VERIFYING CORRECTNESS OF A CHEZ SCHEME COMPILER PASS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Ian Atol

June 2021

COMMITTEE MEMBERSHIP

TITLE: Verifying Correctness of a Chez Scheme Compiler Pass

AUTHOR: Ian Atol

DATE SUBMITTED: June 2021

COMMITTEE CHAIR: John Clements, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Aaron Keen, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Stephen R. Beard, Ph.D.
Assistant Professor of Computer Science

ABSTRACT

Verifying Correctness of a Chez Scheme Compiler Pass

Ian Atol

We present a proof of correctness for a pass of the Chez Scheme compiler over a subset of the Scheme programming language. To improve trust in our proof approach, we provide two different validation frameworks. The first, created with the Coq proof assistant, is a partial mechanization of the proof, notably implementing a formal semantics for our subset of Scheme. This framework was designed to serve as a basis for the future work of a complete mechanization of our proof. The second framework uses an existing implementation of the Scheme semantics to demonstrate correctness of the pass on a variety of individual examples. We discuss our proof and frameworks in-depth, and give a historical background on compiler correctness proofs and their mechanization.

ACKNOWLEDGMENTS

Thanks to:

- My friends and family for supporting me unconditionally, bringing me food on many occasions, and reminding me to step outside every now and then.
- John Clements for putting up with me borrowing books for way too long, for being contagiously excited about programming languages, and for believing in me.
- Aaron Keen for putting in extra effort to help me out several times, for teaching me Rust, and for introducing me to type theory.
- Zoë Wood for pointing me exactly in the right direction, and to exactly the right people.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

As computers become increasingly ubiquitous in our lives, the software running on them becomes even more important, but also more complex. For example, we are lucky to live at a time where computers can help us to live longer by powering advanced medical equipment [32], but the grave consequences of errors in the software of these devices is clear [43]. This simultaneously increasing complexity and importance is troubling, since these two concepts are traditionally at odds.

To confront these issues, many researchers are hard at work with the goal of ensuring reliability of software. One approach to this problem, *static analysis*, aims to prove properties of programs before they are run. For example, static analysis could be performed on a program to eliminate a certain type of error from it [5]. One approach to static analysis uses the *semantics* [44] of a language to devise formal proofs about the "meaning" of a program.

However, the journey from source code written by a human to the code that is actually executed by a computer is not trivial. For many languages, source code must be translated by a tool called a *compiler* to machine code that the computer can run. If the compiler makes an error in translation, any proof about the behavior of that program is now forfeit. Essentially, any proof made about the source code of a program makes the assumption that a compiler will faithfully translate its meaning to the computer.

As our programs grow in size and complexity, so too do our compilers, adding additional features and optimizations. This means that our compilers themselves are

caught in the same troubling convergence of increasing complexity and importance. One natural thought is to look to prove properties about the compilers themselves. Namely, we want to prove that our compilers translate correctly — that the meaning of any program given to them is preserved in the process of translation. For example, the CompCert project [34] formally verifies the correctness of a compiler for a large subset of C. If we make proofs about programs in this subset, we can be sure that their claims are preserved through the compilation process.

In this paper, we provide a proof of correctness for a single pass of the Chez Scheme compiler [19], called *convert-assignments*. To support and validate our reasoning, we provide two different frameworks for producing evidence of our proof. The first is a formal model of the subset of the Scheme semantics that we use to reason about the compiler pass. Built using the Coq proof assistant [7], this model provides a framework for using intuitionistic logic to prove properties about Scheme and Scheme programs with a high degree of trustworthiness. This model was meant to be the basis of a mechanization of our proof, but the full mechanization ended up being outside of the scope of this thesis. The second framework provides a way of testing individual programs for semantic preservation over the *convert-assignments* transformation. This framework, created with the Racket [22] language, uses an existing implementation of the Scheme formal semantics to validate our proof technique on given example programs.

We hope that this work can serve as a trustworthy proof of the correctness of this compiler pass, and more generally hope to see a future where we can wholly trust that our compilers are translating critically important programs correctly.

Chapter 2

BACKGROUND

2.1 Compiler Correctness

"Can you trust your compiler?"

This quote begins the paper on the CompCert project [34], one of the largest proofs of compiler correctness. These proofs aim to provide trust in our compilers by formally verifying that they preserve the meaning of programs that they translate. CompCert provides such a proof for a purpose-built compiler that translates a large subset of C to machine code. For large languages like C, which have correspondingly large compilers, these proofs are notoriously difficult — CompCert totals around 100,000 lines of code in the Coq proof assistant, and took over 6 years to complete [33].

In this section, we will review some background on compiler correctness proofs. To do so, we have to look not only at the history of such proofs themselves, but also the evolution of tooling that embeds mathematical logic systems and allows for management of large-scale proofs. Without such systems, modern, large-scale projects such as CompCert would not be possible, so the advancement of these tools is key to understanding the progression of compiler correctness proofs.

2.1.1 History

The first known formal proof of compiler correctness comes from John McCarthy and James Painter [38]. In their 1967 paper, they prove that a compiler that translates simple arithmetic expressions to machine code is correct. Despite the simplicity of the

example source language, the paper is very important in that it sets up a methodology for computational proofs of compiler correctness. For example, their method of proof by structural induction of expressions is still an oft-used strategy for reasoning about properties of a language's programs.

McCarthy and Painter's proof was intentionally simple, so much so that it was able to be manually formulated and checked. However, when dealing with larger languages, case analysis of language expressions quickly generates too large of a proof to keep track of by hand. Because of this complexity, modern proofs of compiler correctness universally utilize programs called proof assistants that embed mathematical logic systems and can computationally verify the consistency of theorems defined within them. These assistants are necessary to aid with management of proofs at such a large scale. As such, their creation and development has been strongly connected to the progress of compiler correctness proofs.

2.1.1.1 Proof Checkers, Intuitionistic Logic, and Mechanized Proofs

The first large-scale attempt to *mechanize* mathematics, or formally define mathematics in a way tractable by a computer, was Nicolaas Govert de Bruijn's Automath language [10]. Automath was an early example of a correspondence between logic and programs — in the Automath language, theorems are defined as types, and proofs consist of showing that these types are inhabited by some value. This means that the definitions and proofs of theorems within Automath's logical framework are represented as a computer program. This relationship between programs and logic is also at the core of the Curry-Howard correspondence between deductive logic and the simply-typed Lambda Calculus.

Research into these sorts of program-logic relations continued into the 70s and 80s, with the development of the *Intuitionistic Theory of Types* [36] by Per Martin-Löf and the polymorphic Lambda calculus [24] by Girard. These theories also rely on the Curry-Howard correspondence to tie their dependently typed programs to statements in *intuitionistic logic*.

Intuitionistic logic is a kind of logical system that requires *evidence* or *witnesses* of a statement to prove its validity. That is, to prove that a statement $A \to B$ is true in an intuitionistic logic, one must use the axioms of the logic to *construct* evidence that B is true from the existing evidence that A is true. One important property of these intuitionistic logics that follows their constructive foundation is that the law of the excluded middle is not true in these systems — that is, we cannot perform indirect proofs, for example by contradiction. In other words, proving that $\neg A$ is not true does not suffice as proof of A. In this way, intuitionistic logic requires more direct proof of statements. In systems such as Automath, where a correspondence between logical statements and programs is established, a constructive proof of a statement corresponds to an algorithm that generates the program representing that statement. Because of this strong correspondence, these constructive, intuitionistic proofs lend themselves extremely well to automation, just as developers may automate complex parts of a software project.

A further example of the natural connection between intuitionistic logic and dependent type theory is Thierry Coquand's Calculus of Inductive Constructions[15, 41]. This system combines Intuitionistic Type Theory and the polymorphic Lambda Calculus into a single calculus that also provides support for writing specifications that automatically come equipped with powerful induction principles.

While the Calculus of Inductive Constructions is powerful, it is unwieldy and hard to manually construct large programs with. For this reason, the Coq proof assistant

[7] was devised. An extension of the original Automath language, Coq embeds the Calculus of Inductive Constructions, and also provides a high level tactics language [17] on top of the core calculus. This tactics language provides automation in the form of syntactic sugar and algorithmic search of core calculus expressions to assist in the construction of large proof terms. This tactics language greatly increase the size of proofs that Coq can handle — indeed, Coq programmers spend the majority of their time writing, configuring, and refactoring proofs using the tactics language, so at a much higher level than using the calculus itself.

Because of its support for automation, its history, and a large amount of libraries and community support, Coq is a natural choice for large-scale mechanized proofs. One example of this is the proof of the four color theorem. This theorem was famously unsolved until Coq's automation features made its extensive case analysis proof feasible [25]. Other projects realized in Coq include the Univalent Foundations project [49], which attempts to build a foundation for mathematics based on a type theory, and the CompCert project.

Finally, while we focus on Coq because of its usage in this project, a plethora of modern proof assistants exist [23, 6]. Some examples of modern languages that are used as proof assistants include Lean [16], Agda [8], and Idris [9]. More and more frequently, and in various fields, these tools are used to provide mechanized proofs to accompany research papers. In the next section, we will review some modern compiler correctness projects, all of which use proof assistants to validate their approach.

2.1.1.2 Verified Compiler Projects

Table 2.1 gives an overview of some compiler verification projects.

CompCert [34]	A verified compiler from C to various assembly languages, written and proven in Coq.
CakeML [30]	A verified compiler for a subset of ML, verified using Isabelle/HOL [40]
CertiCoq [3]	A verified Coq compiler, also written and proven using Coq
VLISP [27]	A verified (but not mechanized) compiler for an early version of Scheme
ClightTSO [45]	An extension of CompCert that verifies a compiler for a C-like language that supports shared-memory concurrency.
Concurrent Java [35]	A verified compiler for a subset of Java that supports threading, formalized in Isabelle/HOL.

Table 2.1: Sampling of verified compiler projects

So concludes our background on compiler verification projects and history. In the next section, we will review the Scheme language and its compiler, as background for our later proof of correctness of one of its passes.

2.2 Scheme

Scheme is a functional language based on LISP. While LISP supported functional programming (i.e., first-class functions and recursion) with a "lambda" notation [37], Scheme was the first LISP to closely mirror the call-by-value Lambda Calculus by utilizing static scoping for its variable bindings. In addition, as Scheme evolved and added more features, its macro system, syntactic pattern matching, and homoiconity made it a language suited to modelling other programming languages. Because of this, Scheme has seen extensive use in the area of programming languages research.

In this section, we will give some historical background on Scheme, then review the version of Scheme we chose for this project, and finally touch on the Chez Scheme compiler itself.

2.2.1 Lambda Calculus, LISP, and Scheme

The Lambda Calculus is a model which Alonzo Church and his students developed in the late 1930s [13] as a way to categorize a certain kind of number problem. It was later famously shown to be equivalent to Turing Machines, and together define the standard class of known computable problems [14]. While programming languages are inherently based in computation, the Lambda Calculus was later found to be capable of simply and accurately modeling the behavior of parts of ALGOL 60 [31], a programming language based on procedures.

While working on a LISP-based language for modeling actor-based concurrency [48], Steele and Sussman discovered that their new language had a strong, unexpected connection with the Lambda Calculus. While LISP supported lambda notation for functions, it did not handle variable naming or scoping in the same way as Church's original model. However, as Steele and Sussman shaped their version of LISP, they found that they were able to greatly simplify the language by following the semantics of the call-by-value version of the Lambda Calculus closely. The resulting language, called Scheme, was shown to be able to effectively model a wide variety of other programming languages, while still maintaining a small size and a tidy semantics [47].

Because of its small size and close connection to both the well-studied Lambda Calculus and popular LISP language, Scheme saw widespread usage for research in the area of programming languages. Using Scheme's powerful pattern matching and tools for syntactic abstraction, researchers can quickly create a model of their work to accompany a more detailed paper. We use Scheme in a similar manner for our own work in this paper (see Section 5.2).

Scheme has a language standard in the form of the Revised^n Report on the Algorithmic Language Scheme (RnRS). This standard defines a formal syntax and semantics for a large subset of Scheme, while leaving some areas up to the specific implementation. Our work is based on one of the more recent standards for Scheme — R6RS [46].

2.2.2 Scheme Verification

Along with its widespread usage in programming languages research, Scheme has itself been the target of formal verification projects. One such project, VLISP [27], was based on an earlier, denotational semantics for Scheme [1]. The initial VLISP project led to several formally verified extensions as well as explorations into representing Scheme using an operational semantics [26].

2.2.3 The Chez Scheme compiler

The Chez Scheme compiler is an optimizing compiler written in Scheme itself. It provides an implementation of Chez Scheme [20], which follows the R6RS standard and adds some additional features. Notably, it is the compiler for many Scheme dialects, including the Racket language [22]. It notably utilizes the Nanopass framework [29] at its foundation. The framework provides a domain specific language embedded in Scheme, designed for quickly defining languages and translations between them. An example of a Nanopass language definition from the Chez Scheme compiler is shown in Figure 2.1. This example shows an intermediate language (L4) that removes set! expressions from another language that it extends (L3). We will see later that this definition corresponds to the pass that we prove correctness of (see Section 4.1).

```
(define-language L4 (extends L3)
  (entry CaseLambdaExpr)
  (Expr (e body)
    (- (set! x e))))
```

Figure 2.1: Example of a language definition in Nanopass

In this project, we build a framework for reasoning about Scheme and use it to prove correctness of one pass of the Chez Scheme compiler. One reason we chose to target the Chez Scheme compiler for verification was because of its usage of the Nanopass framework. Since passes are distinctly separated and small in size by design, correctness proofs of individual passes should be simpler, and also are able to be easily composed to larger proofs about correctness of a sequence of passes.

Chapter 3

FORMALIZING SCHEME

Any proof about properties of Scheme, such as a proof about semantic preservation over a transformation, needs to be based on a formal definition of the language. Therefore, our first step in writing such a proof is to formally define the Scheme language. For the sake of simplifying the proof, we excluded many features — in general, these are features where our transformation doesn't change very much, but are complex enough to add substantial difficulty to our proof. These excluded features are discussed in more detail in Section 3.1.2.6.

3.1 R6RS Scheme

Fortunately, Scheme is already well-defined. The R6RS language standard [46] provides a formal specification of Scheme's syntax and semantics, and also gives an implementation of the semantic model in the PLT Redex language. As previously mentioned, we excluded many features from this formal definition for ease of reasoning. Below is our modified version of R6RS Scheme

3.1.1 Syntax

Figure 3.1 shows an EBNF representation of the syntax for our subset of Scheme.

```
P        ::=  (store (sf ...)  e)
sf       ::=  (x v)
          |   (pp (cons v v))
e        ::=  nonproc
          |   proc
          |   (begin e e ...)
          |   (if e e e)
          |   (e e ...)
          |   (set!  x e)
          |   (values v)
          |   x
v        ::=  nonproc
          |   proc
nonproc  ::=  pp
          |   null
          |   n
          |   #t
          |   #f
proc     ::=  (lambda (x) e)
          |   (lambda () e)
          |   car
          |   cdr
          |   cons
          |   set-car!
          |   set-cdr!
          |   +
          |   -
          |   /
          |   *
pp       ::=  [pair pointers]
x        ::=  [variables]
n        ::=  [integers]
```

Figure 3.1: The syntax for our subset of Scheme

Variables and pair pointer names are restricted to exclude keywords.

3.1.1.1 Evaluation Contexts

While we have presented a syntax for programs in our language, as we will see in the next section, our semantics operates on programs decomposed into an *evaluation context* and a reducible expression.

Evaluation contexts utilize a syntax of *holes* and *contexts* to isolate certain sub-expressions in order to apply semantic rules on them.

For example, we can decompose the following program to allow for evaluation of the expression in the e_1 position of the if expression.

$(store\ (sf\ \ldots)\ (if\ ((lambda\ (x)\ \#t)\ 5)\ \#t\ \#f)) = C[((lambda\ (x)\ \#t)\ 5)]$ where $C = (store\ (sf\ \ldots)\ (if\ [\,]\ \#t\ \#f)$.

The key feature of evaluation contexts is that they can control where evaluation occurs. These contexts cleverly enforce a canonical order of evaluation by placement of the holes in their syntactic form.

Since we have removed many features, we use only the evaluation contexts relevant to expressions that are still in our language:

```
C   ::=  (store (sf ...)  F*)
F   ::=  [ ]
      |  (v ...  F∘ v ...)
      |  (if F∘ e e)
      |  (set!  x F∘)
      |  (begin F* e e ...)
F*  ::=  [ ]*
      |  F
F∘  ::=  [ ]∘
      |  F
```

Figure 3.2: The evaluation contexts for our subset of Scheme

Here, F_* and $F_\circ$ refer to contexts that perform promotion or demotion between single values and (values v) expressions. The relevant semantics rules are presented in Figure 3.4. Though we do not support multiple values, we wanted to preserve this feature of the formal semantics to increase extensibility of our subset, as well as ensure that our semantics is a true subset of the R6RS semantics. It also means that our proofs will be more easily adaptable if the language is modified to include multiple values.

3.1.2 Semantics

Figures 3.3 and 3.4 show the operational semantics for our subset of Scheme.

We define x being "assigned to" as x being the target of a set! expression.

Some discussion of our semantic rules and how they work together to evaluate Scheme programs is required.

3.1.2.1 Control Flow

beginC works by removing values from the front of a $(begin\ \ldots)$ expression. Together with the begin evalaution context, **beginC** evaluates and removes each sub-expression in a $(begin\ \ldots)$ expression until it has only a single sub-expression remaining. Then, the **beginD** rule applies to consider that expression alone. This is how the requirement that Scheme begin expressions return the value of the last sub-expression is enforced.

ifT and **ifF** behave as expected, with the minor quirk that any value that is not $\#f$ is considered true.

Control Flow

beginC

$$C[(begin\ (values\ v)\ e_1\ e_2\ \ldots)] \longmapsto C[(begin\ e_1\ e_2\ \ldots)]$$

beginD

$$C[(begin\ e_1)] \longmapsto C[e_1]$$

ifT

$$C[(if\ v_1\ e_1\ e_2)] \longmapsto C[e_1] \quad (v_1 \neq \#f)$$

ifF

$$C[(if\ \#f\ e_1\ e_2)] \longmapsto C[e_2]$$

Lists

cons

$$(store\ (sf\ \ldots)\ F[(cons\ v_1\ v_2)]) \longmapsto$$
$$(store\ (sf\ \ldots (pp\ (cons\ v_1\ v_2)))\ F[pp]) \quad (pp\ fresh)$$

car

$$(store\ (sf_1 \ldots (pp\ (cons\ v_1\ v_2))\ sf_2 \ldots)\ F[(car\ pp)] \longmapsto$$
$$(store\ (sf_1 \ldots (pp\ (cons\ v_1\ v_2))\ sf_2 \ldots)\ F[v_1]$$

cdr

$$(store\ (sf_1 \ldots (pp\ (cons\ v_1\ v_2))\ sf_2 \ldots)\ F[(cdr\ pp)] \longmapsto$$
$$(store\ (sf_1 \ldots (pp\ (cons\ v_1\ v_2))\ sf_2 \ldots)\ F[v_2]$$

Mutation & Reference

var

$$(store\ (sf_1 \ldots (x\ v)\ sf_2 \ldots) F[x]) \longmapsto (store\ (sf_1 \ldots (x\ v)\ sf_2 \ldots)\ F[v])$$

set!

$$(store\ (sf_1 \ldots (x\ v_1)\ sf_2 \ldots)\ F[(set!\ x\ v_2)]) \longmapsto (store\ (sf_1 \ldots (x\ v_2)\ sf_2 \ldots)\ F[null])$$

set-car!

$$(store\ (sf_1 \ldots (pp\ (cons\ v_1\ v_2))\ sf_2 \ldots)\ F[(set\text{-}car!\ pp\ v_3)]) \longmapsto$$
$$(store\ (sf_1 \ldots (pp\ (cons\ v_3\ v_2))\ sf_2 \ldots)\ F[null])$$

set-cdr!

$$(store\ (sf_1 \ldots (pp\ (cons\ v_1\ v_2))\ sf_2 \ldots)\ F[(set\text{-}cdr!\ pp\ v_3)]) \longmapsto$$
$$(store\ (sf_1 \ldots (pp\ (cons\ v_1\ v_3))\ sf_2 \ldots)\ F[null])$$

Figure 3.3: Control flow, list, and mutation semantics for our subset of Scheme

Application

mark

$$C[(v\ \ldots\ e_1\ e_2\ \ldots)] \longmapsto$$

$$C[((lambda\ (x)\ (v\ \ldots\ x\ e_2\ \ldots))\ e_1)] \quad (x\ fresh, \exists\ e_i\ \in\ e_2\ \ldots,\ e_i \neq v)$$

appN!

$$(store\ (sf\ \ldots)\ F[((lambda\ (x)\ e)\ v)]) \longmapsto$$

$$(store\ (sf\ \ldots (p\ v))\ F[(\{x \to p\}\ (lambda\ ()\ e))]) \quad (p\ fresh,\ x\ assigned\ to\ in\ e)$$

appN

$$C[((lambda\ (x)\ e)\ v)] \longmapsto C[(\{x \to v\}\ (lambda\ ()\ e))] \quad (x\ not\ assigned\ to\ in\ e)$$

app0

$$C[((lambda\ ()\ e))] \longmapsto C[(begin\ e)]$$

Values

promote

$$C[v_1]_* \longmapsto C[(values\ v1)]$$

demote

$$C[(values\ v_1)]_\circ \longmapsto C[v_1]$$

Arithmetic

+0

$$C[(+)] \longmapsto C[0]$$

+

$$C[(+\ n_1\ n_2 \ldots)] \longmapsto C[\ulcorner \Sigma\{n_1, n_2 \ldots\} \urcorner]$$

u−

$$C[(-\ n_1)] \longmapsto C[\ulcorner -n_1 \urcorner]$$

−

$$C[(-\ n_1\ n_2\ n_3 \ldots)] \longmapsto C[\ulcorner n_1\ -\ \Sigma\{n_2, n_3 \ldots\} \urcorner]$$

***1**

$$C[(*)] \longmapsto C[1]$$

***

$$C[(*\ n_1\ n_2 \ldots)] \longmapsto C[\ulcorner \Pi\{n_1, n_2 \ldots\} \urcorner]$$

u/

$$C[(/\ n_1)] \longmapsto C[(/\ 1\ n_1)]$$

/

$$C[(/\ n_1\ n_2\ n_3 \ldots)] \longmapsto C[\ulcorner n_1 / \Pi\{n_2, n_3 \ldots\} \urcorner] \quad (0 \notin \{n_2, n_3 \ldots\})$$

Figure 3.4: Application, value handling, and arithmetic semantics for our subset of Scheme

3.1.2.2 Lists

cons, **car**, and **cdr** handle pairs (and therefore lists) in our language. Notably, **cons** places pairs into the store and returns a pointer to the store location in return. **car** and **cdr** therefore take a pair pointer as their argument to access the first and second values of the pair respectively.

3.1.2.3 Mutation & Reference

var, **set!**, **set-car!**, and **set-cdr!** handle assignment in the language. **var** is a fairly straightforward step that takes a variable referring to a store location and returns the value at that location. The assignment functions replace values in a store location in a similar way. One thing to note is that **set!** requires that its variable argument already exists in the store. As we are not considering top-level variables (see Section 3.1.2.6), **set!** can only modify existing store locations created during lambda application.

3.1.2.4 Application

mark isolates un-evaluated sub-expressions from an application. By lifting an expression into a single application, either the **appN** or **appN!** will then be applicable. Note that our version of **mark** enforces a left to right evaluation order.

appN! is the application semantic rule for lambda expressions that contain assignment in their bodies. We use the curly brace notation to mean capture-free substitution. Since naive assignment into such lambda expressions would result in erroneous expressions like (set! 4 5), **appN!** first creates a fresh store location containing the value to be substituted and then performs substitution with a pointer to that location rather than with the value itself. Then, assignments inside the lambda expression's

body can properly evaluate. The **appN** rule applies naive substitution in lambda expressions that do not contain assignment. As we will see in Section 4.1, excluding these non-assigned variables from the store is integral to our proof of correctness of the compiler pass we are considering.

3.1.2.5 Values & Arithmetic

Finally the **promote** and **demote** rules deal with wrapping values to satisfy the requirements of their evaluation contexts. While this feature is vestigial in our semantics, we include it for purposes of extensibility.

The following rules present a fairly typical system for performing integer arithmetic. Note the side conditions for ensuring no division by zero takes place.

3.1.2.6 Excluded Features

In the interest of shortening a potentially extensive proof, we chose to remove many features from the formal R6RS semantics. Additionally, some features of Scheme were not originally included in the formal R6RS semantics, likely due to complexity of formalization.

The features we chose to exclude were: quote, multiple argument lambda expressions (and generally multiple values as arguments), exceptions, equivalence testing, call/cc and dynamic wind, and letrec. I/O, the macro system, and the numerical tower were originally excluded from the R6RS semantics. While this list certainly contains major features of Scheme, we believe we have captured enough of the language to accurately represent and prove correctness of our transformation.

Another quirk of the formal R6RS semantics is that evaluation order of expressions is left up to the implementation. However, this means that without modification, the semantics are non-deterministic — application of their version of the **mark** rule results in a set of steps each representing a different choice of evaluation order. In our semantics, we modify **mark** to follow a left-to-right evaluation order. While our transformation shouldn't be affected by a different evaluation order, it is theoretically possible that an implementation of Scheme could enforce an evaluation order that would invalidate our results. Finally, while set! expressions that mutate top-level variables are present in Scheme, these top-level set! expressions are converted to "set-top-level!" expressions in a pass prior to the one we are considering. Therefore, we can assume that all set! terms we encounter refer to variables in their local scope. Further, handling top-level variables in general turned out to be a major difficulty in modeling this pass. Since top-level variables are not affected by this pass, we are not including them in our subset. Therefore, we do not include letrec, as this is how top-level variables are implemented by the semantics.

Chapter 4

PROVING CORRECTNESS OF *CONVERT-ASSIGNMENTS*

While the entire Chez Scheme compiler translates Scheme to machine code, the *convert-assignments* pass performs a single intermediate step in this process by performing a Scheme-to-Scheme transformation on some expressions. As its name suggests, the purpose of the *convert-assignments* pass is to *convert* variable *assignment* expressions to a different form.

We will first review the pass itself and the transformation that it performs. Then, we will define our notion of semantic preservation, and use some properties of the semantics as well as some relations defined atop it to show that the transformation does indeed preserve the semantics of a program.

4.1 The *convert-assignments* pass

4.1.1 Assumptions

One important note about *convert-assignments* is that it uses special, decorated expressions in both its source and target languages. Since we do not have a formal framework for asserting the meaning of these decorated Chez Scheme expressions, we make the critical assumption that they abide to the relevant rules of the R6RS formal semantics. From observation, this assumption seems to hold true, as the decorated expressions can be consistently *erased* down to normal Scheme expressions (see Figure 4.1 for an example of a term in the source language of *convert-assignments* compared to its *erased* version in the language defined by the formal semantics).

```
(case-lambda
  [clause
   ()
   0
   (case-lambda
     [clause
      (#{x lqgmu8nsqgazn77h7whlt2wtp-6})
      1
      (begin
        (set! #{x lqgmu8nsqgazn77h7whlt2wtp-6} '3)
        (#[#{primref a0xltlrcpeygsahopkplcn-2} + 19834 (-1)]
          '4
          #{x lqgmu8nsqgazn77h7whlt2wtp-6}))])])

(lambda (x) (begin (set! x 3) (+ 4 x)))
```

Figure 4.1: Example of *erasing* Chez Scheme term decoration.

Another area where we must make an assumption is in developing a model of *convert-assignments* to reason about. The code for actually executing the pass contains and operates on expressions that are not covered by the R6RS semantic model. Therefore, we attempt to faithfully recreate the effects of the transformation on the expressions in subset of the formal semantics that we are considering. Our analogous *convert-assignments* function was created through careful examination of the source code and observation of input and output of the pass for various examples.

4.1.2 Intuition

The *convert-assignments* pass removes indefinite-extent assignments and replaces them with assignments that have a defined scope. This pass is quite small, defined in about 40 lines in the production Chez Scheme compiler. The size of our proof and accompanying frameworks relative to the size of the transformation is a testament to the difficulty of verifying compilation correctness.

The intuition for the pass is best acquired through observing an example of a transformation on a small program:

ca_{prog}(store ((x v_1)) (lambda (y)
(begin
(set! y v_2)
(+ x y)))) =
(store ((pp_x (cons $\hat{v_1}$ null))) (lambda (t)
((lambda (y)
(begin
(set-car! y $\hat{v_2}$)
(+ (car pp_x) (car y))))
(cons t null))))

Where $\hat{a}$ means the appropriate *convert-assignments* function applied to a.

Figure 4.2: Motivating example for the *convert-assignments* pass.

We can immediately see the primary action of the pass — to remove set! expressions. We can see that the set! expression inside the begin expression is transformed to a set-car! expression. Notice also the recursive call on the value of the set! expression. Depending on if a variable is in the store or enclosed in a lambda, we may transform it into a pointer or not, as shown by the transformation that occurs to the variables x and y respectively.

We can also see how lambda expressions are modified to support the transformation of set! expressions. Since its arguments are now assumed to be cons cells, we have to first wrap its arguments inside of a list. This is also the reason we do not change y to pp_y, as at this point it is still an argument to the lambda expression.

Finally, we can see how *ca* deals with store locations of partially evaluated programs. Since the **appN!** rule is the only one that can create new non-list store locations, we can assume that variables in the store correspond to previously evaluated lambda abstractions. Therefore, we must treat variables and set! expressions that refer to store locations as transformation targets. We also retroactively convert these

previously evaluated store locations. In this case, we do change x to pp_x, so that expressions that contain it can properly evaluate.

4.1.3 Definition

Figure 4.3 defines our model of the transformation as functions on the syntax of our language. For notational brevity, we assume that each function has access to the store and a list of all names that are the target of a set! in the program.

Definition 1. $\boldsymbol{ca_{prog}}$ — *Programs*
$ca_{prog}((store\ (sf \ldots)\ e)) = (store\ (ca_{sf}(sf) \ldots)\ ca_e(e))$

Definition 2. $\boldsymbol{ca_{sf}}$ — *Store Locations*
$ca_{sf}((x\ v)) = (pp_x\ (cons\ ca_e(v)\ null))$
$ca_{sf}((pp\ (cons\ v_1\ v_2))) = (pp\ (cons\ ca_e(v_1)\ ca_e(v_2))$

Definition 3. $\boldsymbol{ca_e}$ — *Expressions*
$ca_e((set!\ x\ e)) = (set\text{-}car!\ pp_x\ ca_e(e))$
where x is in the store.
$ca_e((set!\ x\ e)) = (set\text{-}car!\ x\ ca_e(e))$
where x is not in the store.
$ca_e(x) = (car\ pp_x)$
where x is in the store.
$ca_e(x) = (car\ x)$
where x is not in the store, but is assigned to somewhere in the program.
$ca_e(x) = x$
where x is not in the store or assigned to.
$ca_e((lambda\ (x)\ e)) = (lambda\ (t)\ ((lambda\ (x)\ ca_e(e))\ (cons\ t\ null)))$
where x is assigned to, t is fresh.
$ca_e((lambda\ (x)\ e)) = (lambda\ (x)\ ca_e(e))$
where x is not assigned to.
$ca_e((begin\ e_1\ e_2\ \ldots)) = (begin\ ca_e(e_1)\ ca_e(e_2)\ \ldots)$
$ca_e((e_1\ e_2\ \ldots)) = (ca_e(e_1)\ ca_e(e_2)\ \ldots)$
$ca_e((if\ e_1\ e_2\ e_3)) = (if\ ca_e(e_1)\ ca_e(e_2)\ ca_e(e_3))$
$ca_e((values\ v)) = (values\ ca_e(v))$
All other expressions are unchanged by ca_e.

Finally, we define ca_{ctx} for use in applying the pass to decomposed programs.

Definition 4. $\boldsymbol{ca_{ctx}}$ — *Evaluation Contexts*
$ca_{ctx}((store\ (sf\ \ldots)\ F_*)) = (store\ (ca_{sf}(sf)\ \ldots)\ ca_{ctx}(F_*))$
$ca_{ctx}((v_1\ \ldots\ F_\circ\ v_2\ \ldots)) = (ca_e(v_1)\ \ldots\ ca_{ctx}(F_\circ)\ ca_e(v_2)\ \ldots)$
$ca_{ctx}((if\ F_\circ\ e_1\ e_2)) = (if\ ca_{ctx}(F_\circ)\ ca_e(e_1)\ ca_e(e_2))$
$ca_{ctx}((set!\ x\ F_\circ)) = (set\text{-}car!\ pp_x\ ca_{ctx}(F_\circ))$
where x is in the store.
$ca_{ctx}((set!\ x\ F_\circ)) = (set\text{-}car!\ x\ ca_{ctx}(F_\circ))$
where x is not in the store.
$ca_{ctx}((begin\ F_*\ e_1\ e_2\ \ldots)) = (begin\ ca_{ctx}(F_*)\ ca_e(e_1)\ ca_e(e_2)\ \ldots)$
$ca_{ctx}([\ ]) = [\ ]$
$ca_{ctx}([\ ]_*) = [\ ]_*$
$ca_{ctx}([\ ]_\circ) = [\ ]_\circ$

Figure 4.3: *convert-assignments* functions

4.1.4 Lemmas

Here we prove some properties of the *convert-assignments* functions that are necessary for our later proofs.

Lemma 1. *Transformation Preserves Values*

If v is a value, then $ca_e(v)$ *is a value.*

Proof. By induction on the structure of our value expressions.

For most values, ca_e makes no changes. For these cases, $ca_e(v)$ is trivially a value.

The only value expression which is modified by ca_e is $(lambda\ (x)\ e)$, where x is the target of a set! inside of e. We can observe the transformation:

$ca_e((lambda\ (x)\ e)) = (lambda\ (t)\ ((lambda\ (x)\ ca_e(e))\ (cons\ t\ null)))$

We can clearly see that this is still a value, since it is of the form $(lambda\ (t)\ e')$, where $e' = ((lambda\ (x)\ ca_e(e))(cons\ t\ null))$

Therefore, we have shown that ca_e preserves the value status of values that it transforms. □

Lemma 2. *Transformation Preserves Expressions*

If e is a non-value expression, then $ca_e(e)$ *is a non-value expression.*

Proof. The non-value expressions that get transformed beyond recursive calls on subexpressions (which trivially preserves non-value status) are the following:

1. x

2. $(set!\ x\ v)$

$ca_e(x)$ transforms to either $(car\ x)$, $(car\ pp_x)$, or just x, all of which are still expressions.

$(set!\ x\ v)$ similarly transforms to either $(set\text{-}car!\ x\ \hat{v})$ or $(set\text{-}car!\ pp_x\ \hat{v})$, which are both still expressions.

Therefore, if e is a non-value expression, $ca_e(e)$ will always be a non-value expression.

□

Lemma 3. *Transformation Preserves Decomposition*

$ca_{prog}(C[e]) = ca_{ctx}(C)[ca_e(e)]$

for all evaluation contexts C and expressions p.

Proof. We proceed by induction on the structure of our evaluation contexts.

Consider the cases of C: $(store(sf...)F_*)$

and the cases of F_*: $[\,]_*$, F.

If F_* is a hole, then we must show that

$ca_{prog}((store(sf...)\ e)) = ca_{ctx}((store(sf...)[(ca_e(e))]))$

which follows from the definitions of ca_{prog} and ca_{ctx}.

Otherwise, consider the cases of F: $[\,]$, $(v \ldots F_\circ\ v \ldots)$, $(if\ F_\circ\ e\ e)$, $(set!\ x\ F_\circ)$, and $(begin\ F_*\ e\ e \ldots)$.

$[\,]$ follows by definitions as before.

In the case of $(v \ldots F_\circ \; v \ldots)$, $(if \; F_\circ \; e \; e)$, $(begin \; F_* \; e \; e \ldots)$, ca_{ctx} makes no structural changes to the context other than the expected calls to ca_{ctx} and ca_e on subexpressions. Therefore, it follows from our induction principle and the definitions of ca_{ctx}, ca_e that the transformation preserves decomposition in these cases.

The interesting case, $(set! \; x \; F_\circ)$, has a transformation occur in the evaluation context. There are two possibilities for the transformation of $(set! \; x \; F_\circ)$:

1. $ca_{ctx}((set! \; x \; F_\circ)) = (set\text{-}car! \; pp_x \; ca_{ctx}(F_\circ))$ if x is in the enclosing store.
2. $ca_{ctx}((set! \; x \; F_\circ)) = (set\text{-}car! \; x \; ca_{ctx}(F_\circ))$ if x is not in the store (i.e., if x is bound by a lambda expression in some enclosing context).

In case 1, we need to show that:

$$ca_{prog}((store \; (sf \ldots) \; (set! \; x \; F_\circ[p]))) = ca_{ctx}((store \; (sf \ldots) \; (set! \; x \; ca_{ctx}(F_\circ)[ca_e(p)])))$$

Since we know x must be in the store,

$$ca_{prog}((store \; (sf \ldots) \; (set! \; x \; F_\circ[p]))) = (store \; (ca_{sf}(sf) \ldots) \; (set\text{-}car! \; pp_x \; ca_e(F_\circ[p]))).$$

This case follows by definition of ca_e and ca_{ctx}, and by our induction principle.

Case 2 follows similarly, except that x is not in the store, so

$$ca_{prog}((store \; (sf \ldots) \; (set! \; x \; F_\circ[p]))) = (store \; (ca_{sf}(sf) \ldots) \; (set\text{-}car! \; x \; ca_e(F_\circ[p]))).$$

Therefore, we have shown that our transformation preserves decomposition for all cases. □

Lemma 4. *Transformation Preserves Directly Stuck Expressions*

If e is a directly stuck expression, the $ca_prog(e)$ is as well.

Where directly stuck expressions are one of the following:

- *(set! x v) where x is not in the store.*
- *(set-car! pp v) where pp is not in the store.*
- *(set-car! e v) where e is not a pair pointer.*
- *An application expression not of the form of one of our semantic rules.*

Proof. By case analysis on directly stuck expressions.

We first consider the set! case. Here, x will transform to pp_x. However, since x was not in the store, there will be no corresponding pp_x in the store. Therefore, the expression is directly stuck by definition.

Next, in the first set-car! case, no effect other than a recursive call on v will occur. Therefore, it is still directly stuck.

For the second set-car! case, there is no transformation that will take e to a pair pointer. Therefore, this expression will still be directly stuck.

Finally, for application expressions, we can see by lemmas 1 and 2 that the application will still be the same mix of values and non-values. In the case where the application is stuck and has non-value expressions, this preservation is enough to see that it will still be stuck. In the value case, we have to ensure that the values aren't transformed in a way that makes a step possible. We can see by our proof of lemma 1 that values keep their same form (i.e., lambdas are still lambdas). Therefore, if an application was already stuck, then it will continue to be stuck after transformation. □

4.2 Proof Overview

To prove that ca_{prog} preserves the semantics of programs that it transforms, we must consider what we mean by semantic preservation. Since ca_{prog} entirely removes an expression, set!, we cannot expect the transformed program to take *exactly* the same steps. It seems reasonable, then, to say that a program's semantics are preserved over a transformation if the transformed program takes *equivalent* steps.

To formalize this informal definition, we must first define a way for a program to take multiple steps:

Definition 5. *Transitive, reflexive closure of* $\longmapsto$

1. $a \longmapsto\!\!\!\!\!\rightarrow c$ *if* $a \longmapsto\!\!\!\!\!\rightarrow b$ *and* $b \longmapsto\!\!\!\!\!\rightarrow c$ *(Transitivity)*
2. $a \longmapsto\!\!\!\!\!\rightarrow a$ *(Reflexivity)*
3. $a \longmapsto\!\!\!\!\!\rightarrow b$ *if* $a \longmapsto b$ *(Closure)*

The intuition for the reflexive, transitive closure of $\longmapsto$ is that it relates expressions that have zero or more single steps between them.

Our definition of step *equivalence* uses the idea of simulation. That is, if a relation is a simulation relation, then related programs take equivalent semantic steps. The intuition for such a relation is shown by Figure 4.4. We formally define our notion of a simulation relation in Section 4.2.2.

If we can show that ca_{prog} is a simulation, then we know that P and $ca_{prog}(P)$ take equivalent steps and therefore that ca_{prog} preserves the semantics of the programs it transforms.

As a summary, our definitions so far are as follows:

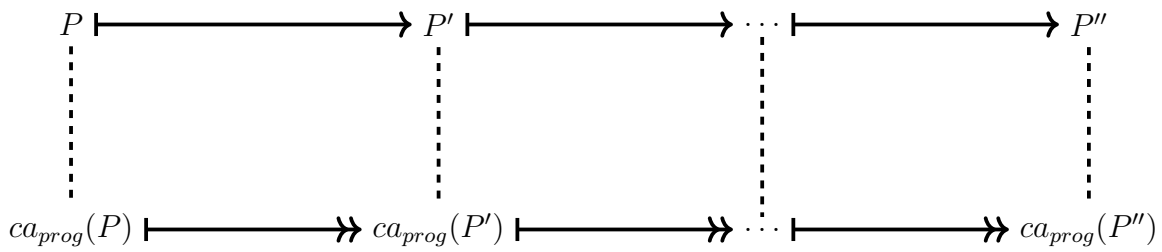

Figure 4.4: Simulation relation visualization

Symbol	Definition
$\longmapsto$	A single, standard, reduction step defined by the semantics.
$\longmapsto\!\!\!\!\!\rightarrow$	The reflexive, transitive closure of $\longmapsto$
ca_{prog}	ca on programs.
ca_{ctx}	ca on evaluations contexts.
ca_{sf}	ca on store locations.
ca_{e}	ca on expressions.

Table 4.1: Definitions

Our proof is structured as follows:

1. Show that our semantics is deterministic (Section 4.2.1).

2. Prove that ca_{prog} is a simulation relation (Section 4.2.2).

4.2.1 Deterministic Semantics

An implicit assumption made by Figure 4.4 is that our programs only have one potential step to take. That is, that a program P will always step to the same P'. Another way of stating this is by saying that our semantics are *deterministic*.

Definition 6. *Deterministic Relation*

If a relation R is deterministic then,

$\forall a\ b_1\ b_2, a\ R\ b_1, a\ R\ b_2 \Rightarrow b_1 = b_2.$

In the context of our semantics, we state this theorem as follows:

Theorem 1. ***Single Step Deterministic***

$\forall P\ P_1\ P_2,\ P \longmapsto P_1,\ P \longmapsto P_2\ \Rightarrow P_1 = P_2$

To prove Theorem 1, we will first show a property of all programs called the VSR property. This property states that P is either a **V**alue, **S**tuck, or **R**educible. In the context of our semantics, we say that that P is reducible if it can be decomposed into a context C and expression e such that $C[e] \longmapsto C'[e']$ for some $C'[e']$ on the right hand side of a semantic step. To show that our semantics are deterministic, we strengthen this property to require uniqueness of the evaluation context C that P decomposes into.

Lemma 5. *VSR Property for Programs*

All programs P are one of the following:

1. $P = (store\ (sf\ \ldots)\ v)$ — *Value*
2. $\exists$ *unique* $C,\ P = C[e]$, *where e is a directly stuck expression* — *Stuck*
3. $\exists$ *unique* $C,\ P = C[e], C[e] \longmapsto C'[e']$ — *Reducible*

where directly stuck expressions are one of the following:

- *(set! x v) where x is not in the store.*
- *(set-car! pp v) where pp is not in the store.*
- *(set-car! e v) where e is not a pair pointer.*
- *An application expression not of the form of one of our semantic rules.*

Proof.

By definition, P must be of the form $(store\ (sf\ \ldots)\ e)$.

We can see that the VSR property holds for a program if the following is true about the program's expression:

If e is a value, then $P = (store\ (sf\ \ldots)\ v)$.

If e is directly stuck, then P is stuck by definition.

Finally, if $e = F[e']$ such that e' is a reducible expression, then $C[e] = C[F[e']]$ has a valid semantic step. We refer to this as e being *reducible*.

We proceed by induction on e.

If e is a value, then we are done.

Otherwise, if e is not a value, consider the cases:

Case 1: $e = (begin\ e_1\ e_2\ \ldots)$

We know by induction hypothesis that all e_i in the body of the begin expression are either values, directly stuck, or reducible.

Consider when $e_2\ \ldots$ is empty. Therefore, $e = (begin\ e_1)$. Hence, we can directly apply the **beginD** rule. Therefore, e is reducible, and the unique context is simply the store and an empty hole.

If $e_2\ \ldots$ is not empty, then consider the case where e_1 is a value. Then, we can decompose e into $F[v]$, where $F = (begin\ [\ ]_*\ e_2\ e_3\ \ldots)$. The **promote** rule is applicable here, and there is no other decomposition we can perform on e. Therefore, this case falls under reducible. Next, consider when e_1 is stuck. Similar to the value case, the evaluation context for begin applies to e, but since e_1 is not a value,

promote is not applicable. Therefore, we are stuck, since P can only decompose into a begin evaluation context and e_1, which is directly stuck. Finally, consider the case where e_1 is reducible. Since e_1 is not a value, **promote** again cannot apply. Then, we can decompose e into $F[e_1']$, where $F = (begin\ [\ ]\ e_2\ e_3\ \ldots)$. Since **promote** and **beginD** are not applicable, F is the unique decomposition, with whichever reduction step $F[e_1']$ takes being the only applicable step.

Case 2: $e = (if\ e_1\ e_2\ e_3)$

If e_1 is a value, then either **ifT** or **ifF** are applicable, and e is clearly reducible, with the unique context just being the store and an empty hole.

Otherwise, if e_1 is stuck and not a value, then P is stuck since it can only decompose using the if evaluation context.

Similarly, if e_1 is reducible, then e is reducible:

$e = F[e_1]$, where $F = (if\ [\ ]\ e_2\ e_3)$.

Case 3: $e = (e_1\ e_2\ \ldots)$

Again, we know that all e_i in the body of e are either values, directly stuck, or reducible.

Consider when all e_i are values. Then, there can be various rules applied, depending on the specific value of each e_i: **cons, car, cdr, set-car!, set-cdr!, appN!, appN, app0** and all of the arithmetic rules are potentially applicable here. If none of these rules are applicable, then e is stuck by definition.

If all of e_i are stuck non-values, then e can take the **mark** step and is therefore reducible. The same follows if all of e_i are reducible non-values, except it will continue to take steps after.

Now, consider if e_i are a mix of stuck, reducible, and value expressions. There are three possible cases here:

1. e can be decomposed into $F[e']$, where $F = (v\ \ldots\ F_\circ\ v\ \ldots)$.
2. The **mark** rule is applicable.
3. e is directly stuck.

We know this because **mark** is the only semantic rule applicable to $(e_1\ e_2\ \ldots)$ expressions with non-value sub-expressions. However, due to the requirement of **mark** that $e_2 \ldots$ contains a non-value expression, it is never the case that **mark** can be applicable and e can be decomposed using the $(v\ \ldots\ F_\circ\ v\ \ldots)$ evaluation context at the same time. If **mark** is not applicable, and a decomposition into the $(v\ \ldots\ F_\circ\ v\ \ldots)$ evaluation context is not possible, then e is directly stuck by definition.

Therefore, consider the possibilities of e' in the decomposition case. e' cannot be a value, since then all of e_i would be values, which contradicts our assumption. If e' is reducible by some semantic step, then that is the semantic step that e must take. Otherwise, if e' is stuck, then e is stuck, since all other e_i are values.

Case 4: $e = (set!\ x\ e_1)$

If x is not in the store, e is directly stuck by definition, so we consider the case where x is in the store.

By induction principle, e_1 must be a value, directly stuck, or reducible.

If e_1 is a value, the **set!** rule is applicable.

Otherwise, if not, e is directly stuck if e_1 is, or reducible if e_1 is, with the $(set!\ x\ F_\circ)$ evaluation context being the only possible decomposition, and therefore unique.

Case 5: $e = (values\ v)$

If e is in a $[\ ]_\circ$ context, then **demote** is applicable, and e is therefore reducible. Otherwise, it is stuck.

Case 6: $e = x$

If x is in the store, then **var** is applicable, and e is therefore reducible. Otherwise, e is directly stuck by definition.

By showing this property for all cases of P and e, we have proven it is true for all programs by induction principle. □

Since we have shown that if a program reduces, it does so by decomposing uniquely into an evaluation context and a reducible expression, Theorem 1 follows from Lemma 5. That is, if $P = C[e]$, $P \longmapsto P_1$, and $P \longmapsto P_2$, then $P_1 = P_2 = C'[e']$ by Lemma 5.

4.2.2 ca_{prog} is a simulation relation

In this section we show that the relation defined by the *convert-assignments* pass is a simulation relation. First, we formally define our notion of simulation:

Definition 7. ***Simulation Relation***

If a relation R is a simulation, and aRb, then:

$a \longmapsto a' \Rightarrow \exists\ b', b \longmapsto\!\!\!\!\!\rightarrow b', a'Rb'$

That is, if a and b are related by a simulation relation, and $a \longmapsto a'$, then there must exist b' such that $b \longmapsto\!\!\!\!\!\rightarrow b'$ and that a' is related to b'. Thus, to show that ca is a

simulation relation, we must show that if $P \longmapsto P'$, then there exists a P^* such that $ca_{prog}(P) \longmapsto\!\!\!\!\!\rightarrow P^*, ca_{prog}(P') = P^*$.

Therefore, if we can show that for each possible single semantic step, that transforming the left hand side and then stepping a finite number of times always arrives at the transformation of the right hand side, then we have shown that ca_{prog} is a simulation relation.

One thing to note is that since we are considering the decomposed programs that $\longmapsto$ operates on, we make implicit use of Lemma 1 throughout our proof to relate these findings back to ca_{prog}.

Theorem 2. ***Step Theorem***

$P \longmapsto P'$ *implies*
exists P^* *such that*
$ca_{prog}(P) \longmapsto\!\!\!\!\!\rightarrow P^*$ *and*
$ca_{prog}(P') = P^*$.

Proof. By induction on the structure of P.

By Lemma 5, P must either a *Value*, *Stuck*, or *Reducible*.

If P is a *Value*, our step theorem follows by contradiction.

If P is *Stuck*, our step theorem follows similarly, but we are also ensured by Lemma 4 that $\hat{P}$ is also *Stuck*

Otherwise, if it is *Reducible*, P must be in the form of the left hand side of one of the semantic rules. Our proof follows by case analysis of the semantic rules.

The most interesting example comes from the rule **appN!**. This is because **appN!** operates on lambda applications that contain assignments, and ca_{prog} removes all assignments. Indeed, we will confirm from the steps that correspond to **appN!** that assignments are removed.

First, we define some convenient notation in Table 4.2:

$$\hat{P} = ca_{prog}(P)$$
$$\hat{C} = ca_{ctx}(C)$$
$$\hat{sf} = ca_{sf}(sf)$$
$$\hat{e} = ca_{e}(e)$$

Table 4.2: Notation for various ca functions

Now, consider the semantic step **appN!**, taking P to P':

$$P = (store\ (sf \ldots)F[((lambda\ (x)\ e)\ v)]) \longmapsto$$

$$P' = (store\ (sf \ldots (p\ v))\ F[(\{x \to p\}\ (lambda\ ()\ e))]) \quad (p\ fresh,\ x\ assigned\ to\ in\ e)$$

P transforms to $\hat{P}$:

$$\hat{P} = (store\ (\hat{sf} \ldots)\hat{F}[((lambda\ (t)\ ((lambda\ (x)\ \hat{e})(cons\ t\ null))\ \hat{v}))]) \quad (t\ fresh)$$

P' transforms to $\hat{P'}$:

$$\hat{P'} = (store\ (\hat{sf} \ldots (pp\ (cons\ \hat{v}\ null))\hat{F}[(\{x \to pp\}\ (lambda\ ()\ \hat{e}))])$$

If ca is a simulation, $\hat{P}$ must take a finite amount of steps and arrive at a P^* such that $P^* = \hat{P'}$.

We propose that $\hat{P}$ will take 5 steps to get to P^*.

1. We first apply the **appN** rule to $\hat{P}$, since t is fresh.

$$(store\ (\hat{sf}\ldots)\hat{F}[((lambda\ (t)\ ((lambda\ (x)\ \hat{e})(cons\ t\ null))\ \hat{v}))])\longmapsto$$

$$(store\ (\hat{sf}\ldots)\hat{F}[((lambda\ ()\ ((lambda\ (x)\ \hat{e})(cons\ \hat{v}\ null))))])$$

2. Then, we apply **app0**.

$$(store\ (\hat{sf}\ldots)\hat{F}[((lambda\ ()\ ((lambda\ (x)\ \hat{e})(cons\ \hat{v}\ null))))])\longmapsto$$

$$(store\ (\hat{sf}\ldots)\hat{F}[(begin\ ((lambda\ (x)\ \hat{e})(cons\ \hat{v}\ null)))])$$

3. Now, we apply **beginD**.

$$(store\ (\hat{sf}\ldots)\hat{F}[(begin\ ((lambda\ (x)\ \hat{e})(cons\ \hat{v}\ null)))])\longmapsto$$

$$(store\ (\hat{sf}\ldots)\hat{F}[((lambda\ (x)\ \hat{e})(cons\ \hat{v}\ null))])$$

4. At this point, to apply the **cons** rule, we must first decompose our program into an evaluation context and a reducible expression. We do so as follows:

$$(store\ (\hat{sf}\ldots)\hat{F}[((lambda\ (x)\ \hat{e})(cons\ \hat{v}\ null))])=$$

$$(store\ (\hat{sf}\ldots)\hat{F_+}[(cons\ \hat{v}\ null)])$$

where $\hat{F_+} = \hat{F}[((lambda\ (x)\ \hat{e})[\ ])]$.

We can then reduce using the **cons** rule:

$$(store\ (\hat{sf}\ \ldots)\hat{F}_+[(cons\ \hat{v}\ null)]) \longmapsto$$

$$(store\ (\hat{sf}\ \ldots(pp\ (cons\ \hat{v}\ null)))\hat{F}_+[pp])$$

And expand our context:

$$(store\ (\hat{sf}\ \ldots(pp\ (cons\ \hat{v}\ null)))\hat{F}_+[pp]) =$$

$$(store\ (\hat{sf}\ \ldots(pp\ (cons\ \hat{v}\ null)))\hat{F}[((lambda\ (x)\ \hat{e})\ pp)]$$

5. Finally, we can apply **appN** again, since pp is fresh.

$$(store\ (\hat{sf}\ \ldots(pp\ (cons\ \hat{v}\ null)))\hat{F}[((lambda\ (x)\ \hat{e})\ pp)]) \longmapsto$$

$$(store\ (\hat{sf}\ \ldots(pp\ (cons\ \hat{v}\ null)))\hat{F}[(\{x \to pp\}\ (lambda\ ()\ \hat{e}))])$$

Comparing this result to $\hat{P'}$, we can see they are the same. Therefore, if P takes step **appN!**, $\hat{P}$ takes the steps **appN, app0, beginD, cons, appN** in order to arrive at a P^* that satisfies our simulation definition. Figure 4.5 visualizes this relationship.

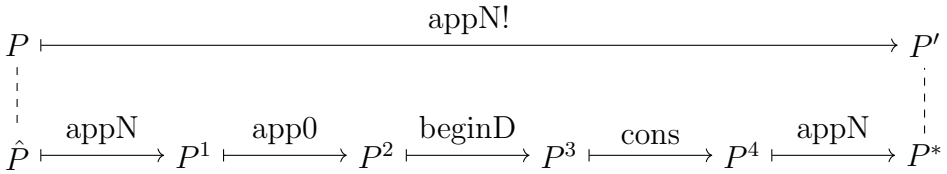

Figure 4.5: Simulation of appN!

This is the only case where more than one equivalent step is needed. In the cases of **var** and **set!**, $\hat{P}$ takes a *different* step, but still only one (see Figure 4.6).

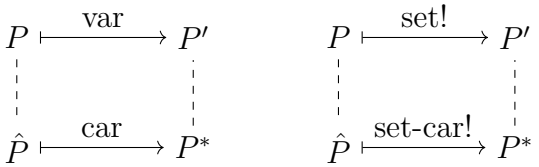

Figure 4.6: Simulation of var and set!

For all other steps, since the *ca* functions do not affect the LHS in a meaningful way, $ca_{prog}(P)$ takes the same step that P does.

This relationship is shown in Figure 4.7:

Semantic Step	Steps for $\hat{P}$ to reach P^*
appN!	appN, app0, beginD, cons, appN
var	car
set!	set-car!
All other steps	Same step

Figure 4.7: Equivalent semantic steps before and after *convert-assignments*.

Therefore, we have shown that ca_{prog} is a simulation relation. □

4.2.3 ca_{prog} is semantic preserving

Now that we know that ca_{prog} is a simulation relation, we can see that it preserves semantics.

For example, if $P \longmapsto\!\!\!\!\!\rightarrow P'$, then by Theorems 1 & 2 and induction on the number of steps taken by P, we can easily see that $ca_{prog}(P) \longmapsto\!\!\!\!\!\rightarrow ca_{prog}(P')$. Therefore, if P gets stuck, gets to a value, or continues infinitely, $ca_{prog}(P)$ will take equivalent steps and get to the transformed version of the stuck expression, of the value expression, or of an arbitrary expression in the infinite sequence.

Chapter 5

VALIDATION FRAMEWORKS

In Chapter 4, we gave our "paper" proof of the correctness of the *convert-assignments* pass. However, we still need to verify that our proof is correct. While we have carefully checked our assumptions and the overall chain of logic of our paper proof, it is always possible to make mistakes or overlook logical flaws when manually reviewing. Fortunately, as covered in Section 2.1.1.1, there are powerful proof assistants available that can aid in *mechanizing* proofs such as ours. In addition, we have the luxury of having access to an existing implementation of the R6RS semantics that allows us to test examples to ensure that ca_{prog} satisfies the requirements of being a simulation relation.

Our initial goal for this project was to provide a full formal mechanization of our proof using the Coq proof assistant. However, the advanced detail required made a complete mechanization out of the scope of this thesis work. Still, we successfully implemented a portion of the R6RS semantics as well as a model of the pass itself in Coq. We will show later in the section some details of the implementation as well as how it could be extended to complete formalization of our proof, and therefore prove to a high degree of certainty the correctness of the *convert-assignments* pass.

Although we were not able to completely formalize the proof in Coq, we wanted to have empirical evidence of the validity of our proof. To do so, we provide a testing framework based on an existing implementation of the R6RS semantics. We use this testing framework to show that the *convert-assignments* pass is, in fact, a simulation relation for a variety of Scheme programs. We use Scheme itself for this framework, as

it is highly adaptable to modeling syntactic transformations like *convert-assignments*, and we have access to an existing implementation of the R6RS semantics.

5.1 Coq Framework

5.1.1 Overview and Functionality

We provide an implementation of a subset of the R6RS formal semantics as well as a model of the *convert-assignments* pass. In addition, we prove some properties of the semantics, such as the property that substituting a fresh variable in an expression returns that same expression. Whereas in our paper proof, we may assume that this is trivially correct, Coq requires you to explicitly prove the validity of seemingly obvious properties such as this one. For example, at one part in our paper proof of the VSR property for programs (Lemma 5 in Section 4.2.1), we needed to show that a program not in the form of any semantic step is stuck. In our paper proof, we do not provide much detail, as it fairly obvious that a program that does not match a semantic step has no applicable semantic steps. However, in Coq, we would need to explicitly show that if a program not in such a form could step, then it would lead to a logical inconsistency. Further, we would have to show this same contradiction for all of the different possible stuck programs. While Coq provides powerful automation tools that can aid in proving such a property, it is still true that mechanization of a proof at this scale requires an amount of detail that is simply not considered in a paper proof. Therefore, while we have a formal model of our subset of the R6RS semantics and some properties of the semantics verified, there are many more minor properties that need to be proven to be able to formally verify the overall correctness of the pass.

In terms of present functionality, our implementation does provide a way of manually applying semantic steps, and we show some simple examples by proving, for example, that $(car\ (cons\ v_1\ v_2)) = v_2$ for all v_1, v_2. Again, while this is quite a trivial proof on paper, Coq requires explicit evidence of its truth, which we show by stepping through the appropriate semantic steps in a systematic way.

Finally, implementing the semantics and pass in Coq required some specific nuance unique to Coq and the context of programming language metatheory proofs. We discuss these implementation details in Section 5.1.2.

5.1.2 Implementation Details

Our implementation of an operational semantics for Scheme programs follows closely from the implementation included with R6RS — however, encoding this semantics in Coq presents challenges unique to the intricacies of the Coq language.

5.1.2.1 Capture-Free Substitution

Historically, handling variable bindings has been a major hurdle in proofs about programming language metatheory [11]. Specifically, the issue of implementing a substitution operation while avoiding unwanted capture of variables is extremely important, but can be difficult to accomplish in mechanized proofs. If a semantic model uses named variables, the proof authors must now reason about all possible names in every relevant lemma that they prove. Since programming languages tend to allow developers close to free reign with defining names for variables, trying to reason about all possible names in expressions quickly becomes intractable.

In paper proofs, this problem is dealt with by freely applying α-conversion if capture would occur, or simply by using the same variables and assuming that no capture occurs. This means that in paper proofs, and in our minds, we tend to implicitly work with α-equivalence classes of expressions rather than directly with expressions themselves. For example, we naturally consider expressions $(\lambda\ (x)\ x)$ and $(\lambda\ (y)\ y)$ as both equivalent to the identity function.

To remedy the problem of reasoning over all possible names, and more closely emulate our intuition and the typical form of reasoning seen in paper proofs, we use the locally nameless style of variable bindings in our formal model. This style uses de Bruijn indices for bound variables (hence *locally nameless*), and names for free variables. Hence, locally nameless lambda expressions are syntactically equivalent if they are in the same α-equivalence class.

Figure 5.1 shows some simple examples of de Bruijn indexed expressions.

(*lambda* (*bvar* 0)) is the identity function
(*lambda* (*lambda* (+ (*bvar* 0) (*bvar* 1)))) is an addition function.
(*lambda* (+ (*bvar* 0) *x*)) adds its argument to a variable from the store.

Figure 5.1: Some examples of expression representations using the *locally nameless* style.

Notice that lambda expressions have no formal arguments. This is because bound variables are "nameless" in this system. Instead, the index of a bound variable refers to the level of abstraction. So (bvar 0) refers to a variable bound by the immediately surrounding abstraction. Whereas (bvar 1) refers to a variable bound by an abstraction surrounding the immediate abstraction. We use free variables to represent variables that are already in the store, because of the previous evaluation of a lambda abstraction.

While the locally nameless style is convenient for reasoning about lambda calculus terms at a formal level, we must note that Scheme is not defined using the locally nameless style. Therefore, we also need to handle converting between the two. To do so, we replace the bound variables at each level with some fresh name, for example to apply a semantic rule that takes a Scheme expression to another.

Finally, because there exist locally nameless expressions that do not correspond to Scheme expressions (for example the expression (bvar 1) outside of an abstraction), we have to define a well-formedness property for locally nameless expressions, and enforce that translation is only performed on such well-formed expressions.

5.1.2.2 Cofinite Quantification

Cofinite quantification is a small stylistic change to the normal way of defining substitution that allows us to have a slightly more powerful induction principle when dealing with fresh variable substitution.

Traditionally, a fresh variable is generated by simply picking (existential quantification) a single variable not in the set of free variables of the expression it is being substituted in. In the cofinite quantification style, we use universal rather than existential quantification to reason about all fresh variables instead of a single one. That is, our definition of freshness generalizes to say all variables x not in some set L are fresh. In our definition for freshness of a variable with respect to an expression, we say the set L is the set of free variables in the expression. Because of this, where we would reason about a single fresh variable, we instead are reasoning about the set of all fresh variables, which can help with showing that a variable that is fresh in an expression is also fresh in its sub-expressions.

The combination of locally nameless and cofinite quantification styles for formal semantics engineering was pioneered by Aydemir, et al. [4].

5.1.2.3 Step-Indexed Functions

To reason about non-terminating programs while still satisfying the requirement that all Coq functions terminate, we use step indexing on many of our functions. This means that our functions are given an index to keep track of how many "steps" they have taken, with a limit that causes termination after the index exceeds it. This ensures that all of our functions are terminating, while being flexible enough to account for large programs For functions dealing with nested sub-expressions, our step index usually refers to recursion depth rather than actual steps taken, so very large terms can be handled without necessarily setting a large step limit.

5.2 Racket Framework

5.2.1 Overview and Functionality

While our Coq framework approaches validation of our proof from the perspective of static analysis, our Racket framework instead provides a more "dynamic" means of checking validity of our proof against specific examples. Given a valid program in our subset of Scheme, our Racket framework uses an existing implementation of the R6RS formal semantics to perform a semantic step on the program. Then, it executes ca_{prog} on the original program and verifies that stepping a finite number of times (in our case up to 5) corresponds to the result from the single semantic step.

5.2.2 Implementation Details

5.2.2.1 Extensibility

Because we are using the full R6RS semantics here, we can easily extend this framework to test examples outside of the scope of our original proof. All that needs to be done to add new kinds of expressions to the framework is extending the framework's implementation of ca_{prog} to appropriately recurse on these new expression's sub-expressions. While this is not a substitute for a formal proof, one can reasonably assure themselves that our proof technique holds over a set of examples that include such new expressions. By constructing the framework in this way, we continue the pattern of high extensiblity in our reasoning.

5.2.2.2 Evaluation Order

As mentioned in Section 3.1.2.6, we assume a left-to-right evaluation order in our proof. However, as we also mention, the R6RS semantics provides a way for implementations to specify the evaluation order by modifying the **mark** semantic rule. In this framework, we are of course relying on the un-modified **mark** rule, since we are directly applying the formal R6RS semantics. However, we circumvent this by simply choosing the path that corresponds to a left-to-right evaluation in all cases. A possible extension to this framework would be to consider all evaluation orders when testing an example for adherence to our proof technique. However, since we do not prove this in our paper proof, we kept the assumption of left-to-right evaluation for this framework.

```
(store () ((lambda (x) (set! x 5)) 4)) steps to (store ((bp 4)) ((lambda () (set! bp 5))))
Using rule: 6appN!
P^ = (store () ((lambda (t0) ((lambda (x) (set-car! x 5)) (cons t0 null))) 4))
P~^ = (store (((-mp bp) (cons 4 null))) ((lambda () (set-car! (-mp bp) 5))))
P^ steps 5 times to P~^, using the following rules:
        (6appN (store () ((lambda () ((lambda (x) (set-car! x 5)) (cons 4 null))))))
        (6app0 (store () (begin ((lambda (x) (set-car! x 5)) (cons 4 null)))))
        (6begind (store () ((lambda (x) (set-car! x 5)) (cons 4 null))))
        (6cons (store (((-mp x1) (cons 4 null))) ((lambda (x) (set-car! x 5)) (-mp x1))))
        (6appN (store (((-mp x1) (cons 4 null))) ((lambda () (set-car! (-mp x1) 5)))))
```

Figure 5.2: Example of a Racket framework test case

5.2.2.3 Testing Framework

As previously mentioned, the purpose of this framework is to give a means of testing the validity of our proof by simulation approach. An example of this testing is shown by Figure 5.2. It should be noted that we do not perform this validation by manual examination.

Figure 5.2 is a text-based visualization of the process that our testing framework uses to validate our proof approach. In the full version of our test, we step until a limit is reached or no further steps are applicable and test the simulation relation for each step.

One very important detail of this framework can also be observed in Figure 5.2 — that the terms we expect to match are not syntactically equivalent, but instead equal over α-equivalence. To counteract this, in our test suites, we first normalize programs such that they are syntactically equal to all programs in their α-equivalence class before comparing them. That way, we get positive results for comparing programs that are entirely the same except for a difference in naming. However, this also means that implicit shadowing or freshness of variables that have the same name is not allowed in this framework. Indeed, this makes it so that this framework does not typically mesh well with examples that include recursion, such as the Y-combinator.

Chapter 6

CONCLUSIONS AND FUTURE WORK

In this chapter, we conclude and summarize the project, reflect on our process of proof and in building our validation frameworks, and discuss future work on this project.

6.1 Reflections

The original intent of this work was to provide a full mechanization of the proof in Coq. To that end, some of the features removed were done so with the motivation of simplifying the Coq formalization. As we discuss in Section 5.1, Coq requires a large amount of detail and specific proof of things that are seemingly given or obvious in a paper proof. For example, we prove in our framework that substitution of a fresh variable is the identity for the *locally nameless* Scheme representation in our Coq framework, something that we take as given in our paper proof.

As our project's scope shifted to no longer target a full mechanization, some features were added back into our model, but a project started with the intention of a paper proof only likely could have included the entire R6RS semantics, as the features we excluded are largely unaffected by ca_{prog}. While complicated features like quote would have added some complexity to the proof, the *convert-assignments* pass does not modify quoted expressions. Therefore, the only additional complexity would be altering our various definitions to support these new features, for example by performing the correct recursive calls on sub-expressions.

Another feature, exceptions, may have actually aided our proof by giving an even stronger definition for *stuck* programs. However, it is clear to see that *convert-assignments* would behave no differently with this change, so its exclusion presents no threat to the overall validity of our claim of correctness.

6.2 Future Work

There are a few paths for potential future work on this project. The first would be extending the paper proof in one of two ways: extending the language we prove over, and/or proving correctness of another pass.

The second future work would be to continue the original scope of this project in fully mechanizing the current proof. This work would entail proving various lemmas to definitively prove Theorems 1 and 2 inside of our Coq framework. While our framework provides definitions for the syntax and semantics of our R6RS subset and some of the auxiliary lemmas, extending it to fully mechanize our proof would likely take some fine-tuning of the definitions, or more specific definitions of, for example, well-formed expressions, which we assume extensively throughout our paper proof.

6.3 Summary and Closing

In this work, we detailed the proof of correctness of our representation of the *convert-assignments* pass, ca_{prog}. Most of our proof follows by induction on the structure of our programs and expressions, with extensive case analysis on programs formed such that a semantic step is applicable to them. We also provide frameworks in Coq and Racket for validation of our proof through computer-checked proof and testing of specific examples respectively. While we consider a subset of the R6RS semantics

in all of our proofs, each has the possibility of extension to include a more complete version, though this would likely be quite difficult in the case of the Coq framework. Overall, we believe this work provides a reasonably trustworthy proof of correctness of the *convert-assignments* pass of the Chez Scheme compiler.

BIBLIOGRAPHY

[1] IEEE standard for the scheme programming language. *IEEE Std 1178-1990*, pages 1–, 1991.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

[3] A. Anand, A. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Belanger, M. Sozeau, and M. Weaver. Certicoq: A verified compiler for coq. In *The third international workshop on Coq for programming languages (CoqPL)*, 2017.

[4] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. *Acm sigplan notices*, 43(1):3–15, 2008. Publisher: ACM New York, NY, USA.

[5] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix. Using static analysis to find bugs. *IEEE software*, 25(5):22–29, 2008.

[6] H. Barendregt and H. Geuvers. Proof-assistants using dependent type systems. *Handbook of automated reasoning*, 2:1149–1238, 2001.

[7] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, and others. *The Coq proof assistant reference manual: Version 6.1.* PhD Thesis, Inria, 1997.

[8] A. Bove, P. Dybjer, and U. Norell. A brief overview of agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[9] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013.

[10] N. G. d. Bruijn. A Survey of the Project Automath**Reprinted from: Seldin, J. P. and Hindley, J. R., eds., To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, p. 579-606, by courtesy of Academic Press Inc., Orlando. In R. P. Nederpelt, J. H. Geuvers, and R. C. d. Vrijer, editors, *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*, pages 141–161. Elsevier, 1994. ISSN: 0049-237X.

[11] A. Charguéraud. The locally nameless representation. *Journal of automated reasoning*, 49(3):363–408, 2012. Publisher: Springer.

[12] A. Chlipala. A verified compiler for an impure functional language. *ACM Sigplan Notices*, 45(1):93–106, 2010.

[13] A. Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[14] B. J. Copeland. The Church-Turing Thesis. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2020 edition, 2020.

[15] T. Coquand and G. Huet. *The calculus of constructions*. PhD Thesis, INRIA, 1986.

[16] L. de Moura, S. Kong, J. Avigad, F. Van Doorn, and J. von Raumer. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[17] D. Delahaye. A tactic language for the system Coq. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 85–95. Springer, 2000.

[18] R. K. Dybvig. *The Scheme Programming Language, 4th Edition*. The MIT Press, 4th edition, 2009.

[19] R. K. Dybvig et al. Chez scheme, 2011.

[20] R. K. Dybvig and B. T. Smith. *Chez Scheme Reference Manual: Version 1.0*. 1983.

[21] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009.

[22] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt. The racket manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[23] H. Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.

[24] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD Thesis, Éditeur inconnu, 1972.

[25] G. Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[26] J. D. Guttman, J. D. Ramsdell, and V. Swarup. *The VLISP Verified Scheme System*, pages 33–110. Springer US, Boston, MA, 1995.

[27] J. D. Guttman, J. D. Ramsdell, and M. Wand. VLISP: A Verified Implementation of Scheme. In *VLISP A Verfied Implementation of Scheme: A Special Issue of Lisp and Symbolic Computation, An International Journal Vol. 8, Nos. 1 & 2 March 1995*, pages 5–32. Springer US, Boston, MA, 1995.

[28] J. Kang, Y. Kim, Y. Song, J. Lee, S. Park, M. D. Shin, Y. Kim, S. Cho, J. Choi, C.-K. Hur, and K. Yi. Crellvm: Verified Credible Compilation for LLVM. *SIGPLAN Not.*, 53(4):631–645, June 2018. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[29] A. W. Keep and R. K. Dybvig. A nanopass framework for commercial compiler development. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 343–350, 2013.

[30] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. Cakeml: a verified implementation of ml. *ACM SIGPLAN Notices*, 49(1):179–191, 2014.

[31] P. J. Landin. Correspondence between algol 60 and church's lambda-notation: part i. *Communications of the ACM*, 8(2):89–101, 1965.

[32] I. Lee, G. Pappas, R. Cleaveland, J. Hatcliff, B. Krogh, P. Lee, H. Rubin, and L. Sha. High-confidence medical device software and systems. *Computer*, 39(4):33–38, 2006.

[33] X. Leroy. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52(7):107–115, July 2009. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[34] X. Leroy. *The CompCert C verified compiler: Documentation and user's manual*. PhD thesis, Inria, 2019.

[35] A. Lochbihler. Verifying a compiler for java threads. In *European Symposium on Programming*, pages 427–447. Springer, 2010.

[36] P. Martin-Löf. An intuitionistic theory of types. *Twenty-five years of constructive type theory*, 36:127–172, 1998.

[37] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[38] J. Mccarthy and J. Painter. Correctness of a compiler for arithmetic expressions. pages 33–41. American Mathematical Society, 1967.

[39] R. Milner and R. W. Weyhrauch. Proving compiler correctness in a mechanised logic. *Machine Intelligence*, 7:51–73, 1972.

[40] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[41] C. Paulin-Mohring. *Introduction to the calculus of inductive constructions*. College Publications, 2015.

[42] B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. Software foundations. *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html*, 2010.

[43] K. Sandler, L. Ohrstrom, L. Moy, and R. McVay. Killed by code: Software transparency in implantable medical devices. *Software Freedom Law Center*, pages 308–319, 2010.

[44] D. A. Schmidt. Programming language semantics. *ACM Computing Surveys (CSUR)*, 28(1):265–267, 1996.

[45] J. Ŝevčik, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 43–54, 2011.

[46] M. Sperber, R. K. Dybvig, M. Flatt, A. Van Straaten, R. Findler, and J. Matthews. Revised6 Report on the Algorithmic Language Scheme. *Journal of Functional Programming*, 19(S1):1–301, 2009. Publisher: Cambridge University Press.

[47] G. L. Steele Jr and G. J. Sussman. The revised report on scheme: A dialect of lisp. Technical report, Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1978.

[48] G. J. Sussman and G. L. Steele. The First Report on Scheme Revisited. *Higher-Order and Symbolic Computation*, 11(4):399–404, Dec. 1998.

[49] V. Voevodsky. Univalent foundations project. *NSF grant application*, 2010.

[50] R. Wilhelm, H. Seidl, and S. Hack. *Compiler design: syntactic and semantic analysis*. Springer Science & Business Media, 2013.