Proving Acceptability Properties of Relaxed Nondeterministic Approximate 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 skipping tasks [29, 30], loop perforation [21, 22, 35], reduction sampling [38], multiple selectable implementations [3, 4, 16, 38], dynamic knobs [16], synchronization elimination [20, 32], approximate function memoization [11], and approximate data types [34] produce programs that can execute at a variety of points in an underlying performance versus accuracy tradeoff space. These transformed programs have the ability to 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 because they have been extended with additional nondeterminism to relax their semantics and enable greater flexibility in their execution.

We present language constructs for developing and specifying relaxed programs. We also present proof rules for reasoning about *acceptability properties* [28], which the program must satisfy to be acceptable. Our proof rules work with two kinds of acceptability properties: *relational* acceptability properties, which characterize desired relationships between the values of variables in the original and relaxed programs, and *unary* acceptability properties, which involve values only from a single (original or relaxed) program. The proof rules support a *staged* reasoning approach in which the majority of the reasoning effort works with the original program. Exploiting the common structure that the original and relaxed programs share, relational reasoning transfers reasoning effort from the original program to prove properties of the relaxed program.

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.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Nondeterministic languages; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

General Terms Languages, Performance, Theory, Verification

Keywords Coq, Acceptability, Relaxed Programs, Relational Hoare Logic

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) [16, 17]. Specific mechanisms include skipping tasks [29, 30], loop perforation (skipping iterations of time-consuming loops) [21, 22, 35], sampling reduction inputs [38], multiple selectable implementations of a given component or components [3, 4, 16, 38], dynamic knobs (configuration parameters that can be changed as the program executes) [16], synchronization elimination (forgoing synchronization not required to produce an acceptably accurate result) [20, 32], approximate function memoization (returning a previously computed value when the arguments of a function call are close to the arguments of a previous call) [11], and approximate data types [32, 34].

All of these mechanisms can produce a *relaxed program* — a program that may adjust its execution by changing one or more variables subject to a specified *relaxation predicate*. For example, a perforated program may dynamically choose to skip loop iterations each time it enters a given loop. A relaxed program is therefore a nondeterministic program, with each execution a variant of the original execution. The different executions typically share a common global structure, with local differences at only those parts of the computation affected by the modified variables.

1.1 Acceptability

To use these relaxation mechanisms, a developer must ensure that the resulting relaxed program is acceptable. We formalize acceptability as *acceptability properties* that the program must satisfy to be acceptable. Acceptability properties include *integrity properties* that the program must satisfy to successfully produce a result and *accuracy properties* that characterize how accurate the produced result must be. For example, an integrity property might state that a function must return a value greater than zero (otherwise the program might crash), while an accuracy property might state that the relaxed program must (potentially only with high probability) produce a result that differs by at most a specified percentage from the result that the original program produces [21, 38].

1.2 Reasoning About Relaxed Programs

We present language constructs for developing relaxed programs and stating acceptability properties. We also present proof rules for verifying the stated acceptability properties. Our language and proof rules support a staged approach in which the developer first develops a standard program and uses standard approaches to reason about this program to determine that it satisfies desired acceptability 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 additional nondeterministic executions. We refer to the dynamic semantics of the program at this stage as the *relaxed semantics* of the program.

Finally, the developer uses *relational reasoning* to verify that the relaxation maintains the desired acceptability properties. Specifically, the developer specifies and verifies additional *relational assertions* that characterize the relationship between the original and relaxed semantics. These relational assertions facilitate the overall verification of the relaxed program. This approach is designed to reduce the overall reasoning effort by exploiting the common structure that the original and relaxed programs share. With this approach the majority of the reasoning effort works with the original program and is then transferred via relational reasoning to verify the nondeterministic relaxed program.

1.3 Relaxed Programming Constructs

Basic relaxed programming constructs include nondeterministic variable assignments (via the `relax` statement), relational assertions that relate the relaxed semantics to the original semantics (via the `relate` statement), unary assertions (via the `assert` statement), and unary assumptions (via the `assume` statement).

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`. When added to the program, the `relax` statement affects only the relaxed semantics of the program; in the original semantics it has no effect.

The Relate Statement. The `relate P` statement asserts that the predicate `P` must hold at the program point where the statement appears. The predicate `P` is a *relational predicate* — it may reference values from both the original and relaxed executions. So, for example, the statement might require the value of a variable `x` in relaxed executions to be greater than or equal to the value of `x` in the original execution.

The Assert Statement. The `assert P` statement states that `P` must hold at the point where the statement appears. In contrast to the `relate` statement, `P` is a *unary predicate* — it only references values from a single execution (original or relaxed) as in a standard assertion. In the original semantics, our proof rules generate an obligation to prove that an `assert` statement holds for all executions. To ensure that the `assert` statement remains valid in the relaxed semantics, our proof rule in the relaxed semantics generates an obligation to prove that if the assertion is valid in the original semantics, then the current relation between the original and relaxed semantics establishes that the assertion is valid in the relaxed semantics. For example, it may be possible to prove that all the variables referenced in an assertion have the same values in the original and relaxed semantics — i.e., the relaxation does not interfere with the assertion. In this way, relational reasoning serves as a bridge to transfer properties of the original program over to the relaxed program.

The Assume Statement. The `assume P` statement states that the unary predicate `P` holds at the point where the statement appears. In the original semantics the `assume` statement does not generate any proof obligations — the proof system simply accepts that `P` holds. To verify that the relaxation does not interfere with the reasoning behind the assumption, the proof rules for the relaxed semantics generate an obligation to prove that if the assumption `P` holds in all original executions, then it holds in all relaxed executions. The proofs work in much the same way as for the `assert` statement except that the proof rules do not generate an obligation to verify that `P` holds in the original semantics.

1.4 Key Properties of Acceptable Relaxed Programs

Our approach makes it possible to formalize key properties that are critical to the development and deployment of acceptable relaxed programs.

Integrity and Noninterference. Essentially all programs have basic integrity properties that must hold for all executions of the program. Examples include the absence of out of bounds array accesses or null pointer dereferences. Developers typically use either `assert` or `assume` statements to formalize these integrity properties. Because successful relaxations do not typically interfere with the basic integrity of the original program, the reasoning that establishes the validity of the integrity properties typically transfers directly from the original program over to the relaxed program. Relational assertions that establish the equality of values of variables in the original and relaxed executions (i.e., *noninterference*) often form the bridge that enables this direct transfer (see Section 5).

Accuracy. Relaxed programs exploit the freedom of the computation to, within limits, produce a range of different outputs. Accuracy properties formalize how accurate the outputs must be to stay within the acceptable range. For example, a perforatable loop may produce a range of acceptable results, with (typically depending on the amount of perforation) some more accurate than others. Because it is often convenient to express accuracy requirements by bounding the difference between results from the original and relaxed executions, developers can use `relate` statements to express accuracy properties (see Section 5).

Debuggability. The `assume` statement provides developers with the ability to state (formally unverified) assumptions that they believe to be true in the original program. But if an assumption is not valid, the program may fail or exhibit unintended behaviors.

Relaxation without verification can therefore complicate debugging — it may cause the relaxed program to violate assumptions that are valid in the original program (and therefore to exhibit unintended behaviors that are not possible in the original program). Our proof rules, by ensuring that if the assumption is valid in the original program, then it remains valid in the relaxed program, simplify debugging by eliminating this potential source of unintended behaviors.

Note that our proof rules can work together effectively to prove important acceptability properties. For example, the developer may use a `relate` statement to establish a relationship between values in variables in the original and relaxed executions, then use this relationship to prove that the property specified by an `assert` or `assume` statement holds in all relaxed executions.

1.5 Proof Rules and Formal Properties

We structure our proof rules as a set of Hoare logics:

- **Axiomatic Original Semantics:** The *original* Hoare-style semantics models the original execution of the program wherein `relax` statements have no effect.
- **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 values of variables in both the original and relaxed executions of the program. A proof with the axiomatic relaxed semantics relates the two semantics of the program in lockstep and, therefore, supports the transfer of reasoning about the original semantics to the relaxed semantics by enabling, for example, noninterference proofs.

One key aspect of the axiomatic relaxed semantics is that it must also give an appropriate semantics for relaxed programs in which the original and relaxed executions may branch in different directions at a control flow construct (at this point the two executions are no longer in lockstep). In particular, we do not support relational reasoning for program points at which the executions are no longer in lockstep (`relate` statements do not have a natural semantics at such program points). Our relaxed semantics therefore incorporates a nonrelational axiomatic *intermediate semantics* that captures the desired behavior of the relaxed execution as it executes without a corresponding original execution. We also appropriately restrict the location of `relate` statements to program points at which the original and relaxed programs execute in lockstep.

Our proof rules are sound and establish the following semantic properties of verified relaxed programs:

- **Original Progress Modulo Assumptions:** If the program verifies under the axiomatic original semantics, then no execution of the program in the dynamic original semantics violates an assertion. By design, a program may still terminate in error if a specified `assume` statement is not valid.
- **Soundness of Relational Assertions:** If the program verifies under the axiomatic relaxed semantics, then all pairs of executions in the dynamic original and relaxed semantics satisfy the `relate` statements in the program.
- **Relative Relaxed Progress:** If the program verifies under the axiomatic relaxed semantics and no executions in the dynamic original semantics violate an assertion or an assumption, then no execution in the dynamic relaxed semantics violates an assertion or an assumption.
- **Relaxed Progress:** If the program verifies under both the original and relaxed axiomatic semantics and no execution in the dynamic original semantics violates an assumption, then no execution in the dynamic relaxed semantics violates an assertion or an assumption.
- **Relaxed Progress Modulo Original Assumptions:** If the program verifies under both the original and relaxed axiomatic semantics, then if an execution in the dynamic relaxed semantics violates an assertion or an assumption, then an execution in the dynamic original semantics violates an assumption.

These properties assure developers that verified relaxation produces a program that satisfies the stated acceptability properties. Our design supports a development process in which developers can use the full range of standard techniques (verification, testing, code reviews) to validate properties that they believe to be true of the original program. They can then use `assume` statements to formally state these properties and incorporate them (via relational reasoning) into the verification of the relaxed program. This verification ensures that if the relaxed program fails because of a violated assumption, then the developer can reproduce the violated assumption in the original program.

1.6 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 our proof rules 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 small relaxed programs.

Our Coq implementation contains approximately 8000 lines of code and proof scripts, with 1300 lines devoted to the original semantics and its soundness proofs and 1900 additional lines devoted to the relaxed semantics and its soundness proofs. A large portion (approximately 3500 lines) is devoted to formalizing the semantics of our relational assertion logic and its soundness with respect to operations such as substitution.

1.7 Contributions

This paper makes the following contributions:

- **Relaxed Programming:** It identifies the concept of relaxed programming as a way to specify nondeterministic variants of an original 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 skipping tasks [29, 30], loop perforation [21, 22, 35], reduction sampling [38], multiple selectable implementations [3, 4, 16, 38], dynamic knobs [16], synchronization elimination [20, 32], approximate function memoization [11], and approximate data types [34].
- **Relational Reasoning and Proof Rules:** It presents a basic reasoning approach and proof rules for verifying acceptability properties of relaxed programs. With this approach, the majority of the reasoning effort works with the original program and is transferred to the relaxed program through relational reasoning.
- **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 original and relaxed 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, 31, 36, 39].
- **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 have confidence that the relaxation satisfies important acceptability properties. This paper presents a foundational formal reasoning system that leverages the structure and relationships 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.

A main point of departure from standard languages 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 ($\mathtt{x}\langle\mathtt{o}\rangle$) and

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

Figure 1. Language Syntax

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

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

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

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

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

Figure 2. Semantics of Expressions

relaxed ($\texttt{x}\langle\texttt{r}\rangle$) executions. These relational expressions enable `relate` statements to specify relationships that must hold between the original and relaxed executions. For example, the statement $\texttt{relate l :} \ \ \texttt{x}\langle\texttt{o}\rangle \ = \ \texttt{x}\langle\texttt{r}\rangle$ asserts that at the current program point (with label `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 $\Sigma = Vars \xrightarrow{fin} \mathbb{Z}$, which is the set of all finite maps from variables to integers.

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 $\langle s, \ \sigma\rangle \Downarrow_o \phi$ denotes that evaluating the statement s in the 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 ("bad assume") denotes that the program has failed at an `assume` statement in the program. The distinguished element wr ("wrong") denotes that the program has failed due to another error, such as an unsatisfied `assert` statement.

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 `relate` 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 `relate` statements in the program — the execution of each `relate` 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.

Evaluation Rules

The rules for `skip`, assignment, `if`, sequential composition, and `while` statements follow the standard semantics for these constructs. The semantics for nonstandard constructs are as follows:

Havoc. The `havoc` (X) `st` (e) statement nondeterministically assigns values to the set of variables in X such that their values satisfy the statement's predicate 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.

Assert. The `assert` e statement checks that the state satisfies its predicate e. If e evaluates to *true*, then execution continues; otherwise, the statement evaluates to wr.

Assume. The `assume` e statement checks that the state satisfies its predicate e. If e evaluates to *true*, then execution continues; otherwise, the statement evaluates to ba.

Relax. The `relax` (X) `st` (e) statement does not modify the state of the program in the original semantics. Because we require the

$$\boxed{\langle s,\ \sigma\rangle \Downarrow_o\ \phi} \qquad \text{skip}\ \frac{}{\langle \texttt{skip},\ \sigma\rangle \Downarrow_o\ \langle \sigma,\ \emptyset\rangle} \qquad \text{assign}\ \frac{\langle e\ ,\ \sigma\rangle \Downarrow_E n}{\langle x\ \texttt{=}\ e,\ \sigma\rangle \Downarrow_o\ \langle \sigma[x \mapsto n],\ \emptyset\rangle}$$

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

$$\text{assert-t}\ \frac{\langle e\ ,\ \sigma\rangle \Downarrow_B \mathit{true}}{\langle \texttt{assert}\ e,\ \sigma\rangle \Downarrow_o\ \langle \sigma,\ \emptyset\rangle} \qquad \text{assert-f}\ \frac{\langle e\ ,\ \sigma\rangle \Downarrow_B \mathit{false}}{\langle \texttt{assert}\ e,\ \sigma\rangle \Downarrow_o\ \mathit{wr}} \qquad \text{assume-t}\ \frac{\langle e\ ,\ \sigma\rangle \Downarrow_B \mathit{true}}{\langle \texttt{assume}\ e,\ \sigma\rangle \Downarrow_o\ \langle \sigma,\ \emptyset\rangle}$$

$$\text{assume-f}\ \frac{\langle e\ ,\ \sigma\rangle \Downarrow_B \mathit{false}}{\langle \texttt{assume}\ e,\ \sigma\rangle \Downarrow_o\ \mathit{ba}} \qquad \text{relax}\ \frac{\langle \texttt{assert}\ e,\ \sigma\rangle \Downarrow_o\ \phi}{\langle \texttt{relax}\ (X)\ \texttt{st}\ (e),\ \sigma\rangle \Downarrow_o\ \phi} \qquad \text{relate}\ \frac{}{\langle \texttt{relate}\ l:\ \ e^*,\ \sigma\rangle \Downarrow_o\ \langle \sigma,\ (l, \sigma)\rangle}$$

$$\text{if-t}\ \frac{\langle b\ ,\ \sigma\rangle \Downarrow_B \mathit{true} \quad \langle s_1,\ \sigma\rangle \Downarrow_o\ \phi}{\langle \texttt{if}\ (b)\ \{s_1\}\ \texttt{else}\ \{s_2\},\ \sigma\rangle \Downarrow_o\ \phi} \qquad \text{if-f}\ \frac{\langle b\ ,\ \sigma\rangle \Downarrow_B \mathit{false} \quad \langle s_2,\ \sigma\rangle \Downarrow_o\ \phi}{\langle \texttt{if}\ (b)\ \{s_1\}\ \texttt{else}\ \{s_2\},\ \sigma\rangle \Downarrow_o\ \phi} \qquad \text{seq}\ \frac{\langle s_1,\ \sigma\rangle \Downarrow_o\ \langle \sigma',\ \psi_1\rangle \quad \langle s_2,\ \sigma'\rangle \Downarrow_o\ \langle \sigma'',\ \psi_2\rangle}{\langle s_1\ ;\ s_2,\ \sigma\rangle \Downarrow_o\ \langle \sigma'',\ \psi_2.\psi_1\rangle}$$

$$\text{while-f}\ \frac{\langle b\ ,\ \sigma\rangle \Downarrow_B \mathit{false}}{\langle \texttt{while}\ (b)\ \{s\},\ \sigma\rangle \Downarrow_o\ \langle \sigma,\ \emptyset\rangle} \qquad \text{while-t}\ \frac{\langle b\ ,\ \sigma\rangle \Downarrow_B \mathit{true} \quad \langle s,\ \sigma\rangle \Downarrow_o\ \langle \sigma',\ \psi_1\rangle \quad \langle \texttt{while}\ (b)\ \{s\},\ \sigma'\rangle \Downarrow_o\ \langle \sigma'',\ \psi_2\rangle}{\langle \texttt{while}\ (b)\ \{s\},\ \sigma\rangle \Downarrow_o\ \langle \sigma'',\ \psi_2.\psi_1\rangle}$$

Figure 3. Dynamic Original Semantics

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

Figure 4. Dynamic Relaxed Semantics

original execution to be one of the relaxed executions, the dynamic original semantics requires the relaxation predicate e to hold in the original execution.

Relate. The $\texttt{relate}\ l:\ \ e^*$ statement is a relational assertion over original and relaxed executions of the program. The dynamic semantics emits an observation consisting of the statement's label l along with the current state of the program. This semantics enables us to define and verify a relation on the observation lists emitted by the original and relaxed programs (see Section 4).

We omit the standard rules for propagating errors (ba and wr) through the program. However, the reader can refer to our previous technical report for further details [9].

2.3 Dynamic Relaxed Semantics

Figure 4 presents an abbreviated version of the dynamic relaxed semantics. The relation $\langle s,\ \sigma\rangle \Downarrow_r\ \phi$ denotes that evaluating the statement s in the state σ yields the output configuration ϕ.

The dynamic relaxed semantics builds upon the original semantics. It differs only in that `relax` statements modify the state of the program. We therefore omit 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`).

Relax. The $\texttt{relax}\ (X)\ \texttt{st}\ (e)$ statement nondeterministically modifies the state of the program in the relaxed semantics so that it satisfies the statement's predicate e. The rule implements the modification by reusing the rule for `havoc` statements.

3. Axiomatic Semantics

We next present axiomatic semantics for relaxed programs.

- **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 (i.e., evaluates to wr). However, the program may dynamically violate an assumption (i.e., evaluate to ba).

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

Figure 5. Relational Assertion Logic Syntax

$$\begin{aligned} [\![P]\!] &\in \mathcal{P}(\Sigma) \\ [\![\texttt{true}]\!] &= \Sigma \qquad [\![\texttt{false}]\!] = \emptyset \\ [\![E_1\ \texttt{cmp}\ E_2]\!] &= \{\sigma \mid [\![E_1]\!](\sigma)\ \mathit{cmp}\ [\![E_2]\!](\sigma)\} \\ [\![P_1\ \texttt{lop}\ P_2\ ]\!] &= \{\sigma \mid \sigma \in [\![P_1]\!]\ \mathit{lop}\ \sigma \in [\![P_2]\!]\} \\ [\![\neg P]\!] &= [\![\texttt{true}]\!] \setminus [\![P]\!] \\ [\![\exists_{\texttt{x}} \cdot P]\!] &= \{\sigma \mid n \in \mathbb{Z},\ \sigma \in [\![P[n/x]]\!]\} \end{aligned}$$

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

Figure 6. Relational Assertion Logic Semantics

- **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 all executions in the original semantics execute without error (i.e., do not evaluate to wr or ba), then all executions in the relaxed semantics execute without error. A proof with these rules also guarantees that pairs of original and relaxed executions satisfy all of the `relate` statements in the program.

3.1 Relational Assertion Logic

Figure 5 presents the concrete syntax of our *relational* assertion logic. This logic extends a nonrelational assertion logic with relational formulas, which then allows us to reason about the validity of relational boolean expressions in `relate` statements. Its presentation follows the style of Benton's Relational Hoare Logic [7].

$$\boxed{\vdash_o \{P\}\ s\ \{Q\}} \qquad \text{skip}\,\frac{}{\vdash_o \{P\}\ \texttt{skip}\ \{P\}} \qquad \text{seq}\,\frac{\vdash_o \{P\}\ s_1\ \{R\} \quad \vdash_o \{R\}\ s_2\ \{Q\}}{\vdash_o \{P\}\ s_1; s_2\ \{Q\}} \qquad \text{assign}\,\frac{}{\vdash_o \{Q[e/x]\}\ \texttt{x = }e\ \{Q\}}$$

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

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

$$\text{relate}\,\frac{}{\vdash_o \{P\}\ \texttt{relate}\ l:\ \ e^*\ \{P\}} \qquad \text{while}\,\frac{\vdash_o \{P \wedge b\}\ s\ \{P\}}{\vdash_o \{P\}\ \texttt{while}\ (b)\ \{s\}\ \{P \wedge \neg b\}} \qquad \text{conseq}\,\frac{\models P \Rightarrow P' \quad \vdash_o \{P'\}\ s\ \{Q'\} \quad \models Q' \Rightarrow Q}{\vdash_o \{P\}\ s\ \{Q\}}$$

Figure 7. Axiomatic Original Semantics

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 formulas to refer to relational integer expressions.

3.1.2 Semantics

Figure 6 presents the semantics of formulas in the logic. The denotation of a nonrelational formula $[\![P]\!]$ is the set of states that satisfy the formula. $[\![P]\!]$ reuses the semantic definitions for integer expressions from Figure 2 to construct a definition for each formula. The denotation of a relational formula $[\![P^*]\!]$ closely follows that of nonrelational formulas: it is the set of pairs of states that satisfy the relation. References to the original semantics (e.g., $\texttt{x}\langle\texttt{o}\rangle$) refer to the first component of the pair and references to the relaxed semantics (e.g., $\texttt{x}\langle\texttt{r}\rangle$) refer to the second component.

Injections. We define injection functions $inj_o(P)$ and $inj_r(P)$, which construct a relational formula in P^* from a nonrelational formula in P. Conceptually, $inj_o(P)$ constructs a relational formula where P holds for the original semantics by replacing variables (e.g., $\texttt{x}$) in P with the relational original variables (e.g., $\texttt{x}\langle\texttt{o}\rangle$); $inj_r(P)$ does the same with the relational relaxed variables such that P holds for the relaxed semantics. This means that $inj_o(P)$ (resp. $inj_r(P)$) creates a formula representing all state pairs where the first (resp. second) component satisfies P:

$$[\![inj_o(P)]\!] = \{(\sigma_1, \sigma_2) \mid \sigma_1 \in [\![P]\!]\}$$
$$[\![inj_r(P)]\!] = \{(\sigma_1, \sigma_2) \mid \sigma_2 \in [\![P]\!]\}$$

We also define the notation $\langle P_1 \cdot P_2 \rangle$ for combining a predicate P_1 over the original semantics with a predicate P_2 over the relaxed:

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

Projections. We define two semantic functions $prj_o(P^*)$ and $prj_r(P^*)$. Each function projects a relational formula in P^* to the set of states corresponding to either the first (prj_o) or second (prj_r) component of each state pair in the denotation of the formula:

$$prj_o(P^*) \equiv \{\sigma_1 \mid (\sigma_1, \sigma_2) \in [\![P^*]\!]\}$$
$$prj_r(P^*) \equiv \{\sigma_2 \mid (\sigma_1, \sigma_2) \in [\![P^*]\!]\}$$

The projection functions allow us to decompose a relational formula over the original and relaxed semantics into the set of states that satisfy the relation for either the original or relaxed semantics individually. We use projection to define the following relations between relational and nonrelational formulas:

$$P^* \models_o P \equiv prj_o(P^*) \subseteq [\![P]\!]$$
$$P^* \models_r P \equiv prj_r(P^*) \subseteq [\![P]\!]$$

Fresh Variables and Substitution. The predicate $\mathit{fresh}(x)$, denoting that x is a fresh variable in the context of an inference rule, is true if $x \in \mathit{Vars}$ and x does not appear in the rule's premises or consequent. Our proof rules also use the standard capture-avoiding substitution $P[e/x]$. We denote multiple substitution $P[e_1/x_1] \cdots [e_n/x_n]$ as $P[e_1, \cdots, e_n/x_1, \cdots, x_n]$. We define substitution over P^* similarly.

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

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

3.2 Original Semantics

Figure 7 presents a manual translation of our Coq formalization of the axiomatic original semantics of the program. The Hoare-style judgment $\vdash_o \{P\}\ s\ \{Q\}$ models the original execution of the program wherein relax statements have no effect. The intended meaning of the semantic judgment $\models_o \{P\}\ s\ \{Q\}$ is: for all states σ, if $\sigma \models P$ and $\langle s,\ \sigma\rangle \Downarrow_o \langle \sigma',\ \psi\rangle$, then $\sigma' \models Q$. In other words, if σ satisfies P and an original execution of s from σ yields a new state σ', then σ' satisfies Q. We note that this definition asserts only partial correctness and not total correctness.

We have elided a discussion of the rules for standard constructs (i.e., `skip`, assign, sequential composition, `if`, `while`, and consequence) because their definitions are the same as in standard presentations (e.g., Floyd [14] and Hoare [15]). We define the nonstandard rules as follows:

The havoc rule requires in its premise that e must be satisfiable by adjusting only the variables in X while retaining the values of other variables. A `havoc` statement evaluates to wr only if there is no way to transform the current state into a new state satisfying e just by changing the variables in X.

The assert rule requires the predicate e to hold in the precondition. The rule therefore requires a proof of e.

The assume rule differs from the assert rule in that it assumes the validity of e and then makes e part of the postcondition. 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 specifies that a `relax` statement is a no-op that does not change the program's state in the original semantics. Our definition of relaxation, however, requires that the original execution must still satisfy e. To enforce this constraint, the rule reuses the rule for the `assert` statement.

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

$$\text{diverge}\ \frac{P^* \models_o P_o \quad P^* \models_r P_r \quad \vdash_o \{P_o\}\ s\ \{Q_o\} \quad \vdash_i \{P_r\}\ s\ \{Q_r\} \quad \mathit{no\_rel}(s)}{\vdash_r \{P^*\}\ s\ \{\langle Q_o \cdot Q_r \rangle\}}$$

$$\text{relax}\ \frac{[\![(\exists_{X'\langle \mathtt{r}\rangle} \cdot P^*[X'\langle \mathtt{r}\rangle / X\langle \mathtt{r}\rangle]) \wedge \mathit{inj}_r(e)]\!] \neq \emptyset \quad \mathit{fresh}(X')}{\vdash_r \{P^*\}\ \mathtt{relax}\ (X)\ \mathtt{st}\ (e)\ \{(\exists_{X'\langle \mathtt{r}\rangle} \cdot P^*[X'\langle \mathtt{r}\rangle / X\langle \mathtt{r}\rangle]) \wedge \langle e \cdot e\rangle\}} \qquad \text{relate}\ \frac{}{\vdash_r \{P^* \wedge e^*\}\ \mathtt{relate}\ l:\ \ e^*\ \{P^* \wedge e^*\}}$$

$$\text{assert}\ \frac{\models P^* \wedge \mathit{inj}_o(e) \Rightarrow \mathit{inj}_r(e)}{\vdash_r \{P^*\}\ \mathtt{assert}\ e\ \{P^* \wedge \langle e \cdot e\rangle\}} \qquad \text{if}\ \frac{\models P^* \Rightarrow \langle b \cdot b\rangle \vee \langle \neg b \cdot \neg b\rangle \quad \vdash_r \{P^* \wedge \langle b \cdot b\rangle\}\ s_1\ \{Q^*\} \quad \vdash_r \{P^* \wedge \langle \neg b \cdot \neg b\rangle\}\ s_2\ \{Q^*\}}{\vdash_r \{P^*\}\ \mathtt{if}\ (b)\ \{s_1\}\ \mathtt{else}\ \{s_2\}\ \{Q^*\}}$$

$$\text{assume}\ \frac{\models P^* \wedge \mathit{inj}_o(e) \Rightarrow \mathit{inj}_r(e)}{\vdash_r \{P^*\}\ \mathtt{assume}\ e\ \{P^* \wedge \langle e \cdot e\rangle\}} \qquad \text{while}\ \frac{\models P^* \Rightarrow \langle b \cdot b\rangle \vee \langle \neg b \cdot \neg b\rangle \quad \vdash_r \{P^* \wedge \langle b \cdot b\rangle\}\ s\ \{P^*\}}{\vdash_r \{P^*\}\ \mathtt{while}\ (b)\ \{s\}\ \{P^* \wedge \langle \neg b \cdot \neg b\rangle\}}$$

Figure 8. Axiomatic Relaxed Semantics

The relate rule gives `relate` statements the same semantics as `skip` because, unlike the axiomatic relaxed semantics (see Section 3.3), the axiomatic original semantics references only a single execution of the program and does not use relational reasoning.

3.3 Relaxed Semantics

Figure 8 presents a manual translation of our Coq formalization of the axiomatic relaxed semantics of the program. The proof rules for the relaxed semantics are relational in that they relate executions of the program under the dynamic relaxed semantics to executions under the dynamic original semantics. The intended meaning of each judgment $\models_r \{P^*\}\ s\ \{Q^*\}$ is the partial correctness assertion: if $(\sigma_o, \sigma_r) \models P^*$, $\langle s,\ \sigma_o\rangle \Downarrow_o \langle \sigma_o',\ \psi_1\rangle$, and $\langle s,\ \sigma_r\rangle \Downarrow_r \langle \sigma_r',\ \psi_2\rangle$, then $(\sigma_o', \sigma_r') \models Q^*$.

The rules are designed to transfer the reasoning from the axiomatic original semantics to prove properties about the axiomatic relaxed semantics. Specifically, the axiomatic relaxed semantics need not re-prove properties about the dynamic original semantics (e.g., the validity of `assert` statements). It can instead simply assume that these properties are established by the axiomatic original semantics and then transfer their validity to the relaxed semantics via relational reasoning. We have elided discussions of most of the standard rules. The nonstandard rules operate as follows:

The relax rule distinguishes the semantics of relaxation in the original and relaxed semantics of the program. The rule is similar to the havoc rule in the axiomatic original semantics except that 1) it deals with a relational formula and 2) the rule only modifies (i.e., substitutes) relaxed variables, such as $\mathtt{x}\langle \mathtt{r}\rangle$, whereas variables over the original semantics $\mathtt{x}\langle \mathtt{o}\rangle$ are not modified. We use a shorthand $X\langle \mathtt{r}\rangle \equiv \{x\langle \mathtt{r}\rangle \mid x \in X\}$ to denote the syntactic extension of a set of variables in X to relaxed variables.

The relate rule enables us to reason about `relate` statements in the program. Similar to a nonrelational assertion, the rule requires e^* to hold in the precondition. This ensures that e^* holds for all pairs of original and relaxed executions that reach the statement.

The assert rule demonstrates how we can use relational reasoning to prove assertions in the relaxed semantics. Specifically, we can first assume that the assertion is true in the original semantics (i.e., $\mathit{inj}_o(e)$) because it has been verified with the axiomatic original semantics. If $P^* \wedge \mathit{inj}_o(e)$ implies $\mathit{inj}_r(e)$, we can then conclude that the assertion is true in the relaxed semantics (i.e., $\mathit{inj}_r(e)$).

For example, if the precondition of the statement `assert` e requires all of the free variables in e to be the same in both semantics (i.e., for all $\mathtt{x} \in \mathit{free}(e)$, $\mathtt{x}\langle \mathtt{o}\rangle$ `==` $\mathtt{x}\langle \mathtt{r}\rangle$), then because the axiomatic original semantics proves that the assertion is true in the dynamic original semantics, we can conclude that the assertion is also true in the dynamic relaxed semantics.

The assume rule demonstrates how relational reasoning allows us to use relations between the original and relaxed semantics of the

$$\boxed{\vdash_i \{P\}\ s\ \{Q\}} \qquad \text{relax}\ \frac{\vdash_i \{P\}\ \mathtt{havoc}\ (X)\ \mathtt{st}\ (e)\ \{Q\}}{\vdash_i \{P\}\ \mathtt{relax}\ (X)\ \mathtt{st}\ (e)\ \{Q\}}$$

$$\text{assume}\ \frac{}{\vdash_i \{P \wedge e\}\ \mathtt{assume}\ e\ \{P \wedge e\}}$$

Figure 9. Axiomatic Intermediate Semantics

program to reason about assumptions in the relaxed semantics in the same way as we do for assertions — i.e., if the assumption is true in the original semantics of the program, then it is also true in the relaxed semantics.

Convergent and Divergent Control Flow. An important aspect of relaxed programs is that the original and relaxed executions of a program may branch in different directions at a control flow construct. Specifically, if two executions branch in the same direction, then we can continue to reason about them relationally in lockstep. However, if the two executions diverge, then the executions execute different statements and, as a result, we lose our relational reasoning power. We note that it is possible for two executions to diverge at a control flow construct and then converge again at the end of the construct, allowing us to regain our relational reasoning power.

The relaxed axiomatic semantics captures this property via a set of proof rules for *convergent control flow* constructs (i.e., the original and relaxed executions always branch in the same direction) and another set for *divergent control flow* constructs. [1]

3.3.1 Convergent Control Flow

The `if` rule allows us to continue to use relational reasoning inside an `if` statement if it has convergent control flow. We establish this convergence 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 semantics. If so, then in all cases, the original and relaxed semantics take the same branch together. Otherwise control flow may diverge and 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 us to continue to use relational reasoning within the body of a `while` statement.

3.3.2 Divergent Control Flow

The *diverge* rule enables a proof to proceed if the original and relaxed semantics diverge at a control flow construct. The rule establishes the postcondition of the statement by independently establishing that $\vdash_o \{P_o\}\ s\ \{Q_o\}$ for the original semantics and that $\vdash_i \{P_r\}\ s\ \{Q_r\}$ for the relaxed semantics, where P_o and P_r are left and right projections of P^*. The judgment $\vdash_i \{P_r\}\ s\ \{Q_r\}$

[1] We provide a detailed formalization of convergent control flow in the supplementary material of this paper on the ACM Digital Library.

is a set of proof rules for the axiomatic *intermediate* semantics of the program. Figure 9 gives an abbreviated presentation of the program logic for the intermediate semantics of the program.

The axiomatic intermediate semantics is a nonrelational characterization of the dynamic relaxed semantics and is very similar to the axiomatic original semantics. The intended meaning of the judgment $\models_i \{P\}\ s\ \{Q\}$ is: for all states σ, if $\sigma \models P$ and $\langle s,\ \sigma\rangle \Downarrow_r \langle \sigma',\ \psi\rangle$, then $\sigma' \models Q$. This semantics differs from the axiomatic original semantics in two ways:

- **The relax rule** specifies that `relax` (X) `st` (e) may apply any modification to the variables in X as long as the new values satisfy e. In the axiomatic original semantics the `relax` statement is a no-op.
- **The assume rule** requires (just as for `assert` statements) a proof that e holds in relaxed executions. The goal is to ensure that the relaxation does not invalidate the reasoning used to establish that e holds in the original program. In the axiomatic original semantics there is no such proof obligation — e is simply assumed to be valid.

We conclude the presentation of the diverge rule by noting that it is also guarded by the predicate *no_rel*(s), which evaluates to *true* if no `relate` statements appear within s. This predicate therefore prevents `relate` statements from appearing in divergent control flow statements where we are unable to use relational reasoning to establish that the `relate` statement is satisfied.

We also note that the use of projections by $\models_o$ and $\models_r$ in this rule means that all relationships between the two semantics are lost and must be reestablished at the end of the statement. Relationships that are not modified by the statement, however, can be preserved via a relational frame rule.

4. Properties

We now present the technical definitions, lemmas, and theorems that establish the semantic properties of programs in the language.

The key property of our language and proof rules is Relative Relaxed Progress (Section 4.3), which states that our proof rules guarantee that if the original program executes without error, then the relaxed program executes without error. This property enables a developer to combine a proof in the axiomatic original semantics with formally unverified assumptions to demonstrate that the original program is error free. The developer can then augment this reasoning with a proof in the axiomatic relaxed semantics to therefore show that the relaxed program is error free. An important consequence of this formulation is that if a formally unverified assumption is not valid and produces unintended behaviors in the program, then these behaviors can be reproduced and debugged in the original program.

In our formalization, we restrict ourselves to terminating relaxed programs. Also, while we only present brief proof sketches, the full sources of our Coq formalization and proofs are available online at

`http://groups.csail.mit.edu/pac/acceptability`

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. Specifically, if you can write a proof in the axiomatic original semantics and an execution in the original semantics terminates, then the resulting state is not wr. This differs from a *strong* form of progress that establishes the same for all programs (including nonterminating programs), which would require a small-step or coinductive formalization of our dynamic semantics.

Lemma 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. More specifically, given a proof $\vdash_o \{P\}\ s\ \{Q\}$, it is the case that for all states $\sigma \models P$, if $\langle s,\ \sigma\rangle \Downarrow_o \langle \sigma',\ \psi\rangle$, then $\sigma' \models Q$.

Proof Sketch. This proof proceeds by induction on the rules of $\vdash_o \{P\}\ s\ \{Q\}$. 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 also requires mutual induction on the lists of modified and fresh variables to establish that the post-condition holds. The cases for structural rules (`if` and sequential composition) follow from induction and the case for the `while` statement proceeds by nested induction on derivations of the evaluation relation. □

Lemma 2 (Original 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. Specifically, given a proof in the original axiomatic semantics, then for all states that satisfy P, if execution terminates, then the execution 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 $\vdash_o \{P\}\ s\ \{Q\}$. We only need to consider three primitive statements where the program may evaluate to wr: `havoc` (the satisfiability check in the premise guards against this), `assert` e (the fact that e must hold in the precondition guards against this), and `relax` (follows by induction because execution of a `relax` statement reduces to execution of an `assert` statement). □

4.2 Intermediate Semantics

The axiomatic relaxed semantics establishes several progress properties about the relaxed execution of the program. However, to prove progress for the axiomatic relaxed semantics, we need to first prove the same for the axiomatic intermediate semantics.

The definition for the axiomatic intermediate semantics is sound and, also, establishes a form of progress for the relaxed semantics of the program. Specifically, the intermediate semantics models the behavior of a relaxed execution after it has branched at a control flow construct in a different direction than an original execution. When this happens, the relaxed execution must not violate assertions (evaluate to wr) or violate assumptions (evaluate to ba).

Lemma 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: given a proof $\vdash_i \{P\}\ s\ \{Q\}$, it is the case that for all states $\sigma \models P$, if $\langle s,\ \sigma\rangle \Downarrow_r \langle \sigma',\ \psi\rangle$, then $\sigma' \models Q$.

Proof Sketch. This proof proceeds by induction on the rules of $\vdash_i \{P\}\ s\ \{Q\}$. Because the axiomatic intermediate semantics reuses a large portion of the axiomatic original semantics, and the definition of the dynamic relaxed semantics is very similar to that of the dynamic original semantics, the vast majority of the proof follows from the proofs in Lemma 1. □

Lemma 4 (Progress)**.**

$$\textit{If } \vdash_i \{P\}\ s\ \{Q\}, \textit{ and } \sigma \models P, \textit{ and } \langle s,\ \sigma\rangle \Downarrow_r\ \phi, \\ \textit{then } \neg err(\phi) \\ \textit{where } err(\phi) \equiv \phi = wr \vee \phi = ba$$

This lemma establishes the progress property that we desire for the axiomatic intermediate semantics. Specifically, given a proof $\vdash_i \{P\}\ s\ \{Q\}$, it is the case that for all states that satisfy P, if execution terminates, then the execution does not yield an error. Note that this guarantee is stronger than that for the axiomatic original semantics in that it does not allow a relaxed execution to violate an assumption whereas an original execution may do so.

Proof Sketch. The proof proceeds by induction on the rules of $\vdash_i \{P\}\ s\ \{Q\}$. As in the proof of soundness (Lemma 3), the vast majority of the proof follows from our proofs about the axiomatic original semantics; in this case most of the proof follows directly from Lemma 2 (Original Progress Modulo Assumptions). The proof differs for two statements: `relax` (the proof follows by induction because execution of a `relax` statement reduces to the execution of a `havoc`) and `assume` (the proof also follows by induction as execution of an `assume` reduces to an `assert`). □

4.3 Relaxed Semantics

Lemma 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 semantics of the program. Specifically, given a proof $\vdash_r \{P^*\}\ s\ \{Q^*\}$, it is the case that for all states $(\sigma_o, \sigma_r) \models P^*$, if $\langle s,\ \sigma_o\rangle \Downarrow_o \langle \sigma_o',\ \psi_1\rangle$ and $\langle s,\ \sigma_r\rangle \Downarrow_r \langle \sigma_r',\ \psi_2\rangle$, then $(\sigma_o', \sigma_r') \models Q^*$.

Proof Sketch. The proof proceeds by induction on the rules of $\vdash_r \{P^*\}\ s\ \{Q^*\}$. The proof is largely similar to that for the original axiomatic semantics in that much of the work lies in proving the semantics of substitution for the `relax` statement, which has a proof that is similar to `havoc` in the original axiomatic semantics. The cases for structural rules (`if` and sequential composition) follow from induction and the case for the `while` statement proceeds by nested mutual induction on derivations of the original and relaxed execution of the statement. The most distinct case is the rule for diverge: this proof uses the soundness of the original axiomatic semantics (Lemma 1) and the soundness of the intermediate axiomatic semantics (Lemma 3) to establish the soundness of the rule for executions in which the original and relaxed executions branch in different directions at a control flow construct. □

Theorem 6 (Soundness of Relational Assertions)**.**

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

This theorem establishes that given a proof in the relaxed axiomatic semantics, if the original execution of the program terminates successfully and the relaxed execution of the program terminates successfully, then the observation lists generated by the executions (ψ_1 and ψ_2, respectively) satisfy the observational compatibility relation $\Gamma \vdash \psi_1 \sim \psi_2$. Observational compatibility implies that the original and relaxed executions of the program satisfy all executed `relate` statements; we define the relation as follows:

$$\frac{}{\Gamma \vdash \emptyset \sim \emptyset} \qquad \frac{[\![\Gamma(l)]\!](\sigma_1, \sigma_2) = \textit{true} \quad \Gamma \vdash \psi_1 \sim \psi_2}{\Gamma \vdash (l, \sigma_1) :: \psi_1 \sim (l, \sigma_2) :: \psi_2}$$

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

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

Proof Sketch. This proof proceeds by induction on the rules of $\vdash_r \{P^*\}\ s\ \{Q^*\}$. The two interesting cases are the *diverge* rule and the rule for `relate` statements. For the diverge rule, we use the fact that the rule requires *no_rel*(s) (which requires that no `relate` statements appear inside s). We can therefore conclude that the observation list for the statement is empty and, therefore, original and relaxed executions of the statement are trivially compatible. In the case of the relate rule, the proof uses the rule's precondition to establish that the two emitted observations satisfy the `relate` statement's condition. □

Theorem 7 (Relative Relaxed Progress)**.**

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

This theorem establishes the relative progress guarantee for the relaxed semantics of the program. Specifically, given a proof $\vdash_r \{P^*\}\ s\ \{Q^*\}$, it is the case that for all pairs of states (σ_o, σ_r) that satisfy P^*, if an original execution terminates and does not produce an error, then if a relaxed execution terminates, it also does not produce an error.

Proof Sketch. This proof proceeds by induction on the rules of $\vdash_r \{P^*\}\ s\ \{Q^*\}$. The most important cases are the `assert` and `assume` statements and the *diverge* rule. The proofs for `assert` and `assume` are similar in that the premise ensures that if the original execution evaluates to *true*, then the condition also evaluates to *true* in the relaxed execution. We do not have to consider the case where the original execution evaluates to *false* because this would imply that $\phi_o = wr \vee \phi_o = ba$, which is inconsistent with the assumption that $\neg err(\phi_o)$.

The diverge rule demonstrates the utility of our design of the axiomatic intermediate semantics. For this rule, we can no longer use facts about the original execution to prove facts about the relaxed execution. Therefore, the relaxed execution must be inherently error free. The proof uses the progress guarantee of the axiomatic intermediate semantics (Lemma 4) to establish exactly that. □

Theorem 8 (Relaxed Progress)**.**

$$\textit{If } \vdash_o \{P\}\ s\ \{Q\}, \textit{ and } \vdash_r \{P^*\}\ s\ \{Q^*\}, \textit{ and } P^* \models_o P_o, \\ \textit{and } (\sigma_o, \sigma_r) \models P^*, \textit{ and } \langle s,\ \sigma_o\rangle \Downarrow_o\ \phi_o, \textit{ and } \phi_o \neq ba, \\ \textit{and } \langle s,\ \sigma_r\rangle \Downarrow_r\ \phi_r, \textit{ then } \neg err(\phi_r).$$

This theorem combines the multiple proofs and assumptions in our programming model to establish the main progress guarantee for relaxed programs: given 1) a proof in the original axiomatic semantics, 2) a proof in the relaxed axiomatic semantics, and 3) that executions in the original semantics terminate and do not violate an assumption, then if a relaxed execution terminates, it does not produce an error.

Proof Sketch. This proof follows directly from our assumptions, Lemma 2 (Original Progress Modulo Assumptions), and Theorem 7 (Relative Progress). Our assumptions and Lemma 2 establish that if an original execution terminates, then it does not terminate in error. By Theorem 7, we can conclude that if a relaxed execution terminates, it does not produce an error. □

Corollary 9 (Relaxed Progress Modulo Original Assumptions)**.**

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

This corollary of Theorem 8 captures an important aspect of how our programming model incorporates assumptions, which may cause errors in both original and relaxed executions. Given proofs in the original and relaxed axiomatic semantics, if an error occurs in a relaxed execution and the original execution terminates, the original execution must violate an assumption. Errors in the relaxed program therefore correspond to invalid assumptions in the original program. To debug an error in a relaxed execution, a developer should therefore look for invalid assumptions in the original program.

5. Example Relaxed Programs

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

5.1 Dynamic Knobs

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 when the server is under heavy load [16]. The rationale for this relaxation is that 1) users are typically only interested 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 advertisement revenue.

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.

Acceptability. 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 `relate` statement captures these constraints:

```
relate (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 `relate` statement uses `num_r<o>` and the (potentially different) value of `num_r` in the relaxed execution (`num_r<r>`) to formalize the desired relationship between the two executions.

Verification. The proof of the `relate` 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 `relate` statement's precondition can be deduced via a proof by cases or, as in our proof environment, verified by an automated theorem prover.

5.2 Statistical Automatic Parallelization

Our next example is drawn from a parallelization of the Water computation [8] with statistical accuracy bounds [20]. 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 an array `RS` execute atomically. The resulting race conditions produce a parallel computation whose result may vary nondeterministically (because of CPU scheduling variations) within acceptable accuracy bounds.

Relaxation. We model the relaxation nondeterminism by relaxing each element in `RS` with no constraints:[2]

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

In a loop that executes after the parallel loop, the Water computation compares `RS[K]` to a cutoff variable `gCUT2` and, if it is less than the cutoff, uses `RS[K]` to update an array `FF` (here `EXP(RS[K])` is an expression involving `RS`):

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

Acceptability. A key acceptability property is that `K` stays within the bounds of the array `FF`.[3] 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 and therefore inserts the statement `assume (K < len_FF)` inside the `if` statement just before the assignment to `FF[K]`.

[2] Although we do not present our treatment of arrays in this paper, our Coq formalization supports reasoning about arrays, which is a straightforward extension of our presented framework.

[3] We note that `K` must also be within the bounds of `RS`; the proof is similar.

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 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 depends on 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, 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 proof step verifies the relational loop invariants `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 Approximate Memory and Data Types

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 [18, 34].

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 [25], 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 nondeterministically adds a bounded error `e` to the result:

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

Acceptability. 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 a `relate` statement:

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

We note that this `relate` statement asserts the *Lipschitz-continuity* of the computation: small changes in the inputs lead to proportionally small changes in the output.

Verification. The Coq verification of this program consists of approximately 315 lines of proof script. The key proof step verifies that `max<o> - max<r> <= e && max<r> - max<o> <= e` (the relation specified by the `relate` statement) is loop invariant.

6. Future Work

Termination. Our proof rules are designed to support the verification of acceptability properties that can be described with `assert` statements—which must hold independently for both original and relaxed executions—and `relate` statements, which relate original and relaxed executions. While such properties establish a partial correctness guarantee (as in standard Hoare logic), other acceptability properties may also prescribe total correctness: proofs of termination as well as partial correctness.

Because relaxed programs may contain additional nondeterministic and dynamically varying control flow, proving termination for a relaxed program may be more challenging than proving nontermination for the original program. One approach is to use relational reasoning to transfer the termination proof from the original program to the relaxed program (in much the same way as for our current set of acceptability properties). And while our proof machinery is not designed to support termination proofs, the rules in our axiomatic relaxed semantics do establish a relative termination guarantee for convergent while loops (whose termination conditions are proved to be equivalent in the original and relaxed semantics). Relative termination for such convergent loops ensures that if the loop terminates in the original program then it also terminates in the relaxed program. We anticipate that such a notion of relative termination (akin to our relative progress) will be a fruitful direction for future work.

Completeness. We have omitted a formal statement of the completeness of the logic because we believe that there are additional acceptability properties of relaxed programs that interact with termination, divergent control flow, and the semantics of errant executions in which the original or relaxed semantics can evaluate to ba or wr. We intend to explore the meaning of completeness and its interactions with these properties in future work.

7. Related Work

Executable Specifications. Executable specifications, via techniques such as refinement and constraint solving, produce concrete outputs that satisfy the specification [13, 19, 23, 27, 33, 37]. 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. Our focus is therefore on enabling developers to specify and prove acceptability requirements that involve relational properties between the original and relaxed programs.

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 stored reliably [10, 18, 34]. These systems focus on data values (such as the values of pixels in an image) that can, in principle, legally take on *any* value. While the techniques presented in this paper support the verification of this class of programs, they also support the verification of a more general class of programs whose legal data values are constrained by relaxation predicates.

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 Credible Compilation [31] and Translation Validation [26] and and has since inspired other forms of relational reasoning. Researchers have also defined relational separation logic [5, 36], probabilistic Hoare logic [6], and have used relational reasoning to verify the correctness of semantics-preserving compiler transformations [12, 31, 39], Lipschitz-continuity [12], access control policies [24], and differential privacy mechanisms [6].

While the majority of previous research has focused on proving that transformed programs retain the semantics of the original program, our goal is different — specifically, to prove that relaxed executions (which typically have different semantics) preserve important acceptability 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).

8. 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, 11, 16–18, 20–22, 29, 30, 32, 34, 35, 38] or responding to load spikes or other fluctuations in the characteristics of the underlying computational platform [16, 17, 29, 34].

We present formal reasoning techniques that make it possible to verify important 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 acceptability properties they need to confidently deploy relaxed programs and exploit the substantial flexibility, performance, and resource consumption advantages that relaxed programs offer.

Acknowledgments

We would like to thank Fan Long and our anonymous reviewers for their insightful comments. We also note our earlier technical report on reasoning about relaxed programs [9]. This research was supported in part by the National Science Foundation (Grants CCF-0811397, CCF-0905244, CCF-1036241, and IIS-0835652), DARPA (Grants FA8650-11-C-7192 and FA8750-12-2-0110), and the United States Department of Energy (Grant DE-SC0005288).

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. FM, 2011.

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

[7] N. 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. *Transactions on Parallel and Distributed Systems*, 3(6), 1992.

[9] M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Reasoning about Relaxed Programs. Technical Report MIT-CSAIL-TR-2011-050, MIT, 2011.

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

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

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

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

[14] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32), 1967.

[15] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10), October 1969.

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

[17] 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.

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

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

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

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

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

[23] C. Morgan. The specification statement. *Transactions on Programming Languages and Systems*, 10(3), 1988.

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

[25] J. Nelson, A. Sampson, and L. Ceze. Dense approximate storage in phase-change memory. ASPLOS-WACI, 2011.

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

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

[28] M. Rinard. Acceptability-oriented computing. OOPSLA Onwards '03.

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

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

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

[32] Martin Rinard. A lossy, synchronization-free, race-full, but still acceptably accurate parallel space-subdivision tree construction algorithm. Technical Report MIT-CSAIL-TR-2012-005, MIT, 2012.

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

[34] 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.

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

[36] H Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3), May 2007.

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

[38] Z. Zhu, S. Misailovic, J. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. POPL, 2012.

[39] L. Zuck, A. Pnueli, and R. Leviathan. Validation of optimizing compilers. Technical report, Weizmann Institute of Science, 2001.