COINDUCTIVE PROGRAM VERIFICATION

BY

BRANDON MICHAEL MOORE

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Doctoral Committee:

Professor Grigore Roșu, Chair
Research Professor Elsa Gunter
Professor Jose Meseguer
Associate Professor Adam Chlipala, MIT

Abstract

We present a program-verification approach based on coinduction, which makes it feasible to verify programs given an operational semantics of a programming language, without constructing intermediates like axiomatic semantics or verification-condition generators. Specifications can be written using any state predicates.

The key observations are that being able to define the correctness of a style of program specification as a greatest fixpoint means coinduction can be used to conclude that a specification holds, and that the number of cases that need to be enumerated to have a coinductively provable specification can be reduced to a feasible number by using a generalized coinduction principle (based on notions of "coinduction up to" developed for proving bisimulation) instead of the simplest statement of coinduction.

We implement our approach in Coq, producing a certifying language-independent verification framework. The soundness of the system is based on a single module proving the necessary coinduction theorem, which is imported unchanged to prove programs in any language.

We demonstrate the power of this approach by verifying algorithms as complicated as Schorr-Waite graph marking, and the flexibility by instantiating it for language definitions covering several paradigms, and in several styles of semantics.

We also demonstrate a comfortable level of proof automation for several languages and domains, using a common overall heuristic strategy instantiated with customized subroutines. Manual assistance is also smoothly integrated where automation is not completely successful.

To Lisa.

Acknowledgments

Thanks to my advisor, Grigore Roșu whose steadfastly supported and encouraged me throughout the development of this approach. Thanks to all of the members of the FSL group over the course of my studies, and our collaborators from Iasi. And finally, thanks to my wife and parents who assisted me through this long process with advice, child care, and love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Formal verification can justify extremely high confidence in the correctness of software, by allowing the use of automated tools and reusable soundness arguments to greatly reduce the amount of material that must be trustworthy to ensure no program error remains undetected. Ideally most of this *trusted base* consists of components reused for all or most verification tasks, and subject to widespread scrutiny.

An effective formal verification framework needs several components. The central component is a formal proof system, which consists of a language of claims about code, a language of *proofs*, and a description of which proofs are acceptable for which claims, which is sufficiently precise that a program can check whether a proof is acceptable for a given claim. Next, there must be a definition of the meaning of claims in terms of a semantics of the programming language, and a proof of soundness of the proof system, in the sense that an acceptable proof of a claim exists only if the meaning of the claim is true.

Given these ingredients, justified confidence in the truth of a claim does not require looking at the code or even the proof, as long as the soundness proof and the proof checker are trusted, the proof checker accepted the proffered proof, and we can be confident that the proof checker execution was not compromised by hardware errors. In this description we regard proofs quite abstractly. As in a program logic, proofs could be derivations in some familiar looking logic which directly concludes claims, but a verification-condition generation approach also fits, if the task of the proof checker includes checking that the verification conditions were correctly calculated in addition to checking the proofs of the verification conditions.

One element of this vision which is unavailable for many programming languages is a sound proof system. For most programming languages, there is not even a sound proof system. Established approaches such as axiomatic semantics or verification-condition generation require designing a new proof system for each target language, and then proving this new proof system is sound. Merely designing a proof system can be complicated by certain language features, such as mutual recursion and nonlocal control flow.[1] Even the familiar while rule in Hoare logic requires a bit of invention beyond the basic rules of its execution to introduce the idea of a loop invariant. Proving the soundness of proof systems requires a substantial effort, and a relatively deep mathematical background. These traditional approaches outline well-trod paths for developing proof systems for programming languages, but they must still be trod afresh for each new language.

This work presents instead a single language-independent proof system. The core of the approach is a generalized coinduction principle, which can be applied to prove reachability in transition relations (by characterizing the set of true claims as a greatest fixpoint). This can be instantiated with any operational semantics expressed as a transition relation, to give a system for proving that a program in that language meets a specification. This is proven once to be sound with respect to any supplied relation. Despite the generality of our approach, verifying a specification using an application of our theorem can be done with a proof whose shape resembles what might be expected from a custom program logic, with most proof steps closely related to execution of the program.

The relation between our approach and ways of reasoning about programs that cannot be described as an independent verification system is less clear. One of particular relevance to the work presented here is the use of powerful type systems, especially in dependently-typed languages which make logical reasoning available as an integral part of a language. This is in fact the logical foundation of the Coq system which is used for the soundness proofs and certifying implementation of our approach. The validity of relying on these features is based on the soundness of the type system and consistency as a logic (perhaps captured through strong normalization), rather than a more direct connection to operational semantics as in the soundness of a Hoare-logic rule. Unfortunately exploring potential connections or interactions between type-based and coinductive program verification is left for future work. The aim of the current evaluation is to show that our system can be easily applied

[1] An axiomatic semantics for a fragment of Java intentionally selected to include several sorts of complexity is presented in [ON02].

to a variety of languages and provide a satisfactory verification experience for a language previously lacking any, rather than to consider competition or synergy on languages fortunate enough to already have options for verification.

Developing a sound program-verification framework for a new language usually requires an operational semantics in any case, as many approaches to program verification define the meaning of specifications in terms of the behavior of programs, meaning such a system can only be proved sound with respect to an operational semantics. We also believe that developing the first faithful formal semantics for a language requires testing to ensure the formalization agrees with existing behavior. This requiring an executable semantics, which generally means an operational semantics. [2]. As such, we believe most approaches to sound program verification usually require developing an operational semantics in any case. However, by working directly with an operational semantics our approach does not require proving soundness of a proof system with respect to that validated operational semantics, which requires considerably more mathematical sophistication than defining an operational semantics.

Our coinductive program-verification approach can be used with any operational semantics and is then correct-by-construction: no additional "program logic" or soundness proofs are needed to be certain any derivations will be sound with respect to the provided operational semantics.

To use a metaphor from software engineering, Hoare Logic [Hoa69] is a design pattern which describes a generally fruitful approach for writing new code for a given class of problems (perhaps requiring a touch of creativity along the way), while our approach is a library that can be directly reused. This becomes literal when we formalize our mathematics in a proof assistant: a single module defines our proof technique and proves its soundness, and verifying example programs in a specific language begins by importing the core module, and providing the language semantics as an argument when using its contents.

The final ingredient for reliable formal verification is a trusted and trust-

[2] A definition by translation to another language is also executable, but has difficulty faithfully reflecting any intentionally unspecified behavior in the source language. An axiomatic semantics might conceivably be "executed" by proof search, but the attempt in [Yan+04] suggests this is feasible only for the simplest cases.

worthy program for checking proofs or proof certificates. By working within an established proof assistant, we can take advantage of their existing proof-checking tools, and prior examination of their soundness. Proving the soundness of a proof system in an established proof assistant improves confidence in the result, and composing this lemma with the verification of an particular specification results in a proof certificate in the logic of the proof assistant which states directly that the code behaves according to the given specifications under the provided operational semantics.

Given a sound verification system the next question is how difficult it is to apply the system, by writing desired specifications and proving that code meets them. Making specification and verification easier is necessary for formal verification to become widely accessible and cost-effective. Proof automation is required to make program verification practical on larger programs. Developing abbreviations for writing specifications and implementing heuristic proof tactics can make it easier to use a system without any threat to soundness.

We evaluate the usability of our approach by defining language in several paradigms and verifying a range of examples in each. Our aim here is a proof of concept, we do not (yet) aim to compete with mature language-specific proof systems but rather to show that our approach does not inherently preclude reaching a comfortable level of concise specification and proof automation. We show that specifications can be written concisely after defining appropriate abbreviations, and describe automated proof tactics which provide nearly-complete automation on examples including heap data structures and recursive functions.

The development of semantics-engineering frameworks such as K [RȘ10] and PLT-Redex [Kle+] reduces the difficulty of defining an operational semantics to little more than that of writing an interpreter (using the language of the framework). Recently full semantics of several real programming languages have been proposed, such as C [HER], Java [BR], JavaScript [Bod+; PȘR], Python [Gut13; Pol+13], PHP [FM], and OCaml [Owe]. Our coinductive program-verification approach could be used with any of these semantics or frameworks.

Developing automatic translators from semantics in these languages into definitions of transition relations in a chosen proof assistant could be a fruitful source of semantics. We describe an initial translator from $\mathbb{K}$ language definitions into Coq.

Chapter 2

Background

This chapter reviews background material in mathematics and programming language semantics. All notation and definitions in this chapter should be standard. Readers should feel free to skip any familiar concepts, except perhaps to confirm details of our notation.

2.1 Basic Mathematics

In this section we recall basic mathematical notions, and the describe the notation we use. This section can likely be skipped.

Given a set A, the *powerset* $\mathcal{P}(A)$ is the set of subsets of A, definable by $\mathcal{P}(A) = \{X \mid X \subseteq A\}$. Given sets A and B, the set $A \times B$ is the set of pairs, $A \times B = \{(a, b) \mid a \in A, b \in B\}$.

Given sets A and B a *relation* from A to B is a subset of $A \times B$. A *relation* A is a relation from A to A, or a subset of $A \times A$. Membership in a relation may be written infix, so xRy means $(x, y) \in R$.

A *sub-relation* of a relation R from A to B is a relation R' from A' to B' for some subsets $A' \subseteq A$ and $B' \subseteq B$, such that $xR'y$ implies xRy. A sub-relation is *induced* from the parent relation if $R' = R \cap A' \times B'$. In this case xRy for $(x, y) \in A' \times B'$ iff $xR'y$.

Definition 2.1 (Relation Properties)**.** A relation is *reflexive* if every element is related to itself, xRx, and *irreflexive* if no element is related to itself, $\neg xRx$.

A relation is *symmetric* when xRy implies yRx, and *antisymmetric* if xRy implies $\neg yRx$ when $x \neq y$.

A relation is *transitive* when $xRy \wedge yRz$ implies xRz.

Definition 2.2 (Orders)**.** A relation which is reflexive, antisymmetric, and transitive is called a *partial order*. A *poset* or *partially ordered set* is a set equipped with a partial order, the relation commonly written $\leq$. A pair of elements a and b are *comparable* if at least one of $a \leq b$ or $b \leq a$ hold. A *total order* is a partial order with the additional property that any pair of elements are comparable.

The collection of total functions with domain A and codomain B is written $A \to B$, and partial functions written $A \rightharpoonup B$. As with relations, a function on a set A is a function from A to A. Unless otherwise specified functions are required to be total. We generally write $f : A \to B$ to indicate the domain and codomain of a function instead of $f \in A \to B$. Given a function f from A to B, perhaps partial, $f[b/a]$ for $a \in A, b \in B$ denotes the function from A to B which has $f[b/a](a) = b$ and agrees with f otherwise.

Definition 2.3 (Monotonicity)**.** A function f on a poset is *monotone* when it preserves order, i.e. $f(a) \leq f(b)$ whenever $a \leq b$.

Definition 2.4 (Bounds)**.** An *upper bound* of a subset X of a partially ordered set is a value c such that $x \leq c$ for any $x \in X$. An upper bound is the *least upper bound*, *supremum* or *lub*, written $\bigvee X$, if it is additionally less than any other bound, $\bigvee X \leq k$ for any upper bound k of X. Conversely, a *lower bound* c of a set X has $c \leq x$ for all $x \in X$. The *greatest lower bound*, *infimum* or *glb* of a set, written $\bigwedge X$ is a lower bound with $k \leq \bigwedge X$ for any lower bound k of X.

In the general case a subset of a poset or even a total order may not have any upper bound, or have several upper bounds with no least upper bound.

Definition 2.5 (Lattice)**.** A *lattice* is a poset where every two-element subset has a lub and a glb, which are written $x \vee y$ for $\bigvee\{x, y\}$ and $x \vee y$ for $\bigvee\{x, y\}$. Any non-empty finite subset of a lattice has a lub and glb.

A *complete lattice* is a poset where every subset has a lub and a glb. Considering in particular the empty set shows that a complete lattice has a least and a greatest element.

We are particularly interested in the complete lattices formed by ordering the powerset of some base set by inclusion. Given a set A, define an order $\leq$ on $\mathcal{P}(A)$ by $X \leq Y$ iff $X \subseteq Y$. With this definition any collection $C \subseteq \mathcal{P}(A)$ of subsets of A has both a least upper bound and greatest lower bound, which are respectively the union $\bigvee C = \bigcup C$ and intersection $\bigwedge C = \bigcap C$ of the collection of sets.

2.1.1 Fixpoints

A fixed point or *fixpoint* of a function is a value mapped to itself by the function, an x in the domain of f with $f(x) = x$. Fixpoints are of interest throughout mathematics, along with fixpoint theorems which give conditions on functions under which fixpoints are guaranteed to exist (perhaps along with constructions of the fixpoint).

We are particularly interested in monotone functions on complete lattices, where the Knaster-Tarski fixpoint theorem applies. As is common in computer science the only thing we need to conclude "by the Knaster-Tarski fixpoint theorem" is that such functions have fixpoints, and in particular a least and/or greatest fixpoint. This is actually shown as a lemma in Tarski's proof [Tar55].

Lemma 1. *A monotone function f on a complete lattice L has a greatest fixpoint, which is the least upper bound of the set $P = \{x \mid x \leq f(x)\}$ of* postfixed points. *Dually, f has a least fixpoint, which is the greatest lower bound of the set $\{x \mid f(x) \leq x\}$ of* prefixed points.

Proof. Fix x in P. By the definition of least upper bound, $x \leq \bigvee P$. By monotonicity of, $f(x) \leq f(\bigvee P)$. By the definition of P, $x \leq f(x)$, so by transitivity $x \leq f(\bigvee P)$. This holds for any $x \in P$, so $f(\bigvee P)$ is an upper bound on P. As $\bigvee P$ is the least upper bound, $\bigvee P \leq f(\bigvee P)$. By monotonicity, $f(\bigvee P) \leq f(f(\bigvee P))$. Thus, $f(\bigvee P) \in P$. As $\bigvee P$ is an upper bound, $f(\bigvee P) \leq \bigvee P$. By antisymmetry of the partial order, $f(\bigvee P) = \bigvee P$, so $\bigvee P$ is a fixpoint of f. By reflexivity of the partial order, P includes every fixpoint of f, so $\bigvee P$ is the greatest fixpoint. Dually, the least fixpoint of f is the greatest lower bound of $\{x \mid f(x) \leq x\}$. □

The full Knaster-Tarski theorem proves more about the structure of the set of fixpoints.

Theorem 1 (Knaster-Tarski Theorem)**.** *Let $(L, \leq)$ be a complete lattice and f a monotone function on L. Then the set $F = \{x \mid f(x) = f\}$ of fixpoints of f is also a complete lattice, under the same order $\leq$.*

Proof. To prove the full result, the least upper bound in F of any subset X of F will be obtained as the least fixpoint of f on an appropriate sublattice of L. Here we use the symbols $\bigvee$ and $\bigwedge$ only for bounds in L.

Let U be the set of all upper bounds in L of X. There is a least upper bound $u = \bigvee X$, and U can be described as $U = \{x \mid u \leq x\}$.

X will have a least upper bound in F if there is a least element among the fixpoints of f which are in U. By the lemma there is such an element if U is closed under f and U is a complete (sub)lattice.

For the first, fix $b \in U$. For any $x \in X$ we have $x \leq b$. As $X \subseteq F$, $x = f(x)$. By monotonicity we have also $f(x) \leq f(b)$, so $x \leq f(b)$. Therefore $f(b)$ is an upper bound on X and thus a member of U.

To show U is a complete lattice, we first recall that u is its least element. Next we note that as a complete sub-order of L, if an element a is the least or greatest member a set $A \subseteq L$, and $a \in U$ then a is also the least or greatest member of $A \cap U$. The greatest element $\top$ of L is certainly an upper bound of X, and thus also the greatest element of U. Now we will show that any nonempty set $B \subseteq U$ has the same least upper bound and greatest lower bound in U as in L. As there is some $b \in B$, by transitivity $u \leq b \leq \bigvee B$. As u is a lower bound on B it is below the greatest, $u \leq \bigwedge B$. Therefore U is a complete lattice.

Now by the lemma, f indeed has a least fixpoint in U. This is the least element of $F \cap U$, and therefore the least upper bound in F of X. Dually, X also has a greatest lower bound in F. Thus the set F of fixpoints of f is a complete lattice. □

Another fixpoint theorem which is commonly used in computer science, especially for denotations semantics, is the Kleene fixpoint theorem. This relaxes the requirement on a poset from being a complete lattice to only requiring least upper bounds exist for totally ordered subsets (including a least element), but requires an additional condition on the function f beyond monotonicity. It gives a more computational description of the leastfixpoint as the least upper bound of the sequence

$$\bot \leq f(\bot) \leq f^2(\bot) \leq ...$$

obtained by iterating the function starting with the least element $\bot$.

However, we will need fixpoints of functions where this theorem does not apply, and are satisfied the simple description of the least fixpoint in Tarski's lemma.

2.2 Coinduction

The definition of greatest fixpoints gives a proof principle called coinduction.

Lemma 2 (Coinduction). *Given a monotone function f on a poset P with greatest fixpoint v, to show $x \leq v$ for some $x \in P$ it suffices to show $x \leq f(x)$, by the description of the greatest fixpoint as the least upper bound of post-fixed points.*

The corresponding principle for least fixpoints gives the more familiar notion of induction.

Lemma 3 (Induction). *Given a monotone function f on a poset P with least fixpoint u, to show $u \leq x$ for some $x \in P$ it suffices to show $f(x) \leq x$.*

This statement may still be somewhat unfamiliar, so let's see how it gives rise to the familiar principle of induction on natural numbers, which we state as follows

Lemma 4 (Natural number induction). *Given a property P on natural numbers, to show that $P(n)$ for every $n \in \mathbb{N}$ it suffices to show that*

- *$P(0)$*
- *for any n, $P(n)$ implies $P(n+1)$*

To connect this to the fixpoint characterization of induction we take as our lattice the set of subsets of $\mathbb{N}$ ordered by inclusion. Our least fixpoint will be shown to be $\mathbb{N}$, Our function will be chosen so the least fixpoint is all of $\mathbb{N}$, and our chosen point will be the *extension* of the predicate, the set of numbers satisfying it, given by $A_P = \{n \mid P(n)\}$. Then $\mathbb{N} \subseteq A_P$ implies the desired conclusion. In terms of A_P, the requirements on P are

- $0 \in A_P$
- for any $n \in A_P$, $n+1 \in A_P$.

This can be put into the form $f(A_P) \subseteq A_P$ by taking as our function this operation on sets of natural numbers

$$\mathrm{suc}(X) = \{0\} \cup \{n+1 \mid n \in X\}$$

Thus we see that induction on natural numbers is equivalent to the fixpoint description using a characterization of $\mathbb{N}$ as the least fixpoint of suc on the complete lattice $\mathcal{P}(\mathbb{N})$.

We work with the intuition that showing an inclusion $A \subseteq B$ establishes that every element of the set A satisfies the predicate whose extension is the set B. Under this point of view, induction lets us do some initial work to find a characterization of a set of particular interest as a least fixpoint, to obtain a lemma which lets us more easily attempt to prove $A \subseteq B$ for a variety of desired predicates B. Coinduction conversely lets us do some initial work to characterize a property of interest as a greatest fixpoint, to obtain a lemma which lets us more easily show that various desired sets contain only elements satisfying the predicate of interest.

In particular, we will work with sets of claims about program execution and characterize the property of a claim being true as a greatest fixpoint, to obtain a coinduction principle for showing that entire sets of claims hold, which we will apply to the sets of claims corresponding to the specifications of programs.

2.3 Semantics

A formal definition of a programming language is necessary to support any formal proofs about the behavior of programs. Two major approaches to semantics are operational semantics and denotational semantics.

A denotational semantics is based on assigning a meaning to terms in the programming language, by giving a mapping from code into a mathematical domain that completely characterizes the behavior of a fragment of code. The value resulting from a fragment of code is called its denotation. Ideally the mapping defines the denotation of any language construct as a function of the denotation rather than the code of its subterms.

For a simple example, arithmetic expressions over constants can be given a denotation as numbers if division (by zero) is avoided, or by adding an additional value for errors to the set of numbers, and each operator is described as a function of the values of its arguments. Expressions over variables can be handled by using partial functions from variables to values as environments, and giving denotations as functions from environments to numbers, extended

$i \in \mathbb{Z}$ – Integers
$x \in \mathit{Id}$ – Identifiers
$b \in \mathbb{B}(= \{\top, \bot\})$ – Booleans
$\sigma \in \mathit{Id} \rightharpoonup \mathbb{Z}$ – Environments (used in semantics)

$e \in \mathit{Exp}$ – Expressions
$e ::= \ i \mid x \mid x\ \texttt{--} \mid e\ \texttt{+}\ e$

$\mathit{be} \in \mathit{BExp}$ – Boolean Expressions
$\mathit{be} ::= \ b \mid \texttt{!}\ \mathit{be} \mid \mathit{be}\ \texttt{\&\&}\ \mathit{be} \mid e\ \texttt{<=}\ e \mid e\ \texttt{==}\ e$

$s \in \mathit{Stmt}$ – Statements
$s ::= \ x\ \texttt{:=}\ e \mid \texttt{skip} \mid s\ \texttt{;}\ s$
$\mid \ \texttt{if (} \mathit{be} \texttt{) \{ } \mathit{stmt} \texttt{ \} else \{ } \mathit{stmt} \texttt{ \}} \mid \texttt{while (} \mathit{be} \texttt{) \{ } s \texttt{ \}}$

Figure 2.1: IMP Syntax

with an error result as before. Simple statements can be given a denotation as functions from environment to environments. More complicated language features such as loops or recursive functions (let alone concurrency) require more sophisticated domains constructed to ensure recursive descriptions are well-defined.

An operational semantics is concerned with describing how code executes. All of the styles discussed are illustrated on a simple imperative language in Fig. 2.1. Our language deliberately includes a postdecrement operator `x--`, so expression evaluation includes side effects, and the examples will show how to deal with this complication.

An operational semantics includes a definition of a set of *states* or *configurations* which combine code with other runtime state needed to describe how a language executes, which may include components such as an environment mapping local variables to values, a heap mapping pointers to values, a call stack, active exception handlers, tables of class or function definitions. An operational semantics also includes a relation describing the execution of these configurations. We show multiple operational semantics for a simple imperative language IMP in Figs. 2.2, 2.3, 2.4 and 2.5 The major division in operational semantics is between *big-step* semantics and *small-step* semantics.

A big-step semantics describes execution with a relation between configurations and a set of results, which immediately relates a program to a final

Configurations are pairs $\langle s, \sigma \rangle$

$\boxed{e, \sigma \Downarrow_e i, \sigma}$ – Arithmetic Expression evaluation

$$\text{Var}\ \frac{\sigma(x) = i}{x, \sigma \Downarrow_e i, \sigma} \quad \text{Dec}\ \frac{\sigma(x) = i}{x\texttt{--}, \sigma \Downarrow_e i, \sigma[i - 1/x]}$$

$$\text{Add}\ \frac{e_1, \sigma \Downarrow_e i_1, \sigma_1 \qquad e_2, \sigma_1 \Downarrow_e i_2, \sigma_2}{e_1\texttt{+}e_2, \sigma \Downarrow_e (i_1 + i_2), \sigma_2}$$

$\boxed{be, \sigma \Downarrow_b b, \sigma}$ – Boolean Expression evaluation

$$\text{Not}\ \frac{be, \sigma \Downarrow_b b, \sigma'}{\texttt{!}be, \sigma \to_b \neg b, \sigma'} \quad \text{And-}\bot\ \frac{be_1, \sigma \Downarrow_b \bot, \sigma'}{be_1\texttt{\&\&}be_2, \sigma \Downarrow_b \bot, \sigma'}$$

$$\text{And-}\top\ \frac{be_1, \sigma \Downarrow_b \top, \sigma_1 \qquad be_2, \sigma_1 \Downarrow_b b, \sigma_2}{be_1\texttt{\&\&}be_2, \sigma \Downarrow_b b, \sigma_2}$$

$$\text{Eq}\ \frac{e_1, \sigma \Downarrow_e i_1, \sigma_1 \qquad e_2, \sigma_1 \Downarrow_e i_2, \sigma_2}{e_1\texttt{=}e_2, \sigma \Downarrow_b (i_1 = i_2), \sigma_2} \quad \text{Le}\ \frac{e_1, \sigma \Downarrow_e i_1, \sigma_1 \qquad e_2, \sigma_1 \Downarrow_e i_2, \sigma_2}{e_1\texttt{<=}e_2, \sigma \Downarrow_b (i_1 \leq i_2), \sigma_2}$$

$\boxed{s, \sigma \Downarrow \sigma}$ – Configuration evaluation

$$\text{Asgn}\ \frac{e, \sigma \Downarrow_e i, \sigma'}{x\texttt{:=}e, \sigma \Downarrow \sigma[i/x]} \quad \text{Seq}\ \frac{s_1, \sigma \Downarrow \sigma_1 \qquad s_2, \sigma_1 \Downarrow \sigma_2}{s_1\texttt{;}s_2, \sigma \Downarrow \sigma_2}$$

$$\text{While}\ \frac{\texttt{if(}be\texttt{)\{}s\texttt{;while(}be\texttt{)\{}s\texttt{\}\}else\{skip\}}, \sigma \Downarrow \sigma'}{\texttt{while(}be\texttt{)\{}s\texttt{\}}, \sigma \Downarrow \sigma'}$$

$$\text{If-}\top\ \frac{be, \sigma \Downarrow_b \top, \sigma_1 \qquad s_1, \sigma_1 \Downarrow \sigma_2}{\texttt{if(}be\texttt{)\{}s_1\texttt{\}else\{}s_2\texttt{\}}, \sigma \Downarrow \sigma_2} \quad \text{If-}\bot\ \frac{be, \sigma \Downarrow_b \bot, \sigma_1 \qquad s_2, \sigma_1 \Downarrow \sigma_2}{\texttt{if(}be\texttt{)\{}s_1\texttt{\}else\{}s_2\texttt{\}}, \sigma \Downarrow \sigma_2}$$

Figure 2.2: IMP Big-Step Semantics

output or summary of its execution. This relation is almost always defined inductively in terms of the big-step results of subprograms. The relation need not be functional - a configuration may be assigned multiple results if execution is not deterministic, and assigning a configuration no result may be used to model errors or diverging execution such as infinite loops (both cases might also be modeled as part of the result set, though handling infinite execution generally requires a coinductive big-step semantics, as in [LG09]).

A small-step semantics describes execution with a relation on the set of configurations which is interpreted as defining primitive execution steps. A possible behavior of a program is described as a sequence of states each related to the next by the step relation.

The semantics also describes how these configurations execute. This divides semantics into small-step and big-step semantics. In a small-step semantics, execution is described by a relation on configurations which connects states and their possible immediate successors. Possible executions of a program consist of sequences of states which are each related to the next by the step relation. Unlike a big-step semantics this gives an inherent notion of intermediate states in an execution, and the only inherent notion of a final result is in reaching a state with no successors, also called a *stuck* state or a *normal form*. An execution which never terminates corresponds to an infinite series of steps that avoids any stuck states.

There are several styles for defining small-step semantics. The main difference is how the evaluation in subterms is handled, such as reducing $1 + 2$ to 3 within $(1 + 2) + 4$ to produce the term $3 + 4$. We describe *structural operational semantics* (abbreviated SOS), *reduction semantics* (or evaluation-context semantics), and *K-style* semantics.

The oldest approach is *structural operational semantics* [Plo04], abbreviated SOS, which defines the step relation inductively and uses congruence rules which allow a compound term to take an execution step when a related configuration for a subterm takes a step.

In the SOS semantics in Fig. 2.3 we have rules such as

$$\frac{e_1, \sigma \rightarrow_e e_1', \sigma'}{e_1\texttt{+}e_2, \sigma \rightarrow_e e_1'\texttt{+}e_2, \sigma'}$$

which allows evaluation to occur within the first argument of an addition expression. This rule explicitly shows how σ is made available to the step in

Configurations are pairs $\langle s, \sigma \rangle$

$\boxed{e, \sigma \to_e e, \sigma}$ – Expression evaluation

$$\text{Var}\ \frac{\sigma(x) = i}{x, \sigma \to_e i, \sigma} \quad \text{Dec}\ \frac{\sigma(x) = i}{x\texttt{--}, \sigma \to_e i, \sigma[i - 1/x]} \quad \text{Add}\ \frac{}{i_1\texttt{+}i_2, \sigma \to_e i_1 + i_2, \sigma}$$

$$\frac{e_1, \sigma \to_e e_1', \sigma'}{e_1\texttt{+}e_2, \sigma \to_e e_1'\texttt{+}e_2, \sigma'} \quad \frac{e_2, \sigma \to_e e_2', \sigma'}{i_1\texttt{+}e_2, \sigma \to_e i_1\texttt{+}e_2', \sigma'}$$

$\boxed{be, \sigma \to_b be, \sigma}$ – Boolean Expression evaluation

$$\text{And-}\top\ \frac{}{\top\texttt{\&\&}be, \sigma \to_b be, \sigma} \quad \text{And-}\bot\ \frac{}{\bot\texttt{\&\&}be, \sigma \to_b \bot, \sigma} \quad \text{Not}\ \frac{}{\texttt{!}b, \sigma \to_b \neg b, \sigma}$$

$$\text{Eq}\ \frac{}{i_1\texttt{=}i_2, \sigma \to_b (i_1 = i_2), \sigma} \quad \text{Le}\ \frac{}{i_1\texttt{<=}i_2, \sigma \to_b (i_1 \leq i_2), \sigma}$$

$$\frac{be_1, \sigma \to_b be_1', \sigma'}{be_1\texttt{\&\&}be_2, \sigma \to_b be_1'\texttt{\&\&}be_2, \sigma'} \quad \frac{be_1, \sigma \to_b be_1', \sigma'}{\texttt{!}be_1, \sigma \to_b \texttt{!}be_1', \sigma'} \quad \frac{e_1, \sigma \to_e e_1', \sigma'}{e_1\texttt{=}e_2, \sigma \to_b e_1'\texttt{=}e_2, \sigma'}$$

$$\frac{e_2, \sigma \to_e e_2', \sigma'}{i_1\texttt{=}e_2, \sigma \to_b i_1\texttt{=}e_2', \sigma'} \quad \frac{e_1, \sigma \to_e e_1', \sigma'}{e_1\texttt{<=}e_2, \sigma \to_b e_1'\texttt{<=}e_2, \sigma'} \quad \frac{e_2, \sigma \to_e e_2', \sigma'}{i_1\texttt{<=}e_2, \sigma \to_b i_1\texttt{<=}e_2', \sigma'}$$

$\boxed{s, \sigma \to s, \sigma}$ – Configuration evaluation

$$\text{Asgn}\ \frac{}{x\texttt{:=}i, \sigma \to \texttt{skip}, \sigma[i/x]} \quad \text{Seq}\ \frac{}{\texttt{skip;}s_2, \sigma \to s_2, \sigma}$$

$$\text{While}\ \frac{}{\texttt{while(}be\texttt{)\{}s\texttt{\}}, \sigma \to \texttt{if(}be\texttt{)\{}s\texttt{;while(}be\texttt{)\{}s\texttt{\}\}else\{skip\}}, \sigma}$$

$$\text{If-}\top\ \frac{}{\texttt{if(}\top\texttt{)\{}s_1\texttt{\}else\{}s_2\texttt{\}}, \sigma \to s_1, \sigma} \quad \text{If-}\bot\ \frac{}{\texttt{if(}\bot\texttt{)\{}s_1\texttt{\}else\{}s_2\texttt{\}}, \sigma \to s_2, \sigma}$$

$$\frac{e, \sigma \to e', \sigma'}{x\texttt{:=}e, \sigma \to x\texttt{:=}e', \sigma'} \quad \frac{s_1, \sigma \to s_1', \sigma'}{s_1\texttt{;}s_2, \sigma \to s_1'\texttt{;}s_2, \sigma'}$$

$$\frac{be, \sigma \to be', \sigma'}{\texttt{if(}be\texttt{)\{}s_1\texttt{\}else\{}s_2\texttt{\}}, \sigma \to \texttt{if(}be'\texttt{)\{}s_1\texttt{\}else\{}s_2\texttt{\}}, \sigma'}$$

Figure 2.3: IMP SOS Semantics

the hypothesis, and the possibly modified σ' is copied back into the conclusion. Even with just one component besides the code in the configuration this is already somewhat tedious. Even worse, this affects the modularity of the semantics. Extending the language in a way that requires extending the configuration could require modifying all existing rules, and even simple rules such as this one cannot be shared between similar semantics. To address this, refined variants of SOS such as MSOS [Mos04] and GSOS [BIM95] include conventions allowing rules to avoid explicitly mentioning state they only pass along rather than directly using. Tool support for writing structural operational semantics is available in the Maude MSOS tool [CB07].

Reduction semantics [FFF09] is based on the idea of explicitly defining the contexts where evaluation may occur, rather than encoding it with a collection of congruence rules. This definition is made by giving a recursive definition of evaluation contexts and a plugging operation which combines a context and a focused term back into a configuration. Then contexts are mentioned in rules such as the addition rule

$$\frac{}{C[i_1\texttt{+}i_2] \to C[i_1 + i_2]}$$

to explicitly allow the expression being reduced (called a *redex*) to occur inside a context. This approach natively allows for some degree of abbreviation over components of states which a rule does not need, by allowing them to be absorbed into the context. Because the recursive structure of terms is handled by the definition of contexts, the definition of the execution relation itself need not be recursive, and every step in the execution of a program is taken by using a single rule, without needing to construct an appropriate stack of congruence rules. However, applying a rule to a given configuration still requires finding the correct decomposition into context and redex. Reduction semantics is the natively supported style of the PLT-Redex tool [FFF09].

The 𝕂 style of operation semantics was developed in parallel with the 𝕂 tool [RȘ10; Șer+14], originating from an effort to define programming language semantics with rewriting logic [ȘRM09]. While big-step, small-step, and reduction semantics were all faithfully expressible, the setting motivated avoiding both the recursive definition of rules needed for SOS and the sophisticated pattern matching needed to decompose a term into context and redex as in reduction semantics. Instead, a 𝕂-style semantics explicitly represents the process of focusing on a current subterm as part of the configuration. A

Configurations are pairs $\langle s, \sigma \rangle$

$C \in \mathit{Cxt}$ – Evaluation Contexts

$C ::= \square \mid \langle \square, \sigma \rangle \mid \mathit{CItem}\ ,\ C$

$\mathit{CItem} ::= \square\ \texttt{+}\ e \mid i\ \texttt{+}\ \square \mid \texttt{!}\ \square \mid \square\ \texttt{\&\&}\ be \mid \square\ \texttt{<=}\ e \mid i\ \texttt{<=}\ \square \mid \square\ \texttt{==}\ e \mid i\ \texttt{==}\ \square \mid$
$\quad x\ \texttt{:=}\ \square \mid \square\ \texttt{;}\ s \mid \texttt{if (} \square \texttt{) \{ } \mathit{stmt} \texttt{ \} else \{ } \mathit{stmt} \texttt{ \}}$

$\boxed{C[e], C[be], C[s]}$ – Context plugging

$$\begin{aligned}
\square[s] &= s \\
\langle \square, \sigma \rangle[s] &= \langle s, \sigma \rangle \\
(\square\texttt{+}e_2, C)[e_1] &= C[e_1\texttt{+}e_2] \\
(i_1\texttt{+}\square, C)[e_2] &= C[i_1\texttt{+}e_2] \\
(\texttt{!}\square, C)[be] &= C[\texttt{!}\,be] \\
(\square\texttt{\&\&}be_2, C)[be_1] &= C[be_1\texttt{\&\&}be_2] \\
(\square\texttt{<=}e_2, C)[e_1] &= C[e_1\texttt{<=}e_2] \\
(i_1\texttt{<=}\square, C)[e_2] &= C[i_1\texttt{<=}e_2] \\
(\square\texttt{==}e_2, C)[e_1] &= C[e_1\texttt{==}e_2] \\
(i_1\texttt{==}\square, C)[e_2] &= C[i_1\texttt{==}e_2] \\
(x\,\texttt{:=}\square, C)[e] &= C[x\,\texttt{:=}e] \\
(\square\texttt{;}s_2, C)[s_1] &= C[s_1\texttt{;}s_2] \\
(\texttt{if(}\square\texttt{)\{}s_1\texttt{\}else\{}s_2\texttt{\}}, C)[be] &= C[\texttt{if(}be\texttt{)\{}s_1\texttt{\}else\{}s_2\texttt{\}}]
\end{aligned}$$

$\boxed{\langle s, \sigma \rangle \rightarrow \langle s, \sigma \rangle}$ – Configuration execution

$$\begin{array}{ll}
\text{Var}\ \dfrac{\sigma(x) = i}{\langle C[x], \sigma \rangle \rightarrow \langle C[i], \sigma \rangle} & \text{Dec}\ \dfrac{\sigma(x) = i}{\langle C[x\texttt{--}], \sigma \rangle \rightarrow \langle C[i], \sigma[i+1/x] \rangle} \\
\text{Add:}C[i_1\texttt{+}i_2] \rightarrow C[i_1 + i_2] & \text{Eq:}C[i_1\texttt{==}i_2] \rightarrow C[(i_1 = i_2)] \\
\text{Not:}C[\texttt{!}\,b] \rightarrow C[\neg b] & \text{Le:}C[i_1\texttt{<=}i_2] \rightarrow C[(i_1 \leq i_2)] \\
\text{And-}\top\text{:}C[\top\texttt{\&\&}be] \rightarrow C[be] & \text{Asgn:}\langle C[x\,\texttt{:=}i], \sigma \rangle \rightarrow \langle C[\texttt{skip}], \sigma[i/x] \rangle \\
\text{And-}\bot\text{:}C[\bot\texttt{\&\&}be] \rightarrow C[\bot] & \text{If-}\top\text{:}C[\texttt{if(}\top\texttt{)\{}s_1\texttt{\}else\{}s_2\texttt{\}}] \rightarrow C[s_1] \\
\text{Seq:}C[\texttt{skip;}s_2] \rightarrow C[s_2] & \text{If-}\bot\text{:}C[\texttt{if(}\bot\texttt{)\{}s_1\texttt{\}else\{}s_2\texttt{\}}] \rightarrow C[s_2] \\
\multicolumn{2}{l}{\text{While:}C[\texttt{while(}be\texttt{)\{}s\texttt{\}}] \rightarrow C[\texttt{if(}be\texttt{)\{}s\texttt{;while(}be\texttt{)\{}s\texttt{\}\}else\{skip\}}]}
\end{array}$$

Figure 2.4: IMP Reduction Semantics

Configurations are pairs $\langle s, \sigma \rangle$

$K ::= \texttt{.K} \mid \mathit{KItem} \curvearrowright K$

$\mathit{KItem} ::= e \mid be \mid s \mid \mathit{Freezer}$

$\mathit{Freezer} ::= \square\ \texttt{+}\ e \mid e\ \texttt{+}\ \square \mid \texttt{!}\ \square \mid \square\ \texttt{\&\&}\ be \mid \square\ \texttt{<=}\ e \mid e\ \texttt{<=}\ \square \mid \square\ \texttt{==}\ e \mid e\ \texttt{==}\ \square \mid x\ \texttt{:=}\ \square \mid \texttt{if (}\square\texttt{) \{ } \mathit{stmt} \texttt{ \} else \{ } \mathit{stmt} \texttt{ \}}$

$\boxed{\langle s, \sigma \rangle \to \langle s, \sigma \rangle}$ – Configuration execution

$$\text{Var}\ \frac{\sigma(x) = i}{\langle x \curvearrowright K, \sigma \rangle \to \langle i \curvearrowright K, \sigma \rangle} \qquad \text{Dec}\ \frac{\sigma(x) = i}{\langle x\texttt{--} \curvearrowright K, \sigma \rangle \to \langle i \curvearrowright K, \sigma[i-1/x] \rangle}$$

Asgn:$\langle x\texttt{:=}i \curvearrowright K, \sigma \rangle \to \langle K, \sigma[i/x] \rangle$ Add:$\langle i_1\texttt{+}i_2 \curvearrowright K, \sigma \rangle \to \langle i_1 + i_2 \curvearrowright K, \sigma \rangle$

Eq:$\langle i_1\texttt{==}i_2 \curvearrowright K, \sigma \rangle \to \langle (i_1 = i_2) \curvearrowright K, \sigma \rangle$Le:$\langle i_1\texttt{<=}i_2, \sigma \rangle \to \langle (i_1 \leq i_2) \curvearrowright K, \sigma \rangle$

Not:$\langle \texttt{!}b \curvearrowright K, \sigma \rangle \to \langle \neg b \curvearrowright K, \sigma \rangle$ And-$\top$:$\langle \top\texttt{\&\&}be \curvearrowright K, \sigma \rangle \to \langle be \curvearrowright K, \sigma \rangle$

Skip:$\langle \texttt{skip} \curvearrowright K, \sigma \rangle \to \langle K, \sigma \rangle$ And-$\bot$:$\langle \bot\texttt{\&\&}be \curvearrowright K, \sigma \rangle \to \langle \bot \curvearrowright K, \sigma \rangle$

Seq:$\langle s_1\texttt{;}s_2 \curvearrowright K, \sigma \rangle \to \langle s_1 \curvearrowright s_2 \curvearrowright K, \sigma \rangle$

If-$\top$:$\langle \texttt{if(}\top\texttt{)\{}s_1\texttt{\}else\{}s_2\texttt{\}} \curvearrowright K, \sigma \rangle \to \langle s_1 \curvearrowright K, \sigma \rangle$

If-$\bot$:$\langle \texttt{if(}\bot\texttt{)\{}s_1\texttt{\}else\{}s_2\texttt{\}} \curvearrowright K, \sigma \rangle \to \langle s_2 \curvearrowright K, \sigma \rangle$

While: $\langle \texttt{while(}be\texttt{)\{}s\texttt{\}} \curvearrowright K, \sigma \rangle \to$
$\langle \texttt{if(}be\texttt{)\{}s\texttt{;while(}be\texttt{)\{}s\texttt{\}\}else\{skip\}} \curvearrowright K, \sigma \rangle$

Heating and Cooling rules

$\langle e_1\texttt{+}e_2 \curvearrowright K, \sigma \rangle \rightleftharpoons \langle e_1 \curvearrowright \square\texttt{+}e_2 \curvearrowright K, \sigma \rangle$ $\to$ when $e_1 \notin \mathbb{Z}$, $\leftarrow$ when $e_1 \in \mathbb{Z}$

$\langle e_1\texttt{+}e_2 \curvearrowright K, \sigma \rangle \rightleftharpoons \langle e_2 \curvearrowright e_1\texttt{+}\square \curvearrowright K, \sigma \rangle$ $\to$ when $e_2 \notin \mathbb{Z}$, $\leftarrow$ when $e_2 \in \mathbb{Z}$

$\langle \texttt{!}be \curvearrowright K, \sigma \rangle \rightleftharpoons \langle be \curvearrowright \texttt{!}\square \curvearrowright K, \sigma \rangle$ $\to$ when $be \notin \mathbb{B}$, $\leftarrow$ when $be \in \mathbb{B}$

$\langle be_1\texttt{\&\&}be_2 \curvearrowright K, \sigma \rangle \rightleftharpoons \langle be_1 \curvearrowright \square\texttt{\&\&}be_2 \curvearrowright K, \sigma \rangle$
$\to$ when $be_1 \notin \mathbb{B}$, $\leftarrow$ when $be_1 \in \mathbb{B}$

$\langle e_1\texttt{<=}e_2 \curvearrowright K, \sigma \rangle \rightleftharpoons \langle e_1 \curvearrowright \square\texttt{<=}e_2 \curvearrowright K, \sigma \rangle$ $\to$ when $e_1 \notin \mathbb{Z}$, $\leftarrow$ when $e_1 \in \mathbb{Z}$

$\langle e_1\texttt{<=}e_2 \curvearrowright K, \sigma \rangle \rightleftharpoons \langle e_2 \curvearrowright e_1\texttt{<=}\square \curvearrowright K, \sigma \rangle$ $\to$ when $e_2 \notin \mathbb{Z}$, $\leftarrow$ when $e_2 \in \mathbb{Z}$

$\langle e_1\texttt{==}e_2 \curvearrowright K, \sigma \rangle \rightleftharpoons \langle e_1 \curvearrowright \square\texttt{==}e_2 \curvearrowright K, \sigma \rangle$ $\to$ when $e_1 \notin \mathbb{Z}$, $\leftarrow$ when $e_1 \in \mathbb{Z}$

$\langle e_1\texttt{==}e_2 \curvearrowright K, \sigma \rangle \rightleftharpoons \langle e_2 \curvearrowright e_1\texttt{==}\square \curvearrowright K, \sigma \rangle$ $\to$ when $e_2 \notin \mathbb{Z}$, $\leftarrow$ when $e_2 \in \mathbb{Z}$

$\langle x\texttt{:=}e \curvearrowright K, \sigma \rangle \rightleftharpoons \langle e \curvearrowright x\texttt{:=}\square \curvearrowright K, \sigma \rangle$ $\to$ when $e \notin \mathbb{Z}$, $\leftarrow$ when $e \in \mathbb{Z}$

$\langle \texttt{if(}be\texttt{)\{}s_1\texttt{\}else\{}s_2\texttt{\}} \curvearrowright K, \sigma \rangle \rightleftharpoons \langle be \curvearrowright \texttt{if(}\square\texttt{)\{}s_1\texttt{\}else\{}s_2\texttt{\}} \curvearrowright K, \sigma \rangle$
$\to$ when $be \notin \mathbb{B}$, $\leftarrow$ when $be \in \mathbb{B}$

Figure 2.5: IMP K-Style Semantics

𝕂-style semantics has a sequence of "komputation" items where other styles might have a single term for the current code. Items include terms introduced to express the semantics, in addition to fragments of code.

A subterm is lifted to the front of the sequence and the result embedded back into a larger term with *heating* and *cooling* rules such as

$$\text{heat-+}_1 \langle e_1 \texttt{+} e_1 \curvearrowright K \mid \sigma\rangle \to \langle e_1 \curvearrowright \square\texttt{+\_}(e_2) \curvearrowright K \mid \sigma\rangle \text{ when } e_1 \notin \mathbb{Z}$$
$$\text{cool-+}_1 \langle i \curvearrowright \square\texttt{+\_}(e_2) \curvearrowright K \mid \sigma\rangle \to \langle i \texttt{+} e_1 \curvearrowright K \mid \sigma\rangle \text{ when } i \in \mathbb{Z}$$

The computation item $\square\texttt{+\_}$ is a *freezer*, retaining the operator and other argument while execution proceeds within the first argument of the addition expression. In practice, both the freezers and the heating and cooling rules are usually generated from annotations on the grammar saying which positions of which operators allow evaluation.

By relying on heating rules to expose subterms as the front of the computation sequence, the remaining rules of the semantics can be written to assume the desired term stands alone at the front of the computation sequence, as in this rule which evaluates an addition.

$$\text{eval-+}\ \frac{}{(i_1 \texttt{+} i_2 \curvearrowright K, \sigma) \to (i_1 + i_2 \curvearrowright K, \sigma)}$$

This part of the approach is related to the idea of a zipper [Hue97] in functional programming, where a datatype allowing quick access to a designated subterm is a larger term is created by defining a set of one-step contexts, and representing a term with a focused subterm as a pair of the focused subterm and a list of contexts recording the structure back up to the root. The 𝕂-style handling can be thought of as a zipper-style implementation of the context/redex decomposition of a reduction semantics.

Chapter 3

Coinductive Verification

This chapter presents the key ideas and mathematics of our coinductive approach to program verification. We begin with a simple problem to motivate basic definitions, and then to give a detailed demonstration of how they can be used to specify and verify a simple program. After seeing the foundational details and hopefully getting an intuition for the soundness of a coinductive proof, we describe how simple abbreviations and the use of proof assistants can handle many of the details we would rarely wish to see. Finally we prevent the full statements and proofs of the coinduction theorems and lemmas that we use.

3.1 Coinductive Proof System

In this section we will demonstrate how coinduction can be used for program verification by giving a detailed proof for a simple example.

We begin by introducing a language-independent notion of specification, by recalling the common notation of Hoare triples with a partial correctness interpretation and eliminating the portions of that notation which rely on assumptions about the language for greater concision. A set of the resulting *partial reachability* claims can capture the meaning of a set of Hoare triples. The assertion that a set of claims are true is amenable to coinductive proof because the collection of true claims can be defined as a greatest fixpoint.

Then we demonstrate how a simple program specification can be proved by coinduction. The most basic coinduction theorem is too unwieldy, but switching to an enhanced coinduction principle makes the proof much easier. In fact, the shape of the proof can closely follow the control flow of the program being verified, even though the statements to be proved may be stated in apparently unrelated terms of fixpoints and set inclusions. A particular point to notice is that loops are nicely handled without needing any language-specific lemmas for reasoning about loops.

We show most details to emphasize that only elementary mathematics is necessary to apply this approach, but it does make a traditional of the proof somewhat long and leaves much redundancy among intermediate stages of the proof. However, the simplicity also makes it easy to translate the statement to be proved into the language of a proof assistant, which will then manage most of the tedious detail such as updating the statement of the current goal. We show in Appendix A an extended transcript of proving this example in Coq, showing what help is provided even without any significant proof automation. In particular, the user need only enter brief commands, while the tool updates and displays the current progress of the proof attempt. Most of our additional example programs rely on the more complete proof automation described in Section 4.3, which proves the examples in Section 5.1 and Appendix B almost completely automatically.

Consider the simple program

```
s=0; while (n-- != 0) {s = s+n}
```

We let `sum` stand for the program and `loop` stand for the `while` statement.

When executed in an environment where `s` is defined and `n` has a non-negative initial value n, it sets `s` to the sum of integers $1, \ldots, n-1$, which is $n(n-1)/2$. This example includes potential nontermination because the semantics use unbounded integers, so the loop runs forever `n`.

This program is written in the language of Fig. 2.1 and will be verified according to the semantics in Fig. 2.3 or Fig. 2.4. The configurations of this semantics consist of pairs $\langle T \mid \sigma \rangle$ of a current statement T and a store σ mapping identifiers to integers. A portion of a possible execution of this program is shown in Fig. 3.1 The exact granularity of the semantics is not critical, as long as a diverging execution will produce an infinite execution. (Using several styles of operational semantics with our implementation is demonstrated and discussed in Section 4.1.1).

While coinductive program verification can be defined in a self-contained way, we discuss it in comparison with a traditional Hoare logic approach, for two reasons. First, it will help motivate some of the definitions we propose, which let use prove similar conclusions despite the different foundations of

```
                               ⟨s=0;loop|n↦ 1⟩
                             ⟨skip;loop|n↦ 1,s↦0⟩
                    ⟨while(n--!=0){s=s+n}|n↦ 1,s↦0⟩
   ⟨if(n--!=0){s=s+n;loop}else{skip}|n↦ 1,s↦0⟩
     ⟨if(1!=0){s=s+n;loop}else{skip}|n↦ 0,s↦0⟩
        ⟨if(⊤){s=s+n;loop}else{skip}|n↦ 0,s↦0⟩
                            ⟨s=s+n;loop|n↦ 0,s↦0⟩
                            ⟨s=0+n;loop|n↦ 0,s↦0⟩
                            ⟨s=0+0;loop|n↦ 0,s↦0⟩
                               ⟨s=0;loop|n↦ 0,s↦0⟩
                             ⟨skip;loop|n↦ 0,s↦0⟩
                    ⟨while(n--!=0){s=s+n}|n↦ 0,s↦0⟩
   ⟨if(n--!=0){s=s+n;loop}else{skip}|n↦ 0,s↦0⟩
     ⟨if(0!=0){s=s+n;loop}else{skip}|n↦−1,s↦0⟩
        ⟨if(⊥){s=s+n;loop}else{skip}|n↦−1,s↦0⟩
                                  ⟨skip|n↦−1,s↦0⟩
```

Figure 3.1: Example execution of sum program

the approaches. Second, it will help put coinductive verification in context, showing how it handles some issues that require some complexity to handle in Hoare logic.

3.1.1 Specifications

We will now consider how to specify that the program computes the sum. Speaking in terms of the operational semantics, execution from any configuration where the current code begins with the sum program and the store maps variable `n` to a value n should either diverge, or reach a configuration that is done executing and has a store which maps variable `s` to the sum $n(n-1)/2$, may have any value for `n`, and is otherwise unchanged.

In a Hoare logic with a partial-correctness interpretation, that specification could be written

$$\{\!|\texttt{n} = N|\!\}\texttt{s = 0; while (n-- != 0) \{s = s+n\}}\ \{\!|\texttt{s} = N(N-1)/2|\!\}$$

This notation is concise and evocative, but some of that convenience relies on details of a particular languages semantics. For example, program variables can be used directly in formulas to stand for their values, but this means the interpretation of the notation must mention details of how the current environment is represented in the configuration. In pursuit of language-independence we will forgo these conveniences for now (showing how to regain some in Section 3.1.6), and write specifications in terms of full configurations:

$$\forall n, T, s_0, \sigma.\, \langle \texttt{s = 0; while (n-- != 0) \{s = s+n\};}\ T \,|\, \texttt{n} \mapsto n, \texttt{s} \mapsto s_0, \sigma \rangle \\ \Rightarrow \{ \langle T \,|\, \texttt{n} \mapsto n', \texttt{s} \mapsto n(n-1)/2, \sigma \rangle \mid n' \in \mathbb{Z} \}$$

We generally regard an operational semantics as the relation $R \subseteq C \times C$ of atomic execution steps on a set C of configurations; we write $a \rightarrow_R b$ for $(a, b) \in R$ and $a \rightarrow_R^+ b$ for the transitive closure of R.

Definition 3.1 (One-Path Partial Reachability)**.** A *reachability claim* $c \Rightarrow_R P$ relates an initial state $c \in C$ and a target set of states $P \subseteq C$. A claim is satisfied or *valid* if the initial state c can either reach a state in P or can take an infinite number of steps. We drop the subscripts, as in $a \rightarrow b$ or $c \Rightarrow P$, when R is understood.

With this definition a claim only requires that some single execution from c falls into either case. This is suitable only for deterministic languages. We present another definition which constrains all paths in Section 3.2, and show both definitions can be used for coinduction. However, these definitions are equivalent on deterministic languages, and the one-path definition is easier to work with because it only requires exhibiting successors rather than reasoning about all possible successors.

The leading universal quantification means the specification above is not an individual reachability claim, but a set of claims which together constitute the specification of the program. This style of specification expresses desired program behavior with claims about the properties of the language semantics itself, asking whether certain sorts of execution paths exist.

3.1.2 Hoare Logic Proof

Before we demonstrate our approach we will review how the sum program would be verified using the axiomatic semantics shown in Fig. 3.2. The first step is transforming the program to remove side effects from the loop condition. This is necessary to allow the proof rules to substitute entire

$$\text{Skip}\ \frac{\quad}{\{\!|P|\!\}\texttt{skip}\{\!|P|\!\}} \qquad \text{Sequence}\ \frac{\{\!|P|\!\}s_1\{\!|R|\!\} \qquad \{\!|R|\!\}s_2\{\!|Q|\!\}}{\{\!|P|\!\}s_1\texttt{;}s_2\{\!|Q|\!\}}$$

$$\text{Assign}\ \frac{E \text{ has no side effects}}{\{\!|P[E/x]|\!\}x := e\{\!|P|\!\}} \qquad \text{While}\ \frac{\{\!|P \wedge b|\!\}s\{\!|P|\!\}}{\{\!|P|\!\}\texttt{while(}b\texttt{)}s\{\!|P \wedge \neg b|\!\}}$$

$$\text{Conseq}\ \frac{\models P \implies P' \qquad \{\!|P'|\!\}s\{\!|Q'|\!\} \qquad \models Q' \implies Q}{\{\!|P|\!\}s\{\!|Q|\!\}}$$

Figure 3.2: Axiomatic Semantics for IMP

expressions into predicates. Directly supporting side effects is also possible, and detailed in [ON02], but it requires expanding the proof system and proofs with a separate proof rule for each operator. The modified program is

```
s = 0; while (n != 0) {n = n-1; s = s+n}; n = n-1
```

The specification uses an auxiliary variable N not mentioned in the program to record the initial value of the variable `n`. We choose $\mathtt{s} + \mathtt{n}(\mathtt{n} - 1)/2 = N(N - 1)/2$ for the loop invariant. All the other intermediate predicates in the proof tree can be filled in by working backwards from the known postconditions, because rules other than While allow any proposition P as the postcondition. Whenever working backwards reaches a give precondition the Conseq rule is used with a hypothesis that the given precondition implies the desired one. Here is the derivation for `s=0`, which uses the Conseq rule to match up with the precondition given by the overall program specification.

$$\text{Conseq}\ \frac{\text{Assign}\ \frac{\quad}{\{\!|0 + \frac{\mathtt{n}(\mathtt{n}-1)}{2} = \frac{N(N-1)}{2}|\!\}\mathtt{s\ =\ 0}\{\!|\mathtt{s} + \frac{\mathtt{n}(\mathtt{n}-1)}{2} = \frac{N(N-1)}{2}|\!\}}}{\{\!|\mathtt{n} = N|\!\}\mathtt{s\ =\ 0}\{\!|\mathtt{s} + \frac{\mathtt{n}(\mathtt{n}-1)}{2} = \frac{N(N-1)}{2}|\!\}}$$

Here is the derivation for the final `n-1`, which uses the Conseq rule to match the precondition imposed the while loop with the predicate propagated from the postcondition of the whole program.

$$\text{Conseq}\ \frac{\text{Assign}\ \frac{\quad}{\{\!|\mathtt{s} = \frac{N(N-1)}{2}|\!\}\mathtt{n\ =\ n-1}\{\!|\mathtt{s} = \frac{N(N-1)}{2}|\!\}}}{\{\!|\mathtt{s} + \frac{\mathtt{n}(\mathtt{n}-1)}{2} = \frac{N(N-1)}{2} \wedge \mathtt{n} = 0|\!\}\mathtt{n\ =\ n-1}\{\!|\mathtt{s} = \frac{N(N-1)}{2}|\!\}}$$

$$\text{While}\;\dfrac{\text{Seq}\;\dfrac{\begin{array}{c}\text{Conseq}\;\dfrac{\text{Assign}\;\dfrac{}{\{\!|\mathtt{s}+\mathtt{n}-1+\frac{(\mathtt{n}-1)(\mathtt{n}-2)}{2}=\frac{N(N-1)}{2}|\!\}\mathtt{n=n-1}\{\!|\mathtt{s}+\mathtt{n}+\frac{\mathtt{n}(\mathtt{n}-1)}{2}=\frac{N(N-1)}{2}|\!\}}}{\{\!|\mathtt{s}+\frac{\mathtt{n}(\mathtt{n}-1)}{2}=\frac{N(N-1)}{2}\wedge n\neq 0|\!\}\mathtt{n=n-1}\{\!|\mathtt{s}+\mathtt{n}+\frac{\mathtt{n}(\mathtt{n}-1)}{2}=\frac{N(N-1)}{2}|\!\}}\\ \vdots\\ \vdots\end{array}\quad \text{Assign}\;\dfrac{}{\{\!|\mathtt{s}+\mathtt{n}+\frac{\mathtt{n}(\mathtt{n}-1)}{2}=\frac{N(N-1)}{2}|\!\}\mathtt{s=s+n}\{\!|\mathtt{s}+\frac{\mathtt{n}(\mathtt{n}-1)}{2}=\frac{N(N-1)}{2}|\!\}}}{\{\!|\mathtt{s}+\frac{\mathtt{n}(\mathtt{n}-1)}{2}=\frac{N(N-1)}{2}\wedge n\neq 0|\!\}\mathtt{n=n-1;s=s+n}\{\!|\mathtt{s}+\frac{\mathtt{n}(\mathtt{n}-1)}{2}=\frac{N(N-1)}{2}|\!\}}}{\{\!|\mathtt{s}+\frac{\mathtt{n}(\mathtt{n}-1)}{2}=\frac{N(N-1)}{2}|\!\}\mathtt{while(n!=0)\{n=n-1;s=s+n\}}\{\!|\mathtt{s}+\frac{\mathtt{n}(\mathtt{n}-1)}{2}=\frac{N(N-1)}{2}\wedge\neg n\neq 0|\!\}}$$

Figure 3.3: Hoare Logic derivation for sum loop

The derivation for the while loop is shown in Fig. 3.3. The applications of the Consequence rule in depend on implications

$$\begin{aligned}\mathtt{n}=N &\implies 0+\frac{\mathtt{n}(\mathtt{n}-1)}{2}=\frac{N(N-1)}{2}\\ \mathtt{s}+\frac{\mathtt{n}(\mathtt{n}-1)}{2}=\frac{N(N-1)}{2}\wedge\mathtt{n}=0 &\implies \mathtt{s}=\frac{N(N-1)}{2}\\ \mathtt{s}+\frac{\mathtt{n}(\mathtt{n}-1)}{2}=\frac{N(N-1)}{2}\wedge\mathtt{n}\neq 0 &\implies\\ &\qquad \mathtt{s}+(\mathtt{n}-1)+\frac{(\mathtt{n}-1)(\mathtt{n}-2)}{2}=\frac{N(N-1)}{2}\end{aligned}$$

The first two are trivial and the third follows from

$$\frac{\mathtt{n}(\mathtt{n}-1)}{2}=\frac{(2+(\mathtt{n}-2))(\mathtt{n}-1)}{2}=(\mathtt{n}-1)+\frac{(\mathtt{n}-1)(\mathtt{n}-2)}{2}$$

These three derivations are combined with the seq rule to produce a proof of the desired conclusion for the overall program:

$$\{\!|n=N|\!\}\mathtt{s=0;\ while(n!=0)\{n=n-1;\ s=s+n\};\ n=n-1}\{\!|\mathtt{s}=\frac{N(N-1)}{2}|\!\}$$

3.1.3 Coinductive Proofs

Now we introduce the proofs of our system. To be able to prove the overall specification we will need to add a specification of the loop itself.

$$\begin{aligned}&\forall x,n,T,\sigma.\ \langle\mathtt{loop;}\ \ T\,|\,\mathtt{n}\mapsto x,\mathtt{s}\mapsto s,\sigma\rangle\\ &\qquad\qquad\Rightarrow\{\langle T\,|\,\mathtt{n}\mapsto n',\mathtt{s}\mapsto s+\frac{n(n-1)}{2},\sigma\rangle\mid n'\in\mathbb{Z}\}\end{aligned}$$

This serves a similar purpose to a loop invariant in that it helps deal with loops by giving a more general precondition which also covers any states that may result from execution around the loop.

Unlike an axiomatic semantics there is nothing special about while loops in our proof system. We could have handled the loop by making an extra claim about states at any point in the loop, perhaps those whose code begins `s=s+n;loop` instead of those just `loop`.

Before stating a coinduction lemma we build some intuition by giving an informal argument that explicitly describes possible executions to show that the loop specification is valid. In the semantics of Fig. 2.3 or fig:imp-sem-reduction we have for any T, σ, s, and nonzero n that

$$\langle \texttt{loop; } T \,|\, \texttt{n} \mapsto 0, \texttt{s} \mapsto s, \sigma \rangle \to^{+} \langle T \,|\, \texttt{n} \mapsto -1, \texttt{s} \mapsto s, \sigma \rangle \tag{3.1}$$

$$\langle \texttt{loop; } T \,|\, \texttt{n} \mapsto n, \texttt{s} \mapsto s, \sigma \rangle \to^{+} \langle \texttt{loop; } T \,|\, \texttt{n} \mapsto n-1, \texttt{s} \mapsto s+n, \sigma \rangle \tag{3.2}$$

in a number of steps independent of n. This suffices to argue that the specification is valid. For initial states with $n \geq 0$ we assemble an explicit path to the target by induction on n. For $n < 0$ we can show execution diverges by concatenating the execution segments given by (3.2) for initial values $n, n-1, n-2, \ldots$.

Constructing specific execution paths, however, is too explicit. One problem is knowing in advance which configurations will diverge. Another problem is that this requires calculating exactly what state to expect at the beginning of the next loop iteration rather than allowing a more flexible predicate. Even in deterministic languages it is better if it is not necessary to predict things like the addresses where newly heap-allocated values will be placed.

Another problem is that calculating explicit execution paths requires describing the exact states reached rather than just using using predicates. Even for a deterministic language where this may be possible it is better if proofs do not need to predict details such as the exact addresses where heap-allocated values will reside.

We can weaken (3.2) to checking that

$$\langle \texttt{while (n-- != 0) \{s = s+n\}; } T \,|\, \texttt{n} \mapsto n, \texttt{s} \mapsto s, \sigma \rangle$$

with $n \neq 0$ reaches after finite steps a state of the form

$$\langle \texttt{while (n-- != 0) \{s = s+n\};}\ T \,|\, \texttt{n} \mapsto n', \texttt{s} \mapsto s', \sigma \rangle$$

such that $s + n(n-1)/2 = s' + n'(n'-1)/2$. This indeed suffices: either the resulting state has $n' = 0$ and reaches the target by (3.1) or $n' \neq 0$ and this claim applies again. Wandering forward will eventually either reach the target set or extend forever to give an infinite path. Either satisfies the reachability claim.

This intuition about "wandering forward" can be formalized with coinduction, specifically coinduction in a lattice of subsets ordered by inclusion. To start connecting coinduction to verification we regard specifications as sets of claims. To phrase the validity of a specification as a question of set inclusion we introduce a trivial definition of the set of all valid claims:

Definition 3.2. If $R \subseteq C \times C$, define $\text{valid}_R \subseteq C \times \mathcal{P}(C)$ by

$$\text{valid}_R = \{(c, P) \mid c \Rightarrow_R P \text{ holds}\}$$

Pairs $(c, P) \in C \times \mathcal{P}(C)$ are called *claims* or *specifications*, and our objective is to prove them true, i.e., $c \Rightarrow_R P$.

Sets of claims $S \subseteq C \times \mathcal{P}(C)$ are all true if $S \subseteq \text{valid}_R$. To show such inclusions by coinduction we express valid_R as a greatest fixpoint (Lemma 7), specifically of the following operator (proven as Lemma 7):

Definition 3.3. Given $R \subseteq C \times C$, let $\text{step}_R : \mathcal{P}(C \times \mathcal{P}(C)) \to \mathcal{P}(C \times \mathcal{P}(C))$ be defined by

$$\text{step}_R(S) = \{(c, P) \mid c \in P \ \vee \ \exists d \,.\, c \to_R d \wedge (d, P) \in S\}$$

The last ingredient we need is a generalized coinduction principle:

Definition 3.4. Given any monotone function $F : \mathcal{P}(D) \to \mathcal{P}(D)$ on a powerset, define its *F-closure* $F^* : \mathcal{P}(D) \to \mathcal{P}(D)$ by

$$F^*(X) = \mu Y.\, F(Y) \cup X$$

This is well-defined because $Y \mapsto F(Y) \cup X$ is monotone for any X.

Definition 3.5 (Transitivity Rule)**.** For any C, trans : $\mathcal{P}(C \times \mathcal{P}(C)) \to \mathcal{P}(C \times \mathcal{P}(C))$ is defined by

$$\mathrm{trans}(S) = \{(c, P) \mid \exists Q.(c, Q) \in S \wedge \forall d \in Q, (d, P) \in S\}$$

For brevity, let $\mathrm{expand}_R(X) = \mathrm{step}_R(X) \cup \mathrm{trans}(X)$.

Lemma 5. *For any $R \subseteq C \times C$ and $S \subseteq C \times \mathcal{P}(C)$,*

$$S \subseteq \mathrm{step}_R(\mathrm{expand}^*_R(S)) \text{ implies } S \subseteq \mathrm{valid}_R$$

This lemma replaces the entire axiomatic semantics shown in Fig. 3.2, and it's proof replaces the soundness proof that would be required for that axiomatic semantics. The only formal definition specific to IMP that is needed is an operational semantics from Fig. 2.3, 2.4 or 2.5. Furthermore, the statement and proof of this lemma do not depend on details of the target relation. Instead of requiring language designers to know how to prove the soundness of a customized proof system they need only provide an operational semantics. This lemma is proven once and covers all languages.

Now we show how to use this lemma to prove the example program. These definitions and results may appear like abstract mathematics quite far from the world of programming but we will see that we can use them in ways directly corresponding to (symbolic) execution in the operational semantics.

Expressed as a set of claims our specification is

$$\begin{aligned} S \equiv\ & \{(\langle \texttt{s = 0; loop; } T \mid \texttt{n} \mapsto n, \texttt{s} \mapsto s, \sigma\rangle \\ & \quad , \{\langle T \mid \texttt{n} \mapsto n', \texttt{s} \mapsto n(n-1)/2, \sigma\rangle \mid \forall n'\}) \mid \forall n, s, T, \sigma\} \\ & \cup \{(\langle \texttt{while (n--!=0) \{s=s+n\}; } T \mid \texttt{n} \mapsto n, \texttt{s} \mapsto s, \sigma\rangle \\ & \quad , \{\langle T \mid \texttt{n} \mapsto n', \texttt{s} \mapsto s + n(n-1)/2, \sigma\rangle \mid \forall n'\}) \mid \forall s, n, T, \sigma\} \end{aligned}$$

The specification holds if $S \subseteq \mathrm{valid}_R$. By Lemma 5, it suffices to show

$$S \subseteq \mathrm{step}_R(\mathrm{expand}^*_R(S)) \tag{3.3}$$

We introduce an abbreviation for the goal sets

$$G(T, n, s, \sigma) = \{\langle T \mid \texttt{n} \mapsto n', \texttt{s} \mapsto s + n(n-1)/2, \sigma\rangle \mid \forall n'\}$$

We begin the proof by splitting the desired inclusion into a separate goal for each family of claims, using the simple fact that $(A \cup B) \subseteq X$ iff $A \subseteq X$ and $B \subseteq X$. This leaves two inclusions to be shown

$$\begin{aligned} \{(\langle \texttt{s=0;loop;}T \,|\, \texttt{n} \mapsto n, \texttt{s} \mapsto s, \sigma\rangle, G(T,n,0,\sigma)) \mid \forall T,n,s,\sigma\} & \\ \subseteq \operatorname{step}_R(\operatorname{expand}^*_R(S)) & \qquad (3.4) \\ \{(\langle \texttt{loop;}\ \ T \,|\, \texttt{n} \mapsto n, \texttt{s} \mapsto s, \sigma\rangle, G(T,n,s,\sigma)) \mid \forall T,n,s,\sigma\} & \\ \subseteq \operatorname{step}_R(\operatorname{expand}^*_R(S)) & \qquad (3.5) \end{aligned}$$

The claims making up the overall specification of the program are handled in Eq. (3.4). The first execution step of the state in each claim executes the assignment to s regardless of the value of the variables

$$\forall n,s,T,\sigma.\, \langle \texttt{s=0;loop;}T \,|\, \texttt{n} \mapsto n, \texttt{s} \mapsto s, \sigma\rangle \to \langle \texttt{skip;loop;}T \,|\, \texttt{n} \mapsto n, \texttt{s} \mapsto 0, \sigma\rangle$$

We can use this with execution step with the second case in the definition of step_R to reduce Eq. (3.4) to showing

$$\{(\langle \texttt{skip;loop;}T \,|\, \texttt{n} \mapsto n, \texttt{s} \mapsto 0, \sigma\rangle, G(T,n,0,\sigma)) \mid \forall n,T,\sigma\} \subseteq \operatorname{expand}^*_R(S)$$

The initial state of these claims can take equivalent steps eliminating `skip`. To use these steps as before we need step_R on the right-hand side. We expose an instance of step_R by strengthening the goal with the following inclusion, shown by unfolding the definition of expand_R and the fixpoint in $-^*$

$$\begin{aligned} \operatorname{step}_R(\operatorname{expand}^*_R(S)) &\subseteq S \cup (\operatorname{step}_R \cup \operatorname{trans})(\operatorname{expand}^*_R(S)) \\ &= S \cup \operatorname{expand}_R(\operatorname{expand}^*_R(S)) = \operatorname{expand}^*_R(S) \end{aligned}$$

Now we can use an execution step as before, reducing the goal to showing

$$\{(\langle \texttt{loop;}T \,|\, \texttt{n} \mapsto n, \texttt{s} \mapsto 0, \sigma\rangle, G(T,n,0,\sigma)) \mid \forall n,T,\sigma\} \subseteq \operatorname{expand}^*_R(S)$$

All these claims are instances of the loop claims from the original specification with $s = 0$. To use the original claims we strengthen the goal by the inclusion

$$S \subseteq S \cup \operatorname{expand}_R(\operatorname{expand}^*_R(S)) = \operatorname{expand}^*_R(S)$$

and finish the proof of Eq. (3.4) by noting that

$$\{(\langle \texttt{loop;}T \,|\, \texttt{n}\mapsto n, \texttt{s}\mapsto 0, \sigma\rangle, G(T,n,0,\sigma)) \mid \forall n, T, \sigma\} \subseteq S$$

Now it remains to prove Eq. (3.5). As before we begin showing inclusion in $\text{step}_R(\text{expand}^*_R(S))$ by using a suitable execution step. In this case the first step always unfolds the while statement, leaving goal

$$\begin{aligned}\{(\langle \texttt{if(n--!= 0)\{s=s+n;loop\}else\{skip\};}T \,|\, \texttt{n}\mapsto n, \texttt{s}\mapsto s, \sigma\rangle& \\ , G(T,n,s,\sigma)) \mid \forall T, n, s, \sigma\} \subseteq \text{expand}^*_R(S)&\end{aligned}$$

Using $\text{step}_R(\text{expand}^*_R(S)) \subseteq \text{expand}^*_R(S)$ as before, we take more execution steps until the if-condition is a single but symbolic Boolean value.

$$\begin{aligned}\{(\langle \texttt{if(}n \neq 0\texttt{)\{s=s+n;loop\}else\{skip\};}T \,|\, \texttt{n}\mapsto n-1, \texttt{s}\mapsto s, \sigma\rangle& \\ , G(T,n,s,\sigma)) \mid \forall T, n, s, \sigma\} \subseteq \text{expand}^*_R(S)&\end{aligned}$$

Further progress requires dividing this into cases for $n \neq 0$ and $n = 0$. A case distinction is made by observing that $A \cup B \subseteq X$ if both $A \subseteq X$ and $B \subseteq X$, leaving two inclusions to check

$$\begin{gathered}\{(\langle \texttt{if(}\top\texttt{)\{s=s+n;loop\}else\{skip\};}T \,|\, \texttt{n}\mapsto n-1, \texttt{s}\mapsto s, \sigma\rangle \\ , G(T,n,s,\sigma)) \mid \forall T, n, s, \sigma.\, n \neq 0\} \subseteq \text{expand}^*_R(S) \\ \{(\langle \texttt{if(}\bot\texttt{)\{s=s+n;loop\}else\{skip\};}T \,|\, \texttt{n}\mapsto -1, \texttt{s}\mapsto s, \sigma\rangle \\ , G(T,0,s,\sigma)) \mid \forall T, s, \sigma\} \subseteq \text{expand}^*_R(S)\end{gathered}$$

In each case we translate more execution steps to proof steps, leaving goals

$$\begin{gathered}\{(\langle \texttt{loop;}T \,|\, \texttt{n}\mapsto n-1, \texttt{s}\mapsto s+(n-1), \sigma\rangle, G(T,n,s,\sigma)) \mid \forall T, n, s, \sigma, n \neq 0\} \\ \cup \{(\langle T \,|\, \texttt{n}\mapsto -1, \texttt{s}\mapsto s, \sigma\rangle, G(T,0,s,\sigma)) \mid \forall T, s, \sigma\} \subseteq \text{expand}^*_R(S)\end{gathered}$$

In the $n = 0$ case, the current configuration is already in the corresponding target set. To conclude, we expose an application of step_R as before but now use the right case $c \in P$ in the definition of step_R to leave the goal

$$\forall T, s, \sigma.\, \langle T \,|\, \texttt{n}\mapsto -1, \texttt{s}\mapsto s, \sigma\rangle \in \{\langle T \,|\, \texttt{n}\mapsto n', \texttt{s}\mapsto s, \sigma\rangle \mid \forall n'\}$$

In the $n \neq 0$ case, the claims would match loop claims in S if

$$G(T, n, s, \sigma) = G(T, n-1, (n-1)+s, \sigma)$$

Expanding the definition of G, this holds if

$$s + n(n-1)/2 = s + (n-1) + (n-1)(n-2)/2$$

which is true for n as we saw earlier. This concludes the proof of Eq. (3.3), and thus by Lemma 5 we have the inclusion $S \subseteq \text{valid}_R$, meaning that the sum program `s=0;loop` satisfies its specification.

Reasoning with fixpoints and functions like step_R can be thought of as reasoning with proof rules, but ones which interact with the target programming language only through its operational semantics. step_R corresponds to two rules: taking an execution step and showing that the current configuration is in the target set. Unions correspond to case analysis. The fixpoint in the closure definition corresponds to iterative uses of these proof rules or to referring back to claims in the original specification.

HIMP

```
append(x, y) {
  var p;
  if (x = 0) return y;
  p := x;
  while (*(p+1) <> 0)
    p := *(p+1);
  *(p+1) := y;
  return x;
}
```

Lambda

```
(λ (λ ifNil 1 0
 ((λ (λ 0 0)
     (λ 1 (λ 1 1 0))) (λ
  (λ (λ (λ 0 1)) (deref 0)
    (λ ifNil (cdr 0)
      ((λ 5) (assign 0
        (cons (car 0) 3)))
      (2 (cdr 0)))))
  1)))
```

Stack

```
: append over if
over begin 1+ dup @ dup while nip repeat drop !
else nip then ;
```

Figure 3.4: Destructive list append in three languages

Before we prove the correctness of our coinductive verification approach, we consider the following pragmatic question: "Can this simple approach really work?". We have implemented it in Coq and specified and verified programs in a variety of languages, each language being defined as an operational semantics. Our experiments can be found at `http://fsl.cs.illinois.edu/coinduction`. We empirically show not only that coinductive program verification is feasible and versatile, but also that it is amenable to highly effective proof automation.

3.1.4 Languages

Here we discuss three languages following different paradigms, each defined operationally. Many more operational semantics are available with the distributions of e.g., $\mathbb{K}$ [RȘ10], PLT-Redex [Kle+], and Ott [Sew+07], but we believe these three language are sufficient to demonstrate that our verification approach works across several language paradigms. Fig. 3.4 shows a destructive linked-list append function in each language.

HIMP (IMP with Heap) is an imperative language with (recursive) functions and a heap. The heap addresses are integers, to demonstrate reasoning about low-level representations, and memory allocation/deallocation are primitives. The configuration is a 5-tuple of current code, local variable environment mapping identifiers to values, call stack with frames as pairs of code and environment, heap, and a collection of functions as a map from function name to definition. HIMP is defined with a $\mathbb{K}$-style semantics.

Indeed the Coq definition of HIMP originates before work began on coinductive program verification, being first developed in connection with work on formalizing the soundness proofs for Reachability Logic Chapter 7 and to experiment with translating $\mathbb{K}$ definitions to Coq. Reachability Logic basically requires $\mathbb{K}$-style semantics, but as we examine in Section 4.1.1 with coinductive program verification there are no major advantages or disadvantages between styles of operational semantics.

Stack is a Forth-like stack based language, though, unlike in Forth, we do make control structures part of the grammar. A single shared data stack is used both for local state and to communicate between function invocations, eliminating the store, formal parameters on function declarations, and the environment of stack frames. Stack's configuration is also a 5-tuple, but instead of a current environment there is a stack of values, and stack frames do not store an environment. Lacking any tree-structured syntax, the distinction between styles of semantics collapses, and the transition relation simply enumerates the stack effect of each command.

Lambda is a simple call-by-value lambda calculus, extended with primitive integer, pair and nil values, and primitive operations for heap access. Unlike HIMP or Stack, fixpoint combinators enable recursive definitions without relying on primitive support for named functions. For further minimalism we also use DeBruijn indices instead of named variables. The semantics is based on a CEK machine, extended with a heap. Lambda's configuration is a 4-tuple of current expression, current environment, the heap, and a current continuation.

3.1.5 Specifying Data Structures

Our coinductive verification approach is orthogonal to how claims in $C \times \mathcal{P}(C)$, or sets of them, are specified. Presented abstractly, specifications can use arbitrary sets or predicates. In our Coq implementation, this corresponds to using any predicates definable in Coq's logic. Within this design space, we began with Matching Logic [RES] for our experiments, building on earlier work on *reachability logic* discussed in Chapter 7. The patterns of matching logic generalize both unification and first order logic formulas. The basic idea is that operator for building values should also be available in patterns, connecting patterns over subterms to form a pattern matching values formed by that operator from values matching the sub-patterns, similarly to unification. In addition, patterns can also be combined logically by the usual FOL connectives, just like formulas. For example, pattern $P \wedge Q$ matches a value if P and Q both match it, $[t]$ matches only the value t, $\exists x.P$ matches if there is any assignment of x under which P matches, and $[\![\varphi]\!]$ where φ is a FOL formula matches any value if φ holds and no values otherwise.[1]

To specify programs manipulating heap data structures we use patterns matching subheaps that contain a data structure representing an abstract value. The operators we use for writing heap patterns are influenced by separation logic and particularly the "computational separation logic" used in Bedrock [Chla]. We define representation predicates for data structures as functions from an abstract description of a value into a primitive pattern describing the its representation in memory. The basic ingredients for this approach are the primitive patterns for describing maps: pattern **emp** for the empty map, $k \mapsto v$ for the singleton map binding key k to value v, and P, Q for maps which are disjoint unions of submaps respectively matching P and Q. We use abbreviation $\langle\varphi\rangle \equiv [\![\varphi]\!] \wedge \mathbf{emp}$ to facilitate inline logical assertions and $p \mapsto [v_0, \ldots, v_i] \equiv p \mapsto v_0, (p+1) \mapsto v_1, \ldots, (p+i) \mapsto v_i$ to describe values at contiguous addresses.

A pattern that matches a heap containing only a linked list starting at address p and representing the list l of integers is defined recursively by

$$\begin{aligned}&\mathrm{list}(\mathrm{nil}, p) = \langle p = 0\rangle\\ &\mathrm{list}(x : l, p) = \langle p \neq 0\rangle, \exists p_l . p \mapsto [x, p_l], \mathrm{list}(l, p_l)\end{aligned}$$

We also define a predicate list_seg(l, e, p) for list segments, useful in algorithms using pointers to the middle of a list, by generalizing the constant 0 to the trailing pointer parameter e. Simple binary trees can be defined by

$$\begin{aligned}&\mathrm{tree}(\mathrm{leaf}, p) = \langle p = 0\rangle\\ &\mathrm{tree}(\mathrm{node}(x, l, r), p) = \langle p \neq 0\rangle, \exists p_l, p_r, p \mapsto [x, lp, rp], \mathrm{tree}(l, lp), \mathrm{tree}(r, rp).\end{aligned}$$

[1] In the presentation of [RES] neither $[t]$ nor $[\![\varphi]\!]$ require a visible marker, but in Coq patterns are a distinct type from terms and pure formulas, requiring explicit injections.

Given such representation predicates, specifications and proofs can be done in terms of the abstract values represented in memory, rather than details of addresses and map entries. By relying on widely reusable (across different languages) proof automation that deals with the primitive patterns, the proof scripting specific to such pattern definitions is concerned exclusively with unfolding the definition when allowed, deciding what abstract value, if any, is represented at a given address in a partially unfolded heap, which helps decide how another claim applies to the current state when attempting a transitivity step.

3.1.6 Specifying Reachability Claims

As mentioned, claims in $C \times \mathcal{P}(C)$ can be specified using any logical formalism, here the full power of Coq. An explicit specification can be verbose and low-level, especially when many semantic components in the configuration stay unchanged. However, any reasonable logic allows making definitions to reduce redundancy. Our use of matching logic particularly facilitates framing conditions, allowing us to regain the compactness and elegance of Hoare logic specifications with definable syntactic sugar. For example, defining

$$\begin{aligned}
&\mathit{call}(f(\mathit{formals})\{\mathit{body}\}, \mathit{args}, P_{\mathit{in}}, P_{\mathit{out}}) = \\
&\quad \{(\langle f(\mathit{args}) \curvearrowright \mathit{rest}, \mathit{env}, \mathit{stk}, \mathit{heap}, \mathit{funs}\rangle, \\
&\qquad \{\langle r \curvearrowright \mathit{rest}, \mathit{env}, \mathit{stk}, \mathit{heap}', \mathit{funs}\rangle \mid \forall r, \mathit{heap}'. \mathit{heap}' \vDash P_{\mathit{out}}(r), [H_f]\}) \\
&\qquad\qquad\qquad\qquad \mid \forall \mathit{rest}, \mathit{env}, \mathit{stk}, \mathit{heap}, H_f, \mathit{funs}. \\
&\qquad\qquad\qquad\qquad\quad \mathit{heap} \vDash P_{\mathit{in}}, [H_f] \wedge f \mapsto f(\mathit{formals})\{\mathit{body}\} \in \mathit{funs}\}
\end{aligned}$$

gives an equivalent of the usual Hoare pre-/post-condition on function calls. The first parameter is the function definition and the second is the list of arguments. The heap effect is described as a pattern P_{in} for the allowable initial states of the heap and function P_{out} from returned values to corresponding heap patterns. For example, we can now write a specification for the definition D of the append function in Fig. 3.4 as

$$\bigcup_{x,y,a,b} \mathit{call}(D, [x, y], (\mathrm{list}(a, x) \star \mathrm{list}(b, y)), (\lambda r.\, \mathrm{list}(a{+\!\!+}b, r))) \tag{3.6}$$

$$
\begin{array}{l}
\mathit{call}(\text{Head}, [x], [H] \wedge \text{list}(v : l, x), \lambda r.\langle r = v\rangle, [H]) \\
\mathit{call}(\text{Tail}, [x], [H] \wedge \text{list}(v : l, x), \lambda r.[H] \wedge \_, \text{list}(l, r)) \\
\mathit{call}(\text{Add}, [y, x], \text{list}(l, x), \lambda r.\,\text{list}(y : l, r)) \\
\mathit{call}(\text{Add}', [y, x], [H] \wedge \text{list}(l, x), \lambda r.\,\text{list}\_\mathit{seg}([y], x, r), H) \\
\mathit{call}(\text{Swap}, [x], \text{list}(a : b : l, x), \lambda r.\,\text{list}(b : a : l, x)) \\
\mathit{call}(\text{Dealloc}, [x], \text{list}(l, x), \lambda r.\mathbf{emp}) \\
\mathit{call}(\text{Length}, [x], [H] \wedge \text{list}(l, x), \lambda r.\langle r = \mathit{len}(l)\rangle, [H]) \\
\mathit{call}(\text{Sum}, [x], [H] \wedge \text{list}(l, x), \lambda r.\langle r = \mathit{sum}(l)\rangle\rangle, [H]) \\
\mathit{call}(\text{Reverse}, [x], \text{list}(l, x), \lambda r.\,\text{list}(\mathit{rev}(l), r)) \\
\mathit{call}(\text{Append}, [x, y], \text{list}(a, x), \text{list}(b, y), \lambda r.\,\text{list}(a{+\!\!+}b, r)) \\
\mathit{call}(\text{Copy}, [x], [H] \wedge \text{list}(l, x), \lambda r.\,\text{list}(l, r), [H]) \\
\mathit{call}(\text{Delete}, [v, x], \text{list}(l, x), \lambda r.\,\text{list}(\mathit{delete}(v, l), r))
\end{array}
$$

Table 3.1: Example list specifications

which is as compact and elegant as it can be. More specifications are given in Table 3.1. A number of specifications assert that part of the heap is left entirely unchanged by writing $[H] \wedge P$ in the precondition so that H must be exactly the portion of the heap that satisfied P, and then using H in the conclusion as well. For example, the Copy specification lets H name the list in the original state, and then mentions H in the postcondition to ensure the original list is untouched, and disjoint from the returned list. The specifications Add and Add' show that it can be a bit more complicated to assert that an input list is used undisturbed as a suffix of a result list. The Add specification simply requires that result has a list containing the new element before the previous entries, but would not forbid an implementation that reallocated all the old list nodes to new locations. Specifications such as Length, Append, and Delete are written in terms of corresponding mathematical functions on the lists represented in the heap. In the proofs, reasoning about properties of those functions are separated from details of memory layout.

When a function contains loops, proving that it meets a specification often requires making some additional claims about configurations which are just about to enter loops, as we saw in Section 3.1.3. We support this with another pattern that describes the current state with some suffix of the function body and a description of the local variables:

$$\begin{aligned}
&\mathit{stmt}(\mathit{code}, \mathit{env}, P_{\mathit{in}}, P_{\mathit{out}}) = \{(\langle \mathit{code}, (\mathit{env}, e_f), \mathit{stk}, \mathit{heap}, \mathit{funs}\rangle, \\
&\quad \{\langle \texttt{return}\ r \curvearrowright \mathit{rest}, \mathit{env}', \mathit{stk}, \mathit{heap}', \mathit{funs}\rangle \ \mid \forall r, \mathit{rest}, \mathit{env}', \mathit{heap}'. \\
&\qquad \mathit{heap}' \vDash P_{\mathit{out}}(r), [H_f]\}) \ \mid \forall e_f, \mathit{stk}, \mathit{heap}, H_f, \mathit{funs}\ .\ \mathit{heap} \vDash P_{\mathit{in}}, [H_f]\}
\end{aligned}$$

Verifying that the definition of append in Fig. 3.4 meeds the specification in Eq. (3.6) requires an auxiliary claim about the loop, which can be written

$$\begin{aligned}
&\mathit{stmt}(\texttt{while (*(p+1) <> 0)}\ldots, (\texttt{x} \mapsto x, \texttt{y} \mapsto y, \texttt{p} \mapsto p), \\
&\quad (\mathrm{list\_seg}(l_x, p, x), \mathrm{list}(l_p, p), \mathrm{list}(l_y, y)), (\lambda r.\, \mathrm{list}(l_x \mathbin{+\!\!+} l_p \mathbin{+\!\!+} l_y, r))).
\end{aligned}$$

The patterns above were described using HIMP's configurations, but we defined similar ones for Stack and Lambda also.

3.2 Correctness of Coinductive Program Verification

The core of our approach is to apply generalized coinduction to a characterization of validity as a greatest fixpoint. The fixpoint theorem we have used for most of our work is equivalent to a specialization of Bartel's λ-coiteration to the setting of lattices (the original work used more general categorical language). A categorical dual of Bartel's theorem was independently presented in "Recursion Schemes from Comonads" [UVP01]. as a scheme for defining recursive functions.

First we fix notation and recall standard definitions and results. Recall from Section 3.1.3 that our specifications are sets of claims from set $C \times \mathcal{P}(C)$, that is elements of $\mathcal{P}(C \times \mathcal{P}(C))$, where C is the set of configurations of the target language, and that an operational semantics $R \subseteq C \times C$ over configurations in C yields a function $\text{step}_R : \mathcal{P}(C \times \mathcal{P}(C)) \to \mathcal{P}(C \times \mathcal{P}(C))$. Our main fixpoint theorem will be stated slightly more generally, for any powerset lattice $(\mathcal{P}(D), \cup)$. This is the same setting used in the original presentation of the Knaster-Tarski theorem [Tar55], and like it our result can also be generalized to any complete lattice, and even to various categories, but our objective here is to keep the presentation as simple and accessible as possible without losing useful generality. Below, F and G range over monotone functions $\mathcal{P}(D) \to \mathcal{P}(D)$, and X, Y, A over sets in $\mathcal{P}(D)$. We lift union and subset pointwise on functions, so $(F \cup G)(X) = F(X) \cup G(X)$ and $F \subseteq G \equiv \forall X . F(X) \subseteq G(X)$.

We recall a few definitions from Chapter 2. A fixpoint of F is a set A with $F(A) = A$. The Knaster-Tarski theorem implies that any monotone function F has a least and greatest fixpoint, respectively denoted μF and νF.

A defining property of least fixpoints is *induction*: $F(A) \subseteq A$ implies $\mu F \subseteq A$ for any set A. Dually, greatest fixpoints allow *coinduction*: $A \subseteq F(A)$ implies $A \subseteq \nu F$ for any set A. A set A is called F*-stable* if $A \subseteq F(A)$. A *closure operation* is a monotone function F satisfying the additional properties of being *extensive* ($\forall X, X \subseteq F(X)$) and *idempotent* ($\forall X, F(F(X)) = F(X)$).

First, we justify the name of the F*-closure* operation F^* (Definition 3.4):

Lemma 6. *If F is monotone then F^* is the least closure operator with $F \subseteq F^*$.*

Proof. First, F^* is a closure operator. For monotonicity, fix $A \subseteq B$. Then $F^*(A) \subseteq F^*(B)$ holds by induction, from $F(F^*(B)) \cup A \subseteq F(F^*(B)) \cup B = F^*(B)$. Extensiveness holds by $X \subseteq F(F^*(X)) \cup X = F^*(X)$. For idempotence it suffices, given extensiveness, to show $F^*(F^*(X)) \subseteq F^*(X)$, which follows by induction from $F(F^*(X)) \cup F^*(X) = F(F^*(X)) \cup F(F^*(X)) \cup X = F^*(X)$.

Next, F^* is above F by $F(X) \subseteq F(F^*(X)) \subseteq F(F^*(X)) \cup X = F^*(X)$. If $F \subseteq G$ for closure operator G, $F^* \subseteq G$ by induction from $F(G(X)) \cup X \subseteq G(G(X)) \cup G(X) \subseteq G(X)$, so F^* is the least closure operator above F. □

Corollary 1. *The operation $-^*$ is monotone. For monotone F and G, if $F \subseteq G$, then G^* is a closure operator with $F \subseteq G \subseteq G^*$, so $F^* \subseteq G^*$.*

Now we are ready to state and prove our key coinduction theorem.

Theorem 2 (Coinduction with Rules)**.** *If $G(F(A)) \subseteq F(G^*(A))$*[2] *for any A, then $X \subseteq F(G^*(X))$ implies $X \subseteq \nu F$ for any X.*

Proof. By extensiveness, $X \subseteq \nu F$ is implied by $G^*(X) \subseteq \nu F$, which follows by coinduction from $G^*(X) \subseteq F(G^*(X))$, which follows by induction from $G(F(G^*(X))) \cup X \subseteq F(G^*(X))$. By idempotence this is equivalent to instance $G(F(G^*(X))) \cup X \subseteq F(G^*(G^*(X)))$ of the hypothesis. □

In our program-verification setting with languages given as relations $R \subseteq C \times C$, F is usually step_R. To verify programs, we express valid_R as a greatest fixpoint:

Lemma 7. $\text{valid}_R = \nu\,\text{step}_R$

Proof. valid_R is step_R-stable. Suppose $(c, P) \in \text{valid}_R$. If $c \in P$ then $(c, P) \in \text{step}_R(\text{valid}_R)$, else c has a successor d which either starts an infinite path in R or reaches an element of P. In either case, $(d, P) \in \text{valid}_R$ so $(c, P) \in \text{step}_R(\text{valid}_R)$.

valid_R is the largest step_R-stable set. Fix $S \subseteq \text{step}_R(S)$ and let $(c, P) \in S$. If there is an infinite path in R beginning at c then $(c, P) \in \text{valid}_R$. Otherwise,

[2] This can be weakened to quantify only over $A \subseteq \nu F$, at the cost of compositionality.

R is well-founded on the set of configurations reachable from c. Then by well-founded induction $(d, P) \in \text{valid}_R$ for any d with $c \to_R d$ and $(d, P) \in S$. If $c \in P$ then $(c, P) \in \text{valid}_R$, else by $(c, P) \in S \subseteq \text{step}_R(S)$ and the induction hypothesis there is a successor d of c with $(d, P) \in \text{valid}_R$, and thus also $(c, P) \in \text{valid}_R$. □

We also need to provide some useful choices for G in Theorem 2. Fixing F, we can see that the set of functions G that meet the condition $G(F(X)) \subseteq F(G^*(X))$ for all X includes F and is closed under union. This motivates the following:

Definition 3.6. Given monotone functions F and H, we say that H is a *valid derived rule for* F whenever $H(F(X)) \subseteq F((F \cup H)^*(X))$ for all X.

This condition is also closed under pointwise union. Using $(F \cup H)^*$ rather than H^* on the right side usefully weakens the condition. For example, the trans function is a valid derived rule for step_R, but it is not the case that $\text{trans}(\text{step}_R(X)) \subseteq \text{step}_R(\text{trans}^*(X))$. Any claim in $\text{trans}(\text{step}_R(X))$ is in $\text{step}_R(\text{trans}(X \cup \text{step}_R(X)))$, but not necessarily in $\text{step}_R(\text{trans}^*(X))$.
Now we show that two useful rules meet this condition. One is the trans rule of Definition 3.5, which was defined by

$$\text{trans}(S) = \{(c, P) \mid \exists Q.(c, Q) \in S \land \forall d \in Q, (d, P) \in S\}$$

and is a valid derived rule for any step_R. The other is a rule for using any claim previously shown valid.

Definition 3.7. Function proved_R is defined by

$$\text{proved}_R(S) = S \cup \text{valid}_R$$

For any transition relation R, proved_R will be a valid derived rule for the instance step_R with the same R.

Lemma 8. trans *is a valid derived rule for* step_R.

Proof. Suppose $(c, P) \in \text{trans}(\text{step}_R(S))$. Fix Q from the definition $(c, Q) \in \text{step}_R(S)$, so either $c \in Q$ or $(c', Q) \in S$ for some c' with $c \to_R c'$.

In the first case let d be c to conclude $(c, P) \in \mathrm{step}_R(S)$. Otherwise we use $\forall d \in Q, (d, P) \in \mathrm{step}_R(S)$ to conclude $(c', P) \in \mathrm{trans}(S \cup \mathrm{step}_R(S))$, and thus $(c, P) \in \mathrm{step}_R(\mathrm{trans}(S \cup \mathrm{step}_R(S)))$.

By monotonicity properties both $\mathrm{step}_R(S)$ and $\mathrm{step}_R(\mathrm{trans}(S \cup \mathrm{step}_R(S)))$ are contained in $\mathrm{step}_R((\mathrm{trans} \cup \mathrm{step}_R)^*(S))$. □

Lemma 9. *Function* proved_R *is a valid derived rule for* step_R*.*

Proof. If $(c, P) \in \mathrm{proved}_R(\mathrm{step}_R(S))$ either $(c, P) \in \mathrm{valid}_R = \mathrm{step}_R(\mathrm{valid}_R)$ or $(c, P) \in \mathrm{step}_R(S)$. Note $\mathrm{valid}_R \cup S = \mathrm{proved}_R(S) \subseteq (\mathrm{proved}_R \cup \mathrm{step}_R)^*(S)$. □

3.2.1 Nondeterministic Languages

Theorem 2 can be used with any specifications whose validity can be characterized as a greatest fixpoint, not just our partial reachability claims.

One example that may be of particular interest is an "all-paths" notion of reachability, which constrains every execution path from an initial configuration rather than just requiring one good path. This can be more difficult to work with, but is more suitable for nondeterministic languages.

Definition 3.8 (All-Path Partial Reachability)**.** We define $c \Rightarrow_R^\forall P$ to hold if every path from c that reaches a stuck configuration passes through set P. More precisely, if $c = \gamma_1, \ldots, \gamma_n$ is any execution path in R such that γ_n has no successors in R, then there exists some i such that $\gamma_i \in P$.

We define $\mathrm{valid}_R^\forall = \{(c, P) \mid c \Rightarrow_R^\forall P\}$ and will prove that this is the greatest fixpoint of

$$\begin{aligned} \mathrm{step}_R^\forall(S) = \{(c, P) \mid {} & c \in P \\ & \vee (\exists d.(c, d) \in R) \wedge \forall d.(c, d) \in R \implies (d, P) \in S\} \end{aligned}$$

Lemma 10. $\mathrm{valid}_R^\forall = \nu\, \mathrm{step}_R^\forall$

Proof. $\mathrm{valid}_R^\forall$ is $\mathrm{step}_R^\forall$-stable. Suppose $(c, P) \in \mathrm{valid}_R^\forall$. If $c \in P$ then also $\mathrm{step}_R^\forall(\mathrm{valid}_R^\forall)$. Otherwise, c must have at least one successor, because the trivial path c avoids P and would otherwise lead from c to a stuck configuration. If any successor d of c was not in (d, P) then there would be some path $d = \gamma_1, \ldots, \gamma_n$ to a stuck configuration that avoids P, but because $c \notin P$

then $c, \gamma_1, \ldots, \gamma_n$ would be a path from c to a stuck configuration that avoids P.

$\mathrm{valid}^{\forall}_R$ is the largest $\mathrm{step}^{\forall}_R$-stable set. Suppose $X \subseteq \mathrm{step}^{forall}_R(X)$, $(c, P) \in X$ and $c = \gamma_1, \ldots, \gamma_n$ is a path in R from c to a stuck configuration. By induction on n we will show that this path encounters P. If $n = 1$ then c has no successors in R so $(c, P) \in \mathrm{step}^{\forall}_R(X)$ can only hold if $c \in P$, in which case $(c, P) \in \mathrm{valid}^{\forall}_R$. Otherwise, $n > 1$. Then we have $(\gamma_2, P) \in X$ by the definition of $\mathrm{step}^{\forall}_R$, so by the inductive hypothesis the path $\gamma_2, \ldots, \gamma_n$ encounters the set P, so the path $c = \gamma_1, \ldots, \gamma_n$ does as well. This concludes the inductive proof. Rearranging quantifiers, we have that for any $(c, P) \in X$ every path from c to a stuck configuration hits P, which is the definition of $(c, P) \in \mathrm{valid}^{\forall}_R$. □

Incidentally, the least fixpoints of step_R and $\mathrm{step}^{\forall}_R$ correspond to total-correctness definitions of reachability, which lack the exceptions allowing infinite executions to avoid the target set, but this observation is not useful for program verification because induction principles have inclusions in the wrong direction to be useful for showing that a specification holds.

This definition also admits transitivity, with exactly the same definition of trans used in the one-path case.

Lemma 11. trans *is also a valid derived rule for* $\mathrm{step}^{\forall}_R$.

Proof. Suppose $(c, P) \in \mathrm{trans}(\mathrm{step}^{\forall}_R(S))$. Fix Q from the definition of trans. Then $(c, Q) \in \mathrm{step}^{\forall}_R(S)$, so either $c \in Q$ or c is not stuck and $(c', Q) \in S$ for every c' with $c \to_R c'$.

In the first case let d be c to conclude $(c, P) \in \mathrm{step}^{\forall}_R(S)$. Otherwise we use $\forall d \in Q, (d, P) \in \mathrm{step}^{\forall}_R(S)$ to conclude that $(c', P) \in \mathrm{trans}(S \cup \mathrm{step}^{\forall}_R(S))$ for every c' with $c \to_R c'$, and thus $(c, P) \in \mathrm{step}^{\forall}_R(\mathrm{trans}(S \cup \mathrm{step}^{\forall}_R(S)))$.

By monotonicity properties both $\mathrm{step}^{\forall}_R(S)$ and $\mathrm{step}^{\forall}_R(\mathrm{trans}(S \cup \mathrm{step}^{\forall}_R(S)))$ are contained in $\mathrm{step}^{\forall}_R((\mathrm{trans} \cup \mathrm{step}^{\forall}_R)^*(S))$. □

3.2.2 Modularity

Verifying a larger program demands some way of dividing the proof into smaller lemmas, corresponding to modules of the program. The proved_R rule gives one way of handling completely self-contained components. If a component can be verified in isolation, then proofs about the rest of the program can use those claims through to proved_R rule rather than including them in an single set of claims to be coinductively supported.

Another approach to modular proof uses monotonicity properties. An inclusion of the form $A \subseteq \text{step}_R((G \cup \text{step}_R)^*(B))$ where G is a valid derived rule for step_R can be seen as an incomplete proof of specification A with unproved assumptions $B \setminus A$. For example, A could be a function's public specification and loop lemmas with B also including specifications of library functions. Two such inclusions compose by monotonicity properties to give $(A_1 \cup A_2) \subseteq \text{step}_R(((G_1 \cup G_2) \cup \text{step}_R)^*(B_1 \cup B_2))$, which proves a larger set of claims $A_1 \cup A_2$ under a potentially smaller set of remaining assumptions $(B_1 \cup B_2) \setminus (A_1 \cup A_2)$. Once enough partial arguments are combined that $B \subseteq A$ then Theorem 2 shows the validity of all the combined claims.

3.3 Relative Completeness

We are not presenting a syntactic proof system, so we carefully consider how to formulate relative completeness. Intuitively, we want to formulate the condition that any desired set X of valid claims can be proven with our approach. This is satisfied if $X \subseteq \text{valid}_R$ is the conclusion of an application of Theorem 2. However, requiring exactly this condition is too strict. Proving a Hoare triple may necessarily require providing additional loop invariants. Likewise, proving a specification of interest in our system may require also making claims about loops and auxiliary functions. In our system this is done by enlarging the original set of claims. The correct notion of relative completeness is thus to ask whether the desired set X of valid claims is contained in some larger set S for which we can conclude $S \subseteq \text{valid}_R$ as an application of Theorem 2.

For any set X and any choice of G we can in fact take the set valid_R of all true claims. As part of showing valid_R is a fixpoint we established that $\text{valid}_R \subseteq \text{step}_R(\text{valid}_R)$. By monotonicity and extensiveness,

$$\text{valid}_R \subseteq \text{step}_R(F^*(\text{valid}_R))$$

This leaves only the goal of showing $X \subseteq S = \text{valid}_R$.

One might complain that this is trivial, but then one should complain all the more about a conventional relative-completeness result. Any general-purpose specification language is necessarily undecidable, so no syntactic proof system can be complete. Instead, any relatively complete proof system has a rule with a hypothesis of semantic validity in some predicate language, and the relative-completeness argument consists of tediously showing how to Gödel-encode validity into the predicate language, and showing that the rules of the proof system are strong enough to make use of such a complicated predicate. We obtain an equally strong result.

Chapter 4

Certifying Implementation

In Chapter 3 we saw how program specifications can be proved by coinduction, and that the unreasonable effort required by naive coinduction can be eliminated by using generalized coinduction principles. To use this mathematics as the basis for a sound verification framework we also need a trustworthy mechanical proof checker. We meet this requirement by working within an established proof assistant, whose existing proof-checking tool and proof-certificate format will then also serve to check proofs about programs.

In particular we use Coq for the vast majority of our work. The higher-order type theory supported by Coq, called the Calculus of Inductive Constructions, is sufficient to define the necessary functions on sets of claims and to reason about inclusions between sets as we need for individual proofs.

It is also sufficient for proving the coinduction theorems we need, and the admissibility of the rules such as transitivity that we use in proofs. This gives a machine-checkable proof of the soundness of our system. Having these lemmas proved in Coq also makes it possible to begin a proof about a particular program by claiming that a specification is valid, and then apply the main theorem so the remainder of the proof consists of showing the inclusion needed to establish the stability of the specification and satisfy the hypothesis of the main theorem.

Working within a proof assistant means on the one hand it is necessary to define the language semantics and program-specification predicates in the chosen system, which will be covered in Section 4.1 and Section 4.2, and on the other hand that programmability and any other convenience features of our chosen proof assistant can be used to automate most of the tedious parts of any proof, as discussed in Section 4.3.

In the next chapter we show how the resulting implementation can be used by verifying a collection of example programs and specifications.

4.1 Defining Semantics

To verify programs in Coq, the next most foundational requirement after the core coinduction lemmas is a semantics of the language in question. This consists of a type of configurations and a step relation over those configurations.

In Coq the most explicit way to introduce a new predicate by explicit cases is as an inductively defined proposition. This has the form of a type definition where the new type will belong to the universe `Prop` of propositions (after supplying sufficient arguments), and each constructor of the definition has a type ending in an instance of the type being defined.

As one of the most trivial examples, consider a transition relation over natural numbers, where positive numbers can take a step to their predecessor and 0 is stuck. That relation is captured in this definition:

```
Inductive countdown : nat → nat → Prop :=
  | decrement : ∀ n, countdown (S n) n.
```

The first line says that `countdown` is a predicate over two `nat`ural numbers, or equivalently that `countdown` *n m* is a proposition if *n* and *m* are in `nat`. The later lines introduce the *constructors* as lemmas of the given type, here just `decrement` of type ∀ n, countdown (S n) n. Their conclusions must be instance of the type being defined. A proposition defined by **Inductive** is the smallest predicate which validates the types of all the postulated constructors, or equivalently countdown n m is true for some m and m only if a term of that type can be assembled as a finite tree of applications of the constructors.

For a programming language, assuming we already have a type cfg of configurations, the semantics can be given as a relation

```
Inductive step : cfg → cfg → Prop :=
  ...
```

which will have one constructor corresponding to each rule of the semantics.

It is also allowed for the hypotheses (or arguments) of a constructor to be other instances of the relation being defined, which nicely accommodates the recursion necessary for defining small-step semantics.

Using these constructions, language definitions presented in familiar styles as in Section 2.3 can be translated straightforwardly into Coq definitions.

The translation might be automated for language definitions written in a language-defining language which is itself formally defined. In particular, Chapter 6 describes an automated translator we have developed.

4.1.1 Semantics Styles

Our definition of reachability is not concerned with how a transition relation is defined, only with which transitions are included in the relation. The details of the definition affect proofs only incidentally, by affecting the difficulty of proving whether a particular step is included in the relation, or finding successor(s) of a state, and this difficulty is completely isolated in proof tactics for finding a step.

To evaluate the difficulty and also to demonstrate this encapsulation, we defined the semantics of a simple IMP language in three different styles, resulting in equivalent relations over the same syntax. Then we implemented a proof tactic for finding the successor of a state for each semantics, and showed that the same specification could be proved by the same proof under each semantics, after only changing which tactic was used to find execution steps.

The three styles were a small-step semantics, a reduction semantics, and a $\mathbb{K}$-style semantics.

The syntax was

Listing 4.1: styles.v - syntax

```
Inductive Exp : Set :=
   BCon : bool -> Exp
 | EVar : string  -> Exp
 | ECon : Z -> Exp
 | EGt : Exp -> Exp -> Exp
 | EPlus : Exp -> Exp -> Exp
with Stmt : Set :=
   SAssign : string  -> Exp -> Stmt
 | Seq : Stmt -> Stmt -> Stmt
 | SIf  : Exp -> Stmt -> Stmt -> Stmt
 | Skip : Stmt
 | SWhile : Exp -> Stmt -> Stmt.
```

The small-step semantics is shown in Listing 4.2, the reduction semantics in Listing 4.3, and the K-style semantics in Listing 4.4.

Here is the property and the proof used in each of the three examples.

```
Definition Spec cfg := cfg -> (cfg -> Prop) -> Prop.

Definition loop :=
  SWhile (EGt (EVar "n") (ECon 0%Z))
    (SAssign "n" (EPlus (EVar "n") (ECon (-1)%Z))).

Inductive loop_spec {code} (wrap : Stmt -> code -> code)
    : Spec (code * Map string Z) :=
  | loop_claim : forall store n rest,
      store ~= "n" |-> n ->
      loop_spec wrap (wrap loop rest, store)
    (fun c => fst c = rest
         /\ exists n, snd c ~= "n" |-> n /\ (n <= 0)%Z).

Ltac step_solver := fail.

Ltac trans_solver := solve[refine (loop_claim _ _ _);equate_maps].
Ltac run :=
  first [eapply dtrans;[trans_solver|];try run
        |solve[intros;apply ddone;eauto]
        |eapply dstep;[step_solver|];instantiate;simpl;try run
        ].
```

Each example defines a relation, redefines `step_solver` as an appropriate tactic, and uses the single command `example_proof` as the proof that the specification holds under the given relation.

In a small-step semantics the semantics is defined recursively, with some steps requiring a step to be taken in a subterm. Clauses of the relation can be matched with the current goal by unification, but this may need to be done recursively. The unification could be handled by `eauto using step, exp_step`, but even modest-size code exceeds the default recursion limit of `eauto`, so the corresponding definition of `step_solver` explicitly increases the depth limit to 20 by writing `eauto 20 with step_db`.

In an evaluation-context semantics or reduction semantics, the transition relation itself is defined non-recursively, so only one clause from `step` is necessary for any execution step, but the rules are written in terms of finding a reducible expression inside of an evaluation context. This requires recursively defining the evaluation contexts and the function for plugging a term into a context in addition to the relation itself. Furthermore, Coq's native unification cannot match a concrete constructor against a function application `plug c t`, so it was necessary to define context-plugging as a relation instead. This allowed a search by unification to find a suitable decomposition, with a search of a similar depth as the small-step semantics required.

Finally, a $\mathbb{K}$-style semantics generalizes the configuration by allowing a sequence of items as the current code, where items include statements or expressions, but also "freezers" which are like one layer of context. Extra clauses of the transition relation handle decomposing and recomposing compound expressions to focus on terms requiring evaluation. The effect is similar to expressing the term and context of a reduction semantics syntactically, with the focused term at the head of the list and the layers of the context following. This makes it possible for steps to be taken using only a single constructor of the transition relation, but also to use simple unification to find applicable rules. This may require a few more execution steps than other styles, but the proof tactic is much simpler and the proof that a step is in the relation is a single constructor.

The execution steps to explicitly focus on a subterm or fold back up the result can be amortized over many evaluation steps within that subterm, unlike a small-step or reduction semantics where several layers of context or rules of the step relation must be used in each step to reach the current evaluation point from the root.

In summary, the reduction semantics required a function to be rewritten as a relation, while the small-step and $\mathbb{K}$-style semantics are relatively trivially automated. The $\mathbb{K}$-style semantics will take more steps, but each step requires only one rule rather than a potentially deeper search. Overall, each style seems to be reasonably easy to automate, and there is no compelling reason to try to change the style of a semantics when translating into Coq.

Listing 4.2: IMP small-step semantics in Coq

```
match goal with
```

```
  |[|- context [(?n >? 0)%Z]] => destruct (Z.gtb_spec n 0)
  end;
  run.

(* Now three styles of semantics *)

(* First small step *)
Module small_step.

Inductive exp_step : relation (Exp * Map string Z) :=
  (* Ordered so trying earlier constructors is a good idea *)
  | eval_var : forall v z rest store,
      store ~= v |-> z :* rest ->
      exp_step (EVar v, store) (ECon z, store)
  | eval_gt : forall z1 z2 store,
      exp_step (EGt (ECon z1) (ECon z2), store)
               (BCon (Z.gtb z1 z2), store)
  | eval_plus : forall z1 z2 store,
      exp_step (EPlus (ECon z1) (ECon z2), store)
               (ECon (Z.add z1 z2), store)
  | eval_gt_r : forall z e1 e2 store1 store2,
      exp_step (e1, store1) (e2, store2) ->
      exp_step (EGt (ECon z) e1, store1)
               (EGt (ECon z) e2, store2)
  | eval_gt_l : forall e1 e2 r store1 store2,
      exp_step (e1, store1) (e2, store2) ->
      exp_step (EGt e1 r, store1) (EGt e2 r, store2)
  | eval_plus_r : forall z e1 e2 store1 store2,
      exp_step (e1, store1) (e2, store2) ->
      exp_step (EPlus (ECon z) e1, store1)
               (EPlus (ECon z) e2, store2)
  | eval_plus_l : forall e1 e2 r store1 store2,
      exp_step (e1, store1) (e2, store2) ->
      exp_step (EPlus e1 r, store1)
               (EPlus e2 r, store2)
  .
```

```
Inductive step : relation (Stmt * Map string Z) :=
  | exec_assign : forall v z z0 rest store,
      store ~= v |-> z0 :* rest ->
      step (SAssign v (ECon z), store) (Skip, (v |-> z :* rest))
  | exec_seq : forall s2 store,
      step (Seq Skip s2, store) (s2, store)
  | exec_if : forall b t e store,
      step (SIf (BCon b) t e, store)
           (if b then t else e, store)
  | exec_while : forall cond body store,
      step (SWhile cond body, store)
           (SIf cond (Seq body (SWhile cond body)) Skip, store)
  | exec_assign_1 : forall e1 e2 store1 store2 v,
      exp_step (e1, store1) (e2, store2) ->
      step (SAssign v e1, store1) (SAssign v e2, store2)
  | exec_seq_1 : forall s1 s2 store1 store2 r,
```

Listing 4.3: IMP reduction semantics in Coq

```
  | exec_if_1 : forall c1 store1 c2 store2 t e,
      exp_step (c1, store1) (c2, store2) ->
      step (SIf c1 t e, store1) (SIf c2 t e, store2)
 .

Create HintDb step_db discriminated.
Hint Extern 1 (_ ~= _) => equate_maps : step_db.
Hint Constructors step exp_step : step_db.

Ltac step_solver ::= solve[eauto 20 with step_db].

Lemma example : sound step (loop_spec Seq).
Proof. example_proof. Qed.

End small_step.

(* Now evaluation context *)
```

```
Module evaluation_contexts.

Inductive context : Set -> Set :=
   Top : context Stmt
 | EGt_left : Exp -> context Exp -> context Exp
 | EGt_right : Z -> context Exp -> context Exp
 | EPlus_left : Exp -> context Exp -> context Exp
 | EPlus_right : Z -> context Exp -> context Exp
 | SAssign_2 : string -> context Stmt -> context Exp
 | Seq_left : Stmt -> context Stmt -> context Stmt
 | SIf_1 : Stmt -> Stmt -> context Stmt -> context Exp
 .

Fixpoint plug {s : Set} (c : context s) : s -> Stmt :=
  match c with
  | Top => fun t => t
  | EGt_left r c => fun l => plug c (EGt l r)
  | EGt_right l c => fun r => plug c (EGt (ECon l) r)
  | EPlus_left r c => fun l => plug c (EPlus l r)
  | EPlus_right l c => fun r => plug c (EPlus (ECon l) r)
  | SAssign_2 v c => fun t => plug c (SAssign v t)
  | Seq_left r c => fun l => plug c (Seq l r)
  | SIf_1 t e c => fun v => plug c (SIf v t e)
 end.

Inductive step : relation (Stmt * Map string Z) :=
  | eval_var : forall c v z rest store,
      store ~= v |-> z :* rest ->
      step (plug c (EVar v), store) (plug c (ECon z), store)
  | eval_gt : forall c z1 z2 store,
      step (plug c (EGt (ECon z1) (ECon z2)), store)
           (plug c (BCon (Z.gtb z1 z2)), store)
  | eval_plus : forall c z1 z2 store,
      step (plug c (EPlus (ECon z1) (ECon z2)), store)
           (plug c (ECon (Z.add z1 z2)), store)
  | exec_assign : forall c v z z0 rest store,
      store ~= v |-> z0 :* rest ->
      step (plug c (SAssign v (ECon z)), store)
           (plug c Skip, (v |-> z :* rest))
  | exec_seq : forall c s2 store,
      step (plug c (Seq Skip s2), store) (plug c s2, store)
  | exec_if : forall c b t e store,
      step (plug c (SIf (BCon b) t e), store)
           (plug c (if b then t else e), store)
  | exec_while : forall c cond body store,
      step (plug c (SWhile cond body), store)
           (plug c (SIf cond (Seq body (SWhile cond body)) Skip), store)
  .

Ltac split_ctx ctx t k :=
  match t with
  (* stop at redexes *)
  | (Seq Skip ?rest) => k ctx t
  | (SWhile _ _) => k ctx t
  | (SIf (BCon _) _ _) => k ctx t
```

Listing 4.4: IMP 𝕂-style semantics in Coq

```
  | (EPlus (ECon _) (ECon _)) => k ctx t
  (* else enter *)
  | (Seq ?l ?r) => let ctx' :=
      constr:(Seq_left r ctx) in split_ctx ctx' l k
  | (SIf ?c ?t ?e) => let ctx' :=
      constr:(SIf_1 t e ctx) in split_ctx ctx' c k
  | (EGt (ECon ?z) ?r) => let ctx' :=
      constr:(EGt_right z ctx) in split_ctx ctx' r k
  | (EGt ?l ?r) => let ctx' :=
      constr:(EGt_left r ctx) in split_ctx ctx' l k
  | (SAssign ?v ?e) => let ctx' :=
      constr:(SAssign_2 v ctx) in split_ctx ctx' e k
  | (EPlus (ECon ?z) ?r) => let ctx' :=
      constr:(EPlus_right z ctx) in split_ctx ctx' r k
  | (EPlus ?l ?r) => let ctx' :=
      constr:(EPlus_left  r ctx) in split_ctx  ctx' l k
  end.
Ltac split_term  t := let ctx := constr:Top in
   split_ctx  ctx t ltac:(fun  c r =>
   change t with (plug c r)).
Ltac step_solver ::=
  match goal with
  |[|- step (?t, ?store) ?P] => split_term t
  end;econstructor(equate_maps).

Lemma example : sound step (loop_spec Seq).
Proof. example_proof. Qed.

End evaluation_contexts.

(* Finally  k style  *)

Module k_style.

Require Import List.

Inductive kitem : Set :=
  | KExp : Exp -> kitem
  | KStmt : Stmt -> kitem
  | KFreezeZ : (Z -> kitem) -> kitem
  | KFreezeB : (bool -> kitem) -> kitem
  .

Inductive step : relation  (list  kitem * Map string Z) :=
  | eval_var : forall c v z rest  store,
      store  ~= v |-> z :* rest  ->
      step (KExp (EVar v) ::  c, store)
           (KExp (ECon z) ::  c, store)
  | eval_gt : forall c z1 z2 store,
      step (KExp (EGt (ECon z1) (ECon z2)) :: c, store)
           (KExp (BCon (Z.gtb z1 z2)) ::  c, store)
```

```
| eval_plus : forall c z1 z2 store,
    step (KExp (EPlus (ECon z1) (ECon z2)) :: c, store)
         (KExp (ECon (Z.add z1 z2)) :: c, store)
| exec_assign : forall c v z z0 rest store,
    store ~= v |-> z0 :* rest ->
    step (KStmt (SAssign v (ECon z)) :: c, store)
         (KStmt Skip :: c, (v |-> z :* rest))
| exec_skip : forall c store,
    step (KStmt Skip :: c, store) (c, store)
| exec_seq : forall c s1 s2 store,
```

4.2 Specifications

The last remaining element of mechanizing the reasoning described in Chapter 3 is defining the predicates to be used in specifications, and the abbreviations for concisely making claims about functions. For making claims about functions Section 3.1.6 introduced a function *call* which took a function definition, a set of values for actual arguments, and preconditions and postconditions on the heap, and expanded in into a full reachability claim over configurations by asserting that the call would evaluate to an acceptable value with the given heap effect, while leaving the stack, local variable environment, function environment, and any frame condition on the remainder of the heap all remain unchanged. For the details of the **HIMP** configuration we can define a corresponding abbreviation:

```
Definition heap_fun (R : Spec kcfg) (deps : list Defn) (d:Defn) :
  forall (args : list KResult) (init_heap : MapPattern k k)
    (ret : Z -> MapPattern k k), Prop :=
  match d with FunDef name formals body =>
    fun args init_heap ret =>
      forall krest store stack heap funs mark otherfuns,
      funs ~= fundefs deps (name s|-> KDefn d) :* otherfuns ->
      (mark > 0)%Z ->
      forall frame,
      heap |= (init_heap :* litP frame) ->
      R (KCfg (kra (ECall name (map KResultToExp args)) krest)
              store  stack  heap funs  mark)
         (value_heap (fun r => existsP v, constraint (r = KInt v)
                                              :* ret v)%pattern
              krest  store  stack  frame funs  mark)
    end.
```

This is named heap_fun because it specifies a function whose only result aside from the return value is an effect on the heap. Note that the first argument is a specification, or a set of claims. This can be instantiated with reaches to make a statement that a claim about a function is actually true, or with the name of a program specification being assembled as part of including a claim about a function in that specification.

Data structures similarly follow the plan described in Section 3.1.5. A predicate asserting that a linked list or some other data structure exists at a given address in the heap takes a mathematical list or tree describing the value to be expanded, and is defined in a way that expands into more specific claims about addresses and memory entries when part of the list is known, and conversely can show the represented list must be empty or nonempty if the pointer is null or not.

```
Fixpoint rep_seg (val : list Z) (tailp p : Z) :=
  match val with
    | nil => constraint (p = tailp)
    | x :: xs => constraint (p <> 0) :* existsP p',
        p h|-> list_node x p' :* rep_seg xs tailp p'
  end%pattern.
Notation rep_list l := (rep_seg l 0).
```

The predicate rep_list corresponds to the list_seg predicate of Section 3.1.5, which matches a heap fragment containing only length l list nodes, with p pointing to the first, and the next pointer of the last being *tailp*. When the abstract list *val* is nonempty, the predicate simplifies to an assertion that a list node exists at address p, an existential quantification over the value p' of the next pointer of that node, and an assertion that the rest of the list segment is represented at address p'.

To see if the preconditions of another claim apply or the target of the current reachability claim has been reached, it may be necessary to recognize whether a list is represented at a given address in the heap even if the current description has some explicit nodes. Folding back up the predicate to make this identification may require some specific proof automation which is discussed further in Section 4.3.

An example which uses both of these abbreviations is a function for prepending a value to a list. The one claim `add_claim` in the specification `add_spec` is stated using `heap_fun` (Coq allows the types of constructors such as `add_claim` to be written using definitions like `heap_fun`, as long as the final result type of the constructor after unfolding all those definitions is an instance of the type being defined, here `add_spec`). The list predicates are used to say that in the initial heap the argument x is the address where some list l is represented, and in the postcondition to say that the returned value is the address of a list beginning with the value v and then continuing with its tail being exactly the original list.

```
Definition add_fun := FunDef "add" ["v";"x"]
 {{Decl "y";"y"<-EAlloc
  ;"y"<<-build_node "v" "x"
  ;SReturn "y"}}.

Inductive add_spec : Spec kcfg :=
  add_claim : forall v H x l,  heap_fun add_spec nil
  add_fun [Int  v;Int  x]
    (asP H (rep_list  l  x))
    (fun r => rep_seg (v::nil)  x r :*  litP  H).

Lemma add_proof : sound kstep add_spec.
Proof. list_solver.   Qed.
```

4.3 Proof Automation

In the previous sections of this chapter we have seen how language semantics and claims about programs can be defined in Coq, and how coinduction lemmas are stated and proved sound. Now we discuss how proof automation is provided.

Any fully formal verification that a specification holds must address all details of its possible executions. Handling the vast majority of this detail automatically is necessary for any verification approach to be usable or cost-effective outside the most critical software, let alone to aspire to widespread adoption.

In keeping with this need, apparent even in verifying our example programs, we have developed proof tactics proving a comfortable level of automation for coinductive program verification. A proof *tactic* is a procedure for selecting and applying some lower-level proof-manipulating commands to achieve some higher-level goal. Our proof automation is implemented in the Ltac [Del] tactic scripting language of Coq, and by using built-in tactics that efficiently provide less flexible sorts of automation.

In this section we describe the proof tactics we developed. Our goal in this section is to share design principles we found useful, and to partially address any concerns about the difficulty of automating coinductive verification by describing our automation effort.

The examples described in the next chapter (and corresponding Coq listings in Appendix B) demonstrate the effectiveness of the resulting system in more detail. In short, for each group of examples sharing some specification predicates connecting program states to an abstract domain, a basic tactic was extended with support for those predicates, and then most example were proved by a single invocation of that command. In most remaining examples the proof was finished by invoking that command, manually applying one or two lemmas about the abstract domain, and then reinvoking proof automation.

Our most basic design goal is for tactics to fail gracefully. When a goal cannot be completely solved, some progress should be made, but without committing to any choices that might preclude a successful proof, and with goals left in a convenient and comprehensible form for the user. In particular, our main tactics only leave goals in the form of reachability claims.

When a procedure cannot completely solve the current goal, some progress should be made, but without committing to any choices that might preclude a successful proof and with goals left in a convenient form, so the user is left with a state from with manual progress may be possible. In particular, our top-level tactics leave goals only in the form of reachability claims.

In contrast to the clear-cut language independence of the basic coinduction principles, proof automation must deal with the individual language semantics as well as predicates introduced for writing specifications in specific domains, so we have no theorems about reusability. Nevertheless, our proof tactics are naturally divide into several levels of organization, which are observed to support various degrees of re-use.

4.3.1 Main Heuristic

A fresh proof starts with a few fixed commands to introduce hypotheses, reduce showing that a specification holds to showing that it is stable by applying the coinduction theorem, and attempting to break the specification down into independent reachability claims. Then the proof is continued with our main heuristic for handling reachability claims.

Our main tactic implements a simple overall heuristic, based on the heuristic used in the MatchC [RȘ12a] system. The idea is to try deal with a reachability goal first in the ways likely to leave fewer remaining execution steps.

The most preferred option is to support the current reachability claim by showing that the current state actually meets the target predicate, leaving no remaining execution steps to cover.

The next best option is to apply a claim from the current specification by transitivity, assuming claims abstract over a significant chunk of execution like a function call or loop.

Taking a single execution step is attempted only if none of the earlier options were successful.

The very last option is attempting a case split, as this increases the number of subgoals to be proved, and does not even advance execution. The last case arises mostly from condition constructs like if-statements, where execution can show that condition will reduce to a boolean constant but have only a symbolic expression like $b > 0$ for the value of that boolean. A case split is needed to reduce to a concrete true or false before execution can proceed, and appropriate hypotheses have to be introduced to record the conditions holding in each case.

In each of these cases, the responsible subroutine can also determine that its case might apply but could not be completed automatically, and make the attempt to advance the goal fail without considering the later options. This happens for example if the current code in the configuration matches that of a claim from the specification, but some preconditions of that claim could not be proved to hold.

The `generic_step` tactical attempts to take one action according to this heuristic, and `generic_run` repeatedly uses `generic_step` to advance the proof as much as possible, recursing down both goals of a case split, until each subgoal has either been proved, intentionally left for the users, or cannot be automatically advanced.

This top-level heuristic is implemented as a tactic parameterized by tactics for the individual subtasks, and is reused between different languages by instantiating it with different implementations of those subtactics.

Listing 4.5: generic proof heuristic

```
Ltac hyp_check := idtac.
Ltac generic_step  trans_tac  step_solver  split_stuck  :=
    (hyp_check
    ||trans_tac
    ||(eapply  dstep;[solve[step_solver]|])
    ||split_stuck
    ||fail
    );instantiate;cbv   beta.
Ltac generic_run  trans_tac  step_solver  done_solver  split_stuck  :=
  repeat  (generic_step  trans_tac  step_solver  split_stuck)
  ;try  solve[eapply  ddone;done_solver].
```

The `hyp_check` step is a debugging feature. By default it does nothing, but `hyp_check` can be overridden to interrupt proof automation on a desired condition. One use is while developing automation for new predicate, if other supporting tactics are supposed to keep hypotheses involving the predicate in a certain normal form, then `hyp_check` can be used to stop automation as soon as an unreduced hypothesis appears. Another use is to help understand a failed proof attempt. When the automated tactic leaves a residual goal with an insufficient, inconsistent, or otherwise undesirable hypothesis, but may have taken many steps since it was introduced, `hyp_check` can be overridden to stop as soon as that hypothesis appears.

4.3.2 Execution Steps

The tactics working most directly with a language definition are those for finding execution steps from a current state. In particular, a tactic must be effective when the next state is a completely unknown existential variable.

We avoid mentioning all the clauses of a transition relation by using primitive tactics that approach goals in an inductively defined predicate by trying every applicable case. The other contribution to reusability between languages is that basic domains like maps or fixed-width integers are defined once and shared between language definitions, so proof tactics for those domains can be shared as well.

In particular, when a language definition is given as a non-recursively defined relation datatype, as in $\mathbb{K}$ or abstract machine styles, the simple tactic

```
econstructor(solve[side_condition_solver])
```

This tactic tries any constructors of the relation type that unify with the current goal, until one is found whose side conditions are all solved by the given `side_condition_solver` tactic.

A single language semantics is defined in several different styles in Section 4.1.1, with a corresponding step tactic presented for each.

4.3.3 Specification Predicates

At an intermediate level of detail, proof automation must also handle predicates used to write specification. These define higher level properties relevant to specifications in terms of the primitive domains and predicates of the language's transition relation. For example, the assertion that a null-terminated linked list starts at a given heap address and contains a specified list of values is defined recursively in terms of basic assertions that a given heap address contains a given value.

Our general strategy is to expand higher-level assertions as much as possible. Keeping predicates decomposed allows the simple step tactics described in the previous subsection to work, without requiring extensions for every new specification predicate.

For example, if a linked list containing some list l of values is represented at p, then after learning that l is nonempty the hypothesis should be expanded to claims that p is nonnull, a list node containing the first value of l exists at address p in the heap, and that the tail of l is represented by a linked list at the "next" address of the list node.

For recursively defined predicates such as the list or tree representations in Section 3.1.5 or Section 4.2, the predicates will unfold by simplification once part of the structure of the list or tree is known, so unfolding can be implemented mostly by delegating to basic tactics for decomposing existentials and (separating) conjunctions of claims.

It is also possible that new lower-level information can imply more information about the abstract values being represented, such as when a case split resulting from an if test adds the assumption that the pointer representing an abstract list l is either null or non-null.

Besides propagating information like this, it is also sometimes necessary to decide whether a specification predicate holds (and perhaps under what abstract arguments it holds), given some more concrete hypotheses. In particular, showing that preconditions hold to use a claim by transitivity, or showing that the target predicate hold to finish off the current subgoal may both require showing that some specification predicate applies in the current state.

4.3.4 Effort

Overall, the work necessary to implement useful proof automation is almost independent of the number of rules of the language semantics, or the number of claims of the specifications to be solved, but depends only on the number and complexity of the domain predicates used in defining the language language and the specification to be proved.

Chapter 5

Evaluation

In this chapter we verify a range of example programs using the Coq implementation of coinductive verification presented in the previous chapter. Our goal here is a basic proof-of-concept, by showing that instantiating our core theorems with operational semantics of several difference languages does indeed give a sufficient foundation for verifying a range of example programs. We verify programs manipulating heap data structures, with the Schorr-Waite graph marking algorithm being the most complicated.

5.1 Verified Programs

Overall statistics are presented in Table 5.1. Times were measured with the 64-bit Linux version of Coq 8.4pl2, on a laptop with a i7-3720QM processor and 1600Mhz memory. We describe the languages, predicates, and proof automation afterwards.

Size attempts to count content, ignoring comments, blank lines, and punctuation. We also ignore some fixed lines such as imports. The specification size includes any functions or predicates defined for a particular example in addition to the set of claims. The proof size similarly includes any lemmas, tactics, and proof hint declarations in addition to the main body of the proof. The one-line proofs are those solved completely by our automation.

Proving time is measured by compiling the Coq file containing an example, and certificate checking time is measured by rechecking the resulting proof certificate file, while skipping checks of any included modules. Checking a proof certificate does not require proof search or executing proof tactic scripts.

The simple examples are those which do not use the heap. The minimum and maximum proofs required a hint to use standard lemmas about the min and max functions. The sum program adds numbers from 1 to n, and required an auxiliary lemma proving an arithmetical formula. We describe one example in each of the remaining categories in more detail.

5.1.1 Lists

The next group of examples deals with linked lists. We implement a list node as a record value containing the integer value of that node, and the address of the rest of the list (which is 0 to represent empty lists).

Here is a statement which returns a copy of an input list, leaving the input list unchanged.

```
if (x == 0)
  { return 0 }
else
  { y := alloc
  ; *y := {val = x->val, next = 0}
  ; iterx := x->next
  ; itery := y
  ; while (not (iterx == 0))
    { node := alloc
    ; *node := {val = iterx->val; next = 0}
    ; *itery := {val = itery->val; next = node}
    ; iterx := iterx->next
    ; itery := itery->next
    }
  ; return y
  }
```

Using abbreviated notation, the desired specification is

```
(<k> copy_code </k>
<store>... "p" |-> p ...</store>
<heap> asP hlist (rep_list A p), hrest </heap>
...)
 =>
(exists p2,
 <k> return p2 </k>
 <store> _ </store>
 <heap> rep_list A p2, hlist, hrest </heap>
 ...)
```

This says that if the code is executed in a state where the store binds program variable `p` to the address of a linked list in the heap holding the sequence of values in the abstract list `A`, it can only return with the address of a freshly-allocated copy of the list, and the rest of the heap unchanged.

For this program, repeating `rep_list A p` in the postcondition would not be a strong enough specification, because it would allow moving intermediate nodes in the input list, but the copy operation should be usable as a subroutine even in code that holds pointers to intermediate nodes in the input list. The `asP` pattern binds the variable `hlist` to the exact subheap that satisfies the pattern `rep_list A p`, which allows specifying that the input list is unchanged.

We have not implemented this abbreviated notation. Our complete specification also includes a claim about the loop, whose precondition asserts that a non-empty initial segment of the list has been copied, and `itery` holds the address of the last list node in the copied segment.

The complete proof script for this example is

```
Proof. list_solver.
rewrite app_ass in * |-.
list_run. Qed.
```

We see associativity of list append was not automatically applied. When the automated solver paused, it left a goal we abbreviate as

```
(<k> return v </k>
 <heap> rep_list ((A++[x])++y::B) v , H </heap>
 ...)
=>
(exists p2,
 <k> return p2 </k>
 <heap> rep_list (A++x::y::B) p2 , H </heap>
 ...)
```

The abstract list is described with an unexpectedly associated append, but the current state does satisfy the target. It was not necessary to automate reasoning about associativity or to manually complete this branch of the proof. After reassociating the expression membership in the target set can be shown automatically, and this is the first thing attempted when the `list_run` tactic resumes automation.

The three examples with three-line proof required this assistance with associativity. The delete example removes all copies of a value from a linked list. The specification defined the desired operation as a Coq function, and we did not automate reasoning about it.

The Bedrock [Chla] example `SinglyLinkedList.v` verifies a module defining length, reverse, and append functions on linked lists. It takes approximately 150 seconds to prove, and 50s to recheck. Our results in Table 5.1 may be an unfair comparison as Bedrock's language can only store a scalar value in a heap location and our code as presented in this section keeps an entire structure at an address. For a more even comparison we modified the program and specification to expect the value and next pointers at consecutive addresses. To ensure no undesired functional was used, we made a modified copy of our main language with records and built-in memory allocation removed, under the `bytewise` directory in our development. However, rather than costing performance, the ability to make a single-field update without copying the other field actually improved performance. The modified examples can be verified in respectively 6.5, 13, and 15 seconds. Rechecking these proof certificates take 1.7, 2.4, and 2.6 seconds.

5.1.2 Trees

The next data structure we deal with is a binary tree. Tree nodes are implemented as records containing a value and the addresses of left and right children.

Two examples flatten a binary tree into a linked list by a preorder traversal, deallocating the input tree.

One implementation is a recursive helper function which flattens a tree onto the front of a given list

```
(<k> flatten_code </k>
 <store>... "t" |-> t , "l" |-> l ...</store>
 <heap>rep_tree T t, rep_list A l, H</heap>
 ...)
```

```
=>
(exists v,
 <k> return v </k>
 <heap>rep_list (tree2list T ++ A) v, H</heap>
 ...)
```

Flattening is defined as a Coq function `tree2list`. This implementation was easily verified.

A more interesting implementation avoids recursion by using an explicit stack, implemented as a linked list of pointers to subtrees. To specify this list, the list predicate is generalized. Instead of just taking a list of integers, the generalized predicate takes a list of any type, plus a representation predicate for that type, and asserts that the values in the abstract list are represented by the corresponding numbers in the concrete list, along with some possibly-empty subheaps. Instantiating this with the representation predicate for trees gives a predicate `rep_gen_list rep_tree trees` for a stack of trees. Here is the full Coq specification of the loop:

```
loop_claim : ∀ ts t l s tn lv ln sn,
  heap_loop tree_to_list_spec tree_to_list_def 0
  ("t" s↦ KInt t ⋆ "l" s↦ KInt l ⋆ "s" s↦ KInt s
    ⋆ "tn" s↦ tn ⋆ "ln" s↦ ln ⋆ "sn" s↦ sn)
  (rep_prop_list (fun t p ⇒ constraint (p ≠ 0) ⋆ rep_tree t p) ts s
    ⋆ rep_list lv l)
  (rep_list (trees2List (rev ts) ++ lv))
```

The proof of the height function required some assistance with the max function. The (exhaustive) find function required a manual case split on the results of looking for the target value in the left subtree. The recursive flatten function required more substantial assistance because we did not provide automation for the generalized list predicate.

5.1.3 Schorr-Waite

Our experiments so far demonstrate that our coinductive verification approach applies across languages in different paradigms, and can handle usual heap programs with a high degree of automation. Here we show that we can also handle the famous Schorr-Waite graph marking algorithm [SW67], which is a well-known verification challenge, “The Schorr-Waite algorithm is the first mountain that any formalism for pointer aliasing should climb” [Bor00]. To give the reader a feel for what it takes to mechanically verify such an algorithm, previous proofs in [MN05] and [HM] required manually produced proof scripts of about 470 and, respectively, over 1400 lines and they both used conventional Hoare logic. In comparison, our proof is 516 lines. We admit that size is not always a fair comparison of proofs; we only mention these numbers to imply that our novel coinductive verification approach need not sacrifice proof size to eliminate the need for a Hoare logic of a language; proofs based on operational semantics may even be less tedious.

The implementation of Schorr-Waite that we verified is based on [Gri79]. We define graphs by extending the definition of trees by additionally allowing a child of a node in the abstract tree to be a reference back to some existing node, in addition to an explicit subtree or a null pointer for a leaf. To specify that graph nodes are at their original addresses after marking, we include an address along with the mark flag in the abstract data structure in the pattern

$$\begin{aligned}
&\mathrm{grph}(\mathrm{leaf}, m, p') && = \langle p' = 0\rangle \\
&\mathrm{grph}(\mathrm{backref}(p), m, p') && = \langle p' = p\rangle \\
&\mathrm{grph}(\mathrm{node}(p,l,r),m,p') && = \langle p'{=}p\rangle \\
&\qquad \star\exists p_l,p_r.p \mapsto [m,p_l,p_r] \star \mathrm{grph}(l,m,p_l) \star \mathrm{grph}(r,m,p_r)
\end{aligned}$$

The overall specification is $\mathit{call}(\mathit{Mark}, [p], \mathrm{grph}(G, 0, p), \lambda r.\mathrm{grph}(G, 3, p))$.

To describe the intermediate states in the algorithm, including the clever pointer-reversal trick used to encode a stack, we define another data structure for the context, in zipper style. A position into a tree is described by it’s immediate context, which is either the the topmost context, or the point immediately left or right of a sibling tree, in a parent context. These are represented by nodes with intermediate values of the mark field, with one field pointing to the sibling subtree and the other pointing to the representation of the rest of the context.

$$\begin{aligned}
&\mathrm{stack}(\mathrm{Top}, p) && = \langle p = 0\rangle \\
&\mathrm{stack}(\mathrm{LeftOf}(r, k), p) && = \exists p_r, p_k.p \mapsto [1, p_r, p_k], \mathrm{grph}(r, 0, p_r), \mathrm{stack}(k, p_k) \\
&\mathrm{stack}(\mathrm{RightOf}(l, k), p) && = \exists p_l, p_k.p \mapsto [2, p_k, p_l], \mathrm{stack}(k, p_k), \mathrm{grph}(l, 3, p_l)
\end{aligned}$$

This is the second data structure needed to specify the main loop. When it is entered, there are only two live local variables, one pointing to the next address to visit and the other keeping context. The next node can either be the root of an unmarked subtree, with the context as stack, or the first node in the implicit stack when ascending after marking a tree, with the context pointing to the node that was just finished. For simplicity, we write a separate claim for each case.

$$\begin{array}{c} \mathit{stmt}(\mathit{Loop}, (\mathtt{p} \mapsto p, \mathtt{q} \mapsto q), (\mathrm{grph}(G, 0, p), \mathrm{stack}(S, q)), \lambda r.\mathrm{grph}(\mathit{plug}(G, S), 3)) \\ \mathit{stmt}(\mathit{Loop}, (\mathtt{p} \mapsto p, \mathtt{q} \mapsto q), (\mathrm{stack}(S, p), \mathrm{grph}(G, 3, q)), \lambda r.\mathrm{grph}(\mathit{plug}(G, S), 3)) \end{array}$$

The application of all the semantic steps was handled entirely automatically, the manual proof effort being entirely concerned with reasoning about the predicates above, for which no proof automation was developed.

	Size (lines)			Time (s)	
Example	Code	Spec.	Proof	Prove	Check
Simple					
undefined	2	3	1	3.5	2.0
average3	2	5	1	4.0	1.2
min	3	4	2	3.6	1.0
max	3	4	2	3.6	1.0
multiply	9	7	1	12.4	2.4
sum(rec)	6	8	6	5.5	1.3
sum(iter)	5	12	6	8.1	1.5
Lists					
head	1	6	1	2.6	1.2
tail	1	6	1	2.5	1.1
add	4	16	1	4.5	1.4
swap	9	13	1	19.6	4.2
dealloc	4	7	1	8.5	1.8
length(rec)	4	12	1	7.8	1.9
length(iter)	5	17	1	9.4	2.0
sum(rec)	6	7	1	9.3	2.1
sum(iter)	5	11	1	12.6	2.3
reverse	7	11	3	22.0	3.7
append	7	12	3	22.3	4.3
copy	13	23	3	101.5	15.5
delete	15	60	35	83.3	10.8
Trees					
height	8	7	3	26.7	4.2
size	5	7	1	11.4	2.5
find	5	12	2	20.9	3.2
mirror	6	16	1	24.2	5.6
dealloc	14	33	1	33.8	6.8
flatten(rec)	10	18	1	43.7	8.5
flatten(iter)	28	35	28	270.7	49.6
Schorr-Waite					
tree	14	91	116	105.1	14.4
graph	14	91	203	232.9	34.4

Table 5.1: Proof statistics

Chapter 6

Translating Semantics

To verify code with our Coq implementation, a semantics of the target programming language is needed as a Coq definition. For easy access to more languages, and as a step towards making program verification more accessible, we wish to develop automatic translators of language definitions from user-friendly semantics-engineering tools into Coq definitions. Any system with a sufficiently precise semantics and a collection of interesting language definitions would be an attractive source. Working in the FSL group at UIUC, our own $\mathbb{K}$ framework was a natural first choice, but familiarity was hardly the only motivation. Language definitions available in $\mathbb{K}$ include C [HER], Java [BR], JavaScript [Bod+; PȘR], Python [Gut13], and PHP [FM].

The language of the $\mathbb{K}$ framework has many convenient features for specifying programming languages. Fortunately for the task of translation, many of these features are translated away by the $\mathbb{K}$ compiler. The current $\mathbb{K}$ implementation is based on an intermediate language called $\mathbb{KORE}$, which is designed to be as simple as possible while still being to able to faithfully express anything expressible in the full $\mathbb{K}$ language (accepting concision and readability as costs).

The terms of a $\mathbb{KORE}$ definition consists of labeled tree, drawing from a defined set of label names, with values of some additional primitive types allowed at the leaves. The transitions are described by a set of rewrite rules, which use pattern matching and boolean side conditions. The only complexity in pattern matching is that some labels may be declared associative (optionally with unit), and matching should be done modulo associativity. Auxiliary functions may also be defined by giving a set of evaluation rules, which are expected to be total.

The Coq translation defines the labels and terms of a definition, relying on a library of Coq implementations of the primitive types and functions. Pattern matching is translated to make as much use as possible of Coq's unification support, so concrete labels in the left hand side of a rule are left as part of a pattern. Associative matching is accommodated by replacing any portion of a pattern that may require associativity with a fresh variable and addition another side condition capturing the matching.

The relations defining functions are separately translated, but because $\mathbb{K}$ functions are not checked for totality and Coq functions must pass a syntactic totality check it is currently left to the user to define a Coq function equivalent to the $\mathbb{K}$ functions.

Chapter 7

Reachability Logic

A predecessor and direct inspiration for coinductive verification is *Reachability Logic*. This is a proof system for proving partial reachability claims over any language semantics given as a collection of rewrite rules, using either the open-path semantics of claims from Definition 3.1, or the all-path semantics from Definition 3.8.

This gives a portion of the language independence of coinductive verification, imposing only the additional restriction that the semantics be given in the syntax of rewrite rules rather than allowing any approach to defining a transition relation (provided the result captures divergence in a useful way).

Before and besides developing coinductive verification I was also involved in the reachability logic project, and in particular developed Coq proofs of the soundness for the proof systems.

Having such a soundness proof makes it conceivable to build a certifying implementation of the proof system on top of a proof assistant, but an explicit syntactic set of rules is more cumbersome to work with.

Proving soundness of the reachability logic proof system required an *deep embedding*, because some global properties of the set of rules are used to prevent using coinductive assumptions before it can be soundly allowed. In a deep embedding the syntax of well-formed proof trees is explicitly defined, and the soundness theorem states that the reachability claim at the root of any legal tree is correct. This theorem can only be used by assembling tree fragments, instead of allowing free use of other Coq-supported reasoning. In contrast, a *shallow embedding* would have a lemma with the same shape as each proof rule of a system, more like coinductive verification where access to coinductive claims is controlled solely by whether the goal has the form $c \in \text{step}(\text{derived}(X))$ or $c \in \text{derived}(X)$.

Step∃ :

$$\dfrac{\left.\begin{array}{l}\models \varphi \to \exists \mathit{FreeVars}(\varphi_l).\varphi_l \\ \models \exists c\ (\varphi[c/\square] \land \varphi_l[c/\square]) \land \varphi_r \to \varphi'\end{array}\right\} \quad \text{for some} \ \ \varphi_l \Rightarrow^{\exists} \varphi_r \ \in \ \mathcal{S}}{\mathcal{S}, \mathcal{A} \ \vdash_{\mathcal{C}} \varphi \Rightarrow^{\exists} \varphi'}$$

Step∀ :

$$\dfrac{\begin{array}{l}\models \varphi \to \bigvee_{\varphi_l \Rightarrow^{\exists} \varphi_r \ \in \ \mathcal{S}} \exists \mathit{FreeVars}(\varphi_l) \ \varphi_l \\ \models \exists c\ (\varphi[c/\square] \land \varphi_l[c/\square]) \land \varphi_r \to \varphi' \qquad \text{for each} \ \ \varphi_l \Rightarrow^{\exists} \varphi_r \ \in \ \mathcal{S}\end{array}}{\mathcal{S}, \mathcal{A} \ \vdash_{\mathcal{C}} \varphi \Rightarrow^{\forall} \varphi'}$$

Transitivity :

$$\dfrac{\mathcal{S}, \mathcal{A} \ \vdash_{\mathcal{C}} \varphi_1 \Rightarrow^{Q} \varphi_2 \qquad \mathcal{S}, \mathcal{A} \cup \mathcal{C} \ \vdash \varphi_2 \Rightarrow^{Q} \varphi_3}{\mathcal{S}, \mathcal{A} \ \vdash_{\mathcal{C}} \varphi_1 \Rightarrow^{Q} \varphi_3}$$

Circularity :

$$\dfrac{\mathcal{S}, \mathcal{A} \ \vdash_{\mathcal{C} \cup \{\varphi \Rightarrow^{Q} \varphi'\}} \varphi \Rightarrow^{Q} \varphi'}{\mathcal{S}, \mathcal{A} \ \vdash_{\mathcal{C}} \varphi \Rightarrow^{Q} \varphi'}$$

Axiom :

$$\dfrac{\varphi \Rightarrow^{Q} \varphi' \ \ \in \ \mathcal{A}}{\mathcal{S}, \mathcal{A} \ \vdash_{\mathcal{C}} \varphi \Rightarrow^{Q} \varphi'}$$

Reflexivity :

$$\dfrac{\cdot}{\mathcal{S}, \mathcal{A} \ \vdash \varphi \Rightarrow^{Q} \varphi}$$

Consequence :

$$\dfrac{\models \varphi_1 \to \varphi_1' \qquad \mathcal{S}, \mathcal{A} \ \vdash_{\mathcal{C}} \varphi_1' \Rightarrow^{Q} \varphi_2' \qquad \models \varphi_2' \to \varphi_2}{\mathcal{S}, \mathcal{A} \ \vdash_{\mathcal{C}} \varphi_1 \Rightarrow^{Q} \varphi_2}$$

Case Analysis :

$$\dfrac{\mathcal{S}, \mathcal{A} \ \vdash_{\mathcal{C}} \varphi_1 \Rightarrow^{Q} \varphi \qquad \mathcal{S}, \mathcal{A} \ \vdash_{\mathcal{C}} \varphi_2 \Rightarrow^{Q} \varphi}{\mathcal{S}, \mathcal{A} \ \vdash_{\mathcal{C}} \varphi_1 \lor \varphi_2 \Rightarrow^{Q} \varphi}$$

Abstraction :

$$\dfrac{\mathcal{S}, \mathcal{A} \ \vdash_{\mathcal{C}} \varphi \Rightarrow^{Q} \varphi' \qquad X \cap \mathit{FreeVars}(\varphi') = \emptyset}{\mathcal{S}, \mathcal{A} \ \vdash_{\mathcal{C}} \exists X \ \varphi \Rightarrow^{Q} \varphi'}$$

We make the standard assumption that the free variables of $\varphi_l \Rightarrow^{\exists} \varphi_r$ in the Step proof rule are fresh (e.g., disjoint from those of $\varphi \Rightarrow^{\forall} \varphi'$.

Figure 7.1: Reachability Logic proof system

7.1 Proof System

The earliest versions were presented in [RŞ12b; RŞ12a]. The form shown here is one of the most developed versions which presents a unified set of rules combining one-path and all-path semantics, as presented in [Şte+] and the accompanying technical report. The only earlier work presenting an incomparably strong variant is [Roş+13], where the notion of divergence in partial reachability is extended to consider *operational divergence*, allowing semantics to use *conditional rewrite rules*.

Of these rules we can see that Consequence, Case Analysis and

Abstraction are subsumed in coinductive verification by simple logical reasoning on the preconditions of the set of claims being proved. Step and Reflexivity correspond to using the two cases in the definition of our step function, Axiom to taking advantage of the properties of $-^*$ to appeal to one of the claims of our specification, and Transitivity alone corresponds to actually using an enhancement.

Notice the Circularity rule allows introducing new claims at any point throughout a proof, and adding the rule to the subscript $\mathcal{C}$ on the sequent rather than into the set $\mathcal{A}$ of accepted claims prevents appealing to the new claim before at least one Step is taken.

Showing that this system is subsumed by coinductive verification relies on another presentation of the proof system which is also presented in the source work, called *Set Circularity*, which in fact corresponds more closely to the $\mathbb{K}$ implementation of reachability logic proving.

7.2 Set Circularity

In the set circularity form of a proof, a proof consists of a forest of trees, the Circularity rule may not be used, and the initial set $\mathcal{C}$ of circularities of each proof tree can be any subset of the set of conclusions of the trees in the forest, rather than necessarily empty. When such a forest can be assembled, all the claims at the root of trees in the proof forest are simultaneously valid.

This was proved to be a valid derived rule, in the sense that any proof in this form can be translated to yield a legal tree as in Fig. 7.1 for any of the conclusions (in particular, starting with an empty set $\mathcal{C}$ of *circularities*), and any proof tree in that form can be translated into a set circularity proof (eliminating uses of the Circularity rule), whose set of conclusions includes that of the original tree.

A fully detailed proof that set circularity proofs are valid is given in [Roș+12]. Putting aside the sometimes-delicate handling of the Reflexivity rule, the general intuition is that uses of Circularity can be removed from a proof forest (possibly a single-tree forest) by making the subtree rooted at a use of Circularity a new tree of the forest, adding the relevant claim to the set $\mathcal{C}$ assumed at the root of the parent tree, and removing that use of Circularity in the parent, now that the hypothesis is no weaker than the conclusion, because the claim will already be in $\mathcal{A}$ or $\mathcal{C}$ at that point.

Conversely, a set circularity proof forest can be expanded into a single tree for any of the conclusions by starting to copy the tree for the desired conclusion, except beginning with an empty set $\mathcal{C}$ instead of the set from the forest, remembering at each point which of those desired claims is still not available in the current set $\mathcal{A} \cup \mathcal{C}$ in the copy, inserting an appeal to Circularity whenever the current claim is one of the desired claims, and recursively beginning to copy another tree of the forest whenever a use of Axiom appeals to one of the still missing claims.

7.3 Equivalence

Given a set circularity proof, the set of claims at the roots of trees can be gathered into a specification for a coinductive proof. Noting that these claims are only in $\mathcal{C}$ at the root of each tree, and Axiom can use claims only from $\mathcal{A}$ ensures that a step has been taken according to the language semantics before any claims of the specification have been used. Thus we can translate each step of the proof into a step in a coinductive proof of that set of claims.

Chapter 8

Future Work

This chapter describes opportunities for future work, along with preliminary results in the suggested directions.

8.1 Alternative Specification Semantics

The majority of our mechanized proof use the one-path partial reachability predicate defined in Definition 3.2, but proof by generalized coinduction can be used with any style of specification claims whose correctness can be defined as a greatest fixpoint.

We have not verified any significant code under such alternate predicates, but our Coq development does include proofs that a few additional notations of specification can be proved coinductively, and that transitivity and other principles can also be used with each of these.

These includes the all-path partial reachability of Definition 3.8, whose characterization as the greatest fixpoint of $\mathrm{step}^{\forall}_{R}$ is proven, along with a proof that it is compatible with exactly the same transitivity (Definition 3.5) and pre-proved (Definition 3.7) rules which are valid for one-path reachability.

A further generalization restricts partial reachability by also parameterizing over an additional "invariant" relation restricting what steps are permitted to occur before encountering the target set.

The all-path "until" definition says until *c I P* is satisfied if no possible execution from c can become stuck or take a step not included in relation T before encountering a state satisfying P.

```
CoInductive until c (I : relation cfg) (P : cfg → Prop) : Prop :=
  | rdone : P c → until c I P
  | rstep : (∃ c', cstep c c') → (∀ c', cstep c c' → I c c')
        → (∀ c', cstep c c' → until c' I P) → until c I P.
```

```
Definition sound (Rules : Spec) : Prop :=
  ∀ c I P, Rules c I P → until c I P.

Inductive derived (X : Spec)
      c (I : relation cfg) (P : cfg → Prop) : Prop :=
  | drule : X c I P → derived X c I P
  | ddone : P c → derived X c I P
  | dstep : (∃ c', cstep c c') → (∀ c', cstep c c' → I c c')
      → (∀ c', cstep c c' → derived X c' I P) → derived X c I P
  | dstrong : ∀ (I' : cfg → cfg → Prop), (∀ a b, I' a b → I a b)
      → derived X c I' P → derived X c I P
  | dtrans' : ∀ Q, derived X c I Q → (∀ c', Q c' → derived X c' I P)
      → derived X c I P.
Lemma proved_sound (Rules : Spec) :
  (∀ c I P, Rules c I P → step (derived Rules) c I P) → sound Rules.
```

These examples were partially motivated by the appearance of relations in Rely-Guarantee [Jon81] reasoning, as well as other potential specifications that require controlling the entire course of execution, such as ensuring the a program for a microcontroller never exceeds the maximum stack size.

8.2 Combining Semantics

The reduced effort to obtain a proof system for an operational semantics may be even more useful when considering modifications to the semantics being used rather than formalizing completely new languages.

One simple modification to a semantics is to ensure termination by brute force, by merely adding a counter as a new component of the state and decrementing it on each transition. With this amendment any specification is forced to describe also an upper bound on execution time, and proves both total correctness and that the time bound is respected.

Another sort of modification would be combining the semantics of a language with a formal model of a system the program interfaces with, such as a file system, a network, or perhaps some physical hardware.

Making small modification should make it easier to re-use specification predicate and proof tactics from other verification efforts based on the same language.

8.3 Larger Examples

The programs verified so far are aimed as a proof of concept, to demonstrate that the strong sense of language independence which is apparent in the mathematics did not come at an unacceptable cost in expressiveness or automatability. The Schorr-Waite graph marking algorithm shows some degree of difficulty, but none of the examples are very large.

A particularly interesting task in connection with the last proposal might be verifying a filesystem, as much of the real difficulty comes from accounting for the behavior and failure modes of the disk.

8.4 Sound Translations

In our current implementation there is are no proofs that the result of automatically translating a $\mathbb{K}$ definition into a Coq transition is faithful. Manually verifying the translation currently requires understanding the Coq definition of the semantics, and checking it against the intended behavior of the $\mathbb{K}$ semantics.

The necessary effort and the necessary experience with Coq could be greatly reduced given a Coq formalization of the semantics of $\mathbb{K}$ itself. Then a Coq definition of a language could be obtained by translating only the syntax tree of the definition, and applying the definition of $\mathbb{K}$. This would require only checking that the syntax was correctly transliterated.

Even if a translation in the current direct style remains necessary for efficiency or readability, it could be proven equivalent to a more easily trusted translation.

8.5 Additional Sources of Semantics

Several tools besides $\mathbb{K}$ define and encourage the development of language semantics in machine-readable forms. The PLT-Redex [FFF09] framework has been used to define several large languages including Python [Pol+13] and JavaScript [GSK10]. The Lem [Mul+14] and Ott [Sew+07] systems can already generate Coq definitions, but we have not yet attempted to use the resulting definitions.

8.6 Additional Types and Predicates

Our definitions and specifications should be able to use any types or predicates definable in Coq. So far our examples have mostly used definitions from the Coq standard library, but for more complicated domains we hope to make use of other formalization work in Coq. Two libraries of particular interest are the IEEE-compatible floating point numbers of the Flocq [BM11] library, and the (concurrent) separation algebras of [DHA09].

8.7 Higher-Order Specifications

For specifying code such has higher-order functions or just-in-time code generators it would be natural for the target predicate of a claim to make some further reachability claims about the execution behavior of function values or compiled code being returned.

These postconditions could be written by directly referring to validity of the claims, but at best that would require giving a separate and earlier proof about the code being returned. Ideally any reachability claims made in the postcondition could be established coinductively along with any other claims in the specification.

A partial answer in this direction is to make a higher-order specification a function parameterized over a predicate to use in place of validity inside of state predicates that wish to make claims about execution.

To accommodate functions returning higher-order functions, it seems preferable to split the parameter into two predicates, and insure the the specification is monotone in one and anti-monotone in the other. This accommodates re-usable definitions of higher-order specifications by switching the arguments passed to negative positions.

Unfortunately, we have not yet found any entirely satisfactory coinduction principles for establishing the soundness of a specification which takes a reachability predicate as a parameter.

8.8 Typed Functional Languages

Our experiments with Lambda served to demonstrate that we could deal with closures and other functional language features so long as the specifications remained simple, but in the absence of a satisfactory approach for stating and proving claims about higher-order functions we are ill-equipped to investigate functional programming languages more deeply.

If we make progress on the previous proposal, we will of course wish to specify and verify higher-order functions in their natural habitat. A more open question is whether or how reasoning in terms of types and reachability claims might be combined.

A very simple case is the use of type-based abstraction boundaries to protect implementations of abstract data types. It suffices to verify code in the defining module operates correctly on the type, and the type system can then be relied upon to extend that guarantee to know that any value of that type appearing anywhere in well-typed code does in fact respect the internal invariants the implementing module defined.

8.9 Comparison with Axiomatic Semantics

In Section 5.1 we demonstrated a range of programs that can be successfully coinductively verified with reasonable effort. We believe this sufficiently demonstrated that coinductive verification is not inherently intractable compared to other approaches. However, it is still be interesting to establish a more precise comparison between the proofs of various approaches.

Overall, there are three levels of integration at which axiomatic semantics rules might be combined with coinductive verification. We will show that coinductive verification can subsume axiomatic semantics in each of these ways, and at the same time demonstrate that proofs are already simple enough that scarcely anything is gained by explicitly introducing Hoare-style rules.

In [RȘ] it was shown that a Hoare logic for a simple imperative language was equivalent to a methodological use of the Reachability Logic proof system. The first step was to show how a Hoare triple can be expressed as a reachability claim, by adding components to a pattern to handle framing and allow references to program variables.

Using the rule proved, a coinductive proof can appeal to any reachability claim that is somehow independently known to be valid. In particular, if we have a sound axiomatic semantics for a language and a translation from it's conclusions into reachability claims, we can use the axiomatic semantics as once source of true claims.

Another form of integration is proving the soundness of rules of an axiomatic semantics using coinductive verification. To do this we would write a specification where each claims conclusion is the conclusion of some rule of the axiomatic semantics, and those claims have hypotheses asserting the validity of reachability claims corresponding to the hypotheses of that proof rule.

The most intimate integration would be trying to make axiomatic semantics rules available in the middle of a coinductive proof, just as transitivity can be used, with the hypotheses left to be established according to the current specification rather than switching to independent subproofs claiming validity.

This would require proving that the desired proof rules are *valid derived rules* for validity with the instance of step_R for the chosen language, like Lemma 8 proved trans is a valid derived rule for any instance step_R.

Chapter 9

Related Work

Now we discuss related work. We believe this work is unique in that it explicitly establishes the soundness of a specified program as the conclusion of a coinductive argument. The most relevant related work thus divides mostly into two categories, namely program verification systems that address goals similar to ours while using different foundations, and work presenting powerful coinduction principles of the sort we required, developed for other application. We also mention a few other ways which coinduction has been used in connection with program specification or verification, in applications quite unlike our proof.

9.1 Current Verification Systems

The prominent tools Why [FP] and Boogie [Lei08; Bar+06], are explicitly used and promoted as "Intermediate Verification Languages", for developing verification tools by translation into the supported languages and specifications of this system. For example, Frama-C and Krakatoa respectively support C and Java by translation through Why, and Spec# and Havoc respectively support C# and C by translation through Boogie.

These systems are however less foundational, lacking proofs of soundness for translations of full languages.

Both Why and Boogie are based around verification condition generators for a particular programming language, passing the verification conditions on to SMT solvers or proof assistants.

Another tool is Bedrock [Chla; Chlb], which is a foundational verifier implemented in Coq for a machine-level language, with any successful verification resulting in a Coq proof that the specification holds, relying only on the definitions of the programming language, the specification language, and the soundness of Coq. The basic proof system is organized around a verification condition generator, proved sound against a fixed operational semantics for the primitive language.

However, Bedrock allows significant language extensions, accepting new language constructs defined as a package of a translation from the new construct into previously defined features, a verification condition generation function for the new constructs, and a proof of soundness of the verification condition generator with respect to the desugaring. Taking advantage of this flexibility, Bedrock has been used to implement a cooperative threading system, an HTML server, and an in-memory database. A design goal is nearly-complete proof automation, which is provided with powerful heuristics and support for extending the automation with lemmas for handling new definitions. This is generally quite effective, though at a cost in proof compilation time, and results in low textual overhead for full functional verification.

In contrast to these systems, we do not need to develop and prove soundness of an intermediate such as a translator or verification condition generator to offer a proof system and machine-checked proof of soundness for a new operational semantics.

9.2 Verification from Operational Semantics

The verification in ACL2 of code in simplified [Moo99] and JVM [LM04] bytecode was carried out by reasoning directly on an operational semantics defined as an interpreter-like function for advancing a configuration by one step. Without the benefit of coinduction, their approach required that each specification claim include a precise calculation of the number of interpreter steps the starting state needs to reach the target state.

Another approach also explored in ACL2 translated program specifications into a proposed invariant, such that showing the resulting property to be an invariant will prove that the original specification holds. To carry out this approach it is necessary to put a limit on the transition system being explored so that execution cannot proceed past the state where the postcondition is supposed to be called.

With an invariant giving predicates over individual states there is only a limited ability to connect arguments on entry to a function with an expected postcondition at the exit. For non-recursive functions it is sufficient to fix or quantify over the arguments before fixing the invariant and initial state, but this breaks down when recursive functions are considered. The suggested approach to handle recursion is to write a program-specific and semantics-specific function that can dig around in the call stack and local variables values to reconstruct the original arguments to a function call from the residual state remaining when it returns.

9.3 Reachability Logic

Reachability logic [Roș+13] is a closely related approach to program verification which explicitly aims for the same notion of language independence as we pursue in this work, and has a circularity proof rule with a coinductive flavor. This system is presented in detail in Chapter 7

In particular I contributed to the development of the logic, participating in then discussions and experiments with different forms of proof rules that lead the the current versions of these systems. I was also primarily responsible for the mechanized proofs of soundness of reachability logic. Coinductive program verification was inspired by trying isolate the smallest essential use of coinduction from those proofs of soundness of reachability logic.

The practicality of this approach was demonstrated in [Ște+16], adding an implementation of the proof system to the $\mathbb{K}$ framework, and using previously and independently developed semantics of real-world languages such as C, Java, and JavaScript.

Coinductive verification is more general in a few ways. One is that reachability logic requires an operational semantics to be presented as a set of rewrite rules instead of any sort of definition of a relation. Another is that using a generalized coinduction theorem as the basis for reasoning allows effectively adding new proof rules independently, by proving them compatible with the step function as in Definition 3.6, rather than needing to update a central soundness proof. Finally, coinduction is also applicable for other notions of specification as long as they can be defined as a greatest fixpoint, including all-path partial reachability and the other examples presented in Section 8.1. The reachability logic proof system has a variant for proving all-path partial reachability, but that required an alternate proof of soundness of the entire system (which uses an incompatible induction argument to handle the Circularity rule). Coinductive verification does require checking that extra reasoning principles like `trans`(Definition 3.5) are compatible with a new notion of specification

9.4 Separation Logic

Separation logic is a prominent body of work which is not in competition with our approach. We are free to use any predicates we like in writing specifications, and separation logic provides a convenient language for describing data structures. The key idea is to use a logic that accounts for locality or independence. The separating disjunction $P \star Q$ asserts not just that P and Q are both true, but that they are true on disjoint portions of memory. This means that when P describes a mutable data structure, we can be sure that Q will still hold on its portion of the heap after modifying the data structure described by P. If we merely asserted $P \wedge Q$ we would need additional assertions about sharing, otherwise we might have for example two linked lists with a common tail, with the contents seen in one list potentially changed by removing an entry from the other list.

Separation logic need not be presented as part of a program logic, and is studied in its own right in terms of *separation algebras* [COY07], especially for concurrency. Important uses for concurrency include describing lock-guarded invariants in *concurrent separation logic*, or ensuring writes happen only to uniquely-referenced objects with *share accounting* [DHA09].

Our verification examples already specify the heap portion of configurations with a pattern language using ideas from separation logic. Rather than separately defining a syntax and satisfaction relation, we follow Bedrock (and our own inclination to work with semantics instead of syntax) and define the $\star$ operator directly as a higher-order predicate asserting that a heap splits into disjoint subheaps satisfying the two argument predicates.

9.5 Other Coinduction Schemata

The generalized coinduction theorem we use is a specialization of the principle introduced under the name λ-coiteration by Bartels [Bar04]. This was stated and proved in categorical language, and described for use as a principle proving bisimilarity. The categorical dual of this theorem was described as a recursion scheme for functional programming in [UVP01].

An attractive development we were unable to take advantage of was the principle of "parameterized coinduction" presented in [Hur+13]. This allows a coinductive proof of $x \leq \nu f$ to enlarge x as needed as the proof proceeds, rather than requiring a sufficiently large set of claims to be made upfront to meet a stability condition like $x \leq f(x)$. This is done by showing how to define for any monotone f a corresponding function G_f satisfying all the following properties:

$$\begin{aligned} \forall x. x \leq G_f(x) \\ \forall x. f(G_f(x)) \leq G_f(x) \\ G_f(\bot) \leq \nu f \forall xy. x \leq f(G_f(y \wedge x)) \implies x \leq G_f(y) \end{aligned}$$

Unfortunately this did not seem to be compatible with the sort of extension allowed in the previous results, which we rely on to allow proof rules such as transitivity.

The recent paper "Coinduction all the Way Up" [Pou16] presents a quite general result including both sorts of extensions. As we propose for future work, it looks quite promising as a replacement for the coinduction theorem we have been using. This work shows that associated to any monotone function f on a lattice there is an associated "companion function" t, definable as a simple least upper bound, which has many properties for coinductively proving $x \leq \nu f$.

It's properties include

$$\begin{aligned} t(\bot) \leq \nu f \\ x \leq t(x) \\ x \leq f(t(y \wedge x)) \implies x \leq t(y) \\ x \leq g(t(y)) \wedge g \leq t \implies x \leq t(y) \end{aligned}$$

The condition $g \leq t$ is implied by several known conditions for using g as an enhancement in coinduction under a function f, including the condition $f(g^*(x)) \leq g(f(x))$ of the Theorem 2 this work uses. Using this result it would not be necessary to fix the entire set of claims nor the "derived rules" to be used before beginning a proof.

Several generalizations over the simplest form of coinduction which are nonetheless too weak for nicely verifying programs seem to be folklore. For example, Isabelle/HOL's standard library offers this lemma $mono(f) \wedge A \subseteq f(\mu x.\, f(x) \cup A \cup \nu f) \implies A \subseteq \nu(f)$.

9.6 Other Uses of Coinduction in Program Verification

The earliest work we are aware of which uses greatest fixpoints in connection with program verification is by Edmund M. Clarke in [Cla77]. This paper gives a definition of weakest precondition and strongest postcondition transformers for Hoare-style partial correctness triples that defines some cases as greatest fixpoints.

In [LG09] Xavier Leroy and Hervé Grall give a big-step semantics for possibly nonterminating programs, simply by taking the greatest fixpoint of the clauses of a big-step semantics rather than the least fixpoint. This means nonterminating configurations are also included in the relation, with membership shown by infinite trees of rules.

Dafny [Lei10] is a programming language with an integrated specification language, using Boogie and Z3 for verification. It is intended to allow specifying and verifying full functional correctness, within a relatively familiar object-oriented imperative language. It has been extended with support for coinductive datatypes and predicates, suitable for cases such as lazy definitions, indefinite streams, and specifying interactive systems. [Rus13].

A number of dependently typed functional programming languages or proof assistants allow coinductively defining data types, in particular Coq. Other related languages such as Adga and Idris have similar support. We used Coq's native notion of coinduction in proving the soundness of our generalized coinduction principle.

The verification of programs written in JVM bytecodes in ACL [LM04] was carried out by reasoning directly about execution using a function calculating the next state of the abstract machine. However, without the use of coinduction it was necessary to reason extensively about termination, and in fact each specification had to include a formula for the exact number of execution steps the code in question would take.

Chapter 10

Conclusion

We have presented a language independent program verification framework, which gives a sound proof system when instantiated with an operational semantics, without requiring any proof of soundness or defining any intermediates such as verification condition generators. The resulting system allows proofs straightforwardly following control flow, and is amenable to proof automation.

We have proved the core theorems in a proof assistant, and working atop those definitions to verify a program results in a machine checkable proof certificate in the in the logic of the proof assistant that the desired claims are true in the provided operational semantics.

The key observation is that a characterization of partial correctness claims allows proofs by coinduction, and that although using the simplest coinduction principle is utterly infeasible, using a generalized coinduction theorem gives an improved coinduction principle that can actually show reasonably small sets of claim correct.

As desired from a sound program verification system, believing that code is correct requires believing only that the formalized claim adequately captures the desired behavior, that the operational semantics correctly describes the intended language, and that the logic and proof checker of Coq are correct, and no hardware errors invalided the execution of the proof checker. In particular, it is not necessary to examine either the given code nor the text of a purported proof to conclude that the specification holds.

In contrast to our language independence, state-of-the art program verifiers such as Why [FP], Bedrock [Chlb] or Boogie [Lei08] directly target a single language, using a proof system and soundness proof developed by experts. Additional language may be handled by translations targeting these systems, but showing this is sound would require giving an additional proof of the faithfulness of these translation.

With our approach, a language designer or other interested party need only develop and validate an operational semantics to have a sound proof system. It is neither necessary to design a proof system, nor prove it's soundness. We hope this will contribute to making program verification more widely accessible.

References

[Bar+06] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. "Boogie: A Modular Reusable Verifier for Object-oriented Programs". In: *FMCO'05*. Amsterdam, The Netherlands: Springer, 2006, pp. 364–387. ISBN: 3-540-36749-7, 978-3-540-36749-9. DOI: 10.1007/11804192_17.

[Bar04] Falk Bartels. "On Generalised Coinduction and Probabilistic Specification Formats: Distributive Laws in Coalg. Modelling". PhD thesis. Vrije Univ. Amsterdam, 2004.

[BIM95] Bard Bloom, Sorin Istrail, and Albert R. Meyer. "Bisimulation Can't Be Traced". In: *J. ACM* 42.1 (Jan. 1995), pp. 232–268. ISSN: 0004-5411. DOI: 10.1145/200836.200876.

[BM11] Sylvie Boldo and Guillaume Melquiond. "Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq". In: *Proceedings of the 2011 IEEE 20th Symposium on Computer Arithmetic*. ARITH '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 243–252. ISBN: 978-0-7695-4318-5. DOI: 10.1109/ARITH.2011.40.

[Bod+] Martin Bodin, Arthur Chargueraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. "A Trusted Mechanised JavaScript Specification". In: *POPL'14*. San Diego, California, USA: ACM, pp. 87–100. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535876.

[Bor00] Richard Bornat. "Proving Pointer Programs in Hoare Logic". English. In: *Mathematics of Program Construction*. Vol. 1837. LNCS. Springer, 2000, pp. 102–126. ISBN: 978-3-540-67727-7. DOI: 10.1007/10722010_8.

[BR] Denis Bogdănaș and Grigore Roșu. “K-Java: A Complete Semantics of Java”. In: *POPL’15*. Mumbai, India: ACM, pp. 445–456. ISBN: 978-1-4503-3300-9. DOI: `10.1145/2676726.2676982`.

[CB07] Fabricio Chalub and Christiano Braga. “Maude MSOS Tool”. In: *Electronic Notes in Theoretical Computer Science* 176.4 (2007), pp. 133–146. ISSN: 1571-0661. DOI: `10.1016/j.entcs.2007.06.012`.

[Chla] Adam Chlipala. “Mostly-automated Verification of Low-level Programs in Computational Separation Logic”. In: *PLDI’11*. San Jose, California, USA: ACM, pp. 234–245. ISBN: 978-1-4503-0663-8. DOI: `10.1145/1993498.1993526`.

[Chlb] Adam Chlipala. “The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier”. In: *ICFP’13*. Boston, Massachusetts, USA: ACM, pp. 391–402. ISBN: 978-1-4503-2326-0. DOI: `10.1145/2500365.2500592`.

[Cla+03] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. “The Maude 2.0 System”. In: *Rewriting Techniques and Applications (RTA 2003)*. Ed. by Robert Nieuwenhuis. Lecture Notes in Computer Science 2706. Springer-Verlag, June 2003, pp. 76–87.

[Cla77] Edmund Melson Clarke. “Program Invariants As Fixed Points”. In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. SFCS ’77. Washington, DC, USA: IEEE Computer Society, 1977, pp. 18–29. DOI: `10.1109/SFCS.1977.25`.

[COY07] Cristiano Calcagno, Peter W. O’Hearn, and Hongseok Yang. “Local Action and Abstract Separation Logic”. In: *Proceedings of the 22Nd Annual IEEE Symposium on Logic in Computer Science*. LICS ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 366–378. ISBN: 0-7695-2908-9. DOI: `10.1109/LICS.2007.30`. URL: `http://dx.doi.org/10.1109/LICS.2007.30`.

[Del] David Delahaye. "A Tactic Language for the System Coq". In: *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*. LPAR'00. Reunion Island, France: Springer, pp. 85–95. ISBN: 3-540-41285-9.

[DHA09] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. "A Fresh Look at Separation Algebras and Share Accounting". In: *Programming Languages and Systems: 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings.* Ed. by Zhenjiang Hu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 161–177. ISBN: 978-3-642-10672-9. DOI: `10.1007/978-3-642-10672-9_13`.

[FFF09] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. 1st. The MIT Press, 2009. ISBN: 0262062755, 9780262062756.

[FM] Daniele Filaretti and Sergio Maffeis. "An Executable Formal Semantics of PHP". English. In: *ECOOP'14*. Vol. 8586. LNCS. Springer, pp. 567–592. ISBN: 978-3-662-44201-2. DOI: `10.1007/978-3-662-44202-9_23`.

[FP] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3 — Where Programs Meet Provers". English. In: *ESOP/ETAPS'13*. Vol. 7792. LNCS. Springer, pp. 125–128. ISBN: 978-3-642-37035-9. DOI: `10.1007/978-3-642-37036-6_8`.

[GM92] Joseph A. Goguen and José Meseguer. "Order-sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations". In: *Theor. Comput. Sci.* 105.2 (Nov. 1992), pp. 217–273. ISSN: 0304-3975. DOI: `10.1016/0304-3975(92)90302-V`.

[Gog+88] Joseph A. Goguen, C. Kirchner, H. Kirchner, A. Megrelis, J. Meseguer, and T. Winkler. "An Introduction To OBJ -3". In: 308 (1988), pp. 258–263.

[Gri79] David Gries. "The Schorr-Waite graph marking algorithm". English. In: *Acta Informatica* 11.3 (1979), pp. 223–232. ISSN: 0001-5903. DOI: `10.1007/BF00289068`.

[GSK10] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. "The Essence of Javascript". In: *Proceedings of the 24th European Conference on Object-oriented Programming*. ECOOP'10. Maribor, Slovenia: Springer-Verlag, 2010, pp. 126–150. ISBN: 3-642-14106-4, 978-3-642-14106-5. URL: `http://dl.acm.org/citation.cfm?id=1883978.1883988`.

[Gut13] Dwight Guth. "A Formal Semantics of Python 3.3". MA thesis. University of Illinois at Urbana-Champaign (UIUC), Aug. 2013. URL: `http://hdl.handle.net/2142/45275`.

[HER] Chris Hathhorn, Chucky Ellison, and Grigore Roșu. "Defining the Undefinedness of C". In: *PLDI'15*. To Appear. ACM.

[HM] T. Hubert and C. Marche. "A case study of C source code verification: the Schorr-Waite algorithm". In: *SEFM'05*. IEEE, pp. 190–199. ISBN: 0-7695-2435-4. DOI: `10.1109/SEFM.2005.1`.

[Hoa69] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: `10.1145/363235.363259`.

[Hue97] Gérard Huet. "The Zipper". In: *Journal of Functional Programming* 7 (05 Sept. 1997), pp. 549–554. ISSN: 1469-7653. URL: `http://journals.cambridge.org/article_S0956796897002864`.

[Hur+13] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. "The Power of Parameterization in Coinductive Proof". In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '13. Rome, Italy: ACM, 2013, pp. 193–206. ISBN: 978-1-4503-1832-7. DOI: `10.1145/2429069.2429093`.

[Jon81] C. B. Jones. "Development Methods for Computer Programs including a Notion of Interference". Printed as: Programming Research Group, Technical Monograph 25. PhD thesis. Oxford University, June 1981.

[Kle+] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. "Run Your Research: On the Effectiveness of Lightweight Mechanization". In: *POPL'12*. Philadelphia, PA, USA: ACM, pp. 285–296. ISBN: 978-1-4503-1083-3. DOI: 10.1145/2103656.2103691.

[Lei08] K. Rustan M. Leino. *This is Boogie 2*. Tech. rep. Microsoft Research, June 2008. URL: http://research.microsoft.com/jump/147643.

[Lei10] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. LPAR'10. Dakar, Senegal: Springer-Verlag, 2010, pp. 348–370. ISBN: 3-642-17510-4, 978-3-642-17510-7. URL: http://dl.acm.org/citation.cfm?id=1939141.1939161.

[LG09] Xavier Leroy and Hervé Grall. "Coinductive big-step operational semantics". In: *Information and Computation* 207.2 (2009), pp. 284–304. ISSN: 0890-5401. DOI: 10.1016/j.ic.2007.12.004.

[LM04] Hanbing Liu and J. Strother Moore. "Java Program Verification via a JVM Deep Embedding in ACL2". In: *Theorem Proving in Higher Order Logics: 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004. Proceedings.* Ed. by Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 184–200. ISBN: 978-3-540-30142-4. DOI: 10.1007/978-3-540-30142-4_14.

[MN05] Farhad Mehta and Tobias Nipkow. "Proving Pointer Programs in Higher-Order Logic". In: *Inf. and Computation* 199 (2005), pp. 200–227. DOI: 10.1016/j.ic.2004.10.007. URL: http://in.tum.de/~nipkow/pubs/ic05.html.

[Moo99] J. Strother Moore. “Proving Theorems About Java-Like Byte Code”. In: *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the Occasion of His Retirement from His Professorship at the University of Kiel)*. London, UK, UK: Springer-Verlag, 1999, pp. 139–162. ISBN: 3-540-66624-9. URL: `http://dl.acm.org/citation.cfm?id=646005.673749`.

[Mos04] Peter D Mosses. “Modular structural operational semantics”. In: *The Journal of Logic and Algebraic Programming* 60 (2004), pp. 195–228. ISSN: 1567-8326. DOI: `10.1016/j.jlap.2004.03.008`.

[Mul+14] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. “Lem: Reusable Engineering of Real-world Semantics”. In: *SIGPLAN Not.* 49.9 (Aug. 2014), pp. 175–188. ISSN: 0362-1340. DOI: `10.1145/2692915.2628143`.

[ON02] David von Oheimb and Tobias Nipkow. “Hoare Logic for NanoJava: Auxiliary Variables, Side Effects and Virtual Methods revisited”. In: *Formal Methods – Getting IT Right (FME’02)*. Ed. by Lars-Henrik Eriksson and Peter Alexander Lindsay. Vol. 2391. LNCS. `http://isabelle.in.tum.de/Bali/papers/NanoJava.html`. Springer, 2002, pp. 89–105.

[Owe] Scott Owens. “A Sound Semantics for OCaml$_{light}$”. In: ESOP/ETAPS’08. Budapest, Hungary: Springer, pp. 1–15. ISBN: 3-540-78738-0, 978-3-540-78738-9. DOI: `10.1007/978-3-540-78739-6_1`.

[Plo04] Gordon D. Plotkin. “A structural approach to operational semantics”. In: *Journal of Logic and Algebraic Programming* 60-61 (2004). Original version: University of Aarhus Technical Report DAIMI FN-19, 1981, pp. 17–139. DOI: `10.1016/j.jlap.2004.05.001`.

[Pol+13] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. “Python: The Full Monty”. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA ’13. Indianapolis, Indiana, USA: ACM, 2013, pp. 217–232. ISBN: 978-1-4503-2374-1. DOI: 10.1145/2509136.2509536.

[Pou16] Damien Pous. “Coinduction All the Way Up”. In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS ’16. New York, NY, USA: ACM, 2016, pp. 307–316. ISBN: 978-1-4503-4391-6. DOI: 10.1145/2933575.2934564.

[PȘR] Daejun Park, Andrei Ștefănescu, and Grigore Roșu. “KJS: A Complete Formal Semantics of JavaScript”. In: *PLDI’15*. To Appear. ACM.

[RES] Grigore Roșu, Chucky Ellison, and Wolfram Schulte. “Matching Logic: An Alternative to Hoare/Floyd Logic”. In: *AMAST’10*. Vol. 6486. LNCS. Springer, pp. 142–162. ISBN: 978-3-642-17795-8. DOI: 10.1007/978-3-642-17796-5_9.

[Roș+12] Grigore Roșu, Andrei Ștefănescu, Ștefan Ciobâcă, and Brandon M. Moore. *Reachability Logic*. Tech. rep. University of Illinois, July 2012. URL: http://hdl.handle.net/2142/32952.

[Roș+13] Grigore Roșu, Andrei Ștefănescu, Ștefan Ciobâcă, and Brandon M. Moore. “One-Path Reachability Logic”. In: *Proceedings of the 28th Symposium on Logic in Computer Science (LICS’13)*. IEEE, June 2013, pp. 358–367. DOI: 10.1109/LICS.2013.42.

[RȘ] Grigore Roșu and Andrei Ștefănescu. “From Hoare Logic to Matching Logic Reachability”. In: *FM’12*. Vol. 7436. LNCS. Springer, pp. 387–402. DOI: 10.1007/978-3-642-32759-9_32.

[RȘ10] Grigore Roșu and Traian Florin Șerbănuță. “An Overview of the K Semantic Framework”. In: *Journal of Logic and Algebraic Programming* 79.6 (2010). http://kframework.org, pp. 397–434. DOI: 10.1016/j.jlap.2010.03.012.

[RȘ12a] Grigore Roșu and Andrei Ștefănescu. "Checking Reachability using Matching Logic". In: *Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*. ACM, Oct. 2012, pp. 555–574. DOI: 10.1145/2398857.2384656.

[RȘ12b] Grigore Roșu and Andrei Ștefănescu. "Towards a Unified Theory of Operational and Axiomatic Semantics". In: *Proceedings of the 39th International Colloquium on Automata, Languages and Programming (ICALP'12)*. Vol. 7392. LNCS. Springer, July 2012, pp. 351–363. DOI: 10.1007/978-3-642-31585-5_33.

[Rus13] Michal Moskal Rustan Leino. *Co-induction Simply: Automatic Co-inductive Proofs in a Program Verifier*. Tech. rep. July 2013. URL: https://www.microsoft.com/en-us/research/publication/co-induction-simply-automatic-co-inductive-proofs-in-a-program-verifier/.

[Șer+14] Traian Florin Șerbănuță, Andrei Arusoaie, David Lazar, Chucky Ellison, Dorel Lucanu, and Grigore Roșu. "The K Primer (version 3.3)". In: *Electronic Notes in Theoretical Computer Science* 304 (2014), pp. 57–80. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2014.05.003.

[Sew+07] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. "Ott: Effective Tool Support for the Working Semanticist". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. ICFP '07. Freiburg, Germany: ACM, 2007, pp. 1–12. ISBN: 978-1-59593-815-2. DOI: 10.1145/1291151.1291155.

[ȘRM09] Traian Florin Șerbănuță, Grigore Roșu, and José Meseguer. "A Rewriting Logic Approach to Operational Semantics". In: *Information and Computation* 207 (2 2009), pp. 305–340. DOI: 10.1016/j.ic.2008.03.026.

[Ște+] Andrei Ștefănescu, Ștefan Ciobâcă, Radu Mereuță, Brandon M. Moore, Traian Florin Șerbănuță, and Grigore Roșu. "All-Path Reachability Logic". In: *RTA-TLCA'14*. Vol. 8560. LNCS. Springer, pp. 425–440. DOI: 10.1007/978-3-319-08918-8_29.

[Ște+16] Andrei Ștefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roșu. "Semantics-Based Program Verifiers for All Languages". In: *Proceedings of the 31th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*. ACM, Nov. 2016, pp. 74–91. DOI: 10.1145/2983990.2984027.

[SW67] H. Schorr and W. M. Waite. "An Efficient Machine-independent Procedure for Garbage Collection in Various List Structures". In: *Commun. ACM* 10.8 (Aug. 1967), pp. 501–506. ISSN: 0001-0782. DOI: 10.1145/363534.363554.

[Tar55] Alfred Tarski. "A lattice-theoretical fixpoint theorem and its applications." In: *Pacific Journal of Mathematics* 5.2 (1955), pp. 285–309. DOI: 10.2140/pjm.1955.5.285.

[UVP01] Tarmo Uustalu, Varmo Vene, and Alberto Pardo. "Recursion Schemes from Comonads". In: *Nordic Journal of Computing* 8.3 (Sept. 2001), pp. 366–390. ISSN: 1236-6064.

[Yan+04] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. "Nemos: a framework for axiomatic and executable specifications of memory consistency models". In: *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International.* Apr. 2004, pp. 31–. DOI: 10.1109/IPDPS.2004.1302944.

Appendix A

Example Coq Session

Here include an example session with the command-line Coq interpreter, including it's responses to show all the information the tool can maintain instead of the user needing to track, and gradually building up rudimentary proof automation by recognizing and encapsulating repetitive portions of the proof.

Now we will repeat the example proof in Coq.

First we define the code of the program, naming the loop statement here as well. We also define a state predicate which says the current code is empty and the variable `s` will have a desired value.

```
Coq < Definition loop := While (Not (Eq (PostDec "n") (Con 0)))
Coq <                    (Assign "s" (Add (Var "s") (Var "n"))).

Coq < Definition code := Seq (Assign "s" (Con 0)) loop.

Coq <
Coq < Definition s_result (k:Z) := (fun (cfg' : cfg) =>
Coq <   let (code,env) := cfg' in code = Skip
Coq <                          /\ Env.MapsTo "s" k env).
```

With these definitions we state our specification.

In Coq the most natural way to define a set of claims is extensionally, as a predicate accepting the desired claims. Definitions by cases corresponds most directly to an inductive definition with a constructor for each case. This definition introduces new propositions `spec  c  P`, which only hold in the cases covered by `sum_claim` and `loop_claim`. The `sum_claim` constructor covers cases where the code of the current configuration `c` is `code`, the environment of `c` maps variable `n` to integer n, and the predicate `P` is `s_result`$(n(n-1)/2)$. The `loop_claim` constructor covers cases where the code if `loop`, the environment defines varibables `n` and `s`, and the goal predicate is `s_result`$(s + n(n-1)/2)$. As an inductively defined predicate, these are the only cases which can be used to prove `spec c P` holds.

```
Coq < Inductive spec : Spec imp.cfg :=
Coq <   | sum_claim : forall n env,
Coq <       Env.MapsTo "n" n env ->
Coq <         spec (code, env) (s_result ((n * (n-1))/ 2))
Coq <   | loop_claim : forall n s env,
Coq <       Env.MapsTo "n" n env ->
Coq <       Env.MapsTo "s" s env ->
Coq <         spec (loop, env) (s_result (s + (n * (n-1))/ 2)).
```

To prove the specification holds, we need to show that `reaches` c P is true whenver `spec` c P holds. Using an abbreviation from the `proof` module, this is stated as

```
Coq < Lemma example : sound imp_step spec.
1 subgoal

  ============================
   sound imp_step spec
```

To begin proving this by coinduction we apply the theorem `proved_sound`, the mechanized version of a generalized coinduction theorem.

```
Coq < apply proved_sound.
1 subgoal

  ============================
   forall (x : cfg) (P : cfg -> Prop),
   spec x P -> step imp_step (trans imp_step spec) x P
```

This leaves a proof goal equivalent to $X \subseteq \text{step}(\text{step}^*(X))$. Now we simply need to show this implication, with no further mention of `reaches`, or need to directly consider specific (possibly infinite) complete execution traces.

We begin the proof by splitting into cases. The tactic `destruct` makes case distinctions, and the form `destruct 1` examines the first unnamed argument in the conclusion, after raising the variables of a surrounding `forall` to hypotheses.

```
Coq < destruct 1.
2 subgoals

  n : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  ============================
   step imp_step (trans imp_step spec) (code, env)
     (s_result (n * (n - 1) / 2))
subgoal 2 is:
 step imp_step (trans imp_step spec) (loop, env)
   (s_result (s + n * (n - 1) / 2))
```

The first goal corresponds to `sum_claim`, the second to `loop_claim`. The "bullet" `*` declares we will be proving the cases separately, restricts attention to the first goal, and enforces the division by making Coq report an error unless another `*` appears at exactly the point where this case has been resolved an we need to move on to the next. (The `idtac` tactic explicitly does nothing, and is included here for some tools used to typeset this example).

```
Coq < * idtac.
1 focused subgoal (unfocused: 1)

  n : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  ============================
   step imp_step (trans imp_step spec) (code, env)
     (s_result (n * (n - 1) / 2))
```

We must start by using one of the cases of `step`. We are not already at the goal, so we must take an execution step. This corresponds to the constructor `sstep` of `step`. The commands `eapply sstep` tries to prove the current goal as an application of `sstep`, leaving its hypotheses nas new subgoals. If arguments mentioned in the hypotheses cannot be determined by unification with the current goal, then `eapply` fills them with *existential variables*, which are placeholders that can be refined towards specific formulas as the proof proceeds. In this case we get one existential variable, for the configuration after the execution step.

```
Coq < eapply sstep.
2 focused subgoals (unfocused: 1)

  n : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  ============================
   imp_step (code, env) ?39
subgoal 2 is:
 trans imp_step spec ?39 (s_result (n * (n - 1) / 2))
```

To show that a step is valid we can explicitly apply cases from the definition of the step relation. With an existential variable as the second argument of `imp_step`, this simultaneously refine the shape of the successor state. Recalling the definition of `code`, the rules we need are `seq`$_1$ to step in the first statement of `Seq`, and then `assign` to execute the assignment (the expression is already a constant).

```
Coq < apply step_seq1.
2 focused subgoals (unfocused: 1)

  n : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  ============================
   imp_step (Assign "s" (Con 0), env) (?46, ?45)
subgoal 2 is:
 trans imp_step spec (Seq ?46 loop, ?45)
(s_result (n * (n - 1) / 2))
```

```
Coq < apply step_assign.
1 focused subgoal (unfocused: 1)

  n : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  ============================
   trans imp_step spec (Seq Skip loop, Env.add "s" 0 env)
     (s_result (n * (n - 1) / 2))
```

We see, especially in the second goal, that matching the goal to be proved against the rules being used refined the successor state from a complete unknown into an expression with no more existential variables.

We can also see that explicitly choosing which execution rules to use at each step will be tiresome on all but the tiniest examples, so it is time to use more automation.

The `eauto` tactic takes a collection of lemmas and tries to solve a goal by using them as by `eapply`, using a depth-first search to explore choices between the lemmas whose conclusions unify with the goal. The particular form `eauto using imp_step` uses all the constructors of `imp_step` as possible lemmas. We undo the two explicit rule applications above and use `eauto`.

```
Coq < Undo 2.
2 focused subgoals (unfocused: 1)

  n : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  ============================
   imp_step (code, env) ?39
subgoal 2 is:
 trans imp_step spec ?39 (s_result (n * (n - 1) / 2))

Coq < eauto using imp_step.
1 focused subgoal (unfocused: 1)

  n : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  ============================
   trans imp_step spec (Seq Skip loop, Env.add "s" 0 env)
     (s_result (n * (n - 1) / 2))
```

Chosing to take a step and finding the step can be combined into a single command using one of the compound forms of Coq's tactic language.

```
Coq < eapply dstep;[eauto using imp_step| ].
1 focused subgoal (unfocused: 1)

  n : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  ============================
   trans imp_step spec (loop, Env.add "s" 0 env)
     (s_result (n * (n - 1) / 2))
```

Now that execution from the overall claim has reached the loop we should conclude by using the loop claim, and wait until we are supporting the loop claim to consider the execution of the loop.

The current claim is actually equivalent to an instance of `loop_claim`, but Coq's unification procedure does not automatically determine this (because it would require considering computation after matching s with 0). Instead, we use `loop_claim` by transitivity to slightly delay reasoning about the target predicates. To prove the hypotheses of `loop_claim` about variable bindings in the environment we use the `map_lookup` tactic which was defined alongside the environment type.

```
Coq < eapply dtrans.
2 focused subgoals (unfocused: 1)

  n : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  ============================
   spec (loop, Env.add "s" 0 env) ?76
subgoal 2 is:
 forall k' : cfg,
 ?76 k' -> trans imp_step spec k' (s_result (n * (n - 1) / 2))
```

```
Coq < apply loop_claim;map_lookup.
1 focused subgoal (unfocused: 1)

  n : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  ============================
   forall k' : cfg,
   s_result (0 + n * (n - 1) / 2) k' ->
   trans imp_step spec k' (s_result (n * (n - 1) / 2))
```

After using any claim by transitivity, we are left with a goal where the current configuration is represented only as a fresh variable known to satisfy the target predicate of the instance of the claim that was just used.

In this case that hypothesis and the current target predicate are the same except for equivalent expressions for the expected value of `s`, so we can conclude by choosing the "done" case and showing the current state meets the current target predicate.

Luckily Coq's libraries happened to choose the definition of addition that lets $0 + x$ reduce to x by computation (while $x + 0 = x$ must be proved as a lemma), and Coq's logic allows convertible terms to be used interchangeably, so we use the `assumption` tactic, which attempts to conclude a subproof by finding a hypothesis equivalent with the goal.

```
Coq < intros;apply ddone.
1 focused subgoal (unfocused: 1)

  n : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  k' : cfg
  H0 : s_result (0 + n * (n - 1) / 2) k'
  ============================
   s_result (n * (n - 1) / 2) k'

Coq < assumption.
This subproof is complete, but there are still unfocused goals.
1 subgoal

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  ============================
   step imp_step (trans imp_step spec) (loop, env)
     (s_result (s + n * (n - 1) / 2))
```

Now we proceed to the second claim.

```
Coq < * idtac.
1 focused subgoal (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  ============================
   step imp_step (trans imp_step spec) (loop, env)
     (s_result (s + n * (n - 1) / 2))

Coq < eapply sstep;[eauto using imp_step|].
1 focused subgoal (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  ============================
   trans imp_step spec
     (If (Not (Eq (PostDec "n") (Con 0)))
        (Seq (Assign "s" (Add (Var "s") (Var "n")))
           (While (Not (Eq (PostDec "n") (Con 0)))
             (Assign "s" (Add (Var "s") (Var "n"))))) Skip, env)
     (s_result (s + n * (n - 1) / 2))

Coq < eapply dstep;[eauto using imp_step|].
1 focused subgoal (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  ============================
   trans imp_step spec
     (If (Not (Eq (Con n) (Con 0)))
        (Seq (Assign "s" (Add (Var "s") (Var "n")))
           (While (Not (Eq (PostDec "n") (Con 0)))
              (Assign "s" (Add (Var "s") (Var "n"))))) Skip,
     Env.add "n" (n - 1) env) (s_result (s + n * (n - 1) / 2))
```

We see that `eauto` did not find a second execution step. Evaluating the condition of the if statment will also require rules from `imp_cond_step` for evaluating conditional expressions, and rules from `imp_exp_step` for evaluating numerical expressions. These could be explicitly listed with `eauto using imp_step, imp_cond_step, imp_exp_step`, but with even just three types this is getting unwieldy. It's time to start controlling the automatic proof search.

Besides listing explicit lemmas or types with a `using` clause, it is also possible to group a collection of lemmas and other sorts of "hints" into a named *hint database*, which can be used like `eauto with imp_steps`, and incrementally extended with more hints. A hint database named `imp_steps` was created alongside the definition of the imp semantics. Here is how we insure it contains all rules of the three evaluation relations, and use `eauto` with the hint database to find the step.

```
Coq < Hint Constructors
Coq <   imp_step imp_cond_step imp_exp_step : imp_steps.

Coq < eauto with imp_steps.
1 focused subgoal (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  ============================
   trans imp_step spec
     (If (Not (Eq (Con n) (Con 0)))
        (Seq (Assign "s" (Add (Var "s") (Var "n")))
           (While (Not (Eq (PostDec "n") (Con 0)))
              (Assign "s" (Add (Var "s") (Var "n"))))) Skip,
     Env.add "n" (n - 1) env) (s_result (s + n * (n - 1) / 2))
```

With improved automation, we would like a tactic to take many simple steps automatically. We also saw that when the tactic `eauto using imp_step` did not find a step, it left that subgoal alone. A failure would also have undone `eapply dstep` instead of leaving the overall proof committed to taking a step.

Even worse, the second remaining subgoal is a reachability claim to which `dstep` could be applied again, despite the current configuration being a bare existential variable, so a careless attempt to automatically take many steps might go into an infinite loop.

We require a tactic to completely succeed using the compound expression `solve[`*tac*`]`, which applies *tac*, but fails also if leaves any remaining subgoals. Combining this with `repeat` *tac* which applies *tac* until it fails, we write a tactic for automatically taking as many execution steps as can be taken completely automatically.

```
Coq < repeat(eapply dstep;[solve[eauto with imp_steps]|]).
```

```
1 focused subgoal (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  ============================
   trans imp_step spec
     (If (BCon (negb (n =? 0)))
        (Seq (Assign "s" (Add (Var "s") (Var "n")))
           (While (Not (Eq (PostDec "n") (Con 0)))
              (Assign "s" (Add (Var "s") (Var "n"))))) Skip,
     Env.add "n" (n - 1) env) (s_result (s + n * (n - 1) / 2))
```

This tactic successfully carried the proof to a point where a case distinction is needed. The if condition has been evaluated to a boolean, but the boolean is expressed symbolically as negb (n =? 0). We need to split into cases where the expression is a literal true or false, but we should add hypotheses expressing the conditions in forms more convienient for further reasoning.

Our final proof automation handles this case, but here we handle it manually.

```
Coq < destruct (Z.eqb_spec n 0).
2 focused subgoals (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  e : n = 0
  ============================
   trans imp_step spec
     (If (BCon (negb true))
        (Seq (Assign "s" (Add (Var "s") (Var "n")))
           (While (Not (Eq (PostDec "n") (Con 0)))
              (Assign "s" (Add (Var "s") (Var "n"))))) Skip,
     Env.add "n" (n - 1) env) (s_result (s + n * (n - 1) / 2))
subgoal 2 is:
 trans imp_step spec
   (If (BCon (negb false))
      (Seq (Assign "s" (Add (Var "s") (Var "n")))
         (While (Not (Eq (PostDec "n") (Con 0)))
            (Assign "s" (Add (Var "s") (Var "n"))))) Skip,
   Env.add "n" (n - 1) env) (s_result (s + n * (n - 1) / 2))
```

The function `Z.eqb_spec` has a specially designed return type so making this case distinction replaces the expression `n =?  0` with `true` or `false` while adding a hypothesis `n = 0` or `n <> 0`.

In the $n = 0$ case we use the hypothesis to replace n with 0 everywhere, and then automatic execution carries past the loop to the end of the code.

```
Coq < subst n.
2 focused subgoals (unfocused: 0)


  s : Z
  env : Env.t Z
  H0 : Env.MapsTo "s" s env
  H : Env.MapsTo "n" 0 env
  ============================
   trans imp_step spec
     (If (BCon (negb true))
        (Seq (Assign "s" (Add (Var "s") (Var "n")))
           (While (Not (Eq (PostDec "n") (Con 0)))
              (Assign "s" (Add (Var "s") (Var "n"))))) Skip,
     Env.add "n" (0 - 1) env) (s_result (s + 0 * (0 - 1) / 2))
subgoal 2 is:
 trans imp_step spec
   (If (BCon (negb false))
      (Seq (Assign "s" (Add (Var "s") (Var "n")))
         (While (Not (Eq (PostDec "n") (Con 0)))
            (Assign "s" (Add (Var "s") (Var "n"))))) Skip,
   Env.add "n" (n - 1) env) (s_result (s + n * (n - 1) / 2))

Coq < repeat(eapply dstep;[solve[eauto with imp_steps]|]).
2 focused subgoals (unfocused: 0)

  s : Z
  env : Env.t Z
  H0 : Env.MapsTo "s" s env
  H : Env.MapsTo "n" 0 env
  ============================
   trans imp_step spec (Skip, Env.add "n" (0 - 1) env)
     (s_result (s + 0 * (0 - 1) / 2))
subgoal 2 is:
 trans imp_step spec
   (If (BCon (negb false))
      (Seq (Assign "s" (Add (Var "s") (Var "n")))
         (While (Not (Eq (PostDec "n") (Con 0)))
            (Assign "s" (Add (Var "s") (Var "n"))))) Skip,
   Env.add "n" (n - 1) env) (s_result (s + n * (n - 1) / 2))

Coq < eapply ddone;split;[reflexivity|map_lookup].
2 focused subgoals (unfocused: 0)

  s : Z
  env : Env.t Z
  H0 : Env.MapsTo "s" s env
  H : Env.MapsTo "n" 0 env
  ============================
   s = s + 0 * (0 - 1) / 2
subgoal 2 is:
 trans imp_step spec
   (If (BCon (negb false))
      (Seq (Assign "s" (Add (Var "s") (Var "n")))
         (While (Not (Eq (PostDec "n") (Con 0)))
            (Assign "s" (Add (Var "s") (Var "n"))))) Skip,
   Env.add "n" (n - 1) env) (s_result (s + n * (n - 1) / 2))

Coq < auto with zarith.
```

```
1 focused subgoal (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  n0 : n <> 0
  ============================
   trans imp_step spec
     (If (BCon (negb false))
        (Seq (Assign "s" (Add (Var "s") (Var "n")))
           (While (Not (Eq (PostDec "n") (Con 0)))
              (Assign "s" (Add (Var "s") (Var "n"))))) Skip,
     Env.add "n" (n - 1) env) (s_result (s + n * (n - 1) / 2))
```

The target predicate is a conjunction of two assertions, namely that the current code equals `Skip`, which is obviously true, and that the current environment assigns variable `s` the expected value. We see that `map_lookup` leaves an arithmetic equation to be solved when the actual and expected values don't unify. The `zarith` database includes a lemma necessary for `auto` to handle the equation.

In the $n \neq 0$ case we will need to execute through the body of the loop to reach a state where the `loop_claim` claim can be used. We again begin by automatically taking many steps.

```
Coq < repeat (eapply dstep;[solve[eauto with imp_steps]|]).
1 focused subgoal (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  n0 : n <> 0
  ============================
   trans imp_step spec
```

```
     (Seq (Assign "s" (Add (Var "s") (Var "n")))
        (While (Not (Eq (PostDec "n") (Con 0)))
           (Assign "s" (Add (Var "s") (Var "n")))),
     Env.add "n" (n - 1) env) (s_result (s + n * (n - 1) / 2))
```

Now the step that is not automatically taken is looking up variable `s` to begin evaluating the expression `s+n` of the assignment. The `imp_steps` database only contains rules from the step relations of imp.

Manually applying all the appropriate rules leaves the hypothesis that we need to be able to handle automatically.

```
Coq < eapply dstep;
Coq <   [apply step_seq1,step_assign1,step_add1,step_var|].
2 focused subgoals (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  n0 : n <> 0
  ============================
   Env.MapsTo "s" ?317 (Env.add "n" (n - 1) env)
subgoal 2 is:
 trans imp_step spec
   (Seq (Assign "s" (Add (Con ?317) (Var "n")))
      (While (Not (Eq (PostDec "n") (Con 0)))
         (Assign "s" (Add (Var "s") (Var "n")))),
   Env.add "n" (n - 1) env) (s_result (s + n * (n - 1) / 2))
```

This lookup can be handled by the `map_lookup` tactic. Hint databases can also contain instructions to apply arbitrary tactics, not just hints to apply lemmas or constructors. These are specified along with a pattern describing goals where it is worthwhile to try using the tactic. After registering `map_lookup` in the `imp_steps` database to be used on `MapsTo` goals, the hypothesis is completed automatically.

```
Coq < Hint Extern 1 (Env.MapsTo _ _ _)
Coq <   => (map_lookup;solve[auto with zarith]) : imp_steps.
```

```
Coq < eauto with imp_steps.
1 focused subgoal (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  n0 : n <> 0
  ============================
   trans imp_step spec
     (Seq (Assign "s" (Add (Con s) (Var "n")))
        (While (Not (Eq (PostDec "n") (Con 0)))
           (Assign "s" (Add (Var "s") (Var "n")))),
     Env.add "n" (n - 1) env) (s_result (s + n * (n - 1) / 2))
```

With this addition additional variable lookups are handled automatically.

```
Coq < repeat (eapply dstep;[solve[eauto with imp_steps]|]).
1 focused subgoal (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  n0 : n <> 0
  ============================
   trans imp_step spec
     (If (BCon (negb (n - 1 =? 0)))
        (Seq (Assign "s" (Add (Var "s") (Var "n")))
           (While (Not (Eq (PostDec "n") (Con 0)))
              (Assign "s" (Add (Var "s") (Var "n"))))) Skip,
     Env.add "n" (n - 1 - 1)
```

```
       (Env.add "s" (s + (n - 1)) (Env.add "n" (n - 1) env)))
     (s_result (s + n * (n - 1) / 2))
```

Unfortunately this went too far, and entered the while loop again rather than stopping at the loop to allow using `loop_claim`.

One solution is to check whether any claim of the specification might apply before taking any execution steps. We could try to automatically use a claim with a tactic like

```
eapply dtrans;[eapply loop_claim;solve[eauto with imp_steps]|].
```

but automatic execution should if a claim potentially applies, even if manual assistance might be required to show that all its hypotheses hold.

Instead, we can conclude the claim couldn't possibly apply if

```
eapply dtrans;[eapply loop\_claim|]
```

fails. Manually specifying the number of steps to reach the state just before `loop` is reached, and using `try` *`tac`* to turn a failure into leaving the goal unchanged, we see that this test fails until the loop is reached, and succeeds at the loop.

```
Coq < Undo 1.
1 focused subgoal (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  n0 : n <> 0
  ============================
   trans imp_step spec
     (Seq (Assign "s" (Add (Con s) (Var "n")))
        (While (Not (Eq (PostDec "n") (Con 0)))
           (Assign "s" (Add (Var "s") (Var "n")))),
     Env.add "n" (n - 1) env) (s_result (s + n * (n - 1) / 2))

Coq < do 3 (eapply dstep;[solve[eauto with imp_steps]|]).
```

```
1 focused subgoal (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  n0 : n <> 0
  ============================
   trans imp_step spec
     (Seq Skip
        (While (Not (Eq (PostDec "n") (Con 0)))
           (Assign "s" (Add (Var "s") (Var "n")))),
     Env.add "s" (s + (n - 1)) (Env.add "n" (n - 1) env))
     (s_result (s + n * (n - 1) / 2))
Coq < try (eapply dtrans;[eapply loop_claim|]).
1 focused subgoal (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  n0 : n <> 0
  ============================
   trans imp_step spec
     (Seq Skip
        (While (Not (Eq (PostDec "n") (Con 0)))
           (Assign "s" (Add (Var "s") (Var "n")))),
     Env.add "s" (s + (n - 1)) (Env.add "n" (n - 1) env))
     (s_result (s + n * (n - 1) / 2))
Coq < eapply dstep;[solve[eauto with imp_steps]|].
1 focused subgoal (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  n0 : n <> 0
  ============================
   trans imp_step spec
     (While (Not (Eq (PostDec "n") (Con 0)))
        (Assign "s" (Add (Var "s") (Var "n"))),
     Env.add "s" (s + (n - 1)) (Env.add "n" (n - 1) env))
     (s_result (s + n * (n - 1) / 2))
Coq < try (eapply dtrans;[eapply loop_claim|]).
3 focused subgoals (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  n0 : n <> 0
  ============================
   Env.MapsTo "n" ?640
     (Env.add "s" (s + (n - 1)) (Env.add "n" (n - 1) env))
subgoal 2 is:
 Env.MapsTo "s" ?639
   (Env.add "s" (s + (n - 1)) (Env.add "n" (n - 1) env))
subgoal 3 is:
 forall k' : cfg,
 s_result (?639 + ?640 * (?640 - 1) / 2) k' ->
 trans imp_step spec k' (s_result (s + n * (n - 1) / 2))
```

To move onto taking an evaluation step if attempting to apply a claim fails, we use the tactic syntax `tac1 || tac2`, which tries the first tactic, and uses the second only if the first failed or made no changes to the goal. To break out of the `repeat` loop if the application succeds, we use the tactic `fail 1` which creates a "failure at level 1", which breaks past the first surrounding construct that ususally catches a failure (here ||). We undo the manual tests and try the improved tactic.

```
Coq < Undo 4.
1 focused subgoal (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  n0 : n <> 0
  ============================
   trans imp_step spec
     (Seq (Assign "s" (Add (Con s) (Var "n")))
        (While (Not (Eq (PostDec "n") (Con 0)))
           (Assign "s" (Add (Var "s") (Var "n")))),
     Env.add "n" (n - 1) env) (s_result (s + n * (n - 1) / 2))

Coq < repeat ((eapply dtrans;[eapply loop_claim|];fail 1)
Coq <       ||(eapply dstep;[solve[eauto with imp_steps]|])).
1 focused subgoal (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  n0 : n <> 0
  ============================
   trans imp_step spec
     (While (Not (Eq (PostDec "n") (Con 0)))
        (Assign "s" (Add (Var "s") (Var "n"))),
     Env.add "s" (s + (n - 1)) (Env.add "n" (n - 1) env))
     (s_result (s + n * (n - 1) / 2))
```

This stops right at the point where loop_claim might apply.

```
Coq < eapply dtrans;[eapply loop_claim;eauto with imp_steps|].
1 focused subgoal (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  n0 : n <> 0
  ============================
   forall k' : cfg,
   s_result (s + (n - 1) + (n - 1) * (n - 1 - 1) / 2) k' ->
   trans imp_step spec k' (s_result (s + n * (n - 1) / 2))

Coq < destruct k';intros;eapply ddone.
1 focused subgoal (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  n0 : n <> 0
  s0 : Stmt
  t : Env.t Z
  H1 : s_result (s + (n - 1) + (n - 1) * (n - 1 - 1) / 2) (s0, t)
  ============================
   s_result (s + n * (n - 1) / 2) (s0, t)

Coq < destruct H1;split;[assumption|map_lookup].
1 focused subgoal (unfocused: 0)

  n : Z
  s : Z
  env : Env.t Z
  H : Env.MapsTo "n" n env
  H0 : Env.MapsTo "s" s env
  n0 : n <> 0
  s0 : Stmt
  t : Env.t Z
  H1 : s0 = Skip
  H2 : Env.MapsTo "s" (s + (n - 1) + (n - 1) * (n - 1 - 1) / 2) t
  ============================
   s + (n - 1) + (n - 1) * (n - 1 - 1) / 2 = s + n * (n - 1) / 2
```

We invoke the claim, and deal with most of the proof that the goal is met, leaving only a numerical equation to be solved. That's a separate issue from illustrating program verification, so we extract the statement into a lemma and omit the proof.

```
Coq < Lemma sum_ind_algebra : forall s n,
Coq <    s + (n-1) + (n-1) * (n - 1 - 1) / 2 = s + n * (n-1) / 2.

Coq < Qed.

Coq < apply sum_ind_algebra.
No more subgoals.

Coq < Qed.
```

This concludes the proof.

Appendix B

Selected Coq Code

This appendix includes key Coq code from our mechanically checked developments. A full disctribution including a build system and some further files is available from

`http://fsl.cs.illinois.edu/coinduction/`.

This document should also contain an archive as a PDF attachment.

Our two main objectives in selecting the code for these appendices are to present the program proofs corresponding to Table 3.1, and to show that the definitions presented in mathematical notation in this paper are faithfully reflected in the mechanically checked development, and vice versa. We also wished to include the definition of the example languages.

The remainder of this chapter presents the statement and proofs of the core core definitions and theorems of our proof system, the Coq definitions of the representation predicates and specification abbreviations described earlier, and then also the full specification proved for the Schorr-Waite algorithm (in Appendix B.2.3).

In Appendix B.3 we give the definitions of the configurations and transition relations of the example languages. We also show the definitions of maps and primitive map patterns, to conclude the supporting code.

Appendix B.4 gives the complete text of the verifications of each of the example programs, in each language.

B.1 Core Proof System

The following code is the Coq module which defines reachability (`reaches`) validity of a set of claims (`sound`), and the main proof principle used for verification (`proved-sound`). Language independence of the results is shown in the most concrete way with this file, which is compiled once to a Coq object file and imported unchanged into every and imported unchanged into every program verification in any of our example languages. For performance reasons we define a single inductive type `trans` which is equivalent to $(\text{step} - R \cup \text{trans} \cup \text{proved})^*$.

```
Set Implicit Arguments.

Section relations.
Variables (cfg : Set) (cstep : cfg -> cfg -> Prop).

Definition Spec : Type := cfg -> (cfg -> Prop) -> Prop.

CoInductive reaches (k : cfg) (P : cfg -> Prop) : Prop :=
 | rdone : P k -> reaches k P
 | rstep : forall k', cstep k k' -> reaches k' P -> reaches k P.

Definition sound (Rules : Spec) : Prop :=
  forall x P, Rules x P -> reaches x P.

Inductive step (X : Spec) (k : cfg) (P : cfg -> Prop) : Prop :=
 | sdone : P k -> step X k P
 | sstep : forall k', cstep k k' -> X k' P -> step X k P.

CoFixpoint stable_sound (Rules : Spec)
  (Hstable : forall x P, Rules x P -> step Rules x P)
   : sound Rules := fun x P H => match Hstable _ _ H with
    | sdone pf => rdone _ _ pf
    | sstep k' Hstep H' =>
        rstep Hstep (stable_sound Hstable _ _ H')
  end.

Inductive trans (X : Spec) (k : cfg) (P : cfg -> Prop) : Prop :=
 | ddone : P k -> trans X k P
 | dtrans' : forall Q, trans X k Q ->
      (forall k', Q k' -> trans X k' P) -> trans X k P
 | drule : X k P -> trans X k P
```

```
  | dstep : forall k', cstep k k' -> trans X k' P -> trans X k P.
(* The desired transivity rule. Violates positivity rules,
   so the dtrans' constructor used Mendler style *)
Definition dtrans (X : Spec) (k : cfg) (P Q : cfg -> Prop)
  (rule : X k Q) (rest : forall k', Q k' -> trans X k' P) :=
  @dtrans' X k P Q (drule _ _ _ rule) rest.

Lemma trans_stable (Rules : Spec) :
  (forall x P, Rules x P -> step (trans Rules) x P)
  -> (forall x P, trans Rules x P -> step (trans Rules) x P).
Proof. induction 2;eauto using step;
destruct IHtrans;eauto using step,dtrans'. Qed.

Lemma proved_sound (Rules : Spec) :
  (forall x P, Rules x P -> step (trans Rules) x P)
  -> sound Rules.
Proof. unfold sound;intros;eapply stable_sound;
[apply trans_stable|apply drule];eassumption. Qed.
```

Using these definitions, all the proofs of programs in Appendix B.4 begin by directly claiming the correctness of the specification using `sound` and then immediately apply `proved-sound` to move from the correctness claim to showing an inclusion in the coinductive style.

B.2 Representation Predicates

In this section we show the definitions of all the notation we used to simplify writing specifications, for each of the programming languages. The descriptions of heap data structures are fairly similar, while abbreviations for making claims about function definitions or statements in function bodies are more affected by the differences between languages.

B.2.1 HIMP

Here is the definition of lists used in the HIMP examples. The primary definition is that of list segments, complete lists are defined as a segment ending in a zero pointer. The values of HIMP include records with named fields, which can be used to store structures such as list nodes at a single address in the heap. Stack explores a lower-level representation, using consecutive addresses to store fields of structures.

```
Require Export himp_claims.
Require Export patterns.

(** A list node in memory *)
Notation list_node val next :=
  (KStruct (Struct (”val” s|−> KInt val :* ”next” s|−> KInt next))).
(** And some abbreviations for code working with list nodes *)
Notation arr_val v := (EProject (ELoad (EVar v)) ”val”).
Notation arr_next v := (EProject (ELoad (EVar v)) ”next”).
Notation build_node v p := (EBuild (”val” s|−> v :* ”next” s|−> p)).

Fixpoint rep_seg (val : list Z) (tailp p : Z) : MapPattern k k :=
  match val with
    | nil => constraint (p = tailp)
    | x :: xs => constraint (p <> 0) :* existsP p',
        p h|−> list_node x p' :* rep_seg xs tailp p'
  end%pattern.
Notation rep_list l := (rep_seg l 0).
```

Here are the abbreviations used for writing specifications. The function `loop-tails` helps make auxiliary claims about the loops in a definition, without needing to explicitly name or repeat the loops. Note the claim extends from the loop all the way to the end of the function (if there are loops after the one being describe, the specification will need another auxiliary claim covering that loop).

```
Require Export patterns.
Require Export proof.
Require Export himp_steps.

(* Environment or record entry, taking a string and kitem*)
Notation ”x␣'s|−>'␣y” :=
  (mapItem (kra (KId x) kdot) (kra (y : kitem) kdot))
  (at level 50, no associativity) : Map.
```

```
(* Heap entry, from an integer key and kitem value *)
Notation "x␣'h|−>'␣y" :=
  (mapItem (kra (KInt x) kdot) (kra (y : kitem) kdot))
  (at level 50, no associativity) : Map.
Notation "x␣'h|−>'␣y" :=
  (itemP (kra (KInt x) kdot) (kra (y : kitem) kdot))
  (at level 50, no associativity) : MapPattern.
(* Notations to help unify typed and untyped maps *)
Notation load_field v f := (EProject (ELoad (EVar v)) f).
(*
Notation funEntry name args body :=
   (name%string s|−> FunDef name args body).
 *)

Definition value_heap ret krest store stack frame funs mark
  : kcfg −> Prop := fun c' => match c' with
  | KCfg k' store' stack' heap' funs' mark' =>
      exists r', k' = kra r' krest /\ store' ~= store
      /\ stack' = stack /\ heap' |= ret r' :* litP frame
      /\ funs' ~= funs /\ (mark' >= mark)%Z
  end.

Definition heap_fun (R : Spec kcfg) (d:Defn) :
  forall (args : list KResult) (init_heap : MapPattern k k)
    (ret : Z −> MapPattern k k), Prop :=
  match d with FunDef name formals body =>
    fun args init_heap ret =>
      forall krest store stack heap funs mark otherfuns,
      funs ~= name s|−> KDefn d :* otherfuns −>
      (mark > 0)%Z −>
      forall frame,
      heap |= (init_heap :* litP frame) −>
      R (KCfg (kra (ECall name (map KResultToExp args)) krest)
               store stack heap funs mark)
        (value_heap (fun r => existsP v, constraint (r = KInt v)
                                            :* ret v)%pattern
                krest store stack frame funs mark)
  end.

Definition return_heap ret stack frame funs mark : kcfg -> Prop :=
  fun c' => match c' with
  | KCfg k' store' stack' heap' funs' mark' =>
    exists r' krest, k' = kra (KStmt (SReturn r')) krest
    /\ stack' = stack /\ heap' |= ret r' :* litP frame
    /\ funs' ~= funs /\ (mark' >= mark)%Z
  end.

Definition suffix_claim (R : Spec kcfg)
    (body : k) (init_store : Map k k)
    (init_heap : MapPattern k k)
    (final_heap : Z -> MapPattern k k) : Prop :=
  forall stack store store_rest heap frame funs mark,
     store ~= init_store :* store_rest ->
     heap |= (init_heap :* litP frame) ->
     (mark > 0)%Z ->
  R (KCfg body store stack heap funs mark)
    (return_heap (fun r => existsP v, constraint (r = ECon v)
                                          :* final_heap v)%pattern
                  stack frame funs mark).

Fixpoint loop_tails (s : Stmt) (rest : k) : list k :=
  match s with
    | Seq s1 s2 => loop_tails s1 (kra s2 rest) ++ loop_tails s2 rest
    | SIf _ s1 s2 => loop_tails s1 rest ++ loop_tails s2 rest
    | SWhile _ s' => (kra s rest) :: loop_tails s' (kra s rest)
    | _ => nil
  end.

Definition body (def : Defn) :=
  match def with FunDef _ _ body => body end.
```

```
Definition heap_loop (R : Spec kcfg) (def : Defn) (n : nat) :
  Map k k -> MapPattern k k -> (Z -> MapPattern k k) -> Prop :=
  suffix_claim  R (nth n (loop_tails  (body def) kdot) kdot).
```

B.2.2 Stack

Here is the definition of lists used in the Stack examples. This definition uses two consecutive addresses for a list node, demonstrating that we can also handle lower-level reasoning.

```
Require Import stack.
Require Import patterns.

Fixpoint rep_seg lst  (tailp  p : Z) : MapPattern Z Z :=
 match lst with
    | nil  => constraint (p = tailp)
    | x ::  xs => constraint (p <> 0) :* existsP  p',
          p |-> x :* (p + 1) |-> p' :* rep_seg xs tailp  p'
  end%pattern.
Notation rep_list  l := (rep_seg l 0).
```

Here are the abbreviations used for making specifications. The dependent type `stk-pat` is used to allow specifications to describe the stack using predicates taking separate arguments for each expected stack entry, instead of repeating list-destructuring code in each claim.

```
Require Import patterns.
Require Import stack.

Set Implicit  Arguments.

Fixpoint min_length (n : nat) {A} (l : list  A) : Prop :=
  match n, l with
    | O, _ => True
    | S _, nil  => False
    | S n', _ ::  l'  => min_length n' l'
  end.
```

```
Fixpoint stk_pat (n : nat) : Type :=
  match n with
    | O => MapPattern Z Z
    | S n' => Z -> stk_pat n'
  end.
Fixpoint stk_pat_app {n} (P : stk_pat n) (stk : list Z) :=
  match n return stk_pat n -> MapPattern Z Z with
    | O => fun P => P
    | S n' => fun P =>
                 match stk with
                   | nil => constraint False
                   | x :: stk' => stk_pat_app (P x) stk'
                 end
  end P.

Definition fun_claim (spec : cfg -> (cfg -> Prop) -> Prop) name body
  num_in (pre : stk_pat num_in) num_out (post : stk_pat num_out) :=
  forall rest stk rstk heap frame funs otherfuns mark,
    funs ~= name |-> body :* otherfuns ->
    min_length num_in stk ->
    mark > 0 ->
    heap |= stk_pat_app pre stk :* litP frame ->
    spec (Cfg (Call name :: rest) stk rstk heap funs mark)
         (fun c' => match c' with
            | Cfg code' stk' rstk' heap' funs' mark' =>
              code' = rest /\ min_length num_out stk'
             /\ skipn num_in stk = skipn num_out stk'
             /\ heap' |= stk_pat_app post stk' :* litP frame
             /\ rstk = rstk'
             /\ funs ~= funs'
             /\ mark' >= mark'
           end).

Fixpoint loops' i k : list (list Inst) :=
  match i with
    | If thn els =>
```

```
        let fix loops l k :=
          match l with
            | nil => nil
            | i :: l => loops' i (l++k) ++ loops l k
          end
        in loops thn k ++ loops els k
    | While cond body =>
        (i::k)::nil
    | _ => nil
  end.
Fixpoint loops (l : list Inst) : list (list Inst) :=
  match l with
    | nil => nil
    | i :: l' => loops' i l' ++ loops l'
  end.

Definition loop_claim (spec : cfg -> (cfg -> Prop) -> Prop) ix code
  num_in (pre : stk_pat num_in) num_out (post : stk_pat num_out) :=
  forall stk rstk heap frame funs mark,
    min_length num_in stk ->
    heap |= stk_pat_app pre stk :* litP frame ->
    mark > 0 ->
    spec (Cfg (nth ix (loops code) nil) stk rstk heap funs mark)
         (fun c' => match c' with
           | Cfg code' stk' rstk' heap' funs' mark' =>
               (exists rest', code' = Ret :: rest')
            /\ min_length num_out stk'
            /\ skipn num_in stk = skipn num_out stk'
            /\ heap' |= stk_pat_app post stk' :* litP frame
            /\ rstk = rstk'
            /\ funs ~= funs'
            /\ mark' >= mark'
          end).
```

Lambda

Here is the definition of lists used in the Lambda examples. Unlike the earlier languages, lists are described as holding any value, not just integers. We use the value Nil to indicate the end of lists, rather than the zero address.

```
Require Import lambda.
Require Import patterns.

Fixpoint list_seg l (tailp p : val) : MapPattern loc val :=
  match l with
    | nil => constraint (tailp = p)
    | v :: vs => existsP l, constraint (p = Addr l) :*
        existsP p', l |-> Pair v p' :* list_seg vs tailp p'
  end%pattern.
Notation rep_list l := (list_seg l Nil).
```

Unlike Stack and HIMP, Lambda has neither named functions nor dedicated looping constructs. A re-usable function is given as a closed lambda-expression.

Specifications will make a claim about a state that is executing the function body in an environment containing values for all the expected arguments - under the evaluation order of Lambda this is approximately the earliest point where all arguments have been reduced to values.

Recursion is implemented with a fixpoint combinator. By the time arguments reach the body of the function, the fixpoint will have expanded into a closure (closed over other closures) constructed from pieces of the fixpoint combinator, its argument, and the environment when the fixpoint combinator itself was reduces. To assist in writing such a specification, the `fix-env` function expands into the necessary closure, and tail environment.

```
Require Import patterns.
Require Import proof.
Require Import lambda.

Definition fix_env (body:exp) (env0:list val) : list val :=
  Closure (((Var 1) (Var 1)) (Var 0))
    (Env (Closure strict_worker
          (Env (Closure  body (Env env0)::nil))::nil))
  ::env0.
Definition evals  F m k post :  cfg  -> Prop :=
  fun c'  => match c' with
    | Cfg (Val  v)  s'  m' k'  =>
      s'  |= post  v :*  litP  F /\ k'  = k /\ (m' >= m)%Z
    | _ => False
  end.

Definition exp_val (R :  Spec cfg)  c e  pre  post  :=
  forall s F m k, s  |= pre  :*  litP  F ->
  R (Cfg (Exp c (Env e))  s  m k) (evals  F m k post).
```

B.2.3 Schorr-Waite

Here is the code of the Schorr-Waite implementation to be proved.

```
Require Import graph.

Definition schorr_waite_loop  :=
  SWhile ("p"  <> 0)
    {{ "t"  <- "p"->>"left"
     ;  "p"  <<- build_tree ("p"->>"val" + 1)
                    ("p"->>"right") "q"
     ;  SIf  (("p"->>"val" = 3)
              ||  (("t"  <> 0) && ("t"->>"val" = 0)))
          {{ "q" <- "p" ; "p" <- "t" }}
          {{ "q" <- "t" }}
    }}.
Definition schorr_waite_code :=
  SIf  ("root"  = 0) SReturnVoid
   {{ Decl  "p"
    ;  Decl  "q"
    ;  Decl  "t"
    ;  "p"<-"root"
    ;  "q"<-0
    ;  schorr_waite_loop
    ; SReturnVoid
    }}.
Definition schorr_waite_def :=
  FunDef "schorr_waite" ["root"] schorr_waite_code.
```

The specification of the algorithm is given in terms of representations predicates describing a marked graph in terms of a depth-first traversal tree, and the stack in terms of zippers into that tree. Here are the abstract types, the representation predicates, and the specification.

```
(* Represent a graph reachable from a single point by the
   depth-first traversal, as a tree with backlinks allowed *)
Inductive dfs_tree (zs : list Z) : list Z -> Set :=
  | Null : dfs_tree zs zs
  | Ref : forall z, In z zs -> dfs_tree zs zs
  | Node : forall phere zs' zs'', ~In phere zs ->
      dfs_tree (phere::zs) zs' -> dfs_tree zs' zs''
      -> dfs_tree zs zs''.
Inductive dfs_tree_ctx (ls : list Z)
  : list Z -> list Z -> list Z -> Set :=
  | LeftOf : forall phere ls' rs rs' rs'', ~In phere ls'
    -> dfs_tree_ctx ls ls' rs' rs'' -> dfs_tree rs rs'
    -> dfs_tree_ctx ls (phere::ls') rs rs''
  | RightOf : forall phere ls' ls'' rs rs', ~In phere ls'
    -> dfs_tree (phere::ls') ls'' -> dfs_tree_ctx ls ls' rs rs'
    -> dfs_tree_ctx ls ls'' rs rs'
  | Top : forall rs, dfs_tree_ctx ls ls rs rs.
Fixpoint plug {ls ls' rs rs'} (c : dfs_tree_ctx ls ls' rs rs')
  : dfs_tree ls' rs -> dfs_tree ls rs' :=
  match c with
  | LeftOf phere _ _ _ _ n path r =>
      fun t => plug path (@Node _ phere _ _ n t r)
  | RightOf phere _ _ _ _ n l path =>
      fun t => plug path (@Node _ phere _ _ n l t)
  | Top _ => fun t => t
  end.
```

```
Definition dfs_tree_addr {zs} {zs'} (t : dfs_tree zs zs') : Z :=
  match t with
    | Null => 0%Z
    | Ref addr _ => addr
    | Node addr _ _ _ _ _ => addr
  end.
Fixpoint rep_dfs_tree (v : Z) {zs} {zs'} (t : dfs_tree zs zs')
    : MapPattern k k :=
  match t with
    | Null => emptyP
    | Ref z _ => emptyP
    | Node p _ _ _ l r => constraint (p <> 0) :*
        p h|-> tree_node v (dfs_tree_addr l) (dfs_tree_addr r)
         :* rep_dfs_tree v l :* rep_dfs_tree v r
  end%pattern.

Definition dfs_tree_ctx_addr {ls ls' rs rs'}
  (c : dfs_tree_ctx ls ls' rs rs') : Z :=
  match c with
  | LeftOf phere _ _ _ _ _ _ _ => phere
  | RightOf phere _ _ _ _ _ _ _ => phere
  | Top _ => 0%Z
  end.
Fixpoint rep_dfs_ctx {ls ls' rs rs'}
  (ctx : dfs_tree_ctx ls ls' rs rs') : MapPattern k k :=
  match ctx with
  | Top _ => emptyP
  | LeftOf phere _ _ _ _ _ path rtree =>
        constraint (phere <> 0)
     :* phere h|-> tree_node 1 (dfs_tree_addr rtree)
                                (dfs_tree_ctx_addr path)
     :* rep_dfs_ctx path :* rep_dfs_tree 0 rtree
  | RightOf phere _ _ _ _ _ ltree path =>
        constraint (phere <> 0)
     :* phere h|-> tree_node 2 (dfs_tree_ctx_addr path)
                                (dfs_tree_addr ltree)
```

```
      :*  rep_dfs_tree  3  ltree   :*  rep_dfs_ctx  path
end%pattern.

(*  At some points  in  the  loop,  q  points  at  the  path  and
   p  at  an  unmarked  (non−Ref, non−Null)  tree.  At other  points,
     p  is  at  the  root  of  the  path  and q  is  holding  one of  the
   finished   subtrees  of  that  node *)
Definition isWellMarked  (descending  :  bool)  (p  q  :  Z)
   {ls  ls'   rs  rs'}  (tree  :  dfs_tree  ls'   rs)
  (path  :  dfs_tree_ctx  ls  ls'   rs  rs')   :  Prop :=
  if descending  then  p  <> 0
  /\ match tree  with   Node _ _ _ _ _ _ _ => True | _ => False end
  /\    p  = dfs_tree_addr tree  /\ q  = dfs_tree_ctx_addr path
  else  p  = dfs_tree_ctx_addr path /\ q  = dfs_tree_addr tree.

Inductive schorr_waite_spec :  Spec kcfg  :=
  schorr_waite_claim  :  forall c,
  kcell   c = kra schorr_waite_code kdot  −>
  forall {zs'} (t  :  dfs_tree  nil  zs'),
     store  c ~= "root" s|−> KInt (dfs_tree_addr t)  −>
  forall hframe, heap c |= rep_dfs_tree  0  t  :*  hframe −>
  schorr_waite_spec  c (fun c'  => exists rest,
            kcell   c'  = kra SReturnVoid rest
            /\ stk_equiv  (stack  c)  (stack  c')
            /\ functions   c ~= functions  c'
            /\ heap c'  |= rep_dfs_tree  3  t  :*  hframe)
  |  schorr_waite_loop_claim  :  forall c rest,
  kcell   c = kra schorr_waite_loop  rest   −>
  forall r t  p q,  store   c ~= "root" s|−> r :*  "t"  s|−> t
                              :*  "p" s|−> KInt p :*  "q" s|−> KInt q −>
  forall descending {ls'   rs  rs'}   tree
          (path  :  dfs_tree_ctx  nil   ls'   rs  rs'),
  isWellMarked  descending  p q tree  path  −> forall hframe,
  heap c |= rep_dfs_tree  (if descending  then  0 else  3)  tree
          :*  rep_dfs_ctx  path :*  hframe −>
  schorr_waite_spec  c (fun c'  =>
            kcell c' = rest
            /\ stk_equiv (stack c) (stack c')
            /\ functions c ~= functions c'
            /\ heap c' |= rep_dfs_tree 3 (plug path tree) :* hframe).
```

B.3 Language Definitions

Here we present the syntax and operational semantics of the three main languages we used in verification examples, the imperative HIMP, the stack-based Stack, and the simplified functional language Lambda.

B.3.1 Stack

The definition of the stack language is shortest. This begins by defining the basic instructions, and using them to define some common stack manipulations under standard Forth names. Next is the definition of the full configuration, and the transition relation which actually gives the operations semantics. Unlike standard Forth, the control structures are part of the grammar, the generalized `Dup` and `Roll` take the offset as an immediate constant rather than reading from the stack, and `Roll` rotates the stack in the opposite of the standard direction (which is more convenient in terms of Coq's list library).

```
Require Export maps.
Require Export ZArith.
Require Export String.
Require Export List.

Open Scope string_scope.
Open Scope list_scope.
Open Scope Z.

Inductive Inst : Set :=
  | Dup : nat -> Inst
  | Roll : nat -> Inst
  | Drop
  | Push : Z -> Inst
  | Binop : (Z -> Z -> Z) -> Inst
  | While : list Inst -> list Inst -> Inst
  | If : list Inst -> list Inst -> Inst
  | Load
  | Store
  | Call : string -> Inst
  | Ret : Inst
  | Alloc : nat -> Inst
  | Dealloc : Inst.

Definition Swap := Roll 1.
Definition Over := Dup 1.
Definition Nip := Swap::Drop::nil.
Definition Rot := Roll 2::Roll 2::nil.
Definition Plus1 := Push 1::Binop Z.add::nil.

Structure cfg := Cfg {
  code : list Inst;
  stack : list Z;
  rstack : list (list Inst);
  heap : Map Z Z;
  functions : Map string (list Inst);
  alloc_next : Z
  }.

(* Compute a map for initial bindings in Alloc,
   with some extra work to make a pretty term *)
Definition offset (base :Z) (n : nat) : Z :=
  match n with
    | O => base
    | _ => (base + Z.of_nat n)%Z
  end.
Fixpoint init n ix base : Map Z Z :=
  match n with
    | O => mapEmpty
    | S O => offset base ix  |-> 0%Z
    | S n'  => offset base ix  |-> 0%Z :* init n'  (S ix)  base
  end.

Local Notation basic_step step
    code  stk   heap  mark
    code2 stk2  heap2 mark2:=
  (forall rstk funs,
     step (Cfg code  stk   rstk  heap  funs  mark)
          (Cfg code2 stk2  rstk  heap2 funs  mark2)).
Local Notation inst_step  step  inst  stk  stk2  :=
  (forall rest  heap mark, basic_step  step
    (inst::rest)    stk  heap mark rest  stk2  heap mark).

Generalizable  Variables  n x y stk.
Inductive stack_step :  cfg  -> cfg -> Prop :=
  | dup :  `(inst_step  stack_step  (Dup n)
       stk  (nth_default  0 stk  n ::  stk))
  | roll   :  `(inst_step  stack_step  (Roll  n)
       (x::stk)   (firstn   n stk  ++ x :: skipn  n stk))
  | drop  :  `(inst_step  stack_step  Drop
       stk  (tl  stk))
  | push  :  `(inst_step  stack_step  (Push x)
       stk  (x  ::  stk))
  | binop  :  forall f, `(inst_step  stack_step  (Binop f)
       (x  ::  y ::  stk)  (f  y x ::  stk))
  | while  :  forall test body rest  stk  h m, basic_step  stack_step
       (While test  body::rest)   stk  h m
       (test++If  (body++While test body::nil)  nil::rest)     stk  h m
  | ifz   :  forall x t f  rest  stk  h m, basic_step  stack_step
       (If   t  f::rest)                                  (x::stk)   h m
       ((if Zneq_bool x 0 then t  else  f)++rest)       stk    h m
  | load  :  forall x rest  stk  v  heap heap'  mark,
       heap ~= x |-> v :* heap'  -> basic_step stack_step
         (Load::rest)   (x::stk)   heap                 mark
                rest    (v::stk)   (x|-> v :*  heap')  mark
  | store  : forall p v rest  stk  v'  heap heap'  mark,
       heap ~= p |-> v' :* heap'  -> basic_step stack_step
         (Store::rest)    (p::v::stk)    heap                   mark
                 rest              stk   (p |-> v :* heap')  mark
  | alloc1  : forall n rest  stk  heap m, basic_step  stack_step
       (Alloc  n::rest)   stk                        heap    m
        rest           (m::stk)  (init   n 0  m :* heap) (Z.of_nat  n + m)%Z
  | dealloc  : forall rest  x  v  stk  heap heap'  mark,
       heap ~= x |-> v :* heap'  -> basic_step stack_step
         (Dealloc::rest)    (x::stk)    heap   mark
                   rest          stk    heap'  mark
  | call   : forall f body rest  stk  rstk   heap funs  funs'   mark,
       funs  ~= f |-> body :* funs'  -> stack_step
          (Cfg  (Call   f::rest)    stk            rstk    heap funs  mark)
          (Cfg  body                stk  (rest::rstk)     heap funs  mark)
  | ret  : forall rest  rbody stk  rstk   heap funs  mark, stack_step
       (Cfg  (Ret::rest)    stk  (rbody::rstk)    heap funs  mark)
       (Cfg  rbody          stk               rstk    heap funs  mark).
Arguments nth_default / :  simpl  nomatch.
```

B.3.2 Lambda

The definition of the Lambda language is based on Matt Might's presentation of Felleisen's CESK machine.

```
Require Export  maps.
Require Export  ZArith.
Require Export  List.

Open Scope list_scope.

Definition loc  :  Set  := Z.
Inductive val  :  Set  :=
 |  Closure  (body :  exp)  (context  :  env)
 |  Num : nat -> val
 |  Addr :  loc  -> val
 |  Pair  :  val  -> val -> val
 (* Primitive functions *)
 | Inc | Dec | Plus | Plus1 (x:nat)
 | Eq | Eq1 (a:val)
 | Nil | Cons | Cons1 (a:val) | Car | Cdr
with exp : Set :=
 | App (f x : exp)
 | Lit (v : val)
 | Var (ix : nat)
 | Lam (body : exp)
 | Closed (e : exp)
 | If (c t e : exp)
 | Seq (a b : exp)
 | Ref (e : exp) | Dealloc (v : exp)
 | Deref (e : exp) | Assign (v : exp) (e : exp)
with env : Set := Env : list val -> env.

Coercion App : exp >-> Funclass.
Coercion Lit : val >-> exp.

Definition Let := (Lam (Lam ((Var 0) (Var 1)))).
Definition strict_worker : exp := (Var 1)
  ((Closed (Lam (Lam (((Var 1) (Var 1)) (Var 0))))) (Var 0)).
Definition strict_fix : exp :=
  Closed (Lam ((Lam ((Var 0) (Var 0))) (Lam strict_worker))).

Local Ltac eq_assums := pose eq_nat_dec;pose Z_eq_dec;
   pose list_eq_dec;decide equality.
Fixpoint val_eq_dec (x y:val) : {x=y}+{x<>y}
with exp_eq_dec (x y:exp) : {x=y}+{x<>y}
with env_eq_dec (x y:env) : {x=y}+{x<>y}.
eq_assums. eq_assums. eq_assums.
Defined.

Inductive control : Set := Val (v : val)
                         | Exp (e : exp) (env : env).
```

```
Inductive cont : Set :=
    Top
  | AppFun (e' : exp) (scope : env) (k : cont) (* app fun [] e' *)
  | AppArg (v' : val) (k : cont)               (* app arg v' [] *)
  | RefVal (k : cont)                          (* ref [] *)
  | SeqCtx (e : exp) (scope : env) (k : cont)  (* [] ; e *)
  | DerefTgt (k : cont)                        (* ! [] *)
  | AssignTgt (e : exp) (scope : env) (k : cont) (* [] := e *)
  | AssignVal (l : loc) (k : cont)               (* l := [] *)
  | DeallocCtx (k : cont)                      (* dealloc [] *)
  | IfCond (t e : exp) (scope : env) (k : cont).

Inductive cfg : Set :=
  Cfg {code:control
      ;heap:Map loc val
      ;mark:loc
      ;ctx:cont}.

Fixpoint get (ix : nat) {A} (l : list A) : option A :=
  match ix, l with
    | _, nil => None
    | O, cons v _ => Some v
    | S ix', cons _ l' => get ix' l'
  end.
Definition getEnv (ix : nat) (p : env) :=
  match p with
    | Env l => get ix l
  end.
Definition extend (x : val) (p : env) : env :=
  match p with
    | Env l => Env (x :: l)
  end.

Definition apply (f : val) (v : val) : option control :=
  match f with
    | Closure e p' => Some (Exp e (extend v p'))
    | Inc  => match v with
      | Num n => Some (Val (Num (S n)))
      | _ => None
      end
    | Dec => match v with
      | Num n => Some (Val (Num (pred n)))
      | _ => None
      end
    | Plus  => match v with
      | Num n => Some (Val (Plus1 n))
      | _ => None
      end
    | Plus1 x => match v with
      | Num n => Some (Val (Num (x+n)))
      | _ => None
      end
    | Eq => Some (Val (Eq1 v))
    | Eq1 x => Some (if val_eq_dec x v then Val (Num 0)
                                       else  Val Nil)
    | Cons => Some (Val (Cons1 v))
    | Cons1 a => Some (Val (Pair a v))
    | Car =>
      match v with
        | Pair  a _ => Some (Val a)
        | _ => None
      end
    | Cdr =>
      match v with
        | Pair  _ b => Some (Val b)
        | _ => None
      end
    | Pair  _ _ => None
    | Nil  => None
    | Num _ => None
    | Addr _ => None
  end.
```

```
Definition isNil  v :=
 match v with
   | Nil  => true
   | _ => false
 end.

Inductive lam_step : cfg  -> cfg -> Prop :=
 | enter_function  :  forall e0 e1 p s  m k,
     lam_step (Cfg (Exp (App e0 e1)  p)  s  m k)
              (Cfg  (Exp e0  p)            s  m (AppFun e1 p k))
 | enter_argrgument :  forall v s e  p  m k,
     lam_step (Cfg  (Val  v)    s  m (AppFun e p k))
              (Cfg  (Exp e  p)  s  m (AppArg v k))
 | eval_call   :  forall v s f  k  m r,  apply  f  v = Some r ->
      lam_step (Cfg (Val  v)  s  m (AppArg f k))
               (Cfg  r          s  m k)
 | eval_lit   :  forall v p s  m k,
     lam_step (Cfg  (Exp (Lit  v)  p)  s  m k)
              (Cfg  (Val  v)           s  m k)
 | eval_lam :  forall e p s  m k,
     lam_step (Cfg  (Exp (Lam e) p)       s  m k)
              (Cfg  (Val  (Closure  e  p))  s  m k)
 | eval_var  :  forall i p v  s  m k, get  i   p = Some v ->
     lam_step (Cfg  (Exp (Var  i)  (Env p))  s  m k)
              (Cfg  (Val  v)                 s  m k)
 | enter_seq  :  forall e e'  p  s  m k,
     lam_step (Cfg  (Exp (Seq  e  e')  p)  s  m k)
              (Cfg  (Exp e  p)             s  m (SeqCtx e'  p  k))
 | eval_seq  :  forall v e p s  m k,
     lam_step (Cfg  (Val  v)    s  m (SeqCtx e p k))
              (Cfg  (Exp e  p)  s  m k)
 | enter_if   :  forall c t  e  p s  m k,
     lam_step (Cfg  (Exp (If   c  t  e)  p)  s  m k)
              (Cfg  (Exp c  p)              s  m (IfCond t  e  p k))
 | eval_if_nil   :  forall v s t  e  p  m k,  isNil   v = true  ->
```

```
    lam_step (Cfg (Val v)    s m (IfCond t e p k))
             (Cfg (Exp e p) s m k)
| eval_if_nonnil : forall v s t e p m k, isNil v = false ->
    lam_step (Cfg (Val v)    s m (IfCond t e p k))
             (Cfg (Exp t p) s m k)
| enter_ref : forall e p s m k,
    lam_step (Cfg (Exp (Ref e) p) s m k)
             (Cfg (Exp e p)       s m (RefVal k))
| eval_ref : forall v s m k,
    lam_step (Cfg (Val v)         s             m   (RefVal k))
             (Cfg (Val (Addr m)) (m|->v :* s) (m+1) k)%Z
| enter_dealloc : forall e p s m k,
    lam_step (Cfg (Exp (Dealloc e) p) s m k)
             (Cfg (Exp e p)           s m (DeallocCtx k))
| eval_dealloc : forall l s x s' m k, s ~= l|->x :* s' ->
    lam_step (Cfg (Val (Addr l)) s  m (DeallocCtx k))
             (Cfg (Val Nil)      s' m k)
| enter_deref : forall e p s m k,
    lam_step (Cfg (Exp (Deref e) p) s m k)
             (Cfg (Exp e p)         s m (DerefTgt k))
| eval_deref : forall l v s s' m k, s ~= l|->v :* s' ->
    lam_step (Cfg (Val (Addr l)) s m (DerefTgt k))
             (Cfg (Val v)        s m k)
| enter_assign_target : forall t e p s m k,
    lam_step (Cfg (Exp (Assign t e) p) s m k)
             (Cfg (Exp t p)            s m (AssignTgt e p k))
| enter_assign_val : forall l p e' s m k,
    lam_step (Cfg (Val (Addr l)) s m (AssignTgt e' p k))
             (Cfg (Exp e' p)     s m (AssignVal l k))
| eval_assign : forall l v s w s' m k, s ~= l|->w :* s' ->
    lam_step (Cfg (Val v) s             m(AssignVal l k))
             (Cfg (Val v) (l|->v :* s') m k)
| close_exp : forall e v s m k,
    lam_step (Cfg (Exp (Closed e) v)     s m k)
             (Cfg (Exp e        (Env nil)) s m k).
```

B.3.3 HIMP

For HIMP, we show only the syntax. The greater size is due in part to attempting to exactly follow the syntax of a K definition. The auxiliary functions are used to help implement subsorting, simple constructors are not sufficient if there are multiple paths of injections from one sort to another. HIMP also defines nicer notation for programs using Coq's notation system. This syntax can be seen in the example proofs in Appendix B.4.

```
Require Export ZArith.
Require Export String.
Require Export List.
Require Export maps.

Global Open Scope Z_scope.
Global Open Scope string_scope.
Global Open Scope list_scope.

Set Implicit  Arguments.

Inductive Defn : Set :=
  FunDef : string  -> list string  -> Stmt -> Defn
with Exp : Set :=
  BCon : bool -> Exp
| EVar : string  -> Exp
| ECon : Z -> Exp
| EValStruct : sval  -> Exp
| ExpUndef : Undef -> Exp
| ENeg : Exp -> Exp
| EMult : Exp -> Exp -> Exp
| EPlus : Exp -> Exp -> Exp
| EMinus : Exp -> Exp -> Exp
| EProject : Exp -> string -> Exp
| EDiv : Exp -> Exp -> Exp
| BLe : Exp -> Exp -> Exp
| BLt : Exp -> Exp -> Exp
| BEq : Exp -> Exp -> Exp
```

```
 | BAnd : Exp -> Exp -> Exp
 | BOr : Exp -> Exp -> Exp
 | EAlloc :  Exp
 | EBuild :  Map k k -> Exp
 | ECall  :  string   -> list  Exp -> Exp
 | ELoad : Exp -> Exp
 | BNot : Exp -> Exp
 | HOLE_Exp : Exp
with Frame : Set :=
   frame :  k  -> Map k k -> Frame
with kitem :  Set :=
   KBool : bool  -> kitem
 | KDefn : Defn -> kitem
 | KExp : Exp -> kitem
 | KExps : list   Exp -> kitem
 | KFrame : Frame -> kitem
 | KFrames : list   Frame -> kitem
 | KId :  string   -> kitem
 | KIds :  list   string   -> kitem
 | KInt :  Z -> kitem
 | KMap : Map k k -> kitem
 | KPgm : list  Defn -> kitem
 | KStmt : Stmt -> kitem
 | KStruct :  sval  -> kitem
 | KUndef : Undef -> kitem
 | KFreeze :  kitem  -> kitem
with KResult :  Set :=
   Bool :  bool  -> KResult
 | Int   :  Z -> KResult
 | VStruct :  sval  -> KResult
 | VUndef : Undef -> KResult
with Stmt : Set  :=
   HAssign :  Exp -> Exp -> Stmt
 | SAssign  :  string   -> Exp -> Stmt
 | Seq :  Stmt -> Stmt -> Stmt
 | HDealloc :  Exp -> Stmt
 | Decl : string -> Stmt
 | SIf : Exp -> Stmt -> Stmt -> Stmt
 | Jump : Exp -> Stmt
 | SReturnVoid : Stmt
 | SReturn : Exp -> Stmt
 | SCall : string -> list Exp -> Stmt
 | Skip : Stmt
 | SWhile : Exp -> Stmt -> Stmt
with sval : Set :=
   Struct : Map k k -> sval
with Undef : Set :=
   undef : Undef
with k : Set := kdot | kra : kitem -> k -> k
 .

Definition KResultToExp (x : KResult) : Exp :=
 match x with
 | Bool i => BCon i
 | Int i => ECon i
 | VStruct i => EValStruct i
 | VUndef i => ExpUndef i
 end.
Definition KResultFromExp (x : Exp) : option KResult :=
 match x with
 | BCon i => Some (Bool i)
 | ECon i => Some (Int i)
 | EValStruct i => Some (VStruct i)
 | ExpUndef i => Some (VUndef i)
 | _ => None
 end.
Definition KResultToK (x : KResult) : kitem :=
 match x with
 | Bool i => KBool i
 | Int i => KInt i
 | VStruct i => KStruct i
 | VUndef i => KUndef i
```

```
  end.
Definition KResultFromK (x : kitem) : option KResult :=
  match x with
  | KBool i => Some (Bool i)
  | KInt i => Some (Int i)
  | KStruct i => Some (VStruct i)
  | KUndef i => Some (VUndef i)
  | _ => None
  end.
Definition ExpToK (x : Exp) : kitem :=
  match x with
  | BCon i => KBool i
  | EVar i => KId i
  | ECon i => KInt i
  | EValStruct i => KStruct i
  | ExpUndef i => KUndef i
  | _ => KExp x
  end.
Definition ExpFromK (x : kitem) : option Exp :=
  match x with
  | KBool i => Some (BCon i)
  | KExp i => Some i
  | KId i => Some (EVar i)
  | KInt i => Some (ECon i)
  | KStruct i => Some (EValStruct i)
  | KUndef i => Some (ExpUndef i)
  | _ => None
  end.

Require Export himp_syntax.

(* Notations and coercions to make syntax nicer *)
Coercion EVar : string >-> Exp.
Coercion ECon : Z >-> Exp.
Coercion BCon : bool >-> Exp.
Coercion ExpToK : Exp >-> kitem.
Coercion KDefn : Defn >-> kitem.

Delimit Scope code with code.

Notation "{{␣s1␣;␣..␣;␣sn␣}}" :=
  (Seq s1%code .. (Seq sn%code Skip) ..) (at level 5) : code.
Notation "t␣<-␣e" := (SAssign t e)
  (at level 100, e at level 200, no associativity) : code.
Notation "e1␣<<-␣e2" := (HAssign e1 e2)
  (at level 100, e2 at level 200, no associativity) : code.
Infix "+" := EPlus : code.
Infix "=" := BEq : code.
Notation "x␣<>␣y" := (BNot (BEq (x : Exp) (y : Exp))) : code.
Infix "&&" := BAnd : code.
Infix "||" := BOr : code.
Notation "v␣->>␣f" := (EProject (ELoad (v : Exp)) f)
  (at level 10, no associativity) : code.

Bind Scope code with Exp.
Bind Scope code with Stmt.

Arguments SWhile (c b)%code.
Arguments SIf (c t e)%code.
Arguments SReturn e%code.
Arguments FunDef name%string args%list body%code.
```

B.4 List Example Proofs

The rest of this document gives proofs of the example programs. Each file has a rigid style, first defining the source code, then the specification, and then giving the proof.

The code is sometimes split across several definitions. This is mostly done in Lambda, which lacks an equivalence of the convenient loop abbreviations of, to name subterms for easy reference in the specifications. In some of the larger HIMP programs this is done for performance reason, to allow the definitions to stay folded until needed.

The specification is writing as an inductively defined predicate with one constructor per claim, using the specification abbreviations. It is preceded by the definition of auxiliary functions, if any are needed. Most specifications use only list functions from Coq's standard library. The exceptions are all three delete examples, the length programs for Stack and HIMP, and the sum program of lambda.

The proof generally consists of simply asserting that the specification holds in the language. The text of the proof is written between `Proof.` and `Qed.`, and is just a single invocation of the automatic proof tactic. If any auxiliary lemmas were required they are proved immediately before the final correctness lemma.

B.4.1 Examples in Stack

Head

```
Require Import list_examples.

Definition head_code := Load::Ret::nil.

Inductive head_spec : Spec cfg :=
  head_claim : forall H x l, fun_claim head_spec
  "head" head_code
    1 (fun p => asP H (rep_list (x::l) p))
    1 (fun p => constraint (p = x) :* litP H).

Lemma head_proof : sound stack_step head_spec.
Proof. list_solver. Qed.
```

Tail

```
Require Import list_examples.

Definition tail_code := Plus1++Load::Ret::nil.

Inductive tail_spec : Spec cfg :=
  tail_claim  :  forall H x y l,  fun_claim tail_spec
  "tail"  (Plus1++Load::Ret::nil)
    1 (fun p => asP H (rep_seg (x::nil)  y  p :*  rep_list  l  y))
    1 (fun p => constraint (p = y) :*  litP  H).

Lemma tail_proof : sound stack_step tail_spec.
Proof. list_solver.    Qed.
```

Add

```
Require Import list_examples.

Definition add_code :=
  Alloc  2::Swap::Over::Store::Swap::Over::Plus1++Store::Ret::nil.

Inductive add_spec : Spec cfg  :=
  add_claim :  forall x H p l,  fun_claim add_spec
  "add" add_code
    2 (fun a b => constraint (a = x) :*  constraint  (b = p)
      :*  asP H (rep_list  l  b))
    1 (fun r  => rep_seg (x::nil)  p r  :*  litP  H).

Lemma add_proof : sound stack_step add_spec.
Proof. list_solver.    Qed.
```

Swap

```
Require Import list_examples.

Definition swap_code :=
  Dup 0::Load::Over::Plus1++Load::Dup 0::Load
  ::Roll  2::Store::Over::Store::Ret::nil.

Inductive swap_spec : Spec cfg  :=
  swap_claim :  forall x y t,  fun_claim swap_spec
  "swap" swap_code
    1 (rep_seg  (x::y::nil)    t)  1 (rep_seg  (y::x::nil)    t).

Lemma swap_proof : sound stack_step swap_spec.
```

```
Proof. list_solver.   Qed.
```

Dealloc

```
Require Import list_examples.

Definition dealloc_code :=
  While (Dup 0::nil)   (Dup 0::Plus1++Swap::Dealloc
                       ::Dup 0::Load::Swap::Dealloc::nil)
  ::Drop::Ret::nil.

Inductive dealloc_spec : Spec cfg :=
  dealloc_claim  : forall l, fun_claim dealloc_spec
    "dealloc"  dealloc_code 1 (rep_list  l)  0 emptyP
 |loop_claim  : forall l, loop_claim dealloc_spec
              0 dealloc_code 1 (rep_list  l)  0 emptyP.

Lemma dealloc_proof : sound stack_step dealloc_spec.
Proof. list_solver.   Qed.
```

Length Recursive

```
Require Import list_examples.

Definition length_code :=
 Dup 0::If  (Plus1++Load::Call "length"::Plus1)  nil::Ret::nil.

Fixpoint zlength  {A} (l : list  A) :=
  match l with
    | nil  => 0
    | _ :: l'  => 1 + zlength l'
  end.
Inductive length_spec :  Spec cfg  :=
  length_claim  :  forall H l, fun_claim length_spec
  "length"  length_code
    1 (fun p => asP H (rep_list l  p))
    1 (fun n => constraint (n = zlength l)  :* litP  H).

Lemma length_proof : sound stack_step length_spec.
```

```
Proof. list_solver.   Qed.
```

Length Iterative

```
Require Import list_examples.

Definition length_code :=
  Push 0::Swap::While (Dup 0::nil)
   (Swap::Plus1++Swap::Plus1++Load::nil)::Drop::Ret::nil.

Fixpoint zlength {A} (l : list A) :=
  match l with
    | nil  => 0
    | _ :: l'  => (1 + zlength l')
  end.
Inductive length_spec : Spec cfg :=
  length_claim : forall H l, fun_claim length_spec
  "length" length_code
    1 (fun p => asP H (rep_list l p))
    1 (fun n => constraint (n = zlength l) :* litP H)
 |loop_claim : forall H l n, loop_claim length_spec
  0 length_code
    2 (fun p k => constraint (k = n) :* asP H (rep_list l p))
    1 (fun k' => constraint (k' = n + zlength l) :* litP H).

Lemma length_proof : sound stack_step length_spec.
Proof. list_solver.   Qed.
```

Sum Recursive

```
Require Import list_examples.

Definition sum_code :=
  Dup 0::If  (Dup 0::Load::Swap::Plus1++Load
           ::Call  "sum"::Binop Zplus::nil)   nil
  ::Ret::nil.

Definition sum := fold_right Zplus 0.
Inductive sum_spec : Spec cfg :=
  sum_claim : forall H l, fun_claim sum_spec
  "sum" sum_code
    1 (fun p => asP H (rep_list l  p))
    1 (fun n => constraint (n = sum l) :* litP  H).

Lemma sum_proof : sound stack_step sum_spec.
Proof. list_solver.   Qed.
```

Sum Iterative

```
Require Import list_examples.

Definition sum_code :=
  Push 0::Swap::While (Dup 0::nil)
    (Swap::Over::Load::Binop  Zplus::Swap::Plus1++Load::nil)
  ::Drop::Ret::nil.

Definition sum := fold_right Zplus 0.
Inductive sum_spec : Spec cfg :=
  sum_claim : forall H l, fun_claim sum_spec
  "sum" sum_code
    1 (fun p => asP H (rep_list l  p))
    1 (fun n => constraint (n = sum l) :* litP  H)
 |loop_claim : forall H n l, loop_claim sum_spec
  0 sum_code
    2 (fun p k => constraint (k = n) :* asP H (rep_list  l  p))
    1 (fun k => constraint (k = n + sum l) :* litP  H).

Lemma sum_proof : sound stack_step sum_spec.
Proof. list_solver.   Qed.
```

Reverse

```
Require Import list_examples.

Definition reverse_code :=
  Push 0::While (Over::nil)
    (Dup 1::Plus1++Load::Swap::Dup 2::Plus1++Store::Swap::nil)
  ::Swap::Drop::Ret::nil.
```

```
Inductive reverse_spec : cfg -> (cfg -> Prop) -> Prop :=
  reverse_claim : forall l, fun_claim reverse_spec
  ”reverse” reverse_code
    1 (rep_list l) 1 (rep_list (rev l))
 |loop_claim : forall lx lp, loop_claim reverse_spec
  0 reverse_code
    2 (fun p x => rep_list lx x :* rep_list lp p)
    1 (rep_list (rev_append lx lp)).
```

```
Lemma reverse_proof : sound stack_step reverse_spec.
Proof. list_solver;
rewrite <- rev_alt in * |-;
list_run. Qed.
```

Append

```
Require Import list_examples.
```

```
Definition append_code :=
  Over::If (Over::While (Plus1++Dup 0::Load::Dup 0::nil) Nip
  ::Drop::Store::nil) Nip::Ret::nil.
```

```
Inductive append_spec : cfg -> (cfg -> Prop) -> Prop :=
  | append_claim : forall lx ly, fun_claim append_spec
    ”append” append_code
      2 (fun y x => rep_list lx x :* rep_list ly y)
      1 (rep_list (lx++ly))
  | loop_claim : forall lx a lp ly, loop_claim append_spec
    0 append_code
      3 (fun p y x => constraint (p <> 0) :*
         rep_list ly y :* rep_seg lx p x :* p |-> a :*
         existsP p’, p + 1 |-> p’ :* rep_list lp p’)
      1 (rep_list (lx++a::lp++ly)).
```

```
Lemma append_proof : sound stack_step append_spec.
Proof. list_solver;
```

```
rewrite  app_ass in  * |- *;
list_run.   Qed.
```

Copy

```
Require Import list_examples.

Definition init_head  :=
  Over::Load::Over::Store::Swap::Plus1++Load::nil.
Definition copy_code := Dup 0::
  If  (Alloc  2::Swap::Over::init_head++
       While (Dup 0::nil)
          (Swap::Alloc  2::Dup 0::Rot++Plus1++Store::init_head)
     ::Swap::Plus1++Store::nil)    nil
  ::Ret::nil.

Inductive copy_spec : Spec cfg  :=
  copy_claim :  forall H l, fun_claim copy_spec
  ”copy” copy_code
    1 (fun p => asP H (rep_list l  p))
    1 (fun r  => rep_list l  r  :*  litP   H)
 |loop_claim  :  forall H l1 x v l,   loop_claim copy_spec
  0 copy_code
    3 (fun y q r  => constraint (q <> 0)
        :*  rep_seg l1  q r  :*  q |-> x :* q+1 |-> v
        :*  asP H (rep_list   l  y))
    1 (fun r  => rep_list (l1++x::l)  r  :*  litP   H).

Lemma copy_proof : sound stack_step copy_spec.
Proof.  list_solver;
rewrite  app_ass in  * |-;
list_run.   Qed.
```

Delete

```
Require Import list_examples.

Fixpoint  delete  v l  :=
  match l with
```

```
    | nil  => nil
    | (x :: l') => if Z.eqb x v
        then delete v l'
        else x :: delete v l'
  end.
Arguments delete v l : simpl nomatch.

Definition z_test_eq (a b : Z) := if Z.eqb a b then 1 else 0.

Definition delete_code :=
  Swap::While (Dup 0::If  (Dup 0::Load::Dup 2::Binop z_test_eq::nil)
                          (Dup 0::nil)::nil)
         (Dup 0::Dealloc::Plus1++Dup 0::Load::Swap::Dealloc::nil)
  ::Dup 0::If
    (Swap::Over::
     While (Dup 0::Plus1++Load::Dup 0::nil)
       (Dup 0::Load::Dup 3::Binop z_test_eq
        ::If  (Over::Plus1++Store::nil)  Nip::nil)
     ::Drop::Drop::Drop::nil)
    (Swap::Drop::nil)
  ::Ret::nil.

Inductive delete_spec : Spec cfg :=
  delete_claim : forall l v, fun_claim delete_spec
  "delete" delete_code
    2 (fun v' p => constraint (v'=v) :* rep_list l p)
    1 (rep_list (delete v l))
 |init_loop  : forall l v, loop_claim delete_spec
  0 delete_code
    2 (fun p v' => constraint (v'=v) :* rep_list l p)
    1 (rep_list (delete v l))
 |run_loop : forall A x v B , loop_claim delete_spec
  1 delete_code
    3 (fun p v' r => constraint (v'=v) :* rep_seg A p r
      :* constraint (p <> 0) :* p |-> x :* constraint (x <> v)
      :* existsP q, (p+1) |-> q :* rep_list B q)
    1 (rep_list (A++x::delete v B)).

Ltac eqb_solve := intros; match goal with [|- context [?a =? ?b]]
  => destruct (Z.eqb_spec a b);intuition congruence end.
Lemma z_test_eql : forall z v, z_test_eq z v = 0 <-> z <> v.
Proof. unfold z_test_eq;eqb_solve. Qed.
Lemma z_test_neq : forall z v, z_test_eq z v <> 0 <-> z = v.
Proof. unfold z_test_eq;eqb_solve. Qed.
Lemma delete_eq v x l : x = v -> delete v (x::l) = delete v l.
Proof. unfold delete;eqb_solve. Qed.
Lemma delete_ne v x l : x<>v -> delete v (x::l) = x::delete v l.
Proof. unfold delete;eqb_solve. Qed.

Create HintDb delete_rules discriminated.
Hint Rewrite z_test_eql z_test_neq app_ass : delete_rules.
Hint Rewrite delete_eq delete_ne using assumption : delete_rules.

Lemma delete_proof : sound stack_step delete_spec.
Proof. list_solver;repeat progress
  (autorewrite with delete_rules in * |- *;list_run). Qed.
```

B.4.2 Examples in Lambda

Head

```
Require Import list_examples.

Definition head_body := Car (Deref (Var 0)).

Inductive head_spec : Spec cfg :=
  head_claim : forall H x l r env, exp_val head_spec
  head_body (Addr r::env)
    (asP H (rep_list (x::l) (Addr r)))
    (fun v => constraint (v = x) :* litP H).

Lemma heap_proof : sound lam_step head_spec.
Proof. list_solver. Qed.
```

Tail

```
Require Import list_examples.

Definition tail_body := Cdr (Deref (Var 0)).

Inductive tail_spec : Spec cfg :=
  tail_claim : forall H r y x l env, exp_val tail_spec
  tail_body (Addr r::env)
    (asP H (rep_list l y :* list_seg (x::nil) y (Addr r)))
    (fun v => constraint (v = y) :* litP H).

Lemma tail_proof : sound lam_step tail_spec.
Proof. list_solver. Qed.
```

Add

```
Require Import list_examples.

Definition add_body := Ref (Cons (Var 1) (Var 0)).

Inductive add_spec : Spec cfg :=
  add_claim : forall x r env, exp_val add_spec
  add_body (r::x::env) emptyP (list_seg (x::nil) r).

Lemma add_proof : sound lam_step add_spec.
Proof. list_solver. Qed.
```

Swap

```
Require Import list_examples.

Definition swap_body :=
   Let (Deref (Var 0)) (Lam
  (Seq (Assign (Var 1) (Cons (Car (Deref (Cdr (Var 0))))
                              (Cdr (Var 0))))
  (Seq (Assign (Cdr (Var 0)) (Cons (Car (Var 0))
                                    (Cdr (Deref (Cdr (Var 0))))))
       (Var 1)))).
```

```
Inductive swap_spec : Spec cfg :=
  swap_claim : forall x y l  r  env,  exp_val swap_spec
  swap_body (r::env)
    (rep_list   (x::y::l)     r)
    (rep_list   (y::x::l)).

Lemma swap_proof : sound lam_step swap_spec.
Proof. list_solver.    Qed.
```

Dealloc

```
Require Import list_examples.

Definition dealloc_body :=
 (If  (Var 0) (Let (Deref (Var 0)) (Lam
   (Seq (Dealloc (Var 1))
         ((Var 2) (Cdr (Var 0))))))
   Nil).

Inductive dealloc_spec :  Spec cfg  :=
  dealloc_claim  :  forall l r  env,  exp_val dealloc_spec
  dealloc_body (r::fix_env    (Lam dealloc_body) env)
    (rep_list   l  r)  (fun v => constraint (v  = Nil)).

Lemma dealloc_proof : sound lam_step dealloc_spec.
Proof. list_solver.    Qed.
```

Length

```
Require Import list_examples.

Definition length_body :=
  If  (Var 0) (Inc  ((Var 1) (Cdr (Deref (Var 0)))))  (Num 0).

Inductive length_spec :  Spec cfg  :=
  length_claim  :  forall H l r  env,  exp_val length_spec
  length_body (r::fix_env    (Lam length_body) env)
    (asP H (rep_list   l  r))
    (fun v => constraint (v = Num (length l)) :* litP H).

Lemma length_proof : sound lam_step length_spec.
Proof. list_solver. Qed.
```

Sum

```
Require Import list_examples.

Definition sum_body :=
  If (Var 0) (Let (Deref (Var 0)) (Lam
    (Plus (Car (Var 0)) ((Var 2) (Cdr (Var 0))))))
    (Num 0).

Fixpoint num_list l := match l with
  | nil => True
  | Num _ :: l' => num_list l'
  | _ => False
  end.
Fixpoint nsum l := match l with
  | nil => 0
  | Num n :: l' => n + nsum l'
  | _ => 0
  end.

Inductive sum_spec : Spec cfg :=
  sum_claim : forall H l r env, exp_val sum_spec
  sum_body (r::fix_env (Lam sum_body) env)
    (asP H (rep_list l r) :* constraint (num_list l))
    (fun v => constraint (v = Num (nsum l)) :* litP H).

Lemma sum_proof : sound lam_step sum_spec.
Proof. list_solver. Qed.
```

Reverse

```
Require Import list_examples.

Definition rev_app_body :=
  (If  (Var 0)
    (Let (Deref (Var 0)) (Lam
      (Seq (Assign (Var 1) (Cons (Car (Var 0)) (Var 2)))
           ((Var 3) (Var 1) (Cdr (Var 0))))))
    (Var 1)).
Definition list_rev_app :=
  strict_fix   (Lam (Lam (Lam rev_app_body))).
Definition list_rev_body := list_rev_app Nil (Var 0).
Definition list_rev  := Lam list_rev_body.

Inductive reverse_spec : Spec cfg :=
  rev_claim : forall l pl env, exp_val reverse_spec
  list_rev_body (pl::env)
    (rep_list  l pl)
    (rep_list  (rev l))
 |rev_app_claim : forall k l pk pl env, exp_val reverse_spec
  rev_app_body (pl::pk::fix_env  (Lam (Lam rev_app_body)) env)
    (rep_list  k pk :* rep_list  l pl)
    (rep_list  (rev_append l k)).

Lemma reverse_proof : sound lam_step reverse_spec.
Proof. list_solver;
rewrite <- rev_alt in * |- ;
list_run.  Qed.
```

Append

```
Require Import list_examples.

Definition list_app_nonempty_body :=
 Let (Deref (Var 0)) (Lam
 (If  (Cdr (Var 0))
     ((Var 2) (Cdr (Var 0)))
     (Assign (Var 1) (Cons (Car (Var 0)) (Var 3))))).
Definition list_app_nonempty :=
  strict_fix   (Lam (Lam list_app_nonempty_body)).
Definition list_app_body :=
  If (Var 1) (Seq (list_app_nonempty (Var 1)) (Var 1)) (Var 0).
Definition list_app := Lam (Lam list_app_body).

Inductive append_spec : Spec cfg :=
  append_claim : forall lx ly px py env, exp_val append_spec
  list_app_body (py::px::env)
    (rep_list lx px :* rep_list ly py)
    (rep_list (lx++ly))
 |append_nonempty_claim : forall x lx px p ly py env,
  exp_val append_spec list_app_nonempty_body
      (Addr p::fix_env (Lam list_app_nonempty_body) (py::env))
    (p |-> Pair x px :* rep_list lx px :* rep_list ly py)
    (fun _ => rep_list(x::lx++ly) (Addr p)).

Lemma append_proof : sound lam_step append_spec.
Proof. list_solver.   Qed.
```

Copy

```
Require Import list_examples.

Definition copy_body :=
  (If (Var 0) (Let (Deref (Var 0)) (Lam
    (Ref (Cons (Car (Var 0)) ((Var 2) (Cdr (Var 0)))))))
  Nil).

Inductive copy_spec : Spec cfg :=
  copy_claim : forall H l pl env, exp_val copy_spec
  copy_body (pl::fix_env (Lam copy_body) env)
    (asP H (rep_list l pl))
    (fun r => rep_list l r :* litP H).

Lemma copy_proof : sound lam_step copy_spec.
Proof. list_solver.   Qed.
```

Delete

```
Require Import list_examples.
```

```
Definition delete_body :=
 (If  (Var 0)
  (Let (Deref (Var 0)) (Lam
  (If  (Eq (Var 3) (Car (Var 0)))
       (Seq (Dealloc (Var 1)) ((Var 2) (Cdr (Var 0))))
       (Seq (Assign (Var 1) (Cons (Car (Var 0))
                                  ((Var 2) (Cdr (Var 0)))))
            (Var 1)))))
  Nil).
```

```
Fixpoint delete v l :=
 match l with
   | nil  => nil
   | x :: l'  => if val_eq_dec v x then delete v l'
                                   else  x::delete  v l'
 end.
```

```
Inductive delete_spec : Spec cfg :=
 delete_claim  : forall v l pl env, exp_val delete_spec
   delete_body (pl::fix_env  (Lam delete_body) (v::env))
   (rep_list  l pl) (rep_list  (delete  v l)).
```

```
Lemma delete_proof : sound lam_step delete_spec.
Proof. list_solver.
simpl;destruct  (val_eq_dec v v0);list_run.   Qed.
```

B.4.3 Examples in HIMP

Head

```
Require Import list_examples.
```

```
Definition head_def := FunDef "head" ["x"]
 (SReturn ("x"->>"val")).
```

```
Inductive head_spec : Spec kcfg :=
 head_claim : forall p H x l, heap_fun head_spec
```

```
  head_def [Int  p]
    (asP H (rep_list  (x::l)  p))
    (fun r => constraint (r=x) :* litP  H).

Lemma head_proof : sound kstep head_spec.
Proof. list_solver.  Qed.
```

Tail

```
Require Import list_examples.

Definition tail_def  := FunDef ”tail”  [”x”]
  (SReturn (”x”−>>”next”)).

Inductive tail_spec  :  Spec kcfg  :=
  tail_claim  :  forall H p x y l,  heap_fun tail_spec
  tail_def  [Int  p]
    (asP H (rep_seg (x::nil)   y  p :*  rep_list  l  y))
    (fun r => constraint (r  = y) :*  litP  H).

Lemma tail_proof : sound kstep tail_spec.
Proof. list_solver.  Qed.
```

Add

```
Require Import list_examples.

Definition add_fun := FunDef ”add” [”v”;”x”]
 {{Decl ”y”;”y”<−EAlloc
  ;”y”<<−build_node ”v” ”x”
  ;SReturn  ”y”}}.

Inductive add_spec : Spec kcfg  :=
  add_claim :  forall v H x l,  heap_fun add_spec
  add_fun [Int  v;Int  x]
    (asP H (rep_list  l  x))
    (fun r => rep_seg (v::nil)  x  r  :*  litP  H).

Lemma add_proof : sound kstep add_spec.
```

```
Proof. list_solver.   Qed.
```

Swap

```
Require Import list_examples.

Definition swap_def := FunDef ”swap” [”x”]
 {{Decl ”p”;  ”p” <− ”x”
  ;”x”  <− arr_next ”x”
  ;”p”<<−build_node (arr_val ”p”) (arr_next ”x”)
  ;”x”<<−build_node (arr_val ”x”) ”p”
  ;SReturn  ”x”}}.

Inductive swap_spec : Spec kcfg  :=
  swap_claim :  forall a b l  xv,  heap_fun swap_spec
  swap_def [Int  xv]
     (rep_list   (a::b::l)     xv)
     (rep_list   (b::a::l)).

Lemma swap_proof : sound kstep swap_spec.
Proof. list_solver.   Qed.
```

Dealloc

```
Require Import list_examples.

Definition dealloc_def  := FunDef ”dealloc”  [”x”]
 {{Decl ”y”
  ;SWhile (”x”  <> 0)
     {{”y”<−arr_next ”x”;HDealloc ”x”;”x”<−”y”}}
  ;SReturn  0}}.

Inductive dealloc_spec  :  Spec kcfg  :=
  dealloc_claim   :  forall l x,  heap_fun dealloc_spec
  dealloc_def  [Int   x]  (rep_list    l  x)  (fun r  => emptyP)
 |loop_claim    :  forall l xv  yv,  heap_loop dealloc_spec
  dealloc_def  0 (”x”  s|−> KInt xv :*  ”y”  s|−> yv)
     (rep_list   l  xv)  (fun r  => emptyP).
```

```
Lemma dealloc_proof : sound kstep dealloc_spec.
Proof. list_solver.   Qed.
```

Length Recursive

```
Require Import list_examples.

Definition length_def := FunDef "length" ["x"]
 (SIf ("x" = 0)
   (SReturn 0)
   (SReturn (1 + ECall "length" [arr_next "x"]))).

Inductive length_spec : Spec kcfg :=
  length_claim : forall H l x, heap_fun length_spec
  length_def [Int x]
    (asP H (rep_list l x))
    (fun r => constraint (r = zlength l) :* litP H).

Lemma length_proof : sound kstep length_spec.
Proof. list_solver.   Qed.
```

Length Iterative

```
Require Import list_examples.

Definition length_def := FunDef "length" ["x"]
 {{Decl "l";"l"<-0
  ;SWhile ("x"<>0) {{"l"<-"l"+1; "x"<- arr_next "x"}}
  ;SReturn "l"}}.

Inductive length_spec : Spec kcfg :=
  length_claim : forall H x l, heap_fun length_spec
  length_def [Int x]
    (asP H (rep_list l x))
    (fun r => constraint (r = zlength l) :* litP H)
 |loop_claim : forall H k x l, heap_loop length_spec
  length_def 0 ("x" s|-> KInt x :* "l" s|-> KInt k)
    (asP H (rep_list l x))
    (fun r => constraint (r = k + zlength l) :* litP H).
```

```
Lemma length_proof : sound kstep length_spec.
Proof. list_solver.   Qed.
```

Sum Recursive

```
Require Import list_examples.

Definition sum_def := FunDef ”sum_recursive” [”x”]
 (SIf (”x”=0)
   (SReturn 0)
   (SReturn (arr_val ”x”
            + ECall ”sum_recursive” [arr_next ”x”]))).

Inductive sum_spec : Spec kcfg :=
  sum_claim : forall H l x, heap_fun sum_spec
  sum_def [Int x]
    (asP H (rep_list l x))
    (fun r => constraint (r = sum l) :* litP H).

Lemma sum_proof : sound kstep sum_spec.
Proof. list_solver.   Qed.
```

Sum Iterative

```
Require Import list_examples.

Definition sum_def := FunDef ”sum” [”x”]
 {{Decl ”s”;”s”<−0
  ;SWhile (”x”<>0) {{”s”<−”s”+arr_val ”x”; ”x”<−arr_next ”x”}}
  ;SReturn ”s”}}.

Inductive sum_spec : Spec kcfg :=
  sum_claim : forall H l x, heap_fun sum_spec
  sum_def [Int x]
    (asP H (rep_list l x))
    (fun r => constraint (r = sum l) :* litP H)
 |loop_claim : forall k H l x, heap_loop sum_spec
  sum_def 0 (”s” s|−> KInt k :* ”x” s|−> KInt x)
```

```
    (asP H (rep_list  l  x))
    (fun r => constraint (r = k + sum l) :* litP  H).

Lemma sum_proof : sound kstep sum_spec.
Proof. list_solver.   Qed.
```

Reverse

```
Require Import list_examples.

Definition reverse_def := FunDef ”reverse” [”x”]
 {{Decl ”p”;Decl ”y”; ”p”<−0
  ;SWhile (”x”<>0)
    {{”y”<−arr_next ”x”
     ;”x”<<−build_node (arr_val ”x”) (EVar ”p”)
     ;”p”<−”x”
     ;”x”<−”y”}}
  ;SReturn ”p”}}.

Inductive reverse_spec : Spec kcfg :=
  reverse_claim : forall l x, heap_fun reverse_spec
  reverse_def [Int x] (rep_list  l x) (rep_list  (rev l))
 |loop_claim : forall A B x p v, heap_loop reverse_spec
  reverse_def 0 (”x” s|−> KInt x :* ”p” s|−> KInt p :* ”y” s|−> v)
    (rep_list  A x :* rep_list  B p) (rep_list  (rev_append A B)).

Lemma reverse_proof : sound kstep reverse_spec.
Proof. list_solver.
rewrite <− rev_alt in * |− .
list_run.  Qed.
```

Append

```
Require Import list_examples.

Definition append_def := FunDef ”append” [”x”;”y”]
  (SIf (”x” = 0)
    (SReturn ”y”)
    {{Decl ”p”;”p” <− ”x”
```

```
      ;SWhile ("p"->>"next" <> 0) ("p"<- "p"->>"next")
      ;"p" <<- build_node ("p"->>"val") "y"
      ;SReturn "x" }}).

Inductive append_spec : Spec kcfg :=
  append_claim : forall lx x ly y, heap_fun append_spec
  append_def [Int x; Int y]
    (rep_list lx x :* rep_list ly y)
    (fun r => rep_list (lx++ly) r)
 |loop_claim : forall lx x ly y lp p, p <> 0 ->
  heap_loop append_spec append_def 0
  ("x" s|-> KInt x :* "y" s|-> KInt y :* "p" s|-> KInt p)
    (rep_seg lx p x :* rep_list lp p :* rep_list ly y)
    (rep_list (lx++lp++ly)).

Lemma append_proof : sound kstep append_spec.
Proof. list_solver.
rewrite app_ass in * |- *.
list_run. Qed.
```

Copy

```
Require Import list_examples.

(* Splitting the code into several definitions
   reduces proof time by reducing the size of
   terms at any single point in the proof *)
Definition copy_loop :=
  SWhile ("iterx"<>0)
   {{"node"<-EAlloc
     ;"node"<<-build_node (arr_val "iterx") 0
     ;"itery"<<-build_node (arr_val "itery") "node"
     ;"iterx"<-arr_next "iterx"
     ;"itery"<-arr_next "itery"
     }}.
Definition copy_tail :=
 {{"y"<-EAlloc;"y"<<-build_node (arr_val "x") 0
```

```
  ;"iterx"<−arr_next "x"
  ;"itery"<−"y"
  ;copy_loop
  ;SReturn "y"}}%code.
Definition copy_def := FunDef "copy" ["x"]
 {{Decl "y";Decl "iterx";Decl "itery";Decl "node"
  ;SIf ("x"=0) (SReturn 0) copy_tail}}.

Inductive copy_spec : Spec kcfg :=
  copy_claim : forall H l x, heap_fun copy_spec
  copy_def [Int x]
    (asP H (rep_list l x))
    (fun r => rep_list l r :* litP H)
 |loop_claim : forall x n A y v itery H B iterx,
  heap_loop copy_spec copy_def 0
       ("x" s|−> x :* "node" s|−> n :* "y" s|−> KInt y
       :* "iterx" s|−> KInt iterx :* "itery" s|−> KInt itery)
    (constraint (itery <> 0) :* rep_seg A itery y
     :* itery h|−> list_node v 0 :* asP H (rep_list B iterx))
    (fun r => rep_list (A ++ v :: B) r :* litP H).

Lemma copy_proof : sound kstep copy_spec.
Proof. list_solver.
rewrite app_ass in * |− .
list_run. Qed.
```

Delete

```
Require Import list_examples.

(* Splitting the code into several definitions
   reduces proof time by reducing the size of
   terms at any single point in the proof *)
Definition delete_loop2 :=
  (SWhile (arr_next "y"<>0)
  (Seq ("z"<−arr_next "y")
        (SIf (arr_val "z"="v")
            (Seq ("y"<<-build_node (arr_val "y") (arr_next "z"))
                  (HDealloc "z"))
            ("y"<-"z"))))%code.
Definition delete_tail1  :=
 {{SIf ("x"=0) (SReturn 0) Skip
  ;Decl "z"
  ;"y"<-"x"
  ;delete_loop2
  ;SReturn "x"}}%code.
Definition delete_def := FunDef "delete" ["x";"v"]
 (Seq (Decl "y")
 (Seq (SWhile (("x"<>0) && (arr_val "x"="v"))
   {{"y"<-arr_next "x";HDealloc "x";"x"<-"y"}})
   delete_tail1))%code.

Fixpoint delete (v : Z) (l : list Z) : list Z :=
  match l with
    | nil => nil
    | (x :: l') => if Z_eq_dec v x then delete v l'
                   else x :: delete v l'
  end.
Arguments delete v l : simpl nomatch.

Inductive delete_spec : Spec kcfg :=
  delete_claim : forall l x v, heap_fun delete_spec
  delete_def [Int x;Int v] (rep_list l x) (rep_list (delete v l))
 |loop1_claim : forall l x v y, heap_loop delete_spec
  delete_def 0 ("x" s|-> KInt x :* "v" s|-> KInt v :* "y" s|-> y)
    (rep_list l x) (rep_list (delete v l))
 |loop2_claim : forall v A x yv B y z, heap_loop delete_spec
  delete_def 1 ("x" s|-> KInt x :* "v" s|-> KInt v
               :* "y" s|-> KInt y :* "z" s|-> z)
    (rep_seg A y x :* constraint (y <> 0) :* existsP q,
     y h|-> list_node yv q :* constraint (yv <> v) :* rep_list B q)
    (rep_list (A++yv::delete v B)).
```

```
Lemma delete_eq : forall v l, delete v (v :: l) = delete v l.
Proof. unfold delete;intros;destruct (Z.eq_dec v v);congruence. Qed.
Lemma delete_ne : forall v z l, v<>z -> delete v (z::l) = z::delete v l.
Proof. unfold delete;intros;destruct (Z.eq_dec v z);congruence. Qed.

Lemma delete_proof : sound kstep delete_spec.
Proof. list_solver;
rewrite ?delete_eq, ?delete_ne by congruence;
try rewrite app_ass in * |-;
list_run. Qed.
```

Appendix C

Sorts in K Semantics

In this appendix we review definitions of *order-sorted algebra*, which is the inspiration for the user-visible 𝕂 system. The foundational semantics of 𝕂 definitions is based on unsorted terms, but the specification language includes a sort system used for parsing and overloading which is heavily inspired by order-sorted algebra.

One aim of the translation tool is to provide an option to attempt to translate different sorts of a 𝕂 definition into different types in Coq, to resemble the user's view of the languages as closely as possible. The chief difficulty here is mimicking the subsorting allowed in order-sorted algebra. Where then is only one path between a pair of sorts in the subsorting relationship this can be translated simply into a wrapper constructor injecting one type into another, but when there are multiple routes to inject one sort into another a more complicated translation was needed. This generated constructors for one preferred path, and used functions to assist in implementing other injections.

C.1 Order-Sorted Algebra

The terms of a 𝕂definition can be understood as an instance of order-sorted algebra, which we review here, following the presentation of Gougen in [GM92]. In general terms, an *order-sorted algebra* consists of a collection of values and operations over those values which are described by a certain *order-sorted signature*. Basic concerns in the study of order-sorted algebras include finding algebras corresponding to a signature, and studying the relationship between different algebras described by a signature, or the algebras of related signatures.

Order-sorted algebra is the basis of a number of programming and modelling languages, such as Maude [Cla+03] and the OBJ [Gog+88] family.

Order-sorted algebra are a strict generalization of multi-sorted algebra, which corresponds more directly to primitive data-type definitions in more languages, especially proof assistants and functional programming languages. In the task of translating a K definition to a formal definition in a theorem prover, a large portion of the work to generate a definition of terms and configurations of the semantics consists of translating an order-sorted definition into a mutli-sorted definition. Now we recall the formal definitions of order-sorted and multi-sorted algebras, beginnning with multi-sorted algebras.

C.1.1 Multi-Sorted Algebra

Definition C.1. A multisorted signature Σ is a tuple (S, F, w) consisting of

- A set S, whose elements are called *Sorts*.
- A set F, whose elements are called *function symbols*
- A function $w : F \rightarrow S \times S^*$ assigning to each function symbol f a term (r, as) in $S \times S^*$ which is called the *arity* of f, with r and as respectively called the *result* and *arguments* of f (or of the arity of f). The variable σ is conventinally used for arities. An arity $(r, (a_1, \ldots, a_n))$ may also be written $a_1 \star \cdots \star a_n \rightarrow r$, giving $\rightarrow r$ when the arguments of the arity is the empty sequence. Referring to a function symbol as $f : \sigma$ asserts that f has arity σ.

Definition C.2. A *multi-sorted algebra* over a signature Σ, also known as a Σ-algebra, is a tuple (D, ρ) giving an interpretation of the sorts and functions of the signature.

- D is an S-indexed family of sets (not necessarily disjoin)
- ρ assigns to each f F an appropriate function – when f has result r and arguments $a_1, \ldots, a_n$, $\rho(f)$ is a function $D_{a_1} \times \cdots \times D_{a_n} \rightarrow D_r$.

A *(closed) term* over a multi-sorted signature Σ of sort s is inductively defined as a tree with node labels from F, where the root label f has result s, and the sequence $t_1, \ldots, t_n$ of subtrees has the same length (possibly empty) as the arguments $a_1, \ldots, a_n$, and child t_i is a term of sort a_i. The s-indexed set of closed terms over signature Σ is denoted by T_Σ.

A collection Var of *variables* for a signature Σ is an S-indexed family of (countably) infinite and pairwise disjoint sets. An element of X_s in a family of variables X is called a variable of sort s. We write $x : s$ for some element of Var_s.

An *open term* over a multi-sorted signature Σ is defined similarly, but also admits any variable of sort s as an open term of sort s. The s-indexed set of open terms over signature Σ and variables X is denoted $T_\Sigma(X)$.

Given any Σ-algebra, ρ extends to closed terms in the obvious way, sending a tree with root label f and subtrees $t_1, \ldots, t_n$ to $\rho(f)(\rho(t_1), \ldots, \rho(t_n))$. Interpreting an open term over a set X of variables requires an environment sending each variable to a value in the appropriate domain. Formally, an environment $\Gamma : X \to D$ is an s-indexed function sending any $x : s \in X_s$ to some value $\Gamma(x : s) \in D_s$. Then we define $\rho^\dagger(\Gamma) : T_\Sigma \to D$ which sends any variable $x : s$ to $\Gamma(x : s)$ and handles a function symbol with ρ as for closed terms.

Definition C.3. A multi-sorted signature with equations, $\Sigma_= = (S, F, w, Q)$, extends a mutli-sorted signature Σ with an additional component, a set of equations Q. An equation is a pair $a =_s b$ of two open terms over Σ, both of sort s.

Definition C.4. A $\Sigma_=$-algbra is a Σ-algebra which additionally satisfies all the equations - for any $a =_s b \in Q$ and any valuation Γ, $\rho^\dagger(\Gamma)(a) = \rho^\dagger(\Gamma)(b)$ in set D_s.

C.1.2 Order-Sorted Algebra

Order-sorted algebra generalizes multi-sorted algebra by allowing a subsorting relationship between sorts, and overloading function symbols with multiple related arities. For example, we might have a sort Pos for positive integers, declared to a be a subsort of a sort Int for any integers, and declare a function plus to have multiple arities $\mathsf{Int} \star \mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Pos} \star \mathsf{Pos} \to \mathsf{Pos}$.

Definition C.5. An *order-sorted signature* is a tuple $\Sigma = (S, R, F, w)$ where

- S is again a set of *sorts*.
- R is a preorder over S. We write $s_1 \leq s_2$ to mean $(s_1, s_2) \in R$ when an order-sorted signature R is understood. This extends to sequences,

defining $\vec{a} \leq \vec{b}$ to hold iff sequences a and b have the same length and $a_i \leq b_i$ for corresponding entries.

- F is again a set of *function symbols*.
- w assigns arities to function symbols. Generalizing the multisorted case, w may assign multiple arities. Formally, w is a relation from F to $(S \times S^*)$ which includes at least one entry for each f in F. In the order-sorted case, writing $f : \sigma$ indictate that σ is one of the arities assigned to f by w. If w assigns multiple arities to a function symbol f, those arities must obey some simple coherence conditions: If $f : \vec{a} \rightarrow r_1$ and $f : \vec{b} \rightarrow r_2$, then the arguments $\vec{a}$ and $\vec{b}$ must have the same length, and additionally if $\vec{a} \leq \vec{b}$ then $r_1 \leq r_2$.

An order-sorted term is an inductively defined tree over labels, where the root function symbol in a tree of sort s must have some signature $f : \vec{a} \rightarrow s'$ such that the subtrees are valid terms of sorts $\vec{a}$.

An open term additionally allow a base case for a term of sort s to consist of a variable $x : s'$ with $s' \leq s$.

Allowing a choice of signature in the definition of a term suggests that a given tree of labels may be assigned several sorts. This is indeed possible

Example C.1. Consider a signature with

- Four sorts, A, B, X, Y, with $A \leq B$ the only subsort relationship
- Nullary function symbols a, b with arities $a :\rightarrow A$ and $b :\rightarrow B$, and a binary function symbol f with two arities $f : A \star B \rightarrow X$ and $f : B \star A \rightarrow Y$.

The only requirement for this to be a well-formed signature is that the arities for f are coherent, which holds because vectors AB and BA are unrelated in the subsort relation. In this signature, the term $f(a, a)$ can be assigned sort X or Y.

To avoid this possibility, some further conditions may be imposed which suffice to guarantee that any valid term has a unique least sort which it can be validly assigned.

Definition C.6. An arity $\vec{a} \to r$ is *applicable* to a sequence $\vec{b}$ of argument sorts if $\vec{b} \le \vec{a}$, and a function symbol is *applicable* to a sequence of argument sorts if any of its arities are applicable.

A function symbol f is *regular* if given argument sorts $\vec{a}$ to which it is applicable there is some least arity $f : \vec{b} \to s$ of f which is applicable, least in the sense that if $\vec{c} \to r'$ is any other applicable arity then $\vec{b} \le \vec{c}$, and thus by coherence $r \le r'$.

A function symbol f is *pre-regular* if given argument sorts $\vec{a}$ to which it is applicable there is some applicable arity $f : \vec{b} \to s$ such that any other applicable arity $f : \vec{c} \to s'$ has $s \le s'$.

It is proven in Gougen [GM92] that if every function symbol in a singature is at least pre-regular then any valid term has a unique least sort, and that any pre-regular function symbol can be made regular by merely declaring additional arities, without changing the set of terms of each sort admitted by the signature.

Continuing Example C.1 we see that f is not pre-regular, a symbol g with arities $g : AB \to X$ and $g : BA \to X$ would be pre-regular but not regular (consider AA to see that regularity fails), and a symbol h with artities $h : AB \to X$ and $h : BA \to X$ but also $AA \to X$ would indeed be regular.

There are two ways to define an algebra over an order-sorted signature, one which is simpler and one which allows a more flexible interpretation of subsorting.

Definition C.7 (Order-sorted Algebra, by inclusion)**.** An algebra by inclusion for a signature Σ consists of a set D which will represent all terms and a sort-indexed family of subsets of D, $\{D_s\}_{s \in S}$ such that $s \le s'$ implies $D_s \subseteq D_{s'}$. [1] For each function symbol f there is an associated partial function $F_f : D^k \rightharpoonup D$ for appropriate k. For any declared arity $f : \vec{a} \to s$, the function F_f is total on the set $D_{a_1} \times \cdots \times D_{a_k}$, with image contained in D_s.

An alternate definition does not require interpreting subsorting by set inclusion. This is useful when modeling order-sorted algebras in programming languages or theorem provers without a native notion of inclusion or subsorting between separate types.

[1] If the sort graph has multiple (undirected) connected components, this definition can be slightly refined to replace the single D with independent domain sets for each connected component.

Definition C.8 (Order-sorted Algebra, by injection)**.** An algebra by injection for a signature Σ consists of a sort-indexed family of sets $\{D_s\}_{s \in S}$, and a collection of injective functions $i_s^{s'}$ for each $(s, s') \in R$, such that $a \leq b \leq c$ implies $i_b^c(i_a^b(x)) = i_a^c(x)$ for any $x \in D_a$. For each arity $f : \vec{a} \to s$ there is a function $F_{f:\vec{a}\to s} : D_{a_1} \times \cdots \times D_{a_k} \to D_s$. These functions are subject to the coherence condition that if $f : \vec{b} \to r_1$ and $f : \vec{c} \to r_2$ are two arities of a function symbol f with and there are some $\vec{a}$ and s with $\vec{a} \leq \vec{b}$, $\vec{a} \leq \vec{c}$ and $r_1 \leq s$, $r_2 \leq s$, then for any $\vec{x} \in D_{\vec{a}}$,

$$i_{r_1}^s(F_{f:\vec{b}\to r_1}(i_{a_1}^{b_1}(x_1), \ldots, i_{a_k}^{b_k}(x_k))) = i_{r_2}^s(F_{f:\vec{c}\to r_2}(i_{a_1}^{c_1}(x_1), \ldots, i_{a_k}^{c_k}(x_k)))$$

These definitions are equivalent

Any algebra according to Definition C.7 induces an algebra according to Definition C.8 simply by letting the collection $\{D_s\}_{s \in S}$ of subsets of the universe serve as the domains of the models, implementing the injections by inclusions, and letting each signature-specific function $F_{f:\vec{a}\to s}$ be the restriction of F_f to the appropriate domains.

The converse direction requires a quotient construction to construct a suitable universe from an algebra according to Definition C.8. Let D be the disjoint union of the domains, $\{(s, x) \mid s \in S, x \in D_S\}$, quotiented by the equivalence which relates (s, x) and (s', y) if $i_s^{s'}(x) = y$ or $i_{s'}^s(y) = x$. The unsorted interpreteation F_f of a function symbol is a partial function defined on any tuple of arguments falling under any signature $f : \vec{a} \to r$, with result given by $F_{f:\vec{a}\to r}$. The coherence conditions in Definition C.8 ensure that the result is unique, and the definition respects equivalence classes.

C.1.3 Multi-Sorted Encoding

An order-sorted signature can be faithfully encoded into a multi-sorted theory with equations, using function symbols and relations corresponding to the injection-based definition of an algebra in Definition C.8.

The resulting singature has

- The same set S of sorts as the order-sorted signature.
- A set of function symbols with two families of operations. There is one function symbol i_a^b for each $(a, b) \in R$, and one *symbol* $f^{\vec{a}\to s}$ for each *arity* $f : \vec{a} \to s$ in the order-sorted signature.

- A function symbols i_a^b has arity $a \to b$, a function symbol $f^{\vec{a} \to s}$ has arity $\vec{a} \to s$.

- For each $a \le b \le c$ there is an equation $i_b^c(i_a^b(x : a)) =_c i_a^c(x : a)$, and for each function symbol f in the order-sorted signature a family of conditions as above: Given arities $f : \vec{b} \to r_1$, $f : \vec{c} \to r_2$, and a sort sequence $\vec{a}$ and sort s such that $\vec{a} \le \vec{b}, \vec{a} \le \vec{c}$ and $r_1 \le s$, $r_2 \le s$, there is an equation

$$\begin{aligned} & i_{r_1}^s(F_{f:\vec{b}\to r_1}(i_{a_1}^{b_1}(x_1 : a_1), \ldots, i_{a_k}^{b_k}(x_k : a_k))) \\ & \qquad =_s i_{r_2}^s(F_{f:\vec{c}\to r_2}(i_{a_1}^{c_1}(x_1 : a_1), \ldots, i_{a_k}^{c_k}(x_k : a_k))) \end{aligned}$$

Comparing the definitions, we see that a collection of sets and functions is a multi-sorted algebra according to Definition C.4 of the translated theory if and only if it is an order-sorted algebra according to Definition C.4 of the original order-sorted signature.