq^{\text{val}-\text{id}}_{m,l} y_i$ by the memory relation, and we store that value at $m[alloc + (i \times |_{|} val)]$. Finally, we set x to point after the header, at $m[alloc + 8)]$, establishing all the necessary pieces for VR_BCON to hold for $c\ \vec{y}$, and extending the memory relation to have $\rho; x \mapsto (c, \vec{w}) \simeq^{\text{env}}_{\theta,e} (G, m, l, x \mapsto \mathsf{Vptr}\ b_{alloc}\ o_{alloc} + 8)$. Before using the induction hypothesis on $\mathsf{codegen}(e) = s$ and $\rho; x \mapsto (c, \vec{w}) \vdash e \Downarrow v$, we update the allocation pointer's offset to to $o_{alloc} + (|\vec{y}| + 1) \times 8$, and reestablish the assumption that is enough allocatable space for the allocation in the heaviest path in e after the new allocation pointer.

$\rho \vdash \mathsf{match}\ x\ \mathsf{with}\ \vec{b} \Downarrow v$ Code generation for case-statement relies on the correctness of our constructor environment, ensuring that each constructor c of an inductive type has a distinct $\mathsf{hdr}_\Delta\ c$. It also relies on the axiomatized semantics of isptr properly distinguishing between our representation of boxed and unboxed values (see Section 4.1.1). By the evaluation derivation (case E_MATCH), we know that $\rho(x) = c\ \vec{w}$, $(c \Rightarrow e) \in \vec{b}$ and $\rho \vdash e \Downarrow v$. Well-formedness of $\vec{b}$ ensures that only one case matches c. By the environment relation, we have $v \simeq^{\text{val}}_{\theta,m} l\ x$ (as x is matched on, and thus cannot be a function). If c is a nullary constructor, we are in the unboxed case (VR_UCON)

and we switch on the header held unboxed in l. Otherwise, $l\ x$ is a pointer, and we can recover the header and switch on it. In both cases, the recursion is done on $\rho \vdash e \Downarrow v$ and $\mathsf{codegen}(e) = s$ with a **Clight** continuation skipping through the remaining case of the switch.

$\rho \vdash \mathsf{App}\ f\ \vec{y} \Downarrow v$ Application is by far the most complicated case of this proof, as it relies on multiple assumptions of correctness for environments in order to properly save and restore arguments from the arguments array, to perform a call to the right location in memory, and to reestablish environment assumptions about the new code block by calling, if needed, garbage collection.

In the application case (R_APP), $\rightsquigarrow$ first generates statements to place the values corresponding to $\vec{y}$, in appropriate slots of the argument array according to the calling convention of f. By the memory relation, we have corresponding **Clight** values in g (in the case of functions) or l (for constructors) for any $y_i \in \vec{y}$.

Then, we step through the call to f, held in the global memory of the **Clight** program p. The expression relation ensures that a corresponding function info $\mathsf{finfo}$is available in the global environment, and that f maps to a sequence of statements consisting of a conditional statement for garbage collection, assignment statements restoring the arguments of f into local memory, followed by the translation of the body of the function.

At this point, we step through the conditional statement inserted by the code generator to ensure enough allocatable memory is available in the garbage-collected area for the heaviest path of e, the body of function f. If there is enough space, we proceed with the proof using $m' = m$. Otherwise, the garbage collection is called, and we use its axiomatized semantics to prove that the memory after collection, m', is suitable and related to $\rho, \vec{x} \mapsto \vec{y}$ under the memory relation. We provide

in section 4.1.2 details about the interface and axiomatized semantics for garbage collection. For the code generation proof, we concentrate on three properties of m':

1. m is related to m' over $\vec{y}$.

2. tinfo has been properly updated, and there is enough space in m' between talloc and tlimit.

3. nothing outside in tinfo, targs or the garbage-collected area has changed between m and m'.

This ensures all of the assumptions about m are still true about m', in addition to enough space in m' being available.

We then restore the arguments of f from the arguments array to the new local environment l'. Since f is closure-converted, we know that all of the free-variables of its body are bound as arguments (potentially in the environment argument). Meanwhile, because f is hoisted, any function in ρ is also present in ρ', so that portion of the memory relation is preserved. Taken together, these two steps ensure that the memory relation $\rho' \simeq^{\text{env}}_{\theta,e} (G, m', l')$ holds.

4.4 Generated Shims

CertiCoq compiles a Coq expression into a set of mutually recursive C functions, one of which is the designated entry point. We expect that this C-language entry-point function will be called from hand-written C code. This C-language "driver" might, for example, read input, formulate it as a Coq value in the OCaml-like representation described earlier in this chapter, call the entry-point function with an appropriate continuation-function argument, and then when that continuation is called, traverse the Coq result data structure and print output.

The writer of this C code will need to construct and traverse Coq values. For that purpose, we provide a simple library of functions to construct and destruct C representation (as described in Section 4.1.1) of every encountered datatypes.

For example, for the datatype included in Figure 4.1, we generate the following C functions, with the type signatures shown in Figure 4.23:

- an eliminator function `elim_T` taking in a value $c\vec{v}$ of type T and setting the second argument to be the ordinal representing c and the third arguments to be an array containing values $\vec{v}$.
- a function `make_T_A` returning the value 1, corresponding to the unboxed encoding of the ordinal of A, 0.
- a function `make_T_B` taking in a value v as first argument, and returning a representation of $B\ v$ in the value array passed as second argument.
- a function `make_T_C` returning the value 3, corresponding to the unboxed encoding of the ordinal of C, 1.
- a function `make_T_D` taking in a value v_1 as first argument and v_2 as second, and returning a representation of $D\ v_1\ v_2$ in the value array passed as third argument.

```
void elim_T(value val, value *ordinal, value **argv)
value make_T_A(void)
value make_T_B(value arg0, value **argv)
value make_T_C(void)
value make_T_D(value arg0, value arg1, value **argv)
```

Figure 4.23: Type signature of shim functions generated for datatype T (from Figure 4.1)

We also generate constant arrays `names_of_T` holding the names of the constructors of `T`, and `arities_of_T`, holding their arities.

Finally, we generate functions `call_n` and `call_n_export`, where n stands to the number of arguments, taking in a value $(f, \vec{e})$ representing a function closure and n values representing its arguments $\vec{v}$, and call f with $\vec{e}$ and $\vec{v}$. The `export` versions of the function copies the resulting value outside of the garbage-collected area, allowing for safe deallocation of `tinfo` and of the garbage-collected area of memory.

Taking together, these functions can be used to write C programs calling Coq functions as compiled by **CertiCoq**.

4.5 Related Work

Oeuf [41] is a verified extraction pipeline for a restricted subset of Gallina to Cminor, an intermediate representation of CompCert. This project has been developed concurrently to **CertiCoq**. It does not support user-defined datatypes, limiting the users to a predefined set of base types. It avoids dealing with extraction concerns by requiring its source terms to be written using eliminators for the provided base types. It also assumes unbounded memories, which we don't – we formalized the interface with garbage collection, and our proof links with the proof of a verified garbage collector. On the other hand, Oeuf's correctness statement allows reasoning about code that calls Oeuf-compiled code, which is not supported currently by **CertiCoq**'s correctness statement.

GCminor [39] is an intermediate language extending CompCert Cminor with primitives to interact with a garbage-collected heap, together with a library to define and prove the correctness of garbage collectors. A similar setup is presented in the thesis of Dargaye [18]. As mention in Section 4.1.6, this approach suffers from high runtime

overhead. Moreover, neither supports the use of a bit to distinguish pointer, making their framework not compatible with our representation of values.

PVS2C [48] presents a code generator from PVS, an interactive proof assistant based on higher-order logic, to the C programming language. Unlike Gallina, PVS is impure, supporting references and array updates. To support this, and to avoid memory leaks, they implement a reference counting runtime system keeping track of the number of live references to each object, and collecting objects when their reference count drops to zero. A formal model for reference counting is presented by the same author in an earlier paper [22], and shown to not impact the execution of well-typed PVS programs. It instruments the operational semantics with explicit operations to add and subtract from reference counts kept in a new portion of the state. Our solution is more general; while we implement a generational copying garbage collector, which is significantly faster than reference counting, our garbage collection interface could be used by a reference counting collector.

As of 2018, CakeML includes a generational garbage collector [21] which was developed and proved correct concurrently to ours. Their garbage collector is configurable in term of data representation, while ours assumes the representation described in Section 4.1.1. However, their proof does not expose an abstract, modular interface to garbage collection. Instead, they proved the correctness of code generation with respect to a specific garbage collector, and show a simulation relation between that garbage collector and others.

Cogent [43] is a project that aims to generate correct C code from a specification language embedded in HOL4. Compiling a Cogent program generates a C program and a proof, in HOL4, that the semantics of the C program correspond to the original program. In addition to using different proof techniques, with proof-carrying code in place of simulation proofs in **CertiCoq**, the Cogent language is much more restrictive than Gallina, being limited to malloc-free functional programs, and aimed

at the development of system software, for which a garbage collector would not be appropriate.

Other optimizing compilers have been developed, but not proved correct, from functional languages to C. We explore in the rest of this section the similarities and differences in compilation techniques used with **CertiCoq**.

Directly relevant is the sml2c project [56], which include a code generation phase from the SML intermediate representation, a direct inspiration for L6, to the C programming language. They implement a garbage collector, and like us (and SML/NJ before), they insert a single heap check at the beginning of every function. Fewer C compilers supported tail-call elimination at the time sml2c was developed, they instead rely on a dispatch loop.

Zinc→K2 is an optimizing compiler from CaML-light to C[15]. The research effort of this project is centered on optimizing function calls through *explicit specialization*. The compiler also uses a one-bit tag to differentiate between pointers and values, together with a copying garbage collector. A major different with **CertiCoq** is that their intermediate representation is not in continuation-passing style, so that they do not benefit from function entry having a defined sets of roots. Instead, their collector must deal with ambiguous roots spanning the whole accessible heap, a costly process.

4.6 Conclusion

In this section, we presented a code generation phase from a functional intermediate language to a subset of the C programming language.

While code generators have been developed before between a functional language and an imperative one, the design of our intermediate language allows for a straightforward translation to C, which impacts both the performance of the generated code and the size of the proof.

Our garbage collection interface separates the challenge of finding roots from the correctness of garbage compilation, resulting in a more modular proof of correctness.

Chapter 5

Evaluation

In this chapter, we evaluate **CertiCoq** by comparing our extraction pipeline with the unverified one currently included with the Coq theorem prover. We follow with a discussion of future work in the form of additional optimization phases and of alternative code generation that would improve the performance of the generated code.

5.1 Benchmarks

We evaluate the performance of the compiler on benchmarks introduced in Section 3.4, and on a functional implementation of the Secure Hash Algorithm (SHA) developed for a proof of correctness of SHA-256 using VST[1] [6]. *Binom* is a small benchmark consisting of a sequence of operations on binomial queues (89 LOC), *Color* runs the Kempe/Chaitin graph coloring algorithm on a large graph (1359 LOC), *Veristar* runs a paramodulation-based resolution decision procedure for separation logic over a large entailment (1964 LOC), and *SHA* runs SHA-256 on a 200 byte message (1143 LOC).

[1]Section 4 of Appel [6] presents two functional programs, `SHA256` and `SHA256'`, which are proved equivalent but the latter is much faster than the former. We use the slow version (`SHA256`) to facilitate the comparison with Oeuf

Figure 5.1 presents the compilation time (in ms) of the various benchmarks throughout the compiler.

- L0 to L1 records the time taken by MetaCoq to reflect the Coq kernel representation of the program into **PCUIC** (see Section 2.2).
- L1 to L2 records the time taken by MetaCoq to type-check the reflected program and and to erase its propositional portion while going from **PCUIC** to $\lambda\,\square$ (see Section 2.3).[2]
- L2 to L3 η-expands constructors and branches (see Section 2.4).
- L3 to L4 changes the representation to globally nameless and creates a term representing the full program by locally binding the environment (see Section 2.5).
- L4 to L6 performs CPS conversion and a change of binder representation to globally unique identifiers (see Section 2.6).
- L6 to L6c records the time taken by a series of optimizations and transformations over L6: uncurrying, shrink reduction followed by closure conversion and lambda-lifting and a second round of shrink reduction (see Section 2.6 and Chapter 3).
- Finally, L6c to L7 generates a **Clight** program from a closure-converted and lambda-lifted *L6* term (see Section 2.7 and Chapter 4).

The total compilation time includes pretty-printing C to a file from the compiled **Clight**.[3] We include for comparison the time taken by the currect extraction to

[2]Most of the time in this translation (in, in a lot of cases, in the whole pipeline) is spent type checking the reflected term in order to check if the term to erase is a proof in *Prop* or a *type*. Work is ongoing to speed up the type checking process, and to avoid redoing work that is already being performed in the Coq kernel.

[3]We could avoid this pretty-printing by sending the **Clight** abstract syntax tree (AST) generated by L6-to-L7 directly to Compcert.

OCaml. Benchmarks were run on a MacBook Pro (Retina, 15-inch, Mid 2015) with a 2.5 Ghz Intel Core i7 and 16 GB 1600 MHz DDR3 memory, using Coq 8.8.2 and OCaml 4.05.0. `Binom` is a toy benchmark aimed at showing the compilation and runtime overhead. `Color` is computationally simple, but involves the encoding of a very large graph, resulting in slower compilation time. `Veristar` and `SHA256` are both computationally involved, with functions and proofs about them taking most of the lines of code, resulting in most of the compilation time being spent in erasure (L1 to L2) and in functional optimization (L6 to L6c).

Figure 5.2 presents the runtime of the compiled benchmarks and of code extracted to OCaml and compiled using the OCaml byte-code compiler `ocamlc` and the OCaml native-code compiler `ocamlopt` (reported times do not include extraction and compilation time). Also shown are the run time of various evaluation facilities in Coq. `compute` is an interpreter with a call-by-value evaluation strategy. `vm_compute` and `native_compute` rely on reification to OCaml, employing, respectively, the byte-code and the native-code OCaml compiler, before reflecting the value into the Coq kernel. The provided time is the whole run time of the command. In the case of `Veristar`, uninstantiated propositional axioms prevent these facilities from executing the benchmark. **CertiCoq**-compiled programs run faster than programs extracted to OCaml and compiled using the OCaml byte-code compiler, but slower than those compiled using the OCaml native-code compiler. We believe that the optimizations described in Sections 5.2.3 and 5.2.4 will bring our runtimes closer to those of the native-code compiler.

5.1.1 Quantitative comparison with other verified extraction pipelines

While Oeuf [41] (see Section 4.5 for an overview of the project) also compiles Coq to C, it assumes its source is in an eliminator form, instead of recovering it from

Benchmark	Binom	Color	Veristar	SHA256
L0 to L1	< 1	207	179	977
L1 to L2	2	4767	600	2970
L2 to L3	< 1	1	1	1
L3 to L4	< 1	5	15	23
L4 to L6	3	138	157	152
L6 to L6c	12	711	933	901
L6c to L7	2	1654	148	252
Pretty-printing to C	30	1036	800	992
Total CertiCoq compilation time	49	8597	2841	6268
CompCert (ccomp) compilation	422	3860	8359	13669
Extraction time to OCaml	54	225	124	78
ocamlc byte-code compilation	21	177	132	70
ocamlopt native-code compilation	137	338	487	256

Table 5.1: Compilation time (in ms)

Benchmark	Binom	Color	Veristar	SHA256
CertiCoq	< 1	37	83	6019
ocamlc	2	53	126	10659
ocamlopt	2	12	15	1921
compute	< 1	7594	*	> 600000
vm_compute	< 1	228	*	> 600000
native_compute	158	1527	*	> 600000

Table 5.2: Run time performance measurements (in ms)

the Coq kernel representation of the term. The Oeuf team used the same `SHA256` benchmark as we did, from the VST proof of correctness of SHA-256[6]. When running their generated code with an unverified slab allocator, they observe a 20x slowdown compared to extraction followed by `ocamlc`, while we achieve a 2x speedup on the same benchmark.

5.2 Future work

5.2.1 Additional optimizations over L6

While we made sure to include realistic compiler optimizations allowing the current pipeline to handle reasonably sized Coq developments, additional optimization phases would further decrease the size of the generated code, and reduce their execution time. Common subexpressions are introduced at different points in the compiler. For example, in the Coq kernel, mutually recursive function bundles are duplicated for every access point. Continuations are also duplicated in the case of pattern-matching. Common subexpression elimination could be applied over L6 to consolidate the portions of the code that were not simplified by shrink reduction.

5.2.2 Separate Compilation, and Interface with C programs

We envision that most users of **CertiCoq** will link their extracted code with performance-sensitive, or reactive portions of development implemented in the C programming language. For this, the current **CertiCoq** release is insufficient, because the generated shims are not proved to generate adequate representation of values, and because the proof of correctness only considers full programs. Future work will be needed to extend the interface between the generated code and C programs, and to be able to link, at the proof level, proofs about Coq development compiled with **CertiCoq** and proofs (in program logics such as VST) about the C portions of the development.

5.2.3 Extraction directives and support for native datatypes

We believe significant run time performance improvement could be achieved by allowing, as is the case with the Coq extraction pipeline, to translate specific Coq datatypes and functions as native datatypes and optimized functions in the target

language. While this is unsafe in general, we could implement *blessed* extraction directives and prove them correct.For example, programs proved correct with respect to integers using modular arithmetic could be safely extracted to machine integers, and primitive operations on them should be replaced by equivalent operations on machine integers. *L6* already supports a notion of primitive datatypes and functions through constructor Prim, such that most of the work would be in adding the special cases to code generation, and proving their correctness.

5.2.4 Further optimization for code generation

Our parameter passing strategy for function calls, described in Section 4.1.3, was designed to be compatible with a wide array of architectures, including those with a limited number of general-purpose registers such as i386. However, on x86-64 (and on RISC machines such as ARM, PowerPC, RISC-V, etc.), we believe we would benefit for adopting a different calling convention for the generated functions to pass arguments in registers. Since multiple argument functions are introduced by uncurrying, we could limit uncurrying to the number of available registers (for example, 16 on x86-64). Moreover, since we are compiling functions in continuation-passing style, and as such arguments are never needed on return, we should use a compiler directive so that the calling convention for all our generated C functions uses no callee-save registers at all, and uses all available registers for parameter passing, as is done in the GHC→LLVM calling convention for Haskell.

Chapter 6

Conclusion

In this thesis, I have described **CertiCoq**, a verified extraction pipeline for the Coq interactive theorem prover. I have focused on my contributions to the project, including the proof of correctness of optimizations over the L6 intermediate representation and of the code generation phase. The proof framework described in Chapter 3 allows for reusing portions of the proof of correctness of shrink reduction for other optimizations over the same intermediate representation. The proof of correctness of the code generation phase described in Chapter 4 includes a novel interface with garbage collection abstracting away details of specific garbage collectors. For Coq developments using extraction facilities, **CertiCoq** considerably reduces the trusted computing base of Coq, replacing an unverified extraction pipeline targeting an unverified compiler code running with an unverified runtime system. The pipeline described in this thesis is the first step in providing a certified extraction mechanism allowing realistic programs to be developed and proved-correct in a proof-assistant without sacrificing runtime efficiency.

CertiCoq is an ongoing project at Princeton University, INRIA, Cornell University, and the University of Edinburgh. Improvements to the generated code such as

the ones described in Section 5.2, and a certified interface with C programs proved-correct in VST, would bring us closer to that goal.

Another line of future work would be to implement reflection facilities in order to use CertiCoq as a reduction mechanism for large proof goals, as an alternative to `vm_compute` and `native_compute`. While our current pipeline does away with proofs early on, nothing prevents us for keeping them as part of the generated programs, and evaluating them just like we do for the rest of the program. The resulting values could then be decoded as Gallina proof terms and reintegrated into the proof state. Work to reduce the compilation time will be required in order to make this viable.

While our current front end is tailored for Coq, we believe our middle and back end could be reused to efficiently and safely compile other functional languages. L6 is general enough to be used as an intermediate language for a wide array of languages, while being structured to make it easier to optimize and to compile down to lower-level languages.[1]

In 1967, McCarthy and Painter published a paper in which they outlined the "ultimate goal" of "mak[ing] it possible to use a computer to check proofs that compilers are correct"[38]. Advances in programming languages, compilers and in theorem proving have made this goal a reality, as exemplified by projects such as CompCert, a semantically verified optimizing compiler for C. However, the vast majority of the compilers used today are not verified, and contain bugs that have been shown to be absent in verified compilers like CompCert [63]. Even in a proof and programming environments such as Coq, where safety critical applications are developed with formal guarantees about the algorithms used and the correspondence with their implementation, unverified compilation is often a part of the pipeline. I hope that this project

[1] The statement of correctness of the code generation phase (see Figure 4.20) is based on a forward simulation of the big-step semantics of L6, so that it can only be used for terminating programs. While this is not a problem in Coq, where strong reduction holds, the proof would need to be adapted to be useful for general-purpose programming languages. We believe the proof should be easily adapted to simulation over the small-step semantics of L6.

and continued work in the area of compilers and runtime systems verification will reduce the verification overhead of optimizing compilers, and make them available and efficient enough to be widely used.

Bibliography

[1] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. The verisoft approach to systems verification. In Natarajan Shankar and Jim Woodcock, editors, *2nd IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08)*, volume 5295 of *LNCS*, pages 209–224. Springer, 2008.

[2] Abhishek Anand, Andrew W. Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Weaver. Certicoq: A verified compiler for Coq. In *CoqPL 2017: The Third International Workshop on Coq for Programming Languages*, January 2017.

[3] Andrew W. Appel. SSA is functional programming. *SIGPLAN Not.*, 33(4):17–20, April 1998.

[4] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 2007.

[5] 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, pages 1–17, Berlin, Heidelberg, 2011. Springer-Verlag.

[6] Andrew W. Appel. Verification of a cryptographic primitive: Sha-256. *ACM Trans. Program. Lang. Syst.*, 37(2):7:1–7:31, April 2015.

[7] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, New York, NY, USA, 2014.

[8] Andrew W. Appel and Trevor Jim. Shrinking Lambda Expressions in Linear Time. *J. Funct. Program.*, 7(5):515–540, September 1997.

[9] 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, pages 49–60, New York, NY, USA, 2017. ACM.

[10] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the clight subset of the c language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[11] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[12] Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning - JAR*, 49:1–46, 10 2012.

[13] Marie-Christine (Kirsten) Chevalier. Implementing type-based deforestation. Master's thesis, University of California at Berkeley, 2004.

[14] Thierry Coquand and Christine Paulin. Inductively defined types. In *Proceedings of the International Conference on Computer Logic*, COLOG '88, pages 50–66, London, UK, UK, 1990. Springer-Verlag.

[15] Regis Cridlig. An optimizing ML to C Compiler. In SIGPLAN Workshop on ML and its Applications, 1992.

[16] Olivier Danvy and Andrzej Filinski. Representing control: a study of the cps transformation, 1992.

[17] Olivier Danvy and Lasse R. Nielsen. CPS Transformation of Beta-redexes. *Inf. Process. Lett.*, 94(5):217–224, June 2005.

[18] Zaynah Dargaye. *Vérification formelle d'un compilateur optimisant pour langages fonctionnels*. PhD thesis, Paris 7 – Denis Diderot, 2009.

[19] Zaynah Dargaye and Xavier Leroy. *Mechanized Verification of CPS Transformations*, pages 211–225. Springer, 2007.

[20] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically typed language. *SIGPLAN Not.*, 27(7):273–282, July 1992.

[21] Adam Sandberg Ericsson, Magnus O. Myreen, and Johannes Åman Pohjola. A verified generational garbage collector for CakeML. *Journal Automated Reasoning (JAR)*, 2018.

[22] Gaspard Férey and Natarajan Shankar. Code generation using a formal model of reference counting. In *NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*, pages 150–165, 2016.

[23] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[24] 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.

[25] Benjamin Grégoire. *Compilation de termes de preuvres, un (nouveau) mariage entre Coq et OCaml*. PhD thesis, Paris 7 - Denis Diderot, 2003.

[26] Fergus Henderson. Accurate garbage collection in an uncooperative environment. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, pages 150–156, New York, NY, USA, 2002. ACM.

[27] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95, pages 13–22, New York, NY, USA, 1995. ACM.

[28] Andrew Kennedy. Compiling with Continuations, Continued. *SIGPLAN Not.*, 42(9):177–190, 2007.

[29] Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, pages 107–115, 2009.

[30] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release*, 2019. Version 4.08.

[31] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Univ. Paris-Sud, July 2004.

[32] John Li. Verifying the uncurry phase of the CertiCoq compiler. Independent Work Report, 2018.

[33] Gregory Michael Malecha. *Extensible Proof Engineering in Intensional Type Theory*. PhD thesis, Harvard University, 2014.

[34] 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.

[35] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[36] The HOL development team. *HOL Interactive Theorem Prover*.

[37] The SMLNJ development team. *Standard ML of New Jersey User's Guide* .

[38] John McCARTHY and James PAINTER. Correctness of a compiler for arithmetic expressions. *Symposia in Applied Mathematics*, 1967.

[39] 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, pages 273–284, 2010.

[40] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, May 1999.

[41] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. Oeuf: Minimizing the Coq extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 172–185, New York, NY, USA, 2018. ACM.

[42] 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. *SIGPLAN Not.*, 50(9):166–178, 2015.

[43] Liam O'Connor, Christine Rizkallah, Zilin Chen, Sidney Amani, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Alex Hixon, Gabriele Keller, Toby C. Murray, and Gerwin Klein. COGENT: certified compilation for a functional systems language. *CoRR*, 2016.

[44] Zoe Paraskevopoulou and Andrew W. Appel. Closure conversion is safe for space. In *ICFP 2019: International Conference on Functional Programming*, 2019.

[45] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125 – 159, 1975.

[46] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Conference on LISP and Functional Programming*, pages 288–298, 1992.

[47] Amr A. Sabry. The formal relationship between direct and continuation-passing style optimizing compilers: A synthesis of two paradigms, 1994.

[48] Natarajan Shankar. A brief introduction to the PVS2C code generator. In Natarajan Shankar and Bruno Dutertre, editors, *Automated Formal Methods*, volume 5 of *Kalpa Publications in Computing*, pages 109–116, 2018.

[49] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq project. In *ITP 2018: the 9th International Conference on Interactive Theorem Proving*, July 2018.

[50] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Theo Winterhalter. Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq. In *POPL*, 2019.

[51] Guy L. Steele, Jr. Rabbit: A compiler for scheme. Technical report, MIT, Cambridge, MA, USA, 1978.

[52] Gordon Stewart, Lennart Beringer, and Andrew W. Appel. Verified heap theorem prover by paramodulation. *SIGPLAN Not.*, 47(9):3–14, 2012.

[53] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. A New Verified Compiler Backend for CakeML. *SIGPLAN Not.*, 51(9):60–73, 2016.

[54] 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.

[55] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 181–192, 1996.

[56] David Tarditi, Peter Lee, and Anurag Acharya. No Assembly Required : Compiling Standard ML to C. Technical Report 187, CMU-CS.

[57] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, LFP '94, pages 1–11, New York, NY, USA, 1994. ACM.

[58] Janis Voigtländer. Concatenate, reverse and map vanish for free. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 14–25, New York, NY, USA, 2002. ACM.

[59] Jean Vuillemin. A data structure for manipulating priority queues. *Commun. ACM*, 21(4):309–315, April 1978.

[60] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *Proceedings of the Second European Symposium on Programming*, pages 231–248, Amsterdam, The Netherlands, The Netherlands, 1988. North-Holland Publishing Co.

[61] Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. Certifying graph-manipulating C programs via localizations within data structures. In *Proceedings of the ACM on Programming Languages*, OOPSLA, 2019.

[62] Dinghao Wu, Andrew W. Appel, and Aaron Stump. Foundational proof checkers with small witnesses. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, PPDP '03, pages 264–274, New York, NY, USA, 2003. ACM.

[63] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.