Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2011-050 November 15, 2011

Reasoning about Relaxed Programs

Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Reasoning about Relaxed Programs

Michael Carbin Deokhwan Kim Sasa Misailovic Martin C. Rinard
MIT CSAIL
{mcarbin, dkim, misailo, rinard}@csail.mit.edu

Abstract

Approximate program transformations such as task skipping [27, 28], loop perforation [20, 21, 32], multiple selectable implementations [3, 4, 15], approximate function memoization [10], and approximate data types [31] produce programs that can execute at a variety of points in an underlying performance versus accuracy tradeoff space. Namely, these transformed programs trade accuracy of their results for increased performance by dynamically and nondeterministically modifying variables that control their execution.

We call such transformed programs *relaxed* programs — they have been extended with additional nondeterminism to *relax* their semantics and enable greater flexibility in their execution.

We present programming language constructs for developing and specifying relaxed programs. We also present proof rules for reasoning about properties of relaxed programs. Our proof rules enable programmers to directly specify and verify *acceptability properties* that characterize the desired correctness relationships between the values of variables in a program's original semantics (before the transformation) and its relaxed semantics. Our proof rules also support the verification of *safety properties* (which characterize desirable properties involving values in only the current execution). The rules are designed to support a reasoning approach in which the majority of the reasoning effort uses the original semantics. This effort is then reused to establish the desired properties of the program under the relaxed semantics.

We have formalized the dynamic semantics of our target programming language and the proof rules in Coq, and verified that the proof rules are sound with respect to the dynamic semantics. Our Coq implementation enables developers to obtain fully machine checked verifications of their relaxed programs.

1. Introduction

In recent years researchers have developed a range of mechanisms for dynamically varying application behavior. Typical goals include maximizing performance subject to an accuracy constraint, maximizing accuracy subject to a performance constraint, or dynamically adjusting program behavior to adapt to changes in the characteristics of the underlying hardware platform (such as varying load or clock rate) [15, 16]. Specific mechanisms include multiple selectable implementations of a given component or components [3, 4, 36]; sampling inputs to reductions [36], skipping tasks in parallel programs [27, 28]; loop perforation (skipping iterations of time-consuming loops) [20, 21, 32]; approximate function memoization (returning a previously computed value when the arguments of the new function call are close to the arguments of the previous function call) [10]; dynamic knobs (configuration parameters that can be changed as the program executes) [15]; and approximate data types(data types that return approximate results for operations) [31].

All of these mechanisms can produce a *relaxed* program—a program that may, whenever it reaches a *control point*, adjust its execution by changing one or more *control variables* subject to a specified *relaxation predicate*. For example, a program transformed with loop perforation may dynamically choose to skip loop iterations each time it enters a loop. A relaxed program is therefore a nondeterministic program, with each execution a variant of the original execution (which never dynamically changes the control variables). The different executions typically share a common global structure, with local differences at only those parts of the computation that are affected by the control variables (for example, the perforated loops).

1.1 Reasoning About Relaxed Programs

We present a language for writing relaxed programs and proof rules for reasoning about relaxed programs. The language and proof rules are designed to support a staged approach in which the developer first develops a standard program and uses standard approaches to reason about the program to verify that it satisfies desired correctness properties. We refer to the dynamic semantics of the program at this stage as the *original semantics* of the program.

Either the developer or an automated system (such as a compiler that implements loop perforation) then *relaxes* the program to enable the desired additional nondeterministic executions. We refer to the dynamic semantics of the program at this stage as the *relaxed semantics* of the program.

Finally, the original reasoning is augmented to verify that the relaxation preserves the correctness properties. At this point, it also possible for a programmer to specify and verify *acceptability properties* that relate the relaxed semantics to the original semantics.

This approach is designed to reduce the overall reasoning effort by exploiting the structure that the original and relaxed programs share. With this approach the majority of the reasoning effort works with the original program semantics and is reused to verify the nondeterministic relaxed program.

1.2 Relaxed Programming Concepts

The basic concepts of relaxed programming include nondeterministic assignment to control variables (via the `relax` statement); acceptability properties that relate the relaxed semantics to the original semantics (via the `accept` statement); assertions (via the `assert` statement); assumptions (via the `assume` statement); and the concept of convergent and divergent program points:

- **The Relax Statement:** The `relax (X) st (P)` statement specifies a nondeterministic assignment to the set of variables `X`. Specifically, the `relax` statement can assign the variables in `X` to any set of values that satisfies the relaxation predicate `P`. In the original semantics the `relax` statement has no effect—it does not change the variables in `X`.
- **The Accept Statement:** The `accept P` statement asserts that the property `P` must hold at the program point where the `accept` statement appears. The property `P` in `accept` statements may reference values from both the original and relaxed semantics. So, for example, the statement might require that the value of a variable `x` in the relaxed semantics must be greater than or equal

to the value of that variable in the original semantics. Such properties are *relational properties* because they relate states from the two semantics of the program.

- **The Assert Statement:** The `assert P` statement states that `P` must hold at the point where the statement appears. In contrast to the `accept` statement, `P` only references values from a single semantics (original or relaxed). Therefore, `assert` statements specify key correctness properties that must hold for all executions of the program.
- **The Assume Statement:** The vast majority of reasoning systems support an *assume* statement (which instructs the reasoning system to simply assume that a property holds). The `assume P` statement states that the property `P` holds at the point where the assume statement appears. In the original semantics the assume statement does not generate any proof obligations—the proof system simply accepts that `P` holds. To ensure that the reasoning behind the `assume` statement remains valid in the relaxed semantics, the proof rule for the `assume` statement generates an obligation to prove that given the assumption in the original semantics, then the current relation between the original and relaxed semantics establishes that the assumption is valid in the relaxed semantics.

  For example, it may be possible to prove that all the variables referenced in an assumption have the same values in the original and relaxed semantics—namely, relaxation does not interfere with the assumption.
- **Convergent Program Points:** Conceptually, convergent program points occur when executions under the original and relaxed semantics on the same input take corresponding control flow paths to reach corresponding points in the execution. If the two executions take different branches, then the executions have *diverged*.

  We note that it is possible for the relaxed execution to diverge, then converge back again. For example, the relaxed execution may diverge by taking a different branch at a conditional statement than the original execution, then converge again at the end of the conditional statement. If this is the case, then the executions have again reached a convergent program point.

  To ensure that acceptability properties have a well-defined semantics that references states from corresponding points in the original and relaxed semantics, our rules establish that `accept` statements appear only at convergent points.

Note that our design enables these statements to work together to prove important properties. For example, the developer may use an `accept` statement to establish a relationship between values in variables from original and relaxed executions, then use this relationship to prove that a given property in an `assert` statement holds in the relaxed execution.

1.3 Proof Rules

We structure the static semantics of relaxed programs as set of Hoare logic proof rules:

- **Axiomatic Original Semantics:** The Hoare-style semantics models the original execution of the program wherein relax statements have no effect.
- **Axiomatic Intermediate Semantics:** The Hoare-style semantics models the relaxed execution of the program wherein `relax` statements modify the state of the program.
- **Axiomatic Relaxed Semantics:**. The *relational*, Hoare-style semantics relates executions in the relaxed semantics to executions in the original semantics. The predicates of the judgment are given in a relational logic that enables us to express properties over the value of variables in both the original execution of the program and a relaxed execution. It also tracks whether the original and relaxed executions are at convergent or divergent program points and supports the reuse of reasoning effort from the original semantics by enabling, for example, noninterference proofs that show that the relaxation does not affect the values of variables in the original semantics.

Our proof rules are sound. Specifically, our proof rules establish the following semantic properties of relaxed programs:

- **Original Progress:** If the program verifies under the axiomatic original semantics, then no execution of the program in the dynamic original semantics violates an assertion.
- **Relaxed Progress:** If the program verifies under the axiomatic intermediate semantics, then no execution of the program in the dynamic relaxed semantics violates an assertion or an assumption.
- **Relaxed Progress Modulo Original Assumptions:** If the program verifies under the axiomatic relaxed semantics, then if an execution of the program in the dynamic relaxed semantics violates an assertion or an assumption, then an execution of the program in the dynamic original semantics violates an assumption.
- **Acceptability:** If the program verifies under the axiomatic relaxed semantics, then program executions in the dynamic original and relaxed semantics satisfy all of the `accept` statements in the program.

1.4 Coq Verification Framework

We have formalized the dynamic original and relaxed semantics with the Coq proof assistant [1]. We have also used Coq to formalize the proof rules for the static semantics and obtain a fully machine checked proof that the rules are sound with respect to the dynamic semantics and provide the stated semantic properties. Our Coq formalization makes it possible to develop fully machine checked verifications of relaxed programs. We have used our framework to develop and verify several relaxed programs.

Our Coq implementation contains approximately 6000 lines of code and proof scripts, with 471 lines devoted to the original semantics and its soundness proofs, 258 additional lines devoted to the intermediate semantics and its soundness proofs, and 1311 additional lines devoted to the relaxed semantics and its soundness proofs. A large portion of the implementation (approximately 2898 lines) is devoted to formalizing the semantics of our relational assertion logic and its soundness with respect to operations in the logic (e.g., substitution lemmas).

1.5 Contributions

This paper makes the following contributions:

- **Relaxed Programming:** It identifies the concept of relaxed programming as a way to specify nondeterministic variants of a program. The variants often occupy a range of points in an underlying performance versus accuracy trade-off space. Current techniques that can produce relaxed programs include multiple selectable implementations [3, 4], skipping tasks [27, 28], loop perforation [20, 21, 32], approximate function memoization [10], approximate data types [31], and dynamic knobs [15].
- **Reasoning Approach:** It identifies the basic reasoning approach of first verifying important correctness properties of the original program, stating acceptability properties that may reference the states of both the original and relaxed programs,

$$
\begin{aligned}
\texttt{iop} &::= \ + \mid - \mid * \mid / \mid ... \\
\texttt{cmp} &::= < \mid > \mid = \mid ... \\
\texttt{lop} &::= \ \wedge \mid \vee \mid ... \\
X &::= \texttt{x} \mid \texttt{x}, X \\
E &::= \texttt{n} \mid \texttt{x} \mid E \ \texttt{iop} \ E \\
E^* &::= \texttt{n} \mid \texttt{x}\langle\texttt{o}\rangle \mid \texttt{x}\langle\texttt{r}\rangle \mid E^* \ \texttt{iop} \ E^* \\
B &::= \texttt{true} \mid \texttt{false} \mid E \ \texttt{cmp} \ E \mid B \ \texttt{lop} \ B \mid \neg B \\
B^* &::= \texttt{true} \mid \texttt{false} \mid E^* \ \texttt{cmp} \ E^* \mid B^* \ \texttt{lop} \ B^* \mid \neg B^* \\
S &::= \texttt{skip} \mid \texttt{x} = E \mid \texttt{havoc} \ (X) \ \texttt{st} \ (B) \mid \texttt{relax} \ (X) \ \texttt{st} \ (B) \\
&\quad \mid \texttt{if} \ (B) \ \{S_1\} \ \texttt{else} \ \{S_2\} \mid \texttt{while} \ (B) \ \{S\} \\
&\quad \mid \texttt{assume} \ B \mid \texttt{assert} \ B \mid \texttt{accept} \ \texttt{l} : \ B^* \\
&\quad \mid S \ ; \ S
\end{aligned}
$$

Figure 1. Language Syntax

then augmenting the verification of the original program to establish the correctness properties in the relaxed program. With this approach, the majority of the reasoning effort works with the original program.

- **Proof Rules:** It presents Hoare logic proof rules that support the above reasoning approach. These rules support the verification of properties of relaxed programs that may nondeterministically change control variables. The properties include acceptability properties that establish relationships between the states of the original and relaxed program. These properties are restricted to convergent program points that appear at corresponding program points in the two programs.
- **Coq Formalization and Soundness Results:** It presents a formalization of the dynamic semantics and proof rules in Coq. We have used this formalization to prove that the proof rules are sound with respect to the dynamic semantics. We note that our Coq formalization contains a reusable implementation of our relational assertion logic that is, in principle, suitable for other uses such as verifying traditional compiler transformations [7, 29, 34].
- **Verified Programs:** It presents several relaxed programs for which we have used the Coq formalization to develop fully machine checked verifications.

Relaxed programs can deliver substantial flexibility, performance, and resource consumption benefits. But to successfully deploy relaxed programs, developers need to know that the relaxation does not violate important correctness properties. This paper presents a foundational formal reasoning system that leverages the structure that the original and relaxed executions share to enable the verification of these properties.

2. Language Syntax and Dynamic Semantics

Figure 1 presents a simple imperative language with integer variables, integer arithmetic expressions, boolean expressions, conditional statements, while loops, and sequential composition. For generality, we support nondeterminism in the original semantics via the `havoc` (X) `st` (B) statement which nondeterministically assigns the variables in `X` to values that satisfy B. The `relax` (X) `st` (B) statement supports nondeterministic relaxation — in the original semantics it has no effect; in the relaxed semantics it nondeterministically assigns the variables in `X` to values that satisfy B. The language also supports the standard `assume` and `assert` statements.

$$
\begin{aligned}
Vars &\subseteq \{x, y, z, \ ...\} & [\![\texttt{iop}]\!] &\in \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\
\mathbb{Z} &= \{..., -1, \ 0, \ 1, ...\} & [\![\texttt{cmp}]\!] &\in \mathbb{Z} \times \mathbb{Z} \to \mathbb{B} \\
\mathbb{B} &= \{true, false\} & [\![\texttt{lop}]\!] &\in \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\
\sigma \in \Sigma &= Vars \xrightarrow{fin} \mathbb{Z}
\end{aligned}
$$

$$
\begin{aligned}
[\![E]\!] &\in \Sigma \to \mathbb{Z} \\
[\![\texttt{n}]\!](\sigma) &= n \\
[\![\texttt{x}]\!](\sigma) &= \sigma(x) \\
[\![\ E_1 \ \texttt{iop} \ E_2]\!](\sigma) &= [\![E_1]\!](\sigma) \ iop \ [\![E_2]\!](\sigma)
\end{aligned}
$$

$$
\begin{aligned}
[\![E^*]\!] &\in \Sigma \times \Sigma \to \mathbb{Z} \\
[\![\texttt{n}]\!](\sigma_1, \sigma_2) &= n \\
[\![\texttt{x}\langle\texttt{o}\rangle]\!](\sigma_1, \sigma_2) &= \sigma_1(\texttt{x}) \\
[\![\texttt{x}\langle\texttt{r}\rangle]\!](\sigma_1, \sigma_2) &= \sigma_2(\texttt{x}) \\
[\![E_1^* \ \texttt{iop} \ E_2^*]\!](\sigma_1, \sigma_2) &= [\![E_1^*]\!](\sigma_1, \sigma_2) \ iop \ [\![E_2^*]\!](\sigma_1, \sigma_2)
\end{aligned}
$$

$$
\begin{aligned}
[\![B]\!] &\in \Sigma \to \mathbb{B} \\
[\![\texttt{true}]\!](\sigma) &= true \\
[\![\texttt{false}]\!](\sigma) &= false \\
[\![E_1 \ \texttt{cmp} \ E_2]\!](\sigma) &= [\![E_1]\!](\sigma) \ cmp \ [\![E_2]\!](\sigma) \\
[\![B_1 \ \texttt{lop} \ B_2]\!](\sigma) &= [\![B_1]\!](\sigma) \ lop \ [\![B_2]\!](\sigma) \\
[\![\neg B]\!](\sigma) &= \begin{cases} true, \ [\![B]\!](\sigma) = false \\ false, \ [\![B]\!](\sigma) = true \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
[\![B^*]\!] &\in \Sigma \times \Sigma \to \mathbb{B} \\
[\![\texttt{true}]\!](\sigma_1, \sigma_2) &= true \\
[\![\texttt{false}]\!](\sigma_1, \sigma_2) &= false \\
[\![E_1^* \ \texttt{cmp} \ E_2^*]\!](\sigma_1, \sigma_2) &= [\![E_1^*]\!](\sigma_1, \sigma_2) \ cmp \ [\![E_2^*]\!](\sigma_1, \sigma_2) \\
[\![B_1^* \ \texttt{lop} \ B_2^*]\!](\sigma_1, \sigma_2) &= [\![B_1^*]\!](\sigma_1, \sigma_2) \ lop \ [\![B_2^*]\!](\sigma_1, \sigma_2) \\
[\![\neg B^*]\!](\sigma_1, \sigma_2) &= \begin{cases} true, \ [\![B^*]\!](\sigma_1, \sigma_2) = false \\ false, \ [\![B^*]\!](\sigma_1, \sigma_2) = true \end{cases}
\end{aligned}
$$

$$
\text{expr}\frac{[\![e]\!](\sigma) = n}{\langle e \ , \ \sigma \rangle \Downarrow_E n} \qquad \text{bexp}\frac{[\![b]\!](\sigma) = v}{\langle b \ , \ \sigma \rangle \Downarrow_B v}
$$

Figure 2. Semantics of Expressions

A main point of departure is the addition of *relational integer expressions* (E^*) and *relational boolean expressions* (B^*). Unlike standard expressions, which involve values from only the current execution, relational expressions can reference values from both the original ($\texttt{x}\langle\texttt{o}\rangle$) and relaxed ($\texttt{x}\langle\texttt{r}\rangle$) executions. These relational expressions enable `accept` statements to specify relationships that must hold between the original and relaxed executions. For example, the statement $\texttt{accept l} : \ \texttt{x}\langle\texttt{o}\rangle = \texttt{x}\langle\texttt{r}\rangle$ asserts that at the current program point (labeled by `l`), `x` must have the same value in both executions.

2.1 Semantics of Expressions

Figure 2 presents the semantics of expressions in the language. The denotations of expressions are functions mapping a state or pair of states to either an integer ($\mathbb{Z}$) or boolean value ($\mathbb{B}$). A state (σ) is a finite map from program variables, $Vars$, to integers, $\mathbb{Z}$ and is an element of the domain Σ–which is the set of all finite maps from variables to integers.

$\boxed{\langle s,\ \sigma\rangle\ \Downarrow_o\ \phi}$

$$\text{skip}\ \frac{}{\langle \texttt{skip},\ \sigma\rangle\ \Downarrow_o\ \langle \sigma,\ \emptyset\rangle} \qquad \text{assign}\ \frac{\langle e\ ,\ \sigma\rangle \Downarrow_E n}{\langle x = e,\ \sigma\rangle\ \Downarrow_o\ \langle \sigma[x \mapsto n],\ \emptyset\rangle} \qquad \text{havoc-t}\ \frac{\langle e\ ,\ \sigma'\rangle \Downarrow_B true \quad \forall_{x \notin X} \cdot \sigma(x) = \sigma'(x)}{\langle \texttt{havoc}\ (X)\ \texttt{st}\ (e),\ \sigma\rangle\ \Downarrow_o\ \langle \sigma',\ \emptyset\rangle}$$

$$\text{havoc-f}\ \frac{\neg \exists_{\sigma'} \cdot (\langle e\ ,\ \sigma'\rangle \Downarrow_B true \wedge \forall_{x \notin X} \cdot \sigma(x) = \sigma'(x))}{\langle \texttt{havoc}\ (X)\ \texttt{st}\ (e),\ \sigma\rangle\ \Downarrow_o\ wr} \qquad \text{assert-t}\ \frac{\langle e\ ,\ \sigma\rangle \Downarrow_B true}{\langle \texttt{assert}\ e,\ \sigma\rangle\ \Downarrow_o\ \langle \sigma,\ \emptyset\rangle} \qquad \text{assert-f}\ \frac{\langle e\ ,\ \sigma\rangle \Downarrow_B false}{\langle \texttt{assert}\ e,\ \sigma\rangle\ \Downarrow_o\ wr}$$

$$\text{assume-t}\ \frac{\langle e\ ,\ \sigma\rangle \Downarrow_B true}{\langle \texttt{assume}\ e,\ \sigma\rangle\ \Downarrow_o\ \langle \sigma,\ \emptyset\rangle} \qquad \text{assume-f}\ \frac{\langle e\ ,\ \sigma\rangle \Downarrow_B false}{\langle \texttt{assume}\ e,\ \sigma\rangle\ \Downarrow_o\ ba} \qquad \text{relax}\ \frac{\langle \texttt{assert}\ e,\ \sigma\rangle\ \Downarrow_o\ \phi}{\langle \texttt{relax}\ (X)\ \texttt{st}\ (e),\ \sigma\rangle\ \Downarrow_o\ \phi}$$

$$\text{accept}\ \frac{}{\langle \texttt{accept}\ l:\ \ e^*,\ \sigma\rangle\ \Downarrow_o\ \langle \sigma,\ (l,\sigma)\rangle} \qquad \text{if-t}\ \frac{\langle b\ ,\ \sigma\rangle \Downarrow_B true \quad \langle s_1,\ \sigma\rangle\ \Downarrow_o\ \phi}{\langle \texttt{if}\ (b)\ \{s_1\}\ \texttt{else}\ \{s_2\},\ \sigma\rangle\ \Downarrow_o\ \phi} \qquad \text{if-f}\ \frac{\langle b\ ,\ \sigma\rangle \Downarrow_B false \quad \langle s_2,\ \sigma\rangle\ \Downarrow_o\ \phi}{\langle \texttt{if}\ (b)\ \{s_1\}\ \texttt{else}\ \{s_2\},\ \sigma\rangle\ \Downarrow_o\ \phi}$$

$$\text{seq-1}\ \frac{\langle s_1,\ \sigma\rangle\ \Downarrow_o\ \langle \sigma',\ \psi_1\rangle \quad \langle s_1,\ \sigma'\rangle\ \Downarrow_o\ \langle \sigma'',\ \psi_2\rangle}{\langle s_1\ ;\ s_2,\ \sigma\rangle\ \Downarrow_o\ \langle \sigma'',\ \psi_2.\psi_1\rangle} \qquad \text{while-f}\ \frac{\langle b\ ,\ \sigma\rangle \Downarrow_B false}{\langle \texttt{while}\ (b)\ \{s\},\ \sigma\rangle\ \Downarrow_o\ \langle \sigma,\ \emptyset\rangle}$$

$$\text{while-t1}\ \frac{\langle b\ ,\ \sigma\rangle \Downarrow_B true \quad \langle s,\ \sigma\rangle\ \Downarrow_o\ \langle \sigma',\ \psi_1\rangle \quad \langle \texttt{while}\ (b)\ \{s\},\ \sigma'\rangle\ \Downarrow_o\ \langle \sigma'',\ \psi_2\rangle}{\langle \texttt{while}\ (b)\ \{s\},\ \sigma\rangle\ \Downarrow_o\ \langle \sigma'',\ \psi_2.\psi_1\rangle}$$

Figure 3. Dynamic Original Semantics

$\boxed{\langle s,\ \sigma\rangle\ \Downarrow_o\ \phi}$

$$\text{seq-2}\ \frac{\langle s_1,\ \sigma\rangle\ \Downarrow_o\ \phi \quad err(\phi)}{\langle s_1\ ;\ s_2,\ \sigma\rangle\ \Downarrow_o\ \phi}$$

$$\text{seq-3}\ \frac{\langle s_1,\ \sigma\rangle\ \Downarrow_o\ \langle \sigma',\ \psi\rangle \quad \langle s_2,\ \sigma'\rangle\ \Downarrow_o\ \phi \quad err(\phi)}{\langle s_1\ ;\ s_2,\ \sigma\rangle\ \Downarrow_o\ \phi}$$

$$\text{while-t2}\ \frac{\langle b\ ,\ \sigma\rangle \Downarrow_B true \quad \langle s,\ \sigma\rangle\ \Downarrow_o\ \langle \sigma',\ \psi\rangle \quad \langle \texttt{while}\ (b)\ \{s\},\ \sigma'\rangle\ \Downarrow_o\ \phi \quad err(\phi)}{\langle \texttt{while}\ (b)\ \{s\},\ \sigma\rangle\ \Downarrow_o\ \phi}$$

$$\text{while-t3}\ \frac{\langle b\ ,\ \sigma\rangle \Downarrow_B true \quad \langle s,\ \sigma\rangle\ \Downarrow_o\ \phi \quad err(\phi)}{\langle \texttt{while}\ (b)\ \{s\},\ \sigma\rangle\ \Downarrow_o\ \phi}$$

Figure 4. Error Propagation in Dynamic Original Semantics

The semantic function $[\![E]\!]$ defines the semantics for integer expressions, which are composed of the standard integer operations (e.g., $+, -, *, /$, ...) on integer operands. The semantic function $[\![B]\!]$ defines the semantics of boolean expressions, which are composed of the standard comparison operators on integers (e.g., $<, =, >$, ...) and the standard boolean operators (e.g., $\wedge, \vee$, ...).

The semantic function $[\![E^*]\!]$ defines the semantics for relational integer expressions as a function mapping a pair of states (σ_1, σ_2) to an integer number. Our convention is to have the first component of the state pair be a state from the original semantics and the second component a state from the relaxed semantics. Therefore, a reference to a variable in the original semantics ($\texttt{x}\langle\texttt{o}\rangle$) is equivalent to $\sigma_1(x)$ whereas a reference to a variable ($\texttt{x}\langle\texttt{r}\rangle$) in the relaxed semantics is equivalent to $\sigma_2(x)$.

The semantic function $[\![B^*]\!]$ likewise extends the semantics for boolean expressions with the capability to express boolean properties over relational integer expressions.

2.2 Dynamic Original Semantics

Figure 3 presents the dynamic original semantics of the program in a big-step operational style. The evaluation relation $\langle s,\ \sigma\rangle\ \Downarrow_o\ \phi$ means that evaluation of the statement s from a state σ yields the output configuration ϕ. An output configuration is an element in the domain $\Phi : ba \cup wr \cup (\Sigma \times \Psi)$.

The distinguished element ba is intended to mean that the program has failed at an `assume` statement in the program. The distinguished element wr denotes that the program has failed due to another error (such as an unsatisfied assert statement in the program).

An element in the domain $\Sigma \times \Psi$ indicates that the program has terminated successfully, yielding a final state σ and an *observation list*, $\psi \in \Psi$, which is the sequence of *observations* emitted by `accept` statements during the execution of the program.

An observation (l, σ) is an element in the domain $L \times \Sigma$. L is the finite domain consisting of all the labels specified in `accept` statements in the program — the execution of each `accept` statement emits an observation consisting of its label along with the current state of the program.

The structure of an observation list is given by the standard constructors for lists: $\Psi = \emptyset \mid (l, \sigma) :: \Psi$. We also use the notation $\psi_1.\psi_2$ to denote the result of appending two lists.

Standard Rules. the rules for `skip`, assignment, `if`, sequential composition, and `while` statements follow the standard semantics for these statements.

The `havoc` ***statement*** non-deterministically assigns values to the set of variables in X such that their values satisfy the statement's constraint, e. All variables not specified in X retain their previous values. If there does not exist an assignment of values to X that satisfy e, then the statement evaluates to wr.

The `assert` ***statement*** checks that the state satisfies its constraint e. If the boolean expression evaluates to $true$, then evaluation continues; otherwise, the statement evaluates to wr.

The `assume` ***statement*** checks that the state satisfies its constraint e. If the boolean expression evaluates to $true$, then evaluation continues; otherwise, the statement evaluates to ba.

The `relax` ***statement*** does not modify the state of the program in the original semantics. However, our design for relaxation is that a `relax` statement strictly relaxes or generalizes the semantics of a program at a given point. In particular, an original execution of the program is a trivial member of the relaxed semantics of the program. Therefore, the dynamic semantics of the `relax` statement assert that the relaxation predicate holds. We implement this by reusing the evaluation rule for assert.

$$\boxed{\langle s,\ \sigma\rangle\ \Downarrow_r\ \phi} \qquad \text{relax}\frac{\langle \texttt{havoc}\ (X)\ \texttt{st}\ (e),\ \sigma\rangle\ \Downarrow_r\ \phi}{\langle \texttt{relax}\ (X)\ \texttt{st}\ (e),\ \sigma\rangle\ \Downarrow_r\ \phi}$$

Figure 5. Dynamic Relaxed Semantics

The `accept` ***statement*** is a relational assertion over original and relaxed executions of the program. As already discussed, the dynamic semantics of the statement is to emit an observation consisting of the statement's label along with the current state of the program. This semantics enables us to define an *acceptability relation* on the observation lists emitted by the original and relaxed variants of the program (Section 4).

Error Propagation Rules. Figure 4 presents standard rules for the propagation of error values in the semantics. The predicate $err(\phi)$ evaluates to true if and only if $\phi = wr$ or $\phi = ba$.

2.3 Dynamic Relaxed Semantics

Figure 5 presents an abbreviated definition of the dynamic relaxed semantics of a program. The evaluation relation $\langle s,\ \sigma\rangle\ \Downarrow_r\ \phi$ means that evaluation of the statement s from a state σ yields the output configuration ϕ.

The dynamic relaxed semantics builds upon the original semantics and differs from the original semantics only in that `relax` statements modify the state of the program. We have, therefore, elided the presentation of all the rules that are either reused (`skip`, assignment, `havoc`, `assert`, and `assume`) or adapted to refer to the relaxed dynamic semantics in their premises (i.e., sequential composition, `if`, and `while`).

The `relax` ***statement*** extends the definition from the original semantics with the additional property that the statement can modify the state of the program. The rule implements the modification by reusing the rule for `havoc` statements.

3. Axiomatic Semantics

In this section we present axiomatic definitions for programs in our model. We formalize this problem by presenting axiomatic definitions for the original semantics, the intermediate semantics, and the relaxed semantics.

- **Axiomatic Original Semantics.** The proof rules model the dynamic original semantics of the program. If the program verifies with these rules, then no execution of the program in the dynamic original semantics violates an assertion. However, the program may dynamically violate an assumption.
- **Axiomatic Intermediate Semantics.** The proof rules model the dynamic relaxed semantics of the program. If the program verifies with these rules, then no execution of the program in the dynamic relaxed semantics violates an assertion or an assumption.
- **Axiomatic Relaxed Semantics.** The proof rules model pairs of executions of the program in the dynamic original and dynamic relaxed semantics. If the program verifies with these rules, then if an execution of the program in the dynamic relaxed semantics violates an assertion or an assumption, then an execution of the program in the dynamic original semantics violates an assumption.

  A proof with these rules also guarantees that program executions in the dynamic original and relaxed semantics satisfy all of the `accept` statements in the program.

We first present the relational assertion logic that underpins our axiomatic definitions.

$$\begin{aligned} P &::= \texttt{true} \mid \texttt{false} \mid E\ \texttt{cmp}\ E \mid P\ \texttt{lop}\ P \mid \neg P \mid \exists_{\texttt{x}} \cdot P \\ &\quad \mid \texttt{prj}\langle \texttt{o}\rangle\ P^* \mid \texttt{prj}\langle \texttt{r}\rangle\ P^* \\ P^* &::= \texttt{true} \mid \texttt{false} \mid E^*\ \texttt{cmp}\ E^* \mid P^*\ \texttt{lop}\ P^* \mid \neg P^* \end{aligned}$$

Figure 6. Relational Assertion Logic Syntax

$$\begin{aligned} [\![P]\!] &\in \mathcal{P}(\Sigma) \\ [\![\texttt{true}]\!] &= \Sigma \\ [\![\texttt{false}]\!] &= \emptyset \\ [\![E_1\ \texttt{cmp}\ E_2]\!] &= \{\sigma \mid [\![E_1]\!](\sigma)\ cmp\ [\![E_2]\!](\sigma)\} \\ [\![P_1\ \texttt{lop}\ P_2\ ]\!] &= \{\sigma \mid \sigma \in [\![P_1]\!]\ lop\ \sigma \in [\![P_2]\!]\} \\ [\![\neg P]\!] &= [\![\texttt{true}]\!] \setminus [\![P]\!] \\ [\![\exists_{\texttt{x}} \cdot P]\!] &= \{\sigma \mid n \in \mathbb{Z},\ \sigma \in [\![P[n/x]]\!]\ \} \\ [\![\texttt{prj}\langle \texttt{o}\rangle\ P^*]\!] &= \{\ \sigma_1 \mid (\sigma_1, \sigma_2) \in [\![P^*]\!]\} \\ [\![\texttt{prj}\langle \texttt{r}\rangle\ P^*]\!] &= \{\ \sigma_2 \mid (\sigma_1, \sigma_2) \in [\![P^*]\!]\} \\ \\ [\![P^*]\!] &\in \mathcal{P}(\Sigma \times \Sigma) \\ [\![\texttt{true}]\!] &= \Sigma \times \Sigma \\ [\![\texttt{false}]\!] &= \emptyset \\ [\![E_1^*\ \texttt{cmp}\ E_2^*]\!] &= \{(\sigma_1, \sigma_2) \mid \\ &\qquad [\![E_1^*]\!](\sigma_1, \sigma_2)\ cmp\ [\![E_2^*]\!](\sigma_1, \sigma_2)\} \\ [\![P_1^*\ \texttt{lop}\ P_2^*]\!] &= \{(\sigma_1, \sigma_2) \mid \\ &\qquad (\sigma_1, \sigma_2) \in [\![P_1^*]\!]\ lop\ (\sigma_1, \sigma_2) \in [\![P_2^*]\!]\} \\ [\![\neg P^*]\!] &= [\![\texttt{true}]\!] \setminus [\![P^*]\!] \end{aligned}$$

Figure 7. Relational Assertion Logic Semantics

3.1 Relational Assertion Logic

Figure 6 presents the concrete syntax of our *relational* assertion logic. This logic extends a non-relational assertion logic with relational formulas, which then allow us to reason about the validity of relational boolean expressions in `accept` statements. Its presentation follows the style of Benton in his work on Relational Hoare Logic [7].

3.1.1 Syntax

The syntactic category P gives the syntax for formulas in first-order logic with integer expressions and existential quantification. The syntactic category P^* gives corresponding syntax for writing relational formulas. P^* extends P by allowing formula to refer to relational integer expressions. P also incorporates a new predicate form that enables us to transform relational formulas into non-relational formulas: *projection.*

Projection. Projections enable the logic to decompose a relational assertion about the behavior of both the original semantics and the relaxed semantics into a non-relational formula that describes the set of states that satisfy the relation for either the original or relaxed semantics individually.

3.1.2 Semantics

Figure 7 presents the semantics of formulas in the logic. We model a non-relational formula P as the set of states that satisfy the formula. The semantic function $[\![P]\!]$ reuses the semantic definitions for integer expressions from Figure 2 to construct a definition for each formula. The semantics of relational formulas closely follows that of non-relational formulas. We model a relational formula as the set of pairs of states that satisfy the relation. In our use of

$\boxed{\vdash_o \{P\}\ s\ \{Q\}}$

$$\text{skip}\frac{}{\vdash_o \{P\}\ \texttt{skip}\ \{P\}} \qquad \text{seq}\frac{\vdash_o \{P\}\ s_1\ \{R\} \quad \vdash_o \{R\}\ s_2\ \{Q\}}{\vdash_o \{P\}\ s_1; s_2\ \{Q\}} \qquad \text{assign}\frac{fresh(x')}{\vdash_o \{P\}\ \texttt{x = }e\ \{\exists_{x'} \cdot x = e[x'/x] \wedge P[x'/x]\}}$$

$$\text{havoc}\frac{[\![\exists_{X'} \cdot P[X'/X] \wedge e]\!] \neq \emptyset \quad fresh(X')}{\vdash_o \{P\}\ \texttt{havoc}\ (X)\ \texttt{st}\ (e)\ \{(\exists_{X'} \cdot P[X'/X]) \wedge e\}} \qquad \text{assert}\frac{\models P \Rightarrow e}{\vdash_o \{P\}\ \texttt{assert}\ e\ \{P \wedge e\}} \qquad \text{assume}\frac{}{\vdash_o \{P\}\ \texttt{assume}\ e\ \{P \wedge e\}}$$

$$\text{while}\frac{\vdash_o \{P \wedge b\}\ s\ \{P\}}{\vdash_o \{P\}\ \texttt{while}\ (b)\ \{s\}\ \{P \wedge \neg b\}} \qquad \text{if}\frac{\vdash_o \{P \wedge b\}\ s_1\ \{Q\} \quad \vdash_o \{P \wedge \neg b\}\ s_2\ \{Q\}}{\vdash_o \{P\}\ \texttt{if}\ (b)\ \{s_1\}\ \texttt{else}\ \{s_2\}\ \{Q\}} \qquad \text{relax}\frac{\vdash_o \{P\}\ \texttt{assert}\ e\ \{P \wedge e\}}{\vdash_o \{P\}\ \texttt{relax}\ (X)\ \texttt{st}\ (e)\ \{P \wedge e\}}$$

$$\text{accept}\frac{}{\vdash_o \{P\}\ \texttt{accept}\ l:\ \ e^*\ \{P\}}$$

Figure 8. Axiomatic Original Semantics

$\boxed{\vdash_i \{P\}\ s\ \{Q\}}$ $\quad \text{relax}\dfrac{\vdash_i \{P\}\ \texttt{havoc}\ (X)\ \texttt{st}\ (e)\ \{Q\}}{\vdash_i \{P\}\ \texttt{relax}\ (X)\ \texttt{st}\ (e)\ \{Q\}}$

$$\text{assume}\frac{\models P \Rightarrow e}{\vdash_i \{P\}\ \texttt{assume}\ e\ \{P \wedge e\}}$$

Figure 9. Axiomatic Intermediate Semantics

the logic, references to the original semantics (e.g., $\texttt{x}\langle\texttt{o}\rangle$) refer to the first component of the pair whereas references to the relaxed semantics (e.g., $\texttt{x}\langle\texttt{r}\rangle$) refer to the second component of the pair.

Non-relational formulas can contain projection predicates of the form $\texttt{prj}\langle\texttt{o}\rangle\ P^*$ and $\texttt{prj}\langle\texttt{r}\rangle\ P^*$. Following our convention for variable naming, $\texttt{prj}\langle\texttt{o}\rangle\ P^*$ selects the component that corresponds to the original semantics, and $\texttt{prj}\langle\texttt{r}\rangle\ P^*$ selects that for the relaxed semantics—which by our convention corresponds to the first and second component, respectively.

Injections. We also define the injection functions $inj_o(P)$ and $inj_r(P)$, which construct a relational formula from a non-relational formula. The form $inj_o(P)$ constructs a relational formula where P holds for the original semantics and $inj_r(P)$ does the same such that P holds for relaxed semantics. This means that $inj_o(P)$ (resp. $inj_r(P)$) creates a formula consisting of all state pairs where the first (resp. second) component satisfies P. We also define the following syntactic form for combining a predicate over the original semantics with one over the relaxed semantics:

$$\langle P_1 \cdot P_2 \rangle \equiv inj_o(P_1) \wedge inj_r(P_2)$$

Auxiliary Definitions. We also define the following judgments for later use in both the rules of our program logics and the discussion of their semantics:

$$\sigma \models P \equiv \sigma \in [\![P]\!] \qquad (\sigma_1, \sigma_2) \models P^* \equiv (\sigma_1, \sigma_2) \in [\![P^*]\!]$$
$$\models P_1 \Rightarrow P_2 \equiv [\![P_1]\!] \subseteq [\![P_2]\!] \quad \models P_1^* \Rightarrow P_2^* \equiv [\![P_1^*]\!] \subseteq [\![P_2]\!]$$

Free and Fresh Variables. The set of free variables of a term, denoted by $free(P)$, is defined by structural induction on the term. The boolean predicate $fresh(x)$, denoting that x is a fresh variable in the context of an inference rule, is true if $x \in Vars$ and it does not appear in the premises or consequent of the rule.

3.2 Original Semantics

Figure 8 presents a manual translation of our Coq formalization of the axiomatic original semantics of the program. The intended meaning of the judgment is the semantic judgment $\models_o \{P\}\ s\ \{Q\}$: for all states σ, if $\sigma \models P$ and $\langle s,\ \sigma\rangle \Downarrow_o \langle\sigma',\ \psi\rangle$, then $\sigma' \models Q$. This asserts partial correctness and says nothing about non-terminating evaluations.

We have elided a discussion of the rules for standard constructs (i.e., `skip`, assign, sequential composition, `if`, `while`) because their definitions are the same as in standard presentations (e.g., Floyd and Hoare [13, 14]). We define the non-standard rules as follows:

The havoc rule is similar in semantics to assignment except that it assigns multiple variables during its execution. We enforce two properties on havoc statements that enable us to prove that havoc statements do not evaluate to wr: 1) e must be satisfiable (i.e., there exists some satisfying assignment of the variables in X) and 2) e must be consistent with P (i.e., $P \wedge e \not\equiv false$).

The assert rule asserts that a formula e is true at a given point in the program's evaluation. This rule requires that P implies e and then adds e to the consequent.

The assume rule assumes that a formula e is true at a given point in the program's evaluation. The rule differs from the rule for `assert` statements in that it assumes the validity of e and then makes e part of the consequent. Because there is no obligation to prove e for an `assume` statement, e may not hold for all states that satisfy P and, as a result, the `assume` may evaluate to ba. However, by design, we allow `assume` statements to fail in the dynamic original semantics.

The relax rule non-deterministically relaxes the relation on variables in X. In the original semantics of the program, relaxation is a no-op and does not change the program's state. However, in the original semantics, the current state of the program must still satisfy the relaxation. Therefore, the rule asserts that P implies the relaxation specification, e, and then adds it to the consequent.

The accept rule gives `accept` statements the same semantics as `skip` because, unlike the relaxed axiomatic semantics (Section 3.1), the original axiomatic semantics references only a single execution of the program.

Auxiliary Rules. Although not presented here, we have also formalized and verified the standard rules of consequence and constancy to aid in the development of proofs.

3.3 Intermediate Semantics

Figure 9 gives an abbreviated presentation of the program logic for the intermediate semantics of the program. Specifically, we have elided all rules that have the same form as those in the original semantics.

The axiomatic intermediate semantics of the program differs from the original semantics of the program only in that it captures the fact that `relax` statements in the dynamic relaxed semantics of the program affect the program's state. The intended meaning

$\boxed{\vdash_r \{P^*\}\ s\ \{Q^*\}}$

$$\text{accept}\frac{\models P^* \Rightarrow e^*}{\vdash_r \{P^*\}\ \texttt{accept}\ l:\ \ e^*\ \{P^* \wedge e^*\}} \qquad \text{assume}\frac{\models P^* \wedge \langle e \cdot \texttt{true}\rangle \Rightarrow \langle \texttt{true} \cdot e\rangle}{\vdash_r \{P^*\}\ \texttt{assume}\ e\ \{P^* \wedge \langle e \cdot e\rangle\}}$$

$$\text{if}\frac{\models P^* \Rightarrow \langle b \cdot b\rangle \vee \langle \neg b \cdot \neg b\rangle \quad \vdash_r \{P^* \wedge \langle b \cdot b\rangle\}\ s_1\ \{Q^*\} \quad \vdash_r \{P^* \wedge \langle \neg b \cdot \neg b\rangle\}\ s_2\ \{Q^*\}}{\vdash_r \{P^*\}\ \texttt{if}\ (b)\ \{s_1\}\ \texttt{else}\ \{s_2\}\ \{Q^*\}}$$

$$\text{while}\frac{\models P^* \Rightarrow \langle b \cdot b\rangle \vee \langle \neg b \cdot \neg b\rangle \quad \vdash_r \{P^* \wedge \langle b \cdot b\rangle\}\ s\ \{P^*\}}{\vdash_r \{P^*\}\ \texttt{while}\ (b)\ \{s\}\ \{P^* \wedge \langle \neg b \cdot \neg b\rangle\}} \qquad \text{diverge}\frac{\vdash_o \{\texttt{prj}\langle \texttt{o}\rangle\ P^*\}\ s\ \{Q_o\} \quad \vdash_i \{\texttt{prj}\langle \texttt{r}\rangle\ P^*\}\ s\ \{Q_r\} \quad no\_acc(s)}{\vdash_r \{P^*\}\ s\ \{\langle Q_o \cdot Q_r\rangle\}}$$

Figure 10. Axiomatic Relaxed Semantics

of the judgment is $\models_i \{P\}\ s\ \{Q\}$: for all states σ, if $\sigma \models P$ and $\langle s,\ \sigma\rangle \Downarrow_r \langle \sigma',\ \psi\rangle$, then $\sigma' \models Q$. This judgment asserts partial correctness and says nothing about non-terminating evaluations.

The relax rule specifies that a `relax` (X) `st` (e) modifies the variables in X such that their values satisfy e. The modification is non-deterministic. To capture this, the rule uses the havoc rule to specify the effects.

The assume rule differs from its counterpart in the original semantics because the effects of relax statements may violate the programmer's original assumptions about the behavior of the program. For example, relaxation may interfere with the values of the variables in a subsequent `assume` statement. Therefore, to ensure that an `assume` statement does not evaluate to ba, the rule asserts that the assumption holds.

3.4 Relaxed Semantics

Figure 10 presents a manual translation of our Coq formalization of the axiomatic relaxed semantics of the program. This definition enables us to relate executions of the program under the dynamic relaxed semantics to executions under the dynamic original semantics. The intended meaning of the judgment is $\models_r \{P^*\}\ s\ \{Q^*\}$: if $(\sigma_o, \sigma_r) \models P^*$, and $\langle s,\ \sigma_o\rangle \Downarrow_o \langle \sigma_o',\ \psi_1\rangle$, and $\langle s,\ \sigma_r\rangle \Downarrow_r \langle \sigma_r',\ \psi_2\rangle$, then $(\sigma_o', \sigma_r') \models Q^*$. This asserts partial correctness and says nothing about non-terminating evaluations.

We have elided presentations of the rule of consequence, rule of constancy, and sequential composition because they have standard definitions. However, the non-standard rules are as follows:

The accept rule enables reasoning about `accept` statements in the program. The rule requires that P^* satisfy the acceptability property, e^*.

The assume rule demonstrates how relational reasoning allows us to use relations between the original and relaxed semantics of the program to reason about assumptions in the relaxed semantics. Specifically, the form of the rule enables us to use general relations between the original and relaxed semantics of the program along with the validity of the assumption in the original program to prove that the assumption is true in the relaxed semantics.

For example, if the precondition of the statement asserts that all the free variables in the condition of the assume statement are the same (i.e., relaxation does not interfere with the assumption), then if the assumption was true in the original semantics of the program, we can conclude that the assumption is true in the relaxed semantics of the program.

The if rule allows `accept` statements to appear within an `if` statement if control flow is convergent (i.e., the program takes the same branch in both the original and relaxed executions). This is established by checking that for all σ_1, σ_2, if $(\sigma_1, \sigma_2) \models P^*$ then the conditional's boolean expression either evaluates to $true$ in both the original and relaxed semantics or it evaluates to $false$ in both the original and relaxed semantics. This means that in all cases, the original and relaxed semantics take the same branch together. If this condition is not satisfied, then the rule cannot be applied.

The while rule is similar in form to the if rule in that it requires that control flow be convergent to allow `accept` statements within the body of a `while` statement.

The diverge rule enables a proof to proceed if the original and relaxed semantics diverge at a control flow construct. Namely, the rule establishes the consequent of the statement by independently establishing that $\vdash_o \{\texttt{prj}\langle \texttt{o}\rangle\ P^*\}\ s\ \{Q_o\}$ and that $\vdash_i \{\texttt{prj}\langle \texttt{r}\rangle\ P^*\}\ s\ \{Q_r\}$ and then uses injection to establish that $\vdash_r \{P^*\}\ s\ \{\langle Q_o \cdot P_r\rangle\}$.

The predicate $no\_acc(s)$ evaluates to true if no `accept` statements appear within s. This predicate therefore prevents `accept` statements from appearing in control flow statements where it cannot be established that control flow is convergent.

The use of projections in this rule means that all relational properties between the two semantics are lost. Relational properties that are not modified by the statement, however, can be preserved by using a relational variant of the rule of constancy. Alternatively, some relational properties may be restored if they are provable after the convergence of the two statements.

The diverge rule also allows the logic to use the rules from the original and intermediate semantics to give a semantics to the language's primitive statements (i.e., `skip`, assign, `assert`, `havoc`, and `relax`).

4. Properties

We now present the technical definitions, lemmas, and theorems that establish the semantic properties of programs in the language. In our formalization, we restrict ourselves to terminating relaxed programs. Also, we only present brief sketches of our proofs because we have verified each lemma and theorem in the Coq development included as supplemental material to this paper

4.1 Original Semantics

Our axiomatic definition for the original semantics is sound and can be used to establish a *weak* form of the traditional progress theorem for programs. Namely, if an evaluation in the original semantics terminates, as given by the rules, then the evaluation does not go wrong. This is opposed to a *strong* form of progress that establishes the same for all programs (including non-terminating).

Lemma 4.1 (Soundness)**.**

$$\textit{If } \vdash_o \{P\}\ s\ \{Q\}\textit{, then } \models_o \{P\}\ s\ \{Q\}$$

This lemma establishes that our axiomatic definition is sound with respect to the dynamic original semantics of the program.

Proof Sketch. This proof proceeds by induction on the rules of the definition. A large portion of the proof effort involves proving the

semantics of substitution in the case of the assignment rule and the havoc rule. The case of the havoc rule requires mutual induction on the lists of modified and fresh variables to establish that the post-condition holds. The proof consists of approximately 147 lines of Coq proof script. □

Lemma 4.2 (Progress Modulo Assumptions)**.**

$$\textit{If } \vdash_o \{P\}\ s\ \{Q\}\textit{, and } \sigma \models P \textit{ , and } \langle s,\ \sigma\rangle\ \Downarrow_o\ \phi \textit{ , then } \phi \neq wr.$$

This lemma establishes the progress property that we desire for the original semantics. Namely, given a proof in the original axiomatic semantics, then for all states that satisfy P, if evaluation terminates, then the evaluation does not yield wr. By design, the judgment does not preclude the program from evaluating to ba (indicating that it has violated an assumption).

Proof Sketch. This proof proceeds by induction on the rules of the definition. The proof uses Lemma 4.1 for the cases of sequential composition and the while statement. In addition, the case for the while statement proceeds by nested induction on derivations of the evaluation of a while statement. The proof consists of approximately 18 lines of Coq proof script. □

4.2 Intermediate Semantics

Like its counterpart for the original semantics, the axiomatic definition for the intermediate semantics is sound and, also, establishes progress. Because our design does not allow relaxed executions that have been verified with the axiomatic intermediate semantics to violate assumptions, the progress lemma is stronger than that of the original semantics: if a relaxed evaluation terminates (according to the evaluation rules), then it does not evaluate to wr or ba.

Lemma 4.3 (Soundness)**.**

$$\textit{If } \vdash_i \{P\}\ s\ \{Q\}\textit{, then } \models_i \{P\}\ s\ \{Q\}$$

This lemma establishes that our axiomatic definition is sound with respect to the dynamic relaxed semantics.

Proof Sketch. This proof proceeds by induction on the rules of the definition. Because a large portion of dynamic relaxed semantics reuses the definitions from the dynamic original semantics, the proof itself reuses many of the proofs established in Lemma 4.1. The proof is approximately 19 lines of Coq proof script. □

Lemma 4.4 (Progress)**.**

$$\textit{If } \vdash_i \{P\}\ s\ \{Q\} \textit{ and } \langle s,\ \sigma\rangle\ \Downarrow_r\ \phi \textit{ , then } \neg err(\phi).$$

This lemma establishes that given a proof in the intermediate axiomatic semantics, for all states that satisfy P, if the program terminates, producing an output configuration ϕ, then ϕ is not an error configuration (i.e., wr or ba).

Proof Sketch. This proof proceeds by induction on the rules of the definition. The proof uses Lemma 4.3 for the cases of sequential composition and the while statement. In addition, the case for the while statement proceeds by nested induction on derivations of the evaluation of a while statement. The proof differs from the corresponding lemma for the original semantics at the assume rule; the condition on the rule enables us to rule out the situation wherein the program produces ba. The proof consists of approximately 17 lines of Coq proof script. □

4.3 Relaxed Semantics

In addition to the soundness of the definition itself, the axiomatic definition for the relaxed semantics establishes that if the relaxed execution violates an assumption or goes wrong, then an original execution violates an assumption. It also establishes that relaxed executions of the program satisfy the `accept` statements in the program when compared to an original execution of the program.

Lemma 4.5 (Soundness)**.**

$$\textit{If } \vdash_r \{P^*\}\ s\ \{Q^*\}\textit{, then } \models_r \{P^*\}\ s\ \{Q^*\}.$$

This lemma establishes that our axiomatic definition is sound with respect to the original and relaxed executions of the program.

Proof Sketch. The proof proceeds by induction on the rules of the definition. The case for the diverge rule uses Lemma 4.1 and Lemma 4.3 to establish the soundness of the judgment for evaluations in which the original and relaxed semantics diverge. The convergent if and while cases use the convergence condition on the rules to eliminate cases in which a conditional evaluates differently in either the original or relaxed evaluation. In the case for the while rule, the proof proceeds by nested, mutual induction on derivations of the original and relaxed evaluation of the statement. The proof consists of approximately 200 lines of Coq proof script. □

Theorem 4.1 (Relaxed Progress Modulo Original Assumptions)**.**

$$\textit{If } \vdash_r \{P^*\}\ s\ \{Q^*\}\textit{, and } (\sigma_o, \sigma_r) \models P^*\textit{, and } \langle s,\ \sigma_r\rangle\ \Downarrow_r\ \phi_r\textit{,}$$
$$\textit{and } \phi_r = ba \vee \phi_r = wr\textit{, and } \langle s,\ \sigma_o\rangle\ \Downarrow_o\ \phi_o\textit{, then } \phi_o = ba$$

This theorem establishes the progress guarantee that restores the ability for relaxed programs to contain `assume` statements without mandating that their conditions be verified. Namely, if a relaxed execution of the program terminates and evaluates to wr or ba, and an original execution of the program terminates, then the original execution evaluates to ba.

Proof Sketch. This proof proceeds by induction on the rules of the definition. In the case of the diverge rule, the proof uses Lemma 4.4 to contradict the premise that $\langle s_r,\ \sigma_r\rangle\ \Downarrow_r\ ba$. The proof also uses Lemma 4.5 in the proofs for the sequential composition statement and the while statement. In addition, the proof for the while statement proceeds by nested, mutual induction on the evaluation derivations for the while statement in both the original and relaxed semantics. The proof consists of approximately 200 lines of Coq proof script. □

Theorem 4.2 (Acceptability)**.**

$$\textit{If } \vdash_r \{P^*\}\ s\ \{Q^*\}\textit{, and } (\sigma_o, \sigma_r) \models P^*\textit{, and}$$
$$\langle s,\ \sigma_o\rangle\ \Downarrow_o\ \langle \sigma_o',\ \psi_1\rangle\textit{, and } \langle s,\ \sigma_r\rangle\ \Downarrow_r\ \langle \sigma_r',\ \psi_2\rangle \textit{ then}$$
$$\Gamma \vdash \psi_1 \sim \psi_2$$

This theorem establishes that given a proof in the relaxed axiomatic semantics, if the original evaluation of the program terminates successfully and the relaxed evaluation of the program terminates successfully, then the observation lists generated by the executions (ψ_1 and ψ_2, respectively) satisfy the *acceptability relation*:

$$\Gamma \vdash \psi_1 \sim \psi_2.$$

Figure 11 presents the definition of the acceptability relation. The symbol Γ represents a finite map from `accept` statement labels to relational boolean expressions (i.e, $\Gamma \in L \rightarrow B^*$). We define this map by structural induction on the syntax of the program, where the label of each `accept` statement in the program maps to its relational boolean expression. We require that `accept` statements in well-formed programs are uniquely labeled.

$$\boxed{\Gamma \vdash \psi_1 \sim \psi_2} \qquad \frac{}{\Gamma \vdash \emptyset \sim \emptyset}$$

$$\frac{l_1 = l_2 \quad [\![\Gamma(l_1)]\!](\sigma_1, \sigma_2) = true \quad \Gamma \vdash \psi_1 \sim \psi_2}{\Gamma \vdash (l_1, \sigma_1) :: \psi_1 \sim (l_2, \sigma_2) :: \psi_2}$$

Figure 11. Acceptability Relation

The rules specify that if two observations lists are empty, then they are acceptable. Otherwise, for any two lists, the two lists are acceptable if 1) the labels in the head are the same (indicating that they are generated by the same `accept` statement), 2) the relational boolean expression for the label evaluates to true for the states in the head, 3) and the tails of the two lists are acceptable.

Proof Sketch. This proof proceeds by induction on the rules of the definition. For the diverge rule, the proof uses a sublemma that if $no\_acc(s)$ holds and $\langle s,\ \sigma_o\rangle \Downarrow_o \langle \sigma'_o,\ \psi_o\rangle$ (alternatively, $\langle s,\ \sigma_r\rangle \Downarrow_r \langle \sigma'_o,\ \psi_r\rangle$), then $\psi_o = \emptyset$ (alternatively, $\psi_r = \emptyset$). This proves that the observations lists are trivially acceptable. In the case of the accept rule, the proof uses the rule's condition to establish that the two emitted observations satisfy the `accept` statement's condition. The proof also uses Lemma 4.5 in the proofs for the sequential composition statement and the while statement. In addition, the proof for the while statement proceeds by nested, mutual induction on the evaluation derivations for the while statement in both the dynamic original and dynamic relaxed semantics. The proof consists of approximately 109 lines of Coq proof script. □

5. Example Programs

Inspired by programs that have been successfully relaxed in other research programs, we developed several example programs designed to capture the core aspects of the successful relaxations. We then formalized key correctness properties of the relaxations and used our Coq formalization to prove these properties.

5.1 Swish++

Swish++ is an open-source search engine. We work with a successful relaxation that uses Dynamic Knobs to reduce the number of search results that Swish++ presents to the user [15] when the server is under heavy load. The rationale for this relaxation is that 1) in most cases, users are interested only in the top search results and 2) users are very sensitive to how quickly the results are presented — even a short delay can significantly reduce revenue from advertisements [15, 33].

Relaxation. The transformation targets a loop that formats and presents the search query results. The loop keeps track of the number of search results, which we denote by `N`. The loop also has a control variable `max_r` which is a threshold on the number of elements that should be presented to the user: if `N` is smaller than `max_r`, then all results will be presented; otherwise, only the first `max_r` results will be presented.

A relaxed program can nondeterministically change `max_r` to reduce the number of iterations of this loop while still returning the most important results:

```
original_max_r = max_r;
relax (max_r) st
  (original_max_r <= 10 && max_r == original_max_r)
  || (10 < original_max_r && 10 <= max_r);
```

This code first saves the original value of the control variable `max_r` in `original_max_r`. It then relaxes `max_r`. There are two cases: if the original value of this control variable was less than or equal to 10, then the relaxed execution should be the same as the original execution — it presents the same number of results, since the value of `max_r` does not change. If, on the other hand, the original value was greater than 10, the only constraint is that the value of `max_r` is not smaller than 10, meaning that it should return at least the top 10 results when available. The `relax` statement nondeterministically changes `max_r` subject to these constraints.

Correctness. One acceptability property is that the relaxed execution must present either all of the search results from the original execution to the user (if the number of search results in the original execution is less than or equal to 10), or at least the first 10 results (if the number of results in the original execution is greater than 10). The following `accept` statement captures these constraints:

```
accept (num_r<o> < 10 && num_r<o> = num_r<r>) ||
       (10 <= num_r<o> && 10 <= num_r<r>);
```

The loop that formats and presents the search results maintains a count `num_r` of the number of formatted and presented results. This statement therefore uses the value of `num_r` in the original program (denoted `num_r<o>`) to determine how many search results the original execution presents. The `accept` statement uses `num_r<o>` and the (potentially different) value of `num_r` in the relaxed execution (`num_r<r>`) to formalize the desired correctness relationship between the two executions.

Verification. The proof of the `accept` statement involves 330 lines of Coq proof scripts. Because the relaxation changes the number of loop iterations, the proof uses the divergent control flow rule to reason about the loop in the original semantics and relaxed semantics separately. The key proof steps establish that the condition of the `relax` statement holds before entering the loop and that `original_max_r<o> = original_max_r<r>` and `N<o> = N<r>`. The loop invariant in both the original and relaxed execution is `num_r <= max_r /\ num_r <= N`.

Once control flow converges after the loop, the accept statement can, conceptually, be deduced via a proof by cases. Our proof environment reflects the accept statement's proof obligation into a formula that can be discharged by the decision procedure for Presburger arithmetic that is available in the Coq standard library.

5.2 Water Parallelization

Our next example is drawn from a parallelization of the Water computation [8] with statistical accuracy bounds [19]. In this computation a control variable determines whether to execute a loop sequentially or in parallel. To maximize performance, the parallelization eliminates lock operations that make updates to the `RS` variable execute atomically. The resulting race conditions produce a parallel computation whose result may vary nondeterministically (because of processor scheduling variations) within acceptable accuracy bounds [19].

Relaxation. We model the relaxation nondeterminism by relaxing the `RS` variable with no constraints:

```
relax (RS) st (true);
```

The Water computation compares `RS` to a cutoff variable `gCUT2` and, if it is less than the cutoff, uses `RS` to update an array `FF` (here `EXP(RS)` is an expression involving `RS`):

```
while (K < N) {
  relax (RS) st (true);
  if (RS < gCUT2) { FF[K]= EXP(RS); }
  K = K + 1;
}
```

Correctness. A key correctness property is that K stays within the bounds of the array FF. The array bounds are stored in the variable len_FF. We assume that the developer establishes, via some standard reasoning process, that the original execution does not violate the array bounds. The developer therefore inserts the statement assume (K < len_FF) just before the assignment to FF[K].

Verification. Recall that the verification of the relaxed program must verify that the condition in each assume statement holds in the relaxed execution. One approach is noninterference — verify that the relaxation does not affect the values of the variables in the predicate.

However, this is a relational property and because the assume statement appears at a divergent control flow point (it is affected by the value of the relaxed variable RS), this approach does not work.

The developer therefore inserts another assume statement, assume (K < len_FF), just before the if statement. It is possible to verify this statement using noninterference and then propagate the condition through the if statement to verify the second assume statement.

The Coq verification of this program consists of approximately 310 lines of proof script. The key prerequisite of the proof is to establish the relational loop invariants that K<o> = K<r> and len_FF<o> = len_FF<r>. These invariants enable us to prove that the relaxation does not interfere with the assumption.

5.3 LU Decomposition

Our third example is drawn from the LU decomposition algorithm implemented in the SciMark2 benchmark suite [2]. Researchers have demonstrated that lower-power, approximate memories and CPU compute units can be used to lower the energy consumption of this computation at the expense of a small loss in accuracy [31].

We focus on the part of the computation that computes the pivot row p for each column j in a matrix A. The pivot row is the row that contains the maximum element in the column.

```
i = j + 1;
while ( i < N ) {
  a = A[i][j];
  if (a > max) { max = a; p = i; }
  i = i + 1;
}
```

Relaxation. Following the assumptions on errors in approximate memories described in [24], if A is stored in approximate memory, then we can model the range of errors when reading a value from A with a relaxation that non-deterministically adds bounded error (e) to the result:

```
original_a = a;
relax (a) st (a >= original_a - e &&
              a <= original_a + e);
```

Correctness. One acceptability property for this computation is that the value in the selected pivot row (max) in the relaxed execution does not differ from the result in an original execution by more than e. We can specify this with an accept statement:

```
accept max<o> - max<r> <= e && max<r> - max<o> <= e
```

We note that this acceptability statement also corresponds to the notion of the *Lipschitz-continuity* [10] of the computation. Namely, small changes in the inputs lead to small changes in the output.

Verification. The Coq verification of this program consists of approximately 315 lines of proof script. The key prerequisite of the proof is to establish that the condition in the accept statement is loop invariant.

6. Related Work

Executable Specifications. Executable specifications, via techniques such as refinement and constraint solving, produce concrete outputs that satisfy the specification [12, 18, 22, 26, 30, 35] Applications include recovering from errors in existing code and providing alternate implementations for code that may be difficult to develop using standard techniques.

The research in this paper differs in that it promotes nondeterministic relaxation to obtain semantically different but still acceptable variants of the original program. A focus is therefore enabling developers to specify and prove correctness requirements as relational properties between the original and relaxed program.

Unreliable Memory and Critical Data. Researchers have proposed techniques for enabling programs to distinguish data that can be stored in unreliable low-power memory from critical data whose values must be reliably stored [9, 17, 31]. These systems typically focus on data values (such as the values of pixels in an image) that can, in principle, legally take on *any* value. The techniques presented in this paper make it possible to prove specific correctness properties of these kinds of programs (whose relaxations may primarily involve unreliably stored data that can take on any value). They also support programs with control variables whose values must satisfy specific properties identified in relax statements.

Relational Program Logics. Our program logic for the relaxed semantics of the program builds on previous work on the Relational Hoare Logic (RHL) [7]. RHL itself was inspired by work on Translation Validation [25] and Credible Compilation [29] and has since inspired other forms of relational reasoning about programs. Researchers have also defined relational separation logic [5, 34], probabilistic Hoare logic [6], and have used relational reasoning to verify the correctness of semantics-preserving loop optimizations [11], Lipschitz-continuity [11], access control policies [23], and differential privacy mechanisms [6].

While majority of the previous research has been focused on proving that transformed programs retain the semantics of the original version, our goal is different — specifically, to prove that relaxed executions (which typically have different semantics) preserve important correctness properties. We adapt RHL to prove properties that relate the original and relaxed executions and extend RHL to reason about assertions (which reference only the current execution) and assumptions (which are assumed to hold in original executions but must be shown to hold in relaxed executions).

7. Conclusion

The additional nondeterminism in relaxed programs enables programs to operate at a variety of points with different combinations of accuracy, performance, and resource consumption characteristics. It is possible to exploit this flexibility to satisfy a variety of goals, including trading off accuracy for enhanced performance or reduced energy consumption [3, 4, 10, 15–17, 19–21, 28, 31, 32, 36] or responding to load spikes or other fluctuations in the characteristics of the underlying computational platform [15, 16, 27, 31].

We present formal reasoning techniques that make it possible to verify important correctness and acceptability properties of relaxed programs. Standard verification techniques reference only the current execution of the current program under verification. Our techniques, in contrast, aim to reduce the verification effort by taking a relational approach that exploits the close relationship between the original and relaxed executions. Our goal is to give developers the verified correctness and acceptability properties they need to confidently deploy relaxed programs and exploit the substantial flexibility, performance, and resource consumption advantages that relaxed programs offer.

References

[1] The Coq Proof Assistant. http://coq.inria.fr.

[2] Scimark 2.0. http://math.nist.gov/scimark2/.

[3] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: a language and compiler for algorithmic choice. PLDI, 2009.

[4] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. PLDI, 2010.

[5] G. Barthe, J. Crespo, and C. Kunz. Relational verification using product programs. 2011.

[6] G. Barthe, B. Köpf, F. Olmedo, and S. Zanella Béguelin. Probabilistic reasoning for differential privacy. In *POPL*, 2012.

[7] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. POPL, 2004.

[8] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *TOPDS*, 3(6), 1992.

[9] M. Carbin and M. Rinard. Automatically Identifying Critical Input Regions and Code in Applications. ISSTA, 2010.

[10] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving Programs Robust. FSE, 2011.

[11] J.M. Crespo and C. Kunz. A machine-checked framework for relational separation logic. SEFM, 2011.

[12] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. ICSE, 2005.

[13] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19, 1967.

[14] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.

[15] H. Hoffman, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. ASPLOS, 2011.

[16] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard. Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures . Technical Report MIT-CSAIL-TR-2009-042, MIT, 2009.

[17] S. Liu, K. Pattabiraman, T. Moscibroda, and B. Zorn. Flikker: Saving dram refresh-power through critical data partitioning. ASPLOS, 2011.

[18] A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson. Unifying execution of imperative and declarative code. ICSE, 2011.

[19] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. Technical Report MIT-CSAIL-TR-2010-038, MIT, 2010.

[20] S. Misailovic, D. Roy, and M. Rinard. Probabilistically Accurate Program Transformations. SAS, 2011.

[21] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. ICSE, 2010.

[22] C. Morgan. The specification statement. *TOPLAS*, (3), Jul 1988.

[23] A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. SP, 2011.

[24] J. Nelson, A. Sampson, and L. Ceze. Dense approximate storage in phase-change memory. ASPLOS-WACI.

[25] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. TACAS, 1998.

[26] D. Rayside, A. Milicevic, K. Yessenov, G. Dennis, and D. Jackson. Agile specifications. OOPSLA, 2009.

[27] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. ICS, 2006.

[28] M. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. OOPSLA, 2007.

[29] M. C. Rinard and D. Marinov. Credible compilation with pointers. RTRV, 1999.

[30] H. Samimi, E. Aung, and T. Millstein. Falling back on executable specifications. ECOOP, 2010.

[31] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: approximate data types for safe and general low-power computation. PLDI, 2011.

[32] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard. Managing Performance vs. Accuracy Trade-offs With Loop Perforation. FSE '11.

[33] Steve Souders. Velocity and the bottom line. 2009.

[34] H Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, May 2007.

[35] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. POPL, 2012.

[36] Z. Zhu, S. Misailovic, J. Kelner, and M. Rinard. Optimal accuracy-aware transformations for approximate computations. POPL, 2012.