Logical Reasoning for Approximate and Unreliable Computation

by

Michael James Carbin

B.S., Stanford University, 2006
S.M., Massachusetts Institute of Technology, 2009

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

Author . . . Signature redacted . . . . . .
Department of Electrical Engineering and Computer Science
February 13, 2015

Certified by . . . . Signature redacted . . . . . .
Martin C. Rinard
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . . . Signature redacted . . . . . .
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Theses
Professor of Electrical Engineering and Computer Science

Logical Reasoning for Approximate and Unreliable Computation

by

Michael James Carbin

Submitted to the Department of Electrical Engineering and Computer Science
on February 13, 2015, in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

Abstract

Improving program performance and resilience are long-standing goals. Traditional approaches include a variety of transformation, compilation, and runtime techniques that share the common property that the resulting program has the same semantics as the original program.

However, researchers have recently proposed a variety of new techniques that set aside this traditional restriction and instead exploit opportunities to change the semantics of programs to improve performance and resilience. Techniques include skipping portions of a program's computation, selecting different implementations of program's subcomputations, executing programs on unreliable hardware, and synthesizing values to enable programs to skip or execute through otherwise fatal errors.

A major barrier to the acceptance these techniques in both the broader research community and in industrial practice is the challenge that the resulting programs may exhibit behaviors that differ from that of the original program, potentially jeopardizing the program's resilience, safety, and accuracy. This thesis presents the first general programming systems for precisely verifying and reasoning about the programs that result from these techniques.

This thesis presents a programming language and program logic for verifying worst-case properties of a transformed program. Specifically the framework, enables verifying that a transformed program satisfies important assertions about its safety (e.g., that it does not access invalid memory) and accuracy (e.g., that it returns a result within a bounded distance of that of the original program).

This thesis also presents a programming language and automated analysis for verifying a program's quantitative reliability – the probability the transformed program returns the same result as the original program – when executed on unreliable hardware.

The results of this thesis, which include programming languages, program logics, program analysis, and applications thereof, present the first steps toward reaping the benefits of changing the semantics of programs in a beneficial yet principled way.

Thesis Supervisor: Martin C. Rinard
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

It is difficult to know where to begin because, by and large, the journey described in this thesis is a product of the interactions with and support by a whole cast of people in my life. Without the following individuals – and many more – this thesis would not exist.

I would like to thank my advisor Martin Rinard for providing me with an environment in which I was able to be productive — and also lending his keen sense for what it takes to change the way people think. Being bold was not a characteristic that I attributed to myself when I first entered graduate school. However, in no small part due to Martin, I do consider being bold a necessary and critical aspect of my research going forward.

Sasa Misailovic, my closest collaborator, and I have burned through many midnight (Samoa Time) deadlines, shared many paper acceptance celebrations, and even had many tense debates about the direction of a project. However, through all the stresses and successes, the thought and care Sasa has put into the random goings-on in my life demonstrate that we have always been friends and not just collaborators. I also thank Sasa for his contributions to the Rely system (presented in Chapter 4).

I would like to thank Stelios Sidiroglou for being Stelios Sidiroglou. There was never a day where a long sought out moment of peace and quiet wasn't pleasantly interrupted by a moment of laughter or the requisite chat about a genius and/or terrible research idea. Moreover, the way Stelios has used the variety of his creative interests to inspire and motivate his research agenda has inspired me to seek out how to do the same with my own.

I would like to thank Deokhwan Kim for sharing with me his appreciation and talent for elegance. The 36 pages of inference rules contained within this thesis are a tribute to Deokhwan. My only hope is that they elicit from him fewer than one "oops" per page. I also thank Deokhwan for his contributions to the notation for relaxed programs, the formalization of convergent control flow points, and the SMT solving tool included with the Coq-based proof environment (presented in Chapter 2).

Fan Long entered the research group right around the time I started to feel like an "old" graduate student. It was Fan's eagerness, focus, and hacking skills that motivated me to think back to my time as a young graduate student and re-evaluate and rediscover aspects of my own energy, focus, and methods.

My overlap with the younger graduate students of the group, Sara Achour, Zichao Qin, Fereshte Khani, Jiasi Shen, Zichao Qi, and Christopher Musco, has been unfortunately short. However, their enthusiasm and zeal for research has reminded me of what is ultimately fun and exciting about being a graduate student. I hope that as a professor I will be able to do the same for them as they fight their inevitable graduate student battles.

Over the years, our research group has also included a number of people and collaborators, including Vijay Ganesh, Hank Hoffmann, Michael Gordon, Jeff Perkins, Karen Zee, Michael Kling, Cam Tenny, Paolo Piselli, Jordan Eikenberry, and Rocky Kramm. With all of these members I have shared invaluable experiences in both research and life.

At both MIT and other institutions I have been privileged to receive the support and advice of many senior academics. In particular, I would like to thank Monica Lam, Jim Larus, Arvind, Saman Amarasinghe, and Armando Solar-Lezama, for the advice they have given me at multiple stages in my career about how to succeed.

I would like to thank Adrian Sampson, Todd Mytkowicz, Dan Grossman, and Luis Ceze for being excellent colleagues in this emerging field of approximate and unreliable computing. Together we have shared many conversations that have guided the way I think about the important problems of our field.

In my time on the academic job market I met and spoke with many of the brightest professors and students in the world. Of those many, the following people in particular gave advice, critique, and support that tailored the way I view not only the job market process, but also how I view myself: Patrick Lam, Brian Demsky, Andrew Chien, Ben Liblit, Thomas Reps, Ras Bodik, Ranjit Jhala, Sorin Lerner, Rajesh Gupta, Vijay Janapa Reddi, Sarfraz Khurshid, Brian Kernighan, Joseph Devietti, Stephen Edwards, Alfred Aho, Jason Mars, Marios Papaefthymiou, William Sanders, Steven Lumetta, Wen-Mei Hwu, Ross Tate, Kathryn McKinley, Trishul Chilimbi, Tom Ball, and Ben Zorn.

I would like to thank my family: Eddie Carbin, Sr., Eddie Carbin, Jr., Cynthia Carbin, and Sandra Dunn. The passing of my mother, Rowena Joyce Carbin, reminded me of what I know to be important in life: having all the success in the world doesn't mean much without family and friends. The hurdles that we as a family have jumped make plain that the high expectations I have for myself have been learned from the high expectations that

my family members have for themselves. Moreover, much of my success can be attributed the fact that we as a family have and always will support each other in our endeavors.

My wife Angelee Russ-Carbin has endured all the trials and tribulations that have accompanied my desire to push my career forward. It is only through her love, support, and dedication that I have had a solid foundation on which I have been able to build and pursue my research career. Without her support, it would not have been possible to achieve all that I have in this time.

My family has also been extended to include Jennifer Russ, Redding Hordijk, Don Hordijk, and Jim and Helen Russ. Living 3000 miles away from my West Coast family would not have been possible without the support of my new East Coast family.

My childhood friends Eric Ooi, John Bedard, Lakshmi Narayan, West Alexander, and Rufus Olivier have always so diligently kept me grounded and aware of where I've come from (particularly, the many mistakes I've made along the way).

Christopher Reeder – as a colleague, a housemate, and as a friend – has been an essential pillar of my support. Many of the moments that have defined my years at MIT have included Chris. It's hard to imagine what these years would have been like without his presence and perspective.

I would also like to thank the self-titled friend collective, "Slizzard," which includes Robert Rykowski, Stephanie Brown, Heather Rykowski, Radthavis Sayabovorn, Drew Bisset. Joined with Chris, Angelee, and I, the memories we share together are some of the most definitive and formative of my life.

The passing of my mother motivated me to expand myself and reach out for support. In that immediate time after, my circle of support grew enormously to include (and is not limited to) Lin Zhu, Philippe Suter, Irene Sung, Katharina Nimptsch, Felix Kurz, Anja Hohmann, Chee Xu, and Chistopher Betrand. The impact of these friends on my life is far larger than perhaps they would expect.

This thesis has been a journey. And while this particular journey has come to and end, the relationships that I have built during this time will endure.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Approximate computations, such as multimedia, financial, machine learning, and big data analytics, have emerged as a major component of many computing environments. Motivated in part by the observation that these computations can often acceptably tolerate occasional errors in their execution and/or data [77, 58, 21], researchers have recently developed a range of new mechanisms that forgo the exact correctness of the computation's underlying computational stack to optimize other objectives. Typical goals include maximizing program performance subject to an accuracy constraint and modifying program execution to recover from otherwise fatal errors.

1.1 Improved Performance

In the search for increased performance, researchers have proposed a variety of both software transformations and hardware modifications to enable applications to trade the quality of their results for better performance and/or lower energy consumption.

Software Transformations. Proposed mechanisms for transforming programs include skipping tasks [77, 78], loop perforation (skipping iterations of time-consuming loops) [58, 57, 88], sampling reduction inputs [94], multiple selectable implementations of a given component or components [5, 3, 39, 94], dynamic knobs (configuration parameters that can be changed as the program executes) [39] and synchronization elimination (forgoing synchronization not required to produce an acceptably accurate result) [55, 79].

I, along with my collaborators, have worked on a number of these techniques. Our experimental results show that aggressive techniques – for example transforming loops to skip some (or all) iterations through loop perforation – can yield up to a four-fold improvement in an application's performance with acceptable changes in the quality of its results [39, 19].

Approximate Hardware. Programs that are amenable software transformations that augment program behavior typically implement computations that are also amenable to executing on hardware systems that occasionally expose unmasked errors that occur during execution. For these computations, operating without (or with at most selectively applied) mechanisms that detect and mask soft errors can produce 1) fast and energy efficient execution that 2) delivers acceptably accurate results often enough to satisfy the needs of their users despite the presence of unmasked soft errors.

Motivated in part by this observation, the computer architecture community has begun to investigate new designs that improve performance by breaking the traditional fully reliable digital abstraction that computer hardware was traditionally sought to provide. The goal is to reduce the cost of implementing a reliable abstraction on top of physical materials and manufacturing methods that are inherently unreliable. For example, researchers are investigating designs that incorporate aggressive device and voltage scaling techniques to provide low-power ALUs and memories. A key aspect of these components is that they forgo traditional correctness checks and instead expose timing errors and bitflips with some non-negligible probability [27, 31, 32, 45, 49, 64, 66, 84].

1.2 Increased Resilience

Maintaining the security and availability of both approximate and traditional computing systems is an active and open research area. In addition to using fortified programming languages, automatic program analysis, and program verification systems to eliminate a system's vulnerabilities before deployment, researchers have proposed a number of techniques that enable deployed systems to recover from otherwise fatal attacks and errors.

I, along with my collaborators, have worked on a number of techniques that modify the semantics of the program to make it more resilient to programming errors [68, 50]. The experimental results show that if the program encounters an error that threatens its stability or security, these techniques enable the runtime system to modify the program's execution to steer around the problem and still execute acceptably. For example, my work on the Jolt and Bolt systems demonstrates that it is often possible to enable an application that is stuck in an infinite loop to produce an acceptable output by simply exiting the loop and continuing on with the remaining execution of the program [20, 41].

1.3 Thesis

Changing the semantics of a program violates the traditional contract that the programming system must preserve the semantics of the program. Building programs in this environment therefore necessitates new techniques for reasoning about the semantics of programs.

One key aspect of the computations that are amenable to these techniques is that they typically contain *critical* regions (which must execute without error) and *approximate* regions (which can execute acceptably even in the presence of occasional errors) [77, 21].

Existing systems, tools, and type systems have focused on helping developers identify, separate, and reason about the binary distinction between critical and approximate regions [77, 78, 21, 49, 84, 86, 32]. However, in practice, no computation can tolerate an unbounded accumulation of errors — to execute acceptably, even the approximate regions must execute correctly with some minimal requirements.

Changing the semantics of programs therefore raises a number of new and fundamental questions. For example, what is the probability that the resulting program will produce the same result as the original program? How much do the results differ from those produced by the original program? And is the resulting program safe and secure?

This thesis investigates the hypothesis that it is possible to design and use programming languages, programming analyses, and program verification to reason about the *integrity*, *accuracy*, and *reliability* of transformed programs.

Integrity. The *integrity properties* of a program are the properties that the program must satisfy to successfully produce a result. Consider for example the following simplified code from bodytrack, a machine vision application in the PARSEC benchmark suite [13]:

```
float *err = new float;
int num_samples = 0
int acc = 0;
for (int i = 0; i < n; ++i)
{
    int *x = new int;
    int *y = new int;

    GetCoord(i, x, y);
    num_samples = num_samples + Sample(x, y, err);
}
float err_v = *err;
assert (0 < num_samples);
float avg_error = err_v / num_samples;
```

This code implements a sampling-based computation that enumerates over a set of `n` coordinates (`x`, `y`) to take samples (`Sample(x, y, err)`) of an image. In previous work, my collaborators and I demonstrated that this loop is amenable to a loop perforation transformation that modifies the loop to skip some of its iterations, therefore decreasing the computation's sampling rate [19]. However, a key concern for this computation is that the number of samples taken, `num_samples`, is used as the divisor of a division operation. Specifically, if the program is not carefully transformed then the value of `num_samples` may cause the program to produce a divide-by-zero error and be unable to produce a result. The property that `num_samples` is not equal to zero is therefore one of the computation's integrity properties.

Accuracy. The *accuracy properties* of a program are the properties that characterize how accurate the produced result must be. For example, an accuracy property might state that the transformed program must produce a result that differs by at most a specified percentage from the result that the original program produces [57, 94]. Consider for example the following code from LU, a numerical application in the SciMark2 benchmark suite [2]:

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

This code is part of a LU matrix factorization implementation and is responsible for computing the pivot row, p, of the matrix A. Researchers have shown that this computation can be profitably transformed to execute on approximate hardware in a way that allocates the matrix in an unreliable memory that may produce incorrect results when read from or written to [84]. An accuracy property for this computation is that, for example, the error in the value associated with the final selected pivot row, max, is a linear function of the error introduced from loading the value of a from A (i.e., Lipschitz-continuity).

Reliability. The *reliability* of a computation is the probability that computation computes a correct result. Many emerging designs for approximate hardware provide a probabilistic execution model: each approximate operation may with some probability produce an incorrect value. Executing a computation (such as the LU factorization above) on an approximate hardware substrate therefore lends itself to the question of, what's the probability that the computation produces the correct result?

1.4 Approach

To enable reasoning about these fundamental properties of transformed and approximately executed programs, this thesis presents the *relaxed programing* and Rely frameworks.

1.4.1 Reasoning about Relaxed Programs

The transformations that researchers have developed for changing the semantics of programs produce new programs that can be modeled as *relaxed programs* — programs that

may adjust their 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.

Reasoning about a program transformed with one of these mechanisms therefore reduces to the task of reasoning about a relaxed program. Specifically, a developer or system must specify and verify the relaxed program's *acceptability properties*. A relaxed program's acceptability properties are the properties of the program's behavior and results that must be true for the program to be acceptably usable in its designated context.

This relaxed programming framework enables reasoning about acceptability properties that include both integrity properties and accuracy properties (as described above). The framework specifically makes integrity and accuracy first-class members in the syntax, semantics, and specification of a program.

The relaxed programming language also includes general primitives for *relaxing* the semantics of programs, enabling a developer or compilation system to transform the semantics of a program in a directed and controlled way.

The relaxed programming framework also includes a programming logic and verification system that enables developers and/or compilation systems to verify that relaxation preserves the integrity and accuracy of the computation.

The 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 the desired acceptability properties. Either the developer or an automated system (such as a compiler that implements loop perforation) then relaxes the program to enable additional nondeterministic executions. 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 program. These relational assertions facilitate the overall verification of the relaxed program. This approach is de-

signed 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.

Taken together, the components of the relaxed programming system make it possible to specify, relax, and verify the acceptability of programs that incorporate transformations to change their semantics.

1.4.2 Reasoning about Quantitative Reliability

This thesis also presents Rely, a new programming language, and an associated program analysis that computes the *quantitative reliability* of the computation — i.e., the probability with which the computation produces a correct result when its approximate regions execute on unreliable hardware. Specifically, given a hardware specification and a Rely program, the analysis computes, for each value that the computation produces, a conservative probability that the value is computed correctly despite the possibility of soft errors.

In contrast to existing approaches – which support only a binary distinction between critical and approximate regions – and in contrast with relaxed programs – which provides worst-case reasoning about the program's behavior – quantitative reliability can provide precise static probabilistic acceptability guarantees for computations that execute on unreliable hardware platforms.

Rely makes reliability a first-class member in the syntax, semantics, and specification of a program. Rely includes a language that enables developers to specify the reliability requirements of their application. Rely also includes an execution model and language support for exposing and exploiting operations of an approximate hardware platform. The Rely framework also provides an automated program analysis system that enables developers to automatically verify that a program targeted to approximate hardware satisfies its reliability specification.

1.5 Contributions

The relaxed programming and Rely frameworks are the first programming models and reasoning systems that work together to enable developers to specify, manipulate, and verify programs that incorporate semantic transformations in the pursuit of increased performance and resilience.

Taken together, my work demonstrates that giving programming systems the freedom to change the semantics of the program can open up new, simple, and effective ways to boost performance and/or enhance resilience and still verify that the resulting behavior is acceptable. This work therefore holds out the promise of enabling us to reap the performance and resilience benefits made available by these techniques.

Chapter 2

Relaxed Programming

While there is a wide diversity of transformations that change the semantics of programs, the successful transformations that researchers have developed typically modify a small portion of the program, leaving the overall structure of the program the same. For example, loop perforation and dynamic knobs change the value of variables that control the number of iterations of a loop or the choice of a specific implementation of a function, therefore locally modifying the execution path of the program, but largely preserving the overall structure of the execution.

This observation has enabled me to develop the concept of a relaxed programs, which, within the same program text, represents both the original meaning and new meaning of program. In this shared representation, I call the original meaning of the program the *original semantics* of the program whereas the new meaning is the program's *relaxed semantics*. Representing these two semantics in the same program text has enabled me to develop a program logic and corresponding proof rules that leverage the correspondence between the structure and behavior of the two programs to lower the verification burden of verifying the relaxed program.

2.1 Overview and Contributions.

I have designed the relaxed programming model to achieve three general goals.

Specification. The programming model enables developers to specify *acceptability properties*, which are the primary properties that ensure that a program is acceptable for use.

Relaxation. The program model enables systems and developers to specify relaxations of the program's semantics, which modify the program's behavior to achieve a goal such as reduced energy consumption or increased performance.

Verification. The programming model provides a program logic that enables proofs of the acceptability of relaxed programs. I have designed the program logic to enable proofs that use a *relational reasoning* approach that leverages acceptability properties that are true of the original program, along with relationships that hold between values in the original and relaxed program to establish that the acceptability properties still hold for the relaxed program. For example, a proof may demonstrate that relaxation does not interfere with the variables involved in an acceptability property and, therefore, the property is true in the relaxed program – provided that it was true in the original program. By enabling this kind of relational reasoning, the proof system admits proofs that can be simpler than proving the acceptability of the relaxed program outright.

In this section, I present an overview of the relaxed programming framework, including its programming language constructs and proof system, along with an overview of the framework's primary contributions.

2.1.1 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 introduces the relaxed semantics of a program. Specifically, in the relaxed semantics of the program the `relax` statement modifies its specified set of variables whereas in the original semantics the `relax` statement has no effect. Therefore `relax` statements are the point at which execution of the program deviates between the original and relaxed semantics.

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, the program logic's 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, the 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` state-

ment 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.

2.1.2 Key Properties of Acceptable Relaxed Programs

My 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 2.5).

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 2.5).

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). My 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 the 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.

2.1.3 Proof Rules and Formal Properties

The proof rules are a set of Hoare-style 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 me 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, the logic does 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). The 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. The logic also appropriately restricts the location of `relate` statements to program points at which the original and relaxed programs execute in lockstep.

The 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. The programming model's 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.

2.1.4 Coq Verification Framework

I have formalized the dynamic original and relaxed semantics with the Coq proof assistant [1]. I have also used Coq to formalize the 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. The Coq formalization makes it possible to develop fully machine-checked verifications of relaxed programs. I have used the framework to develop and verify several small relaxed programs.

The 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 my relational assertion logic and its soundness with respect to operations such as substitution.

Appendix A presents the full Coq development of the relaxed programming framework, including the semantics, proof rules, and proofs of the properties of the framework.

2.1.5 Contributions

This chaper presents 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 [77, 78], loop perforation [58, 57, 88], reduction sampling [94], multiple selectable implementations [5, 3, 39, 94], dynamic knobs [39], synchronization elimination [55, 79], approximate function memoization [24], and approximate data types [84].

- **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. I have used this formalization to prove that the proof rules are sound with respect to the dynamic original and relaxed semantics. In addition, the Coq formalization contains a reusable implementation of my relational assertion logic that is, in principle, suitable for other uses such as verifying traditional compiler transformations [82, 12, 92, 95].

- **Verified Programs:** It presents several relaxed programs for which I 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 chapter presents a foundational reasoning system that leverages the structure and relationships shared by the original and relaxed executions to enable verification of these properties.

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

Figure 2-1: Language Syntax

2.2 Language Syntax and Dynamic Semantics

Figure 2-1 presents a simple imperative language with integer variables, integer arithmetic expressions, boolean expressions, conditional statements, while loops, and sequential composition. For generality, the language supports 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* ($\hat{E}$) and *relational boolean expressions* ($\hat{B}$). 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 `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.

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

$$
\begin{aligned}
[\![\hat{E}]\!] &\in \Sigma \times \Sigma \to \mathbb{Z} \\
[\![\mathtt{n}]\!](\sigma_1, \sigma_2) &= n \\
[\![\mathtt{x}\langle\mathtt{o}\rangle]\!](\sigma_1, \sigma_2) = \sigma_1(x) &\qquad [\![\mathtt{x}\langle\mathtt{r}\rangle]\!](\sigma_1, \sigma_2) = \sigma_2(x) \\
[\![\hat{E}_1 \ \mathtt{iop}\ \hat{E}_2]\!](\sigma_1, \sigma_2) &= [\![\hat{E}_1]\!](\sigma_1, \sigma_2)\ \mathit{iop}\ [\![\hat{E}_2]\!](\sigma_1, \sigma_2)
\end{aligned}
$$

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

$$
\begin{aligned}
[\![\hat{B}]\!] &\in \Sigma \times \Sigma \to \mathbb{B} \\
[\![\mathtt{true}]\!](\sigma_1, \sigma_2) = \mathit{true} &\qquad [\![\mathtt{false}]\!](\sigma_1, \sigma_2) = \mathit{false} \\
[\![\hat{E}_1 \ \mathtt{cmp}\ \hat{E}_2]\!](\sigma_1, \sigma_2) &= [\![\hat{E}_1]\!](\sigma_1, \sigma_2)\ \mathit{cmp}\ [\![\hat{E}_2]\!](\sigma_1, \sigma_2) \\
[\![\hat{B}_1 \ \mathtt{lop}\ \hat{B}_2]\!](\sigma_1, \sigma_2) &= [\![\hat{B}_1]\!](\sigma_1, \sigma_2)\ \mathit{lop}\ [\![\hat{B}_2]\!](\sigma_1, \sigma_2) \\
[\![\neg \hat{B}]\!](\sigma_1, \sigma_2) &= \begin{cases} \mathit{true}, & [\![\hat{B}]\!](\sigma_1, \sigma_2) = \mathit{false} \\ \mathit{false}, & [\![\hat{B}]\!](\sigma_1, \sigma_2) = \mathit{true} \end{cases}
\end{aligned}
$$

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

Figure 2-2: Semantics of Expressions

2.2.1 Semantics of Expressions

Figure 2-2 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 $\Sigma = \mathit{Vars} \xrightarrow{\mathit{fin}} \mathbb{Z}$ (the set of all finite maps from variables to integers).

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

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

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

$$\text{havoc-f}\ \frac{\neg \exists_{\sigma'} \cdot (\langle e\ ,\ \sigma'\rangle \Downarrow_B \mathit{true} \wedge \forall_{x \notin X} \cdot \sigma(x) = \sigma'(x))}{\langle \texttt{havoc}\ (X)\ \texttt{st}\ (e),\ \sigma\rangle \Downarrow_o \mathit{wr}}$$

$$\text{assert-t}\ \frac{\langle e\ ,\ \sigma\rangle \Downarrow_B \mathit{true}}{\langle \texttt{assert}\ (e),\ \sigma\rangle \Downarrow_o \langle \sigma,\ \emptyset\rangle}$$

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

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

$$\text{relate}\ \frac{}{\langle \texttt{relate}\ l\colon (\hat{e}),\ \sigma\rangle \Downarrow_o \langle \sigma,\ (l, \sigma)\rangle} \qquad \text{if-t}\ \frac{\langle b\ ,\ \sigma\rangle \Downarrow_B \mathit{true} \quad \langle s_1,\ \sigma\rangle \Downarrow_o \phi}{\langle \texttt{if}\ (b)\ \{s_1\}\ \texttt{else}\ \{s_2\},\ \sigma\rangle \Downarrow_o \phi}$$

$$\text{if-f}\ \frac{\langle b\ ,\ \sigma\rangle \Downarrow_B \mathit{false} \quad \langle s_2,\ \sigma\rangle \Downarrow_o \phi}{\langle \texttt{if}\ (b)\ \{s_1\}\ \texttt{else}\ \{s_2\},\ \sigma\rangle \Downarrow_o \phi}$$

$$\text{seq}\ \frac{\langle s_1,\ \sigma\rangle \Downarrow_o \langle \sigma',\ \psi_1\rangle \quad \langle s_2,\ \sigma'\rangle \Downarrow_o \langle \sigma'',\ \psi_2\rangle}{\langle s_1\ ;\ s_2,\ \sigma\rangle \Downarrow_o \langle \sigma'',\ \psi_2.\psi_1\rangle}$$

$$\text{while-f}\ \frac{\langle b\ ,\ \sigma\rangle \Downarrow_B \mathit{false}}{\langle \texttt{while}\ (b)\ \{s\},\ \sigma\rangle \Downarrow_o \langle \sigma,\ \emptyset\rangle}$$

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

Figure 2-3: 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., $\wedge, \vee$, ...).

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

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

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

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

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

Figure 2-4: Error Propagation in Dynamic Original Semantics

The semantic function $[\![\hat{E}]\!]$ defines the semantics for relational integer expressions as a function mapping a pair of states (σ_1, σ_2) to an integer number. The first component of the state pair is 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 $[\![\hat{B}]\!]$ likewise extends the semantics for boolean expressions with the capability to express boolean properties over relational integer expressions.

2.2.2 Dynamic Original Semantics

Figure 2-3 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: $\Psi = \emptyset \mid (l, \sigma) :: \Psi$. The notation $\psi_1.\psi_2$ denotes the result of appending two lists.

Evaluation Rules

Skip. The `skip` is a no-op whose execution does not modify the state of the program and therefore produces the original input state as output.

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 the programming model dictates that the original execution is one of the relaxed executions, the dynamic original semantics requires the relaxation predicate e to hold in the original execution.

Relate. The `relate` l : $(\hat{e})$ 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 me to define and verify a relation on the observation lists emitted by the original and relaxed programs (see Section 2.4).

If. The `if (`b`) {`s_1`} else {`s_2`}` statement has the standard semantics of `if` statements. If the boolean condition b evaluates to *true* then execution of the statement continues with the nested statement s_1. If b evaluates *false* then execution continues with the statement s_1.

Sequence. The sequential composition construct s_1 ; s_2 also has a standard semantics. The construct first executes the statement s_1 and then executes the statement s_2.

While. The `while (b) { s }` statement also has the standard semantics of while loops. If the boolean condition b evaluates to *true*, then the statement executes one iteration of the statement s and then repeats. If b evaluates to *false* then the statement does not modify the state of the program.

Error Propagation. Figure 2-4 presents standard rules for the propagation of error values in the semantics of compound statements (i.e., sequential composition and `while` statements). The predicate $err(\phi)$ evaluates to true if and only if $\phi = wr$ or $\phi = ba$. For a sequential composition s_1 ; s_2, an error may occur either during the execution of s_1 (Rule [seq-3]) or during the execution of s_2 (Rule [seq-4]). For `while` statements, an error may occur either during the first iteration of the loop (Rule [while-t2]) or during a subsequent iteration of the loop (Rule [while-t3]).

2.2.3 Dynamic Relaxed Semantics

Figure 2-5 presents 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. I therefore include a discussion of only the rule for the `relax` statement.

Relax. The `relax (`X`) 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.

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

$$\text{skip}\frac{\langle \texttt{skip}, \sigma\rangle \Downarrow_o \phi}{\langle \texttt{skip}, \sigma\rangle \Downarrow_r \phi} \qquad \text{assign}\frac{\langle x = e, \sigma\rangle \Downarrow_o \phi}{\langle x = e, \sigma\rangle \Downarrow_r \phi}$$

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

$$\text{assume}\frac{\langle \texttt{assert}\ (e), \sigma\rangle \Downarrow_o \phi}{\langle \texttt{assume}\ (e), \sigma\rangle \Downarrow_r \phi} \qquad \text{seq}\frac{\langle s_1, \sigma\rangle \Downarrow_r \langle \sigma', \psi_1\rangle \quad \langle s_2, \sigma'\rangle \Downarrow_r \langle \sigma'', \psi_2\rangle}{\langle s_1\ ;\ s_2, \sigma\rangle \Downarrow_r \langle \sigma'', \psi_2 \cdot \psi_1\rangle}$$

$$\text{if-t}\frac{\langle b\ , \sigma\rangle \Downarrow_B \mathit{true} \quad \langle s_1, \sigma\rangle \Downarrow_r \phi}{\langle \texttt{if}\ (b)\ \{s_1\}\ \texttt{else}\ \{s_2\}, \sigma\rangle \Downarrow_r \phi} \qquad \text{if-f}\frac{\langle b\ , \sigma\rangle \Downarrow_B \mathit{false} \quad \langle s_2, \sigma\rangle \Downarrow_r \phi}{\langle \texttt{if}\ (b)\ \{s_1\}\ \texttt{else}\ \{s_2\}, \sigma\rangle \Downarrow_r \phi}$$

$$\text{while-f}\frac{\langle b\ , \sigma\rangle \Downarrow_B \mathit{false}}{\langle \texttt{while}\ (b)\ \{s\}, \sigma\rangle \Downarrow_r \langle \sigma, \emptyset\rangle}$$

$$\text{while-t}\frac{\langle b\ , \sigma\rangle \Downarrow_B \mathit{true} \quad \langle s, \sigma\rangle \Downarrow_r \langle \sigma', \psi_1\rangle \quad \langle \texttt{while}\ (b)\ \{s\}, \sigma'\rangle \Downarrow_r \langle \sigma'', \psi_2\rangle}{\langle \texttt{while}\ (b)\ \{s\}, \sigma\rangle \Downarrow_r \langle \sigma'', \psi_2 \cdot \psi_1\rangle}$$

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

Figure 2-5: Dynamic Relaxed Semantics

2.3 Axiomatic Semantics

I next present the axiomatic semantics of both the original and relaxed semantics of a relaxed program.

- **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 violate an assumption (i.e., evaluate to *ba*).

- **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.

$$
\begin{aligned}
P &::= \texttt{true} \mid \texttt{false} \mid E\ \texttt{cmp}\ E \mid P\ \texttt{lop}\ P \mid \neg P \mid \exists \texttt{x} \,.\, P \\
\hat{P} &::= \texttt{true} \mid \texttt{false} \mid \hat{E}\ \texttt{cmp}\ \hat{E} \mid \hat{P}\ \texttt{lop}\ \hat{P} \mid \neg \hat{P} \\
&\quad \mid \exists \texttt{x}\langle \texttt{o} \rangle \,.\, \hat{P} \mid \exists \texttt{x}\langle \texttt{r} \rangle \,.\, \hat{P}
\end{aligned}
$$

Figure 2-6: Relational Assertion Logic Syntax

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

$$
\begin{aligned}
[\![\hat{P}]\!] &\in \mathcal{P}(\Sigma \times \Sigma) \\
[\![\texttt{true}]\!] &= \Sigma \times \Sigma \quad [\![\texttt{false}]\!] = \emptyset \\
[\![\hat{E}_1\ \texttt{cmp}\ \hat{E}_2]\!] &= \{(\sigma_1, \sigma_2) \mid [\![\hat{E}_1]\!](\sigma_1, \sigma_2)\ cmp\ [\![\hat{E}_2]\!](\sigma_1, \sigma_2)\} \\
[\![\hat{P}_1\ \texttt{lop}\ \hat{P}_2]\!] &= \{(\sigma_1, \sigma_2) \mid (\sigma_1, \sigma_2) \in [\![\hat{P}_1]\!]\ lop\ (\sigma_1, \sigma_2) \in [\![\hat{P}_2]\!]\} \\
[\![\neg \hat{P}]\!] &= [\![\texttt{true}]\!] \setminus [\![\hat{P}]\!] \\
[\![\exists \texttt{x}\langle \texttt{o} \rangle \,.\, \hat{P}]\!] &= \{(\sigma_1, \sigma_2) \mid n \in \mathbb{Z},\ (\sigma_1, \sigma_2) \in [\![\hat{P}[n/x\langle \texttt{o} \rangle]]\!]\} \\
[\![\exists \texttt{x}\langle \texttt{r} \rangle \,.\, \hat{P}]\!] &= \{(\sigma_1, \sigma_2) \mid n \in \mathbb{Z},\ (\sigma_1, \sigma_2) \in [\![\hat{P}[n/x\langle \texttt{r} \rangle]]\!]\}
\end{aligned}
$$

Figure 2-7: Relational Assertion Logic Semantics

2.3.1 Relational Assertion Logic

Figure 2-6 presents the concrete syntax of the program logic's *relational* assertion logic. This logic extends a nonrelational assertion logic with relational formulas, enabling proofs that verify the relational boolean expressions in `relate` statements. Its presentation follows the style of Benton's Relational Hoare Logic [12].

Syntax. The syntactic category P gives the syntax for formulas in first-order logic with integer expressions and existential quantification. The syntactic category $\hat{P}$ gives corresponding syntax for writing relational formulas. $\hat{P}$ extends P by allowing formulas to refer to relational integer expressions.

Semantics

Figure 2-7 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 2-2 to construct a definition for each formula. The denotation of a relational formula $[\![\hat{P}]\!]$ 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., $\mathtt{x}\langle\mathtt{o}\rangle$) refer to the first component of the pair and references to the relaxed semantics (e.g., $\mathtt{x}\langle\mathtt{r}\rangle$) refer to the second component.

Injections. The injection functions $inj_o(P)$ and $inj_r(P)$ construct a relational formula in $\hat{P}$ 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., $\mathtt{x}$) in P with the relational original variables (e.g., $\mathtt{x}\langle\mathtt{o}\rangle$); $inj_r(P)$ 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:

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

The notation $\langle P_1 \cdot P_2 \rangle$ denotes the combination of 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. The semantic functions $prj_o(\hat{P})$ and $prj_r(\hat{P})$ project a relational formula in $\hat{P}$ 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:

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

The projection functions 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. I use projection to define the following relations between relational and nonrelational formulas:

$$\hat{P} \models_o P \equiv prj_o(\hat{P}) \subseteq [\![P]\!]$$
$$\hat{P} \models_r P \equiv prj_r(\hat{P}) \subseteq [\![P]\!]$$

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

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

$$\sigma \models P \equiv \sigma \in [\![P]\!] \qquad (\sigma_1,\sigma_2) \models \hat{P} \equiv (\sigma_1,\sigma_2) \in [\![\hat{P}]\!]$$
$$\models P_1 \Rightarrow P_2 \equiv [\![P_1]\!] \subseteq [\![P_2]\!] \qquad \models \hat{P}_1 \Rightarrow \hat{P}_2 \equiv [\![\hat{P}_1]\!] \subseteq [\![P_2]\!]$$

2.3.2 Original Semantics

Figure 2-8 presents a manual translation of the 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. Note that this definition asserts only partial correctness and not total correctness.

$$\boxed{\vdash_o \{P\}\ s\ \{Q\}} \qquad \text{skip}\frac{}{\vdash_o \{P\}\ \texttt{skip}\ \{P\}} \qquad \text{seq}\frac{\vdash_o \{P\}\ s_1\ \{R\} \quad \vdash_o \{R\}\ s_2\ \{Q\}}{\vdash_o \{P\}\ s_1;s_2\ \{Q\}}$$

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

$$\text{assert}\frac{}{\vdash_o \{P \wedge e\}\ \texttt{assert (}e\texttt{)}\ \{P \wedge e\}} \qquad \text{assume}\frac{}{\vdash_o \{P\}\ \texttt{assume (}e\texttt{)}\ \{P \wedge e\}}$$

$$\text{relax}\frac{\vdash_o \{P\}\ \texttt{assert (}e\texttt{)}\ \{Q\}}{\vdash_o \{P\}\ \texttt{relax (}X\texttt{) st (}e\texttt{)}\ \{Q\}} \qquad \text{relate}\frac{}{\vdash_o \{P\}\ \texttt{relate}\ l\texttt{: (}\hat{e}\texttt{)}\ \{P\}}$$

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

$$\text{conseq}\frac{\models P \Rightarrow P' \quad \vdash_o \{P'\}\ s\ \{Q'\} \quad \models Q' \Rightarrow Q}{\vdash_o \{P\}\ s\ \{Q\}}$$

$$\text{const}\frac{\vdash_o \{P\}\ s\ \{Q\} \quad \mathit{free}(R) \cap \mathit{mods}(s) = \emptyset}{\vdash_o \{P \wedge R\}\ s\ \{P \wedge R\}}$$

Figure 2-8: Axiomatic Original Semantics

Skip. The skip rule has a standard semantics. Because the `skip` statement does not modify the state of the program, if P holds before the execution of the statement, then P holds after.

Sequence. The sequential composition rule has a standard semantics. If a predicate P holds before the execution of the sequence, then given proofs that 1) R holds after the execution of s_1 and 2) starting from a state satisfying R, execution of s_2 yields a state satisfying Q, then Q holds after execution of the sequence of statements.

Assignment. The assignment statement follows the standard backwards formalization of assignment [38]. Specifically, for any state that results from the execution of the assignment statement and satisfies Q, the statement must have began its execution from a state satisfying $Q[e/x]$.

Havoc. 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.

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

Assume. 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, `assume` statements may fail in the dynamic original semantics.

Relax. The relax rule specifies that a `relax` statement is a no-op that does not change the program's state in the original semantics. However, the programming model's definition of relaxation also requires that the original execution must still satisfy e. To enforce this constraint, the rule reuses the rule for the `assert` statement.

Relate. The relate rule gives `relate` statements the same semantics as `skip` because, unlike the axiomatic relaxed semantics (see Section 2.3.3), the axiomatic original semantics references only a single execution of the program and does not use relational reasoning.

If. The rule for `if` statements has a standard semantics. If P holds before the execution of the statement then Q holds after the statement's execution given proofs that 1) if $P \wedge b$ holds before the execution of s_1, then Q holds after and 2) if $P \wedge \neg b$ before holds before the execution of s_2, then Q holds after.

While. The rule for `while` loops also has a standard semantics. Specifically, given an *invariant* P that holds before the execution of the loop and a proof that each iteration of the loop preserves the invariant (i.e., if $P \wedge b$ holds before the execution of s, then P still holds after), then if the loop terminates, the resulting state satisfies $P \wedge \neg b$.

Consequence. The rule of consequence is a standard component of Hoare-style program logics that enables weakening of a program's predicates. The rule states that the judgment $\vdash_o \{P\}\ s\ \{Q\}$ can alternatively hold given a proof $\vdash_o \{P'\}\ s\ \{Q'\}$ where P' is weaker than

P (i.e., $\models P \Rightarrow P'$) and Q is weaker than Q' (i.e., $\models Q' \Rightarrow Q$. A proof from the initial states captured by P' implies that the proof is also valid for the subset of those states captured by P. Similarly, if the proof establishes that the resulting states are members of the set captured by Q', then those states are also members of the larger set captured by Q.

Constancy. The rule of constancy is standard component of Hoare-style program logics that enables modular reasoning about predicates that are not modified by the side-effects of a statement. The rule states that if a conjunction of predicates $P \wedge R$ holds before the execution of a statement s and it can be shown that 1) $\vdash_o \{P\}\ s\ \{Q\}$ and 2) the set of free variables of R is disjoint from the set of variables modified by s, then $Q \wedge R$ holds after the statement's execution.

2.3.3 Relaxed Semantics

Figure 2-9 presents a manual translation of my 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 $\models_r \{\hat{P}\}\ s\ \{\hat{Q}\}$ is the partial correctness assertion: if $(\sigma_o, \sigma_r) \models \hat{P}$, $\langle s,\ \sigma_o\rangle \Downarrow_o \langle \sigma_o',\ \psi_1\rangle$, and $\langle s,\ \sigma_r\rangle \Downarrow_r \langle \sigma_r',\ \psi_2\rangle$, then $(\sigma_o', \sigma_r') \models \hat{Q}$.

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.

Skip. The `skip` statement does not modify the state of the program therefore, similar to the axiomatic original semantics, if the relation $\hat{P}$ holds before the execution of the statement, then $\hat{P}$ also holds after.

$$\text{skip}\frac{}{\vdash_r \{\hat{P}\}\ \texttt{skip}\ \{\hat{P}\}} \qquad \text{assign}\frac{}{\vdash_r \{\hat{Q}[inj_o(e)/x\langle \texttt{o}\rangle][inj_r(e)/x\langle \texttt{r}\rangle]\}\ x = e\ \{\hat{Q}\}}$$

$$\text{havoc}\frac{[\![(\exists_{X'\langle \texttt{o}\rangle}\cdot \exists_{X'\langle \texttt{r}\rangle}\cdot \hat{P}[X'\langle \texttt{o}\rangle/X\langle \texttt{o}\rangle][X'\langle \texttt{r}\rangle/X\langle \texttt{r}\rangle]) \wedge inj_o(e) \wedge inj_r(e)]\!] \neq \emptyset \quad fresh(X')}{\vdash_r \{\hat{P}\}\ \texttt{havoc}\ (X)\ \texttt{st}\ (e)\ \{(\exists_{X'\langle \texttt{o}\rangle}\cdot \exists_{X'\langle \texttt{r}\rangle}\cdot \hat{P}[X'\langle \texttt{o}\rangle/X\langle \texttt{o}\rangle][X'\langle \texttt{o}\rangle/X\langle \texttt{o}\rangle]) \wedge inj_o(e) \wedge inj_r(e)\}} \qquad \boxed{\vdash_r \{\hat{P}\}\ s\ \{\hat{Q}\}}$$

$$\text{relax}\frac{[\![(\exists_{X'\langle \texttt{r}\rangle}\cdot \hat{P}[X'\langle \texttt{r}\rangle/X\langle \texttt{r}\rangle]) \wedge inj_r(e)]\!] \neq \emptyset \quad fresh(X')}{\vdash_r \{\hat{P}\}\ \texttt{relax}\ (X)\ \texttt{st}\ (e)\ \{(\exists_{X'\langle \texttt{r}\rangle}\cdot \hat{P}[X'\langle \texttt{r}\rangle/X\langle \texttt{r}\rangle]) \wedge \langle e\cdot e\rangle\}} \qquad \text{assert}\frac{\models \hat{P} \wedge inj_o(e) \Rightarrow inj_r(e)}{\vdash_r \{\hat{P}\}\ \texttt{assert}\ (e)\ \{\hat{P} \wedge \langle e\cdot e\rangle\}}$$

$$\text{assume}\frac{\models \hat{P} \wedge inj_o(e) \Rightarrow inj_r(e)}{\vdash_r \{\hat{P}\}\ \texttt{assume}\ (e)\ \{\hat{P} \wedge \langle e\cdot e\rangle\}} \qquad \text{relate}\frac{}{\vdash_r \{\hat{P} \wedge \hat{e}\}\ \texttt{relate}\ l : (\hat{e})\ \{\hat{P} \wedge \hat{e}\}} \qquad \text{seq}\frac{\vdash_r \{\hat{P}\}\ s_1\ \{\hat{R}\} \quad \vdash_r \{\hat{R}\}\ s_2\ \{\hat{Q}\}}{\vdash_r \{\hat{P}\}\ s_1; s_2\ \{\hat{Q}\}}$$

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

$$\text{while}\frac{\models \hat{P} \Rightarrow \langle b\cdot b\rangle \vee \langle \neg b\cdot \neg b\rangle \quad \vdash_r \{\hat{P} \wedge \langle b\cdot b\rangle\}\ s\ \{\hat{P}\}}{\vdash_r \{\hat{P}\}\ \texttt{while}\ (b)\ \{s\}\ \{\hat{P} \wedge \langle \neg b\cdot \neg b\rangle\}} \qquad \text{diverge}\frac{\hat{P} \models_o P_o \quad \hat{P} \models_r P_r \quad \vdash_o \{P_o\}\ s\ \{Q_o\} \quad \vdash_i \{P_r\}\ s\ \{Q_r\} \quad no\_rel(s)}{\vdash_r \{\hat{P}\}\ s\ \{\langle Q_o\cdot Q_r\rangle\}}$$

$$\text{conseq}\frac{\models \hat{P} \Rightarrow \hat{R} \quad \vdash_r \{\hat{R}\}\ s\ \{\hat{S}\} \quad \models \hat{S} \Rightarrow \hat{Q}}{\vdash_r \{\hat{P}\}\ s\ \{\hat{Q}\}} \qquad \text{const}\frac{\vdash_r \{\hat{P}\}\ s\ \{\hat{Q}\} \quad \hat{free}_r(\hat{R}) \cap \hat{mods}_r(s) = \emptyset}{\vdash_r \{\hat{P} \wedge \hat{R}\}\ s\ \{\hat{Q} \wedge \hat{R}\}}$$

Figure 2-9: Axiomatic Relaxed Semantics

Assignment. The rule for assignment extends the backwards semantics of assignment present in the axiomatic original semantics to capture the fact that the assignment updates values in both the original and relaxed executions of the program. The rule works by apply two substitutions to the predicate $\hat{Q}$, which holds after the statement's execution. The first substitution $\hat{Q}[inj_o(e)/x\langle \mathtt{o} \rangle]$ substitutes all references to the value of x in the original program ($x\langle \mathtt{o} \rangle$) with the expression e as renamed to refer to variables in the original program $inj_o(e)$. The second substitution $[inj_r(e)/x\langle \mathtt{r} \rangle]$ operates similarly for the relaxed semantics of the program.

Havoc. The rule havoc statements is similar to the havoc rule in the axiomatic original semantics except that 1) it deals with a relational formula and 2) characterizes the changes in both the original and relaxed executions of the program. The uses 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 (and similarly $X\langle \mathtt{o} \rangle$ for the extension to original variables). Following the form of the havoc from the axiomatic original semantics, the rule uses these shorthands to check the satisfiability of the condition of the `havoc` statement and generate the final predicate that holds after the statement's execution.

Relax. 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 except that 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.

Assert. The assert rule demonstrates how it is possible 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., $inj_o(e)$) because it has been verified with the axiomatic original semantics. If $\hat{P} \wedge inj_o(e)$ implies $inj_r(e)$, we can then conclude that the assertion is true in the relaxed semantics (i.e., $inj_r(e)$).

Assume. The assume rule demonstrates how relational reasoning enables one to use relations between the original and relaxed semantics of the program to reason about assumptions in the relaxed semantics in the same way as 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.

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

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, one can conclude that the assertion is also true in the dynamic relaxed semantics.

Sequence. The rule for sequential composition is similar in form to that of the axiomatic original semantics with the key difference being that the rule operates over relations.

Consequence. The rule of consequence is also similar in form to that of the axiomatic original semantics.

Constancy. The rule of constancy also has a similar form as that of the axiomatic original semantics. The primary difference is that the rule uses the functions $\hat{\mathit{free}}$ and $\hat{\mathit{mods}}$, which compute the free variables of a relational predicate and the set of variables modified by the statement under both the original and relaxed semantics. Because both $\hat{\mathit{free}}$ and $\hat{\mathit{mods}}$ are relational, they may return either original or relaxed variables.

Convergent Control Flow 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, precludes relational reasoning. 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, therefore reenabling relational reasoning.

$\boxed{\langle s, \langle \sigma_o, \sigma_r \rangle\rangle \Downarrow \langle \sigma_o', \sigma_r' \rangle, C}$

$$\text{if-t-f}\frac{\langle b\ ,\ \sigma_o\rangle \Downarrow_B \mathit{true} \qquad \langle b\ ,\ \sigma_r\rangle \Downarrow_B \mathit{false} \qquad \langle \texttt{if (}b\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}},\ \sigma_o\rangle \Downarrow_o \sigma_o' \qquad \langle \texttt{if (}b\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}},\ \sigma_r\rangle \Downarrow_r \sigma_r'}{\langle \ell_b \texttt{if (}b\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}}_{\ell_a}, \langle \sigma_o, \sigma_r\rangle\rangle \Downarrow \langle \sigma_o', \sigma_r'\rangle, \{\ell_b, \ell_a\}}$$

$$\text{if-f-t}\frac{\langle b\ ,\ \sigma_o\rangle \Downarrow_B \mathit{false} \qquad \langle b\ ,\ \sigma_r\rangle \Downarrow_B \mathit{true} \qquad \langle \texttt{if (}b\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}},\ \sigma_o\rangle \Downarrow_o \sigma_o' \qquad \langle \texttt{if (}b\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}},\ \sigma_r\rangle \Downarrow_r \sigma_r'}{\langle \ell_b \texttt{if (}b\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}}_{\ell_a}, \langle \sigma_o, \sigma_r\rangle\rangle \Downarrow \langle \sigma_o', \sigma_r'\rangle, \{\ell_b, \ell_a\}}$$

$$\text{if-t-t}\frac{\langle b\ ,\ \sigma_o\rangle \Downarrow_B \mathit{true} \qquad \langle b\ ,\ \sigma_r\rangle \Downarrow_B \mathit{true} \qquad \langle s_1, \langle \sigma_o, \sigma_r\rangle\rangle \Downarrow \langle \sigma_o', \sigma_r'\rangle, C}{\langle \ell_b \texttt{if (}b\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}}_{\ell_a}, \langle \sigma_o, \sigma_r\rangle\rangle \Downarrow \langle \sigma_o', \sigma_r'\rangle, \{\ell_b, \ell_a\} \cup C \cup \mathrm{in}(s_2)}$$

$$\text{if-f-f}\frac{\langle b\ ,\ \sigma_o\rangle \Downarrow_B \mathit{false} \qquad \langle b\ ,\ \sigma_r\rangle \Downarrow_B \mathit{false} \qquad \langle s_2, \langle \sigma_o, \sigma_r\rangle\rangle \Downarrow \langle \sigma_o', \sigma_r'\rangle, C}{\langle \ell_b \texttt{if (}b\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}}_{\ell_a}, \langle \sigma_o, \sigma_r\rangle\rangle \Downarrow \langle \sigma_o', \sigma_r'\rangle, \{\ell_b, \ell_a\} \cup C \cup \mathrm{in}(s_1)}$$

$$\text{while-t-f}\frac{\langle b\ ,\ \sigma_o\rangle \Downarrow_B \mathit{true} \qquad \langle b\ ,\ \sigma_r\rangle \Downarrow_B \mathit{false} \qquad \langle \texttt{while (}b\texttt{) \{}s\texttt{\}},\ \sigma_o\rangle \Downarrow_o \sigma_o'}{\langle \ell_b \texttt{while (}b\texttt{) \{}s\texttt{\}}_{\ell_a}, \langle \sigma_o, \sigma_r\rangle\rangle \Downarrow \langle \sigma_o', \sigma_r\rangle, \{\ell_b, \ell_a\}}$$

$$\text{while-f-t}\frac{\langle b\ ,\ \sigma_o\rangle \Downarrow_B \mathit{false} \qquad \langle b\ ,\ \sigma_r\rangle \Downarrow_B \mathit{true} \qquad \langle \texttt{while (}b\texttt{) \{}s\texttt{\}},\ \sigma_r\rangle \Downarrow_r \sigma_r'}{\langle \ell_b \texttt{while (}b\texttt{) \{}s\texttt{\}}_{\ell_a}, \langle \sigma_o, \sigma_r\rangle\rangle \Downarrow \langle \sigma_o, \sigma_r'\rangle, \{\ell_b, \ell_a\}}$$

$$\text{while-f-f}\frac{\langle b\ ,\ \sigma_o\rangle \Downarrow_B \mathit{false} \qquad \langle b\ ,\ \sigma_r\rangle \Downarrow_B \mathit{false}}{\langle \ell_b \texttt{while (}b\texttt{) \{}s\texttt{\}}_{\ell_a}, \langle \sigma_o, \sigma_r\rangle\rangle \Downarrow \langle \sigma_o, \sigma_r\rangle, \mathrm{in}(\texttt{while (}b\texttt{) \{}s\texttt{\}})}$$

$$\text{while-t-t}\frac{\langle b\ ,\ \sigma_o\rangle \Downarrow_B \mathit{true} \qquad \langle b\ ,\ \sigma_r\rangle \Downarrow_B \mathit{true} \qquad \langle s, \langle \sigma_o, \sigma_r\rangle\rangle \Downarrow \langle \sigma_o', \sigma_r'\rangle, C \qquad \langle \texttt{while (}b\texttt{) \{}s\texttt{\}}, \langle \sigma_o, \sigma_r\rangle\rangle \Downarrow \langle \sigma_o'', \sigma_r''\rangle, C'}{\langle \ell_b \texttt{while (}b\texttt{) \{}s\texttt{\}}_{\ell_a}, \langle \sigma_o, \sigma_r\rangle\rangle \Downarrow \langle \sigma_o'', \sigma_r''\rangle, (\{\ell_b, \ell_a\} \cup C) \cap C')}$$

$$\text{seq}\frac{\langle s_1, \langle \sigma_o, \sigma_r\rangle\rangle \Downarrow \langle \sigma_o', \sigma_r'\rangle, C \qquad \langle s_2, \langle \sigma_o', \sigma_r'\rangle\rangle \Downarrow \langle \sigma_o'', \sigma_r''\rangle, C'}{\langle \ell_b s_1\ ;\ s_2 \ell_a, \langle \sigma_o, \sigma_r\rangle\rangle \Downarrow \langle \sigma_o'', \sigma_r''\rangle, \{\ell_b, \ell_a\} \cup C \cup C'}$$

Figure 2-10: The Convergent Program Points of Dynamic Original and Relaxed Executions

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.

Convergent Program Points. I formally define the *convergent program points* of a pair of dynamic original and relaxed execution of a program, and then the convergent program points of the whole program. Figure 2-10 presents how the convergent program

points are defined for a pair of dynamic original and relaxed execution. The relation $\langle s, \langle \sigma_o, \sigma_r \rangle \rangle \Downarrow \langle \sigma_o', \sigma_r' \rangle, C$ denotes that C is the set of convergent program points when the dynamic original evaluation of a statement s from a state σ_o yields an output state σ_o' and its relaxed evaluation from a state σ_r yields a state σ_r'.

I use the following notation in Figure 2-10. In each rule, ℓ_b and ℓ_a denote static program points just before and after the given statement, respectively. The set of all program points in a statement s, including its before/after program points, is denoted by $\text{in}(s)$. Some rules use the relations $\Downarrow_o$ and $\Downarrow_r$ (defined in Figure 2-3 and 2-5), but I omit the observation list from an output configuration because it is not relevant here.

Figure 2-10 shows only the rules for control flow statements. The convergent program points of execution of the other statements that do not relate to control flow are just their before/after program points.

The `if` statement. If the condition b evaluates to different truth values in the dynamic original and relaxed semantics (if-t-f and if-f-t), the convergent program points are just the before/after program points of the `if` statement because its control flow diverges within the `if` statement.

If the condition b evaluates to the same truth value in both the dynamic original and relaxed semantics (if-t-t and if-f-f), the convergent program points of the two execution consist of its before/after program points and the convergent program points included in the executed branch (C). Also it includes all program points in the un-executed branch ($\text{in}(s_2)$ or $\text{in}(s_1)$), which allows the `accept` statement to appear at any unreachable program point. Note that an unreachable `accept` statement is trivially true.

The `while` statement. If the condition b evaluates differently in the dynamic original and relaxed semantics (while-t-f and while-f-t), the convergent program points are just its before/after program points. Therefore, the `accept` statement cannot exist in the loop body.

If the condition b evaluates to false in both the dynamic original and relaxed semantics (while-f-f), the convergent program points are all program points inside the `while` statement, which I denote by $\text{in}(\texttt{while (}b\texttt{) \{}s\texttt{\}})$. In other words, all the unreachable program points inside the loop body are convergent program points as well as the before/after program points of the `while` statement.

If the condition b evaluates to true in both the dynamic original and relaxed semantics (while-t-t), the convergent program points are the intersection of the convergent program points of the current iteration ($\{\ell_b, \ell_a\} \cup C$) and those of the following iterations (C'). Note that either the while-t-f or while-f-t rule is guaranteed to be applied when the loop body is executed different number of times in the original and relaxed semantics. In that case, the convergent program points of the later iterations (C') are the before/after program points of the `while` statement because the while-t-f and while-f-t rules do not include any program points inside the loop body s. So, the convergent program points of a `while` statement that executes a different number of times in the dynamic original and relaxed semantics are just the before/after program points of the `while` statement.

Sequential Composition. The convergent program points of execution of sequential composition are the union of the before/after program points, and the first and second statements' own convergent programs points (seq).

Definition 1 (Convergent Program Points). *Given a program s, its* convergent program points *are* $\mathrm{CPP}(s) = \bigcap_{\sigma \in \Sigma}(\bigcap\{C \mid \langle s, \langle \sigma, \sigma \rangle\rangle \Downarrow \langle \sigma', \sigma'' \rangle, C\})$.

A program's convergent program points are the intersection of the convergent program points for all possible pairs of dynamic original and relaxed executions of the program.

Convergent Control Flow Rules. The `if` rule allows us to continue to use relational reasoning inside an `if` statement if it has convergent control flow. I establish this convergence by checking that for all σ_1, σ_2, if $(\sigma_1, \sigma_2) \models \hat{P}$ 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, therefore enabling the use of relational reasoning within the body of a `while` statement.

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

$$\text{relax}\frac{\vdash_i \{P\}\ \texttt{havoc}\ (X)\ \texttt{st}\ (e)\ \{Q\}}{\vdash_i \{P\}\ \texttt{relax}\ (X)\ \texttt{st}\ (e)\ \{Q\}}$$

$$\text{assume}\frac{}{\vdash_i \{P \wedge e\}\ \texttt{assume}\ (e)\ \{P \wedge e\}} \qquad \text{skip}\frac{\vdash_o \{P\}\ \texttt{skip}\ \{P\}}{\vdash_i \{P\}\ \texttt{skip}\ \{P\}} \qquad \text{assign}\frac{\vdash_o \{P\}\ \texttt{x} = e\ \{Q\}}{\vdash_i \{P\}\ \texttt{x} = e\ \{Q\}}$$

$$\text{havoc}\frac{\vdash_o \{P\}\ \texttt{havoc}\ (X)\ \texttt{st}\ (e)\ \{Q\}}{\vdash_i \{P\}\ \texttt{havoc}\ (X)\ \texttt{st}\ (e)\ \{Q\}} \qquad \text{assert}\frac{\vdash_o \{P\}\ \texttt{assert}\ (e)\ \{P \wedge e\}}{\vdash_i \{P\}\ \texttt{assert}\ (e)\ \{P \wedge e\}}$$

$$\text{seq}\frac{\vdash_i \{P\}\ s_1\ \{R\} \quad \vdash_i \{R\}\ s_2\ \{Q\}}{\vdash_i \{P\}\ s_1; s_2\ \{Q\}} \qquad \text{if}\frac{\vdash_i \{P \wedge b\}\ s_1\ \{Q\} \quad \vdash_i \{P \wedge \neg b\}\ s_2\ \{Q\}}{\vdash_i \{P\}\ \texttt{if}\ (b)\ \{s_1\}\ \texttt{else}\ \{s_2\}\ \{Q\}}$$

$$\text{while}\frac{\vdash_i \{P \wedge b\}\ s\ \{P\}}{\vdash_i \{P\}\ \texttt{while}\ (b)\ \{s\}\ \{P \wedge \neg b\}} \qquad \text{conseq}\frac{\models P \Rightarrow P' \quad \vdash_i \{P'\}\ s\ \{Q'\} \quad \models Q' \Rightarrow Q}{\vdash_i \{P\}\ s\ \{Q\}}$$

Figure 2-11: Axiomatic Intermediate Semantics

Divergent Control Flow Rules. 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 $\hat{P}$. The judgment $\vdash_i \{P_r\}\ s\ \{Q_r\}$ is a set of proof rules for the axiomatic *intermediate* semantics of the program. Figure 2-11 presents 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 valid.

I 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 it is not possible to use relational the relational assertion logic establish that the `relate` statement is satisfied.

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.

2.4 Properties

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

The key property of my language and proof rules is Relative Relaxed Progress (Section 2.4.3), which states that the 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 my formalization, I restrict myself to terminating relaxed programs. Also, while the following sections only present proof sketches, the full sources of my Coq formalization and proofs are included in Appendix A.

2.4.1 Original Semantics

The 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 my dynamic semantics.

Lemma 1 (Soundness)**.**

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

This lemma establishes that the 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 `original_axiomatic_soundness` in Section A.10 of the Appendix presents the full Coq source of the proof. □

Lemma 2 (Original Progress Modulo Assumptions)**.**

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

This lemma establishes the progress property 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 $\vdash_o \{P\}\ s\ \{Q\}$. One only needs to consider three primitive statements for which 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). Lemma `original_axiomatic_progress` in Section A.10 of the Appendix presents the full Coq source of the proof. □

2.4.2 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, I 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)**.**

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

This lemma establishes that the 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 `intermediate_axiomatic_soundness` in Section A.11 presents the full Coq source of the proof. □

Lemma 4 (Progress).

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

This lemma establishes the progress property 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 the 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`). Lemma `intermediate_axiomatic_progress` in Section A.11 presents the full Coq source of the proof. □

2.4.3 Relaxed Semantics

Lemma 5 (Soundness).

$$\textbf{\textit{If}}\ \vdash_r \{\hat{P}\}\ s\ \{\hat{Q}\}, \textbf{\textit{then}} \models_r \{\hat{P}\}\ s\ \{\hat{Q}\}$$

This lemma establishes that the axiomatic definition is sound with respect to the original and relaxed semantics of the program. Specifically, given a proof $\vdash_r \{\hat{P}\}\ s\ \{\hat{Q}\}$, it is the case that for all states $(\sigma_o, \sigma_r) \models \hat{P}$, 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 it is also the case that $(\sigma_o', \sigma_r') \models \hat{Q}$.

Proof Sketch. The proof proceeds by induction on the rules of $\vdash_r \{\hat{P}\}\ s\ \{\hat{Q}\}$. 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 `relaxed_axiomatic_soundness` in Section A.12 presents the full Coq source of the proof. □

Theorem 6 (Soundness of Relational Assertions).

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

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; I define the relation as follows:

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

The symbol Γ represents a finite map from `relate` labels to relational boolean expressions (i.e, $\Gamma \in L \to \hat{B}$). I define this map by structural induction on the syntax of the program, where the label of each `relate` statement in the program is unique and maps to its relational boolean expression.

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 $\vdash_r \{\hat{P}\}\ s\ \{\hat{Q}\}$. The two interesting cases are the *diverge* rule and the rule for `relate` statements. For the diverge rule, I use the fact that the rule requires $no\_rel(s)$ (which requires that no `relate` statements appear inside s). I 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 `relational_assertion_soundness` in Section A.12 presents the full Coq source of the proof. □

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

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

This theorem establishes the relative progress guarantee for the relaxed semantics of the program. Specifically, given a proof $\vdash_r \{\hat{P}\}\ s\ \{\hat{Q}\}$, it is the case that for all pairs of states (σ_o, σ_r) that satisfy $\hat{P}$, 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 $\vdash_r \{\hat{P}\}\ s\ \{\hat{Q}\}$. 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. One does 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 the design of the axiomatic intermediate semantics. For this rule, one 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 `relaxed_axiomatic_relative_progress` in Section A.12 presents the full Coq source of the proof. □

Theorem 8 (Relaxed Progress)**.**

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

This theorem combines the multiple proofs and assumptions in the 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 the assumptions, Lemma 2 (Original Progress Modulo Assumptions), and Theorem 7 (Relative Progress). The assumptions and Lemma 2 establish that if an original execution terminates, then it does not terminate in error. By Theorem 7, I can conclude that if a relaxed execution terminates, it does not produce an error. Theorem `relaxed_axiomatic_progress` in Section A.12 presents the full Coq source of the proof.

□

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

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

This corollary of Theorem 8 captures an important aspect of how the 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.

2.5 Example Relaxed Programs

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

2.5.1 Dynamic Knobs

Swish++ is an open-source search engine. I 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 [39]. 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 I 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 my proof environment, verified by an automated theorem prover.

2.5.2 Statistical Automatic Parallelization

My next example is drawn from a parallelization of the Water computation [16] with statistical accuracy bounds [55]. 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. I model the relaxation nondeterminism by relaxing each element in `RS` with no constraints:[1]

```
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;
}
```

[1] Although I do not present a treatment of arrays in this chapter, I provide a full presentation in Chapter 3.

Acceptability. A key acceptability property is that K stays within the bounds of the array FF.[2] The array bounds are stored in the variable len_FF. I 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].

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 original assume statement at the point at which it appears.

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 me to prove that the relaxation does not interfere with the assumption.

2.5.3 Approximate Memory and Data Types

My 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 [49, 84].

I 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.

[2] I note that K must also be within the bounds of RS; the proof is similar.

```
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 [65], if `A` is stored in approximate memory, then it is possible to 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`. This property can be specified with a `relate` statement:

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

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.

2.6 Related Work

Executable Specifications. Executable specifications, via techniques such as refinement and constraint solving, produce concrete outputs that satisfy the specification [60, 72, 83, 93, 28, 53]. Applications for executable specifications 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 thesis differs in that it promotes nondeterministic relaxation to obtain semantically different but still acceptable variants of the original program. My 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 [21, 49, 84]. 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 thesis 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. My program logic for the relaxed semantics of the program builds on previous work on the Relational Hoare Logic (RHL) [12]. RHL itself was inspired by Credible Compilation [82] and Translation Validation [70] and and has since inspired other forms of relational reasoning. Researchers have also defined relational separation logic [92, 7], probabilistic Hoare logic [11], and have used relational reasoning to verify the correctness of semantics-preserving compiler transformations [82, 95, 26], Lipschitz-continuity [26], access control policies [63], and differential privacy mechanisms [11].

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

2.7 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 [77, 78, 94, 5, 3, 58, 57, 88, 24, 55, 79, 39, 40, 84, 49] or responding to load spikes or other fluctuations in the characteristics of the underlying computational platform [39, 40, 77, 84].

I 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. My 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. My 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.

Chapter 3

From Core Calculus to Programming Language

The previous chapter presents a core calculus for relaxed programming that consists of a basic imperative programming language and a program logic that demonstrates the core concepts of verifying relaxed programs. Specifically, a key theme of the work is that relational reasoning can enable simpler proofs of the acceptability of relaxed programs that reuse the reasoning done for the original program.

In this chapter, I extend this core calculus – including its relational reasoning approach – to include features in common programming languages along with their associated reasoning principles. These extensions include extending the logic's support for control flow, adding dynamically allocated data structures (i.e., primitive types, structured types, and arrays), and adding procedures. Extending the logic's control flow support enables additional opportunities for relational verification for case where the original and relaxed program diverge at a control flow point. Adding reasoning principles for dynamically allocated structures enables proofs of rich properties of the broad range of data structures of relaxed programs. Adding procedures enables reasoning about modularly decomposed relaxed programs where the scope of both the relaxation and the properties that the program must satisfy span more than a small computational kernel.

$$
\begin{aligned}
x, y, p, a &\in \mathit{Vars} \\
f &\in F \\
\mathit{Prim} &:= \texttt{int} \mid \texttt{float} \\
T &:= \mathit{Prim} \mid \mathit{id} \\
\mathit{ArrayType} &:= \mathit{Prim}\{n\} \mid T*\{n\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Struct} &:= \texttt{struct}\ \mathit{id}\ \{\ \mathit{SDecl}\ \} \\
\mathit{SDecl} &::= \mathit{Prim} f;\ \mathit{SDecl} \mid T*f;\ \mathit{SDecl} \mid \mathit{ArrayType} f;\ \mathit{SDecl}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Proc} &::= \mathit{Ret}\ \mathit{fn}(\mathit{Params})\, \mathit{Pre}?\ \mathit{Post}?\ \mathit{Pre}_{rel}?\ \mathit{Post}_{rel}?\ \{\ S\ \} \\
\mathit{Ret} &::= T \mid \mathit{ArrayType} \mid \texttt{void} \\
\mathit{Param} &::= \mathit{Prim}\, x \mid T*\, x \mid \mathit{ArrayType}\, a \\
\mathit{Params} &::= \varepsilon \mid \mathit{Param}, \mathit{Params} \\
\mathit{Pre} &::= \texttt{requires}\ P \\
\mathit{Post} &::= \texttt{ensures}\ P \\
\mathit{Pre}_{rel} &::= \texttt{requires-r}\ \hat{P} \\
\mathit{Post}_{rel} &::= \texttt{ensures-r}\ \hat{P}
\end{aligned}
$$

$$
\begin{aligned}
E &::= \texttt{n} \mid \texttt{x} \mid E\ \texttt{iop}\ E \\
\mathit{Mval} &::= x \mid *p \mid p.f \mid a[E, ..., E] \\
\dot{E} &::= \texttt{n} \mid \mathit{Mval} \mid \texttt{old}(\mathit{Mval}) \mid\ \dot{E}\ \texttt{iop}\ \dot{E} \\
\hat{E} &::= \texttt{n} \mid \texttt{x}\langle\texttt{o}\rangle \mid \texttt{x}\langle\texttt{r}\rangle \mid \hat{E}\ \texttt{iop}\ \hat{E}
\end{aligned}
$$

$$
\begin{aligned}
B &::= \texttt{true} \mid \texttt{false} \mid E\ \texttt{cmp}\ E \mid B\ \texttt{lop}\ B \mid \neg B \\
\dot{B} &::= \texttt{true} \mid \texttt{false} \mid \dot{E}\ \texttt{cmp}\ \dot{E} \mid \dot{B}\ \texttt{lop}\ \dot{B} \mid \neg \dot{B} \\
\hat{B} &::= \texttt{true} \mid \texttt{false} \mid \hat{E}\ \texttt{cmp}\ \hat{E} \mid \hat{B}\ \texttt{lop}\ \hat{B} \mid \neg \hat{B}
\end{aligned}
$$

$$
\begin{aligned}
S ::= {} & \texttt{skip} \\
& \mid \mathit{Prim}\, x = E \mid T*\, x = (0 \mid y \mid \texttt{new}\ T) \mid (0 \mid y \mid \mathit{ArrayType}\, x = \texttt{new}\ T[E, ..., E]) \\
& \mid x = E \mid *p = E \mid p.f = E \mid a[E, ..., E] = E \\
& \mid x = *p \mid x = p.f \mid x = a[E, ..., E] \\
& \mid x = \texttt{len}(a, n) \\
& \mid \texttt{havoc}\ (A, X)\ \texttt{st}\ (\dot{B}) \mid \texttt{relax}\ (A, X)\ \texttt{st}\ (\dot{B}) \\
& \mid x = \mathit{fn}(E, ..., E) \mid \mathit{fn}(E, ..., E) \mid \texttt{return}\ E \\
& \mid \texttt{assume}\ (B) \mid \texttt{assert}\ (B) \mid \texttt{relate}\ \texttt{l}: (\hat{B}) \\
& \mid \texttt{if}\ (B)\ \{S\}\ \texttt{else}\ \{S\} \mid \texttt{while}\ (B)\ \{S\} \mid S\ ;\ S
\end{aligned}
$$

Figure 3-1: Language Syntax

3.1 Language Syntax

Figure 3-1 presents the extended syntax of the language. Many of the basic statements of the language (production *S*) are similar to those in the core calculus. However, the basic operands of these operations differ in that they may include references to *structured data types*, first-class arrays, and calls to procedures.

3.1.1 Data Types

The language first extends the singular *primitive* integer type of the core calculus to also include floating-point primitive types (production *Prim*).

The production *Struct* extends the language to include structured data types. A structured data type declaration begins with the `struct` keyword, then specifies the name of the type (*id*) and provides a list of *field* declarations, which specify the names and types of the structure's data members. A field is either a primitive value (*Prim f*), a pointer to a heap-allocated structure (*T* f*), or a pointer to a heap-allocated array (*ArrayType f*). The production *ArrayType* specifies the type and structure of the array. Specifically, an array is *n*-dimensional and contains either primitive values (*Prim* {*n*}) or pointers to structured data types (*T** {*n*}).

3.1.2 Procedures

The production *Proc* specifies the syntax of procedure declarations using a C-style declaration syntax. A procedure declaration first specifies a return value (*Ret*) that is either a primitive value, a pointer to a heap-allocated structured type, or a pointer to a heap-allocated array. The return value may also be `void`, which indicates that the function does not return a value.

The declaration syntax follows the return value specification with the name of the function (*fn*), a list of parameters to the function, a list of *precondition* and *postcondition* specifications, and then the function body itself. A procedure may receive as parameters a list of values that may either be a primitive value, a pointer to a heap-allocated structured type, or a pointer to an array. The procedure's precondition and postcondition specifications use a syntax that is similar to that of other verification-enabled languages (e.g., ESC/Java [36])

to specify that the function has a precondition – a predicate that most hold before execution of the function (`requires` P) – or a postcondition – a predicate that is guaranteed (through verification) to hold after the execution of the function (`ensures` P). Unlike existing languages, procedures may also have *relational* preconditions and postconditions, which specify relations between the values in the original program and the relaxed program that must hold before and after the execution of the program, respectively. The annotation `requires-r` $\hat{P}$ specifies a relational precondition and `ensures-r` $\hat{P}$ specifies a relational postcondition. Section 3.3.1 presents the syntax and semantics of the assertion logic for specifying P and $\hat{P}$.

3.1.3 Expressions and Statements

Introducing heap-allocated primitives, structures, and arrays into the language necessitates new expressions and statements for allocating, writing, and reading these data structures.

Expressions. The productions E and $\hat{E}$ present the syntax of *hygienic* numerical and relational numerical expressions, respectively. The syntax for these expressions expressions derives from that of the core calculus. Similar to numerical expressions, the productions B and $\hat{B}$ present the syntax of hygienic boolean and relational boolean expressions, which is also similar to that of the corresponding expressions in the core calculus. Where the syntax of expressions in the full language differs from that of the core calculus is in the addition of *non-hygienic* numerical and boolean expressions, $\dot{E}$ and $\dot{B}$, respectively.

Non-hygienic expressions differ from hygienic expressions in that, in addition to local variables, they may also reference values of data allocated in the heap. Specifically, non-hygienic expressions may reference heap-allocated primitives, $*p$, fields of heap-allocated structured types, $p.f$, and elements of heap-allocated arrays, $a[e_1, ..., e_k]$.

Non-hygienic expressions can only be used within a `havoc` or `relax` statement's condition. Specifically, non-hygienic expressions are used to express constraints on modifications to heap-allocated data within a `havoc` or `relax`. To additionally increase the expressivity of these statements, a non-hygienic expression may also use the syntax `old`$(\cdot)$ to refer to the *old* value of a variable or heap-allocated structure: the value before modification by the `havoc` or `relax`.

Declaration and Allocation. The statements in the program operate on either *local variables* or heap-allocated data. Local variables are allocated in a local *frame* of the *program stack*, which (as is standard) records a stack of local variable-to-value bindings, one for each live function. A local variable is either a primitive type, a pointer to a heap-allocated primitive or structured type, or a pointer to an array. Correspondingly, a local variable declaration has one of three forms:

- **Primitive.** A declaration of a primitive typed variable, *Prim* $x = e$, initializes the value of the variable to the value of a hygienic expression.

- **Pointer.** A declaration of a pointer typed variable, $T* x = (0 \mid y \mid \texttt{new}\ T)$, initializes the value of the variable to either zero, the value of another pointer-typed variable, or a new dynamically allocated instance of the base type.

- **Array.** A declaration of an n-dimensional array variable, $ArrayType\{n\}\ x = (0 \mid y \mid \texttt{new}\ ArrayType[E,...,E])$, initializes the variable to either 1) zero, 2) the value of another variable that points to an array, or 3) a new dynamically allocated array of the base type type with the length of each of the n dimensions given by a sequence of n hygienic expressions.

Stores. Heap-allocated data structures introduce new syntax for storing into the heap. A store into the heap writes the value of an expression into either a heap-allocated primitive type ($*p = e$), the field of a heap-allocated structured type ($p.f = e$), or an element of a heap-allocated array ($a[E,...,E] = x$).

Loads. Heap-allocated data structures also introduce new syntax for reading values stored in the heap. A load from the heap reads into a local variable the value of either a heap-allocated primitive type ($x = p*$), the field of a heap-allocated structured type ($x = p.f$), or an element of a heap-allocated array ($x = a[E,...,E]$.)

For both loads and stores, the dereferenced pointers, fields, and array indices must correspond to valid, previously allocated regions of the heap. The rules for verifying programs that manipulate the heap therefore include proof obligations that ensure that the program does not access invalid memory regions.

Havoc and Relax. Both `havoc` and `relax` statements extend their definitions in the core calculus to not only jointly modify a set of variables X, but to also jointly modify a set of addresses A. Each address in the set can be either a pointer (p), a field ($p.f$), or an array element ($a[e_1,...,e_k]$). The set of addresses in A is a subset of the addresses specified in each `havoc` or `relax` statement's condition.

Procedure Calls. Procedure calls introduce new syntax for calling functions either with or without a return value, $x = fn(E,...,E)$ and $fn(E,...,E)$, respectively. For functions that return a value, the `return` E statement provides this functionality in a standard way.

Unmodified Statements. The syntax for `assume`, `assert`, `relate`, `if`, `while`, and sequential composition statements are the same as that of the core calculus, with each statement's subexpressions and nested statements (e.g., each branch of an `if` statement) adapted to use the syntax of the new language.

3.2 Semantics.

I next present the semantics of the extended language. The new semantics' primary deviation from that of the core calculus is the addition of a heap to the program's environment.

Preliminaries. A heap $h \in H = A \rightarrow Z$ is a finite map from addresses $a \in A$ to integers that represent data that has been stored in the heap. The function $\mathbf{dom} \in H \rightarrow \mathcal{P}(N)$ returns the *domain* of the heap, which is the set of address allocated in the heap.

To give a structure to the heap, a *descriptor store* $m \in M = A \rightarrow D$ is a finite map from addresses to *descriptors*. A *descriptor* $d \in D = (F \rightarrow N) + (\bigcup_i N^i)$ describes the type of the data structure. A descriptor is either a finite map from fields to positive integer offsets – denoting the offset of each field of a structured type – or a n-arity tuple, denoting the length of each dimension of an n-dimensional array.

In this representation, a primitive data type allocated in the heap has a descriptor of the form $\langle 1 \rangle$ – which denotes a single-dimension, single element array. A n-dimensional array has a descriptor of the form $\langle l_1,...,l_n \rangle$, where each l_i denotes the length of the array on the i-th dimension. A structured data type allocated in the heap has a descriptor of the form $\{f_1 \mapsto 0,...,f_k \mapsto (k-1)\}$, where each f_k is a declared field of the structure and the

associated mapping gives the field's offset from the structure's base address. The function **size** $\in D \to N$ returns the size (in number of allocated addresses) of the data structure described by a descriptor.

To incorporate these new elements of the semantics, I extend the definition of an environment $\varepsilon \in \mathrm{E} = \Delta \times M \times H$ to consist of a *stack*, a descriptor store, and a heap. A stack $\delta \in \Delta := \cdot \mid \sigma :: \Delta$ is a list of *frames* $\sigma \in \Sigma = \mathit{Vars} \to Z$, where each frame is a finite map from local variables to integers that records the values of variables in the current function call scope. The value of a local variable is either a primitive value or a pointer to a data structured allocated in the heap.

Auxiliary Definitions. To support manipulating environments, I define the following projections that take as input an environment and return the environment's topmost frame, descriptor store, and heap, respectively.

$$\pi_{\text{frame}} \in \mathrm{E} \to \Sigma \qquad \pi_{\text{desc}} \in \mathrm{E} \to M \qquad \pi_{\text{heap}} \in \mathrm{E} \to H$$

$$\pi_{\text{frame}}(\langle \sigma :: \delta, m, h \rangle) = \sigma \qquad \pi_{\text{desc}}(\langle \delta, m, h \rangle) = d \qquad \pi_{\text{heap}}(\langle \delta, m, h \rangle) = h$$

3.2.1 Expression Semantics.

Figure 3-2 presents the semantics of both hygienic and non-hygienic expressions. The semantics of hygienic numerical and boolean expressions, E and B respectively, retain the same semantics as that of the core calculus. Non-hygienic numerical and boolean expression,$\dot{E}$ and $\dot{B}$ respectively, build upon the semantics of their corresponding hygienic forms to incorporate references to heap-allocated data and references to old values.

The semantics of non-hygienic numerical expressions is given by the semantic function $[\![\dot{E}]\!] \in \mathrm{E} \times \mathrm{E} \to \mathbb{Z}$, which given a pair of environments, returns the value of the expression. The first frame passed as an argument of the denotion denotes the old environment of the expression (before execution of a `havoc` or `relax`) whereas the second frame denotes the new environment. Therefore, for each of the denotations that follow, the denotations of references to old values (e.g., **old**(x)) operates on the first environment whereas the denotations of current values (x) operates on the second environment.

$$[\![E]\!] \in \Sigma \to \mathbb{Z} \qquad [\![B]\!] \in \Sigma \to \mathbb{B}$$

$$[\![\mathtt{n}]\!](\sigma) = n \qquad [\![\mathtt{true}]\!](\sigma) = \mathit{true} \quad [\![\mathtt{false}]\!](\sigma) = \mathit{false}$$

$$[\![\mathtt{x}]\!](\sigma) = \sigma(x) \qquad [\![E_1 \ \mathtt{cmp}\ E_2]\!](\sigma) = [\![E_1]\!](\sigma)\ \mathit{cmp}\ [\![E_2]\!](\sigma)$$

$$[\![E_1 \ \mathtt{iop}\ E_2]\!](\sigma) = [\![E_1]\!](\sigma)\ \mathit{iop}\ [\![E_2]\!](\sigma) \qquad [\![B_1 \ \mathtt{lop}\ B_2]\!](\sigma) = [\![B_1]\!](\sigma)\ \mathit{lop}\ [\![B_2]\!](\sigma)$$

$$[\![\neg B]\!](\sigma) = \begin{cases} \mathit{true}, & [\![B]\!](\sigma) = \mathit{false} \\ \mathit{false}, & [\![B]\!](\sigma) = \mathit{true} \end{cases}$$

$$[\![\dot{E}]\!] \in \mathrm{E} \times \mathrm{E} \to \mathbb{Z}$$

$$[\![\mathtt{n}]\!](\varepsilon, \varepsilon) = n$$

$$[\![\mathtt{x}]\!](\varepsilon, \langle \sigma :: \delta, m, h \rangle) = \sigma(x)$$

$$[\![\mathtt{old(x)}]\!](\langle \sigma :: \delta, m, h \rangle, \varepsilon) = \sigma(x)$$

$$[\![*x]\!](\varepsilon, \langle \sigma :: \delta, m, h \rangle) = h(\sigma(x))$$

$$[\![\mathtt{old}(*x)]\!](\langle \sigma :: \delta, m, h \rangle, \varepsilon) = h(\sigma(x))$$

$$[\![x.f]\!](\varepsilon, \langle \sigma :: \delta, m, h \rangle) = h(\sigma(x) + m(\sigma(x))(f))$$

$$[\![\mathtt{old}(x.f)]\!](\langle \sigma :: \delta, m, h \rangle, \varepsilon) = h(\sigma(x) + m(\sigma(x))(f))$$

$$[\![\mathtt{x[}e_1, ..., e_2\mathtt{]}]\!](\varepsilon, \langle \sigma :: \delta, m, h \rangle) = \mathrm{let} \langle l_1, ..., l_k \rangle = m(\sigma(x)) \ \mathrm{in}$$
$$h(\sigma(x) + [\![E_k]\!](\sigma) + \sum_{i=1}^{k-1} [\![E_i]\!](\sigma) \cdot l_i)$$

$$[\![\mathtt{old(x[}e_1, ..., e_2\mathtt{])}]\!](\langle \sigma :: \delta, m, h \rangle, \varepsilon) = \mathrm{let} \langle l_1, ..., l_k \rangle = m(\sigma(x)) \ \mathrm{in}$$
$$h(\sigma(x) + [\![E_k]\!](\sigma) + \sum_{i=1}^{k-1} [\![E_i]\!](\sigma) \cdot l_i)$$

$$[\![\dot{E}_1 \ \mathtt{iop}\ \dot{E}_2]\!](\varepsilon_1, \varepsilon_2) = [\![\dot{E}_1]\!](\varepsilon_1, \varepsilon_2)\ \mathit{iop}\ [\![\dot{E}_2]\!](\varepsilon_1, \varepsilon_2)$$

$$[\![\dot{B}]\!] \in \mathrm{E} \times \mathrm{E} \to \mathbb{B}$$

$$[\![\mathtt{true}]\!](\varepsilon_1, \varepsilon_2) = \mathit{true} \quad [\![\mathtt{false}]\!](\varepsilon_1, \varepsilon_2) = \mathit{false}$$

$$[\![\dot{E}_1 \ \mathtt{cmp}\ \dot{E}_2]\!](\varepsilon_1, \varepsilon_2) = [\![\dot{E}_1]\!](\varepsilon_1, \varepsilon_2)\ \mathit{cmp}\ [\![\dot{E}_2]\!](\varepsilon_1, \varepsilon_2)$$

$$[\![\dot{B}_1 \ \mathtt{lop}\ \dot{B}_2]\!](\varepsilon_1, \varepsilon_2) = [\![\dot{B}_1]\!](\varepsilon_1, \varepsilon_2)\ \mathit{lop}\ [\![\dot{B}_2]\!](\varepsilon_1, \varepsilon_2)$$

$$[\![\neg \dot{B}]\!](\varepsilon_1, \varepsilon_2) = \begin{cases} \mathit{true}, & [\![\dot{B}]\!](\varepsilon_1, \varepsilon_2) = \mathit{false} \\ \mathit{false}, & [\![\dot{B}]\!](\varepsilon_1, \varepsilon_2) = \mathit{true} \end{cases}$$

Figure 3-2: Hygienic and Non-hygienic Expression Semantics

The denotation of constants ($[\![n]\!]$) and references to local variables ($[\![x]\!]$) have the same semantics as hygienic expressions as they do not access the heap. For heap-accessing expressions, the denotation first computes the corresponding address in the heap and then returns the value of the heap at that location. The denotation of a dereference of pointer to a primitive type $*x$ is the value of the heap at the address stored for x in the given frame. The denotation of a reference to a field f of the structured type pointed to by x ($x.f$) is the value stored in the heap at the address of the given field. The denotation computes the address of the field as the sum of the base of the structured type ($\sigma(x)$) and the offset of the field from the base of the structure. The descriptor recorded in the descriptor store at the address of the structure gives this offset: $m(\sigma(x))(f)$. The denotation of an array reference $x[E_1,...,E_k]$ is the base address of the array plus the offset of the array element as computed by fetching the array's length descriptor from the descriptor store ($\langle l_1,...,l_k \rangle$) and computing the offset of the element given the dimension indices $E_1,...,E_k$.

The semantics of non-hygienic boolean expressions is similar to that of hygienic boolean expressions. The difference between the two semantics is that non-hygienic boolean expressions include non-hygienic numerical expressions in their comparisons operations.

3.2.2 Dynamic Original Semantics.

Figures 3-3 and 3-5 present the dynamic original semantics of the full language's heap manipulating statements and procedures, respectively.

Judgment. The judgment $\langle s,\ \sigma \rangle \Downarrow_o \bar{\phi}$ extends the core calculus's dynamic original semantics judgment with a new output configuration structure $\bar{\phi} \in \bar{\Phi} = \{ba\} \cup \{wr\} \cup (\mathrm{E} \times \Psi \times \{return, continue\})$. The new output configuration preserves the *ba* and *wr* symbols that denote failures of assumptions and assertions, respectively, and adds an additional value to the tuple that records the output of a successful execution of a statement. The new value is either the symbol *return* or *continue*, which denotes whether the evaluation of s completed by executing a `return` statement or if statements occurring after s should instead be allowed to execute (respectively). For clarity of presentation, I use the notation $\langle \varepsilon, \psi \rangle$ to map by default to the tuple $\langle \varepsilon, \psi, continue \rangle$.

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

$$\text{FO-Assign}\frac{\langle e\ ,\ \sigma\rangle \Downarrow_E n}{\langle x = e,\ \langle \sigma :: \delta, m, h\rangle\rangle \Downarrow_o \langle\langle \sigma[x \mapsto n] :: \delta, m, h\rangle, \cdot\rangle} \qquad \text{FO-Decl}\frac{\langle x = e,\ \varepsilon\rangle \Downarrow_o \phi}{\langle pt\ x = e,\ \varepsilon\rangle \Downarrow_o \phi}$$

$$\text{FO-Alloc}\frac{d = \mathbf{desc}(t) \qquad \langle a, h'\rangle = \mathbf{new}(h, \mathbf{size}(d))}{\langle t* x = \texttt{new}\ t,\ \langle \sigma :: \delta, m, h\rangle\rangle \Downarrow_o \langle\langle \sigma[x \mapsto a] :: \delta, m[a \mapsto d], h'\rangle, \cdot\rangle}$$

$$\text{FO-Alloc-Array}\frac{\begin{array}{cc}\forall i.\ \langle e_i\ ,\ \sigma\rangle \Downarrow_E n_i & \langle a, h'\rangle = \mathbf{new}(h, \mathbf{size}(\langle n_1, ..., n_k\rangle)) \\ \sigma' = \sigma[x \mapsto a] & m' = m[a \mapsto \langle n_1, ..., n_k\rangle]\end{array}}{\langle at\{n\}\ x = \texttt{new}\ at[e_1, ..., e_k],\ \langle \sigma :: \delta, m, h\rangle\rangle \Downarrow_o \langle\langle \sigma' :: \delta, m', h'\rangle, \cdot\rangle}$$

$$\text{FO-Load}\frac{a = \sigma(p) \qquad \langle 1\rangle = m(a) \qquad n = h(a)}{\langle x = *p,\ \langle \sigma :: \delta, m, h\rangle\rangle \Downarrow_o \langle\langle \sigma[x \mapsto n] :: \delta, m, h\rangle, \cdot\rangle} \qquad \text{FO-Store}\frac{\langle e\ ,\ \sigma\rangle \Downarrow_E n \qquad a = \sigma(p) \qquad \langle 1\rangle = m(a)}{\langle *p = e,\ \langle \sigma :: \delta, m, h\rangle\rangle \Downarrow_o \langle\langle \sigma :: \delta, m, h[a \mapsto n]\rangle, \cdot\rangle}$$

Figure 3-3: Dynamic Original Semantics of Heap-manipulating Statements

$$\text{FO-Field-Load}\ \frac{a = \sigma(p) \qquad d = m(a) \qquad n = h(a + d(f))}{\langle x = p.f,\ \langle \sigma :: \delta, m, h \rangle \rangle \ \Downarrow_o \ \langle \langle \sigma[x \mapsto n] :: \delta, m, h \rangle, \cdot \rangle}$$

$$\text{FO-Field-Store}\ \frac{\langle e\ ,\ \sigma \rangle \Downarrow_E n \qquad a = \sigma(p) \qquad d = m(a)}{\langle p.f = e,\ \langle \sigma :: \delta, m, h \rangle \rangle \ \Downarrow_o \ \langle \langle \sigma :: \delta, m, h[(a + d(f)) \mapsto n] \rangle, \cdot \rangle}$$

$$\text{FO-Array-Load}\ \frac{\begin{array}{c} \forall i.\ \langle e_i\ ,\ \sigma \rangle \Downarrow_E n_i \qquad a = \sigma(y) \qquad \langle l_1, ..., l_k \rangle = m(a) \\ \forall i.\ 0 \leq n_i < l_i \qquad n_o = l_k + \sum_{i=0}^{k-1} n_i \cdot l_i \qquad n = h(a + n_o) \end{array}}{\langle x = y[e_1, ..., e_k],\ \langle \sigma :: \delta, m, h \rangle \rangle \ \Downarrow_o \ \langle \langle \sigma[x \mapsto n] :: \delta, m, h \rangle, \cdot \rangle}$$

$$\text{FO-Array-Store}\ \frac{\begin{array}{c} \forall i.\ \langle e_i\ ,\ \sigma \rangle \Downarrow_E n_i \qquad a = \sigma(x) \qquad \langle l_1, ..., l_k \rangle = m(a) \\ \forall i.\ 0 \leq n_i < l_i \qquad \langle e\ ,\ \sigma \rangle \Downarrow_E n_e \qquad n_o = l_k + \sum_{i=0}^{k-1} n_i \cdot l_i \end{array}}{\langle a[e_1, ..., e_k] = e,\ \langle \sigma :: \delta, m, h \rangle \rangle \ \Downarrow_o \ \langle \langle \sigma :: \delta, m, h[(a + n_o) \mapsto n_e \rangle, \cdot \rangle}$$

$$\text{FR-Havoc}\ \frac{\begin{array}{c} [\![\dot{B}]\!](\varepsilon, \varepsilon') = \mathit{true} \qquad \forall x \notin X.\ \pi_{\text{frame}}(\varepsilon)(x) = \pi_{\text{frame}}(\varepsilon')(x) \\ \mathbf{dom}(\pi_{\text{heap}}(\varepsilon)) = \mathbf{dom}(\pi_{\text{heap}}(\varepsilon')) \qquad \forall a \notin A\ \pi_{\text{heap}}(\varepsilon)(a) = \pi_{\text{heap}}(\varepsilon')(a) \end{array}}{\langle \texttt{havoc (}A\texttt{,}X\texttt{) st (}[\![b]\!]\texttt{)}, \varepsilon \rangle \ \Downarrow_r \ \langle \varepsilon', \cdot \rangle}$$

Figure 3-4: Dynamic Original Semantics of Heap-manipulating Statements (Continued)

Standard Assignment. The rule [FO-ASSIGN] specifies the semantics of standard assignment to local variables. The semantics are the same as that of the core calculus.

Declarations and Allocations. Declarations declare a local variable whereas allocations create a new heap-allocated data structure (i.e., a primitive, structured type, or array).

The rule [FO-DECL] declares a local variable of primitive type. Because this rule does not allocate data in the heap, its semantics is the same as that of standard assignment: it evaluates e and records the resulting value as the value of x in the current frame.

The rule [FO-ALLOC] allocates a primitive or structured type in the heap. The function **desc** $\in T \rightarrow D$ maps a type (such as `int` or the name of a structured data type) to the descriptor that describes the type. For primitive types, the descriptor is a single-element array (see Preliminaries in this Section) and for structured types the descriptor is map from field names to offsets (see Preliminaries in this Section). Given a heap h and the size of the type to be allocated, the function **new** $\in H \times N \rightarrow N \times H$ allocates a new block of memory within h according to the specified size and initializes the block to 0. The function itself returns the address a of the block, along with a new heap, h', in which the block has been allocated. The following axioms capture the semantics of **new**:

$$(\langle a, h' \rangle = \mathbf{new}(h, n)) \Rightarrow (\forall i.\ (0 \leq i < n) \Rightarrow ((a+i) \notin \mathbf{dom}(h) \wedge (a+i) \in \mathbf{dom}(h'))$$

$$(\langle -, h' \rangle = \mathbf{new}(h, n)) \Rightarrow (\forall a.\ a \in \mathbf{dom}(h) \Rightarrow a \in \mathbf{dom}(h'))$$

$$(\langle -, h' \rangle = \mathbf{new}(h, n)) \Rightarrow (\forall a.\ a \in \mathbf{dom}(h) \Rightarrow (\forall v.\ h(x) = v \Rightarrow h'(x) = v))$$

$$(\langle a, h' \rangle = \mathbf{new}(h, n)) \Rightarrow (\forall i.(\ 0 \leq i < n) \Rightarrow (h(a+i) = 0))$$

As a final step, the semantics records the address a as the value of x in the frame and records d as the descriptor for a in the descriptor store.

The rule [FO-ALLOC-ARRAY] specifies the semantics for allocating a k-dimensional array. The rule evaluates each of the k expressions that specify the length of each dimension of the array, producing a descriptor $\langle n_1, ..., n_k \rangle$ where each l_i corresponds to the length of the i-th dimension. The rule uses the function **new** to produce a new heap, h', that reflects the allocation of the new array at address a. As the final step, the semantics records the address a as the value of x in the current frame and also records $\langle n_1, ..., n_k \rangle$ as a's descriptor.

Stores. In correspondence with the three types of heap-allocated data, there are also three separate forms for storing to heap-allocated data.

- **Primitive Types.** The rule [FO-STORE] specifies the semantics of storing a value e to the memory address pointed to by the pointer p. The rule operates by evaluating e to a value n and then fetching from the local frame the address, a, that p points to $(a = \sigma(p))$. The rule concludes by updating the contents of the heap at address a to contain n $(h[a \mapsto n])$.
- **Structured Types.** The rule [FO-FIELD-STORE] specifies the semantics of storing a value, e, into the field f of the structured type pointed to by p. The rule evaluates e to a value n and then fetches from the local frame both the address of the structured type $(a = \sigma(p))$ and the descriptor for the type $(d = m(a))$. Using the descriptor, the rule updates the heap at the location $a + d(f)$ with the value n, where d gives the offset of the field from the base of the structure.
- **Arrays.** The rule [FO-ARRAY-STORE] specifies the semantics of storing the value of e into the index $e_1, ..., e_k$ of the array x. The rule first evaluates each index, producing for each index a value n_i. The rule next loads the address of the array from the current frame $a = \sigma(y)$ and the descriptor for the array from the descriptor store $(\langle l_1, ..., l_k \rangle = m(a))$. The rule next verifies that the value of each index is within the bounds of the index's dimension (greater than or equal to 0 and less than the length of the dimension, l_i). Finally, the rule evaluates e, producing a value n_e, computes the offset of the target array element, n_o, and then stores n_e into the address of the element $(h[(a + n_o) \mapsto n_e])$.

Loads. For each of the three types of stores to heap-allocated, there is also a corresponding statement for reading the stored data:

- **Primitive Types.** The rule [FO-LOAD] specifies the semantics of loading a value from pointer into the heap. The pointer p is a local variable in the current frame that contains the address a of the data item. The rule fetches a from the current frame $a = \sigma(p)$ and then loads addresses value n from the heap $n = h(a)$. The rule concludes by recording the value of n as the value of x in the current frame.

$$\text{FO-Call-E}\ \frac{\forall i.\ \langle e_i\ ,\ \sigma\rangle \Downarrow_E n_i \wedge \sigma_c(\mathbf{param}(fn,i)) = n_i \quad \langle \mathbf{code}(fn),\ \langle \sigma_c :: \sigma :: \delta\rangle\rangle\ \Downarrow_o\ \langle \sigma_c' :: \sigma :: \delta, m', h', \psi\rangle \quad n = \sigma_c(\mathbf{ret})}{\langle x = fn(e_1,...,e_k),\ \langle \sigma :: \delta, m, h\rangle\rangle\ \Downarrow_o\ \langle \sigma[x \mapsto n] :: \delta, m', h', \psi\rangle}$$

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

$$\text{FO-Call}\ \frac{\forall i.\ \langle e_i\ ,\ \sigma\rangle \Downarrow_E n_i \wedge \sigma_c(\mathbf{param}(fn,i)) = n_i \quad \langle \mathbf{code}(fn),\ \langle \sigma_c :: \sigma :: \delta, m, h\rangle\rangle\ \Downarrow_o\ \langle \sigma_c' :: \sigma :: \delta, m', h', \psi\rangle}{\langle fn(e_1,...,e_k),\ \langle \sigma :: \delta, m, h\rangle\rangle\ \Downarrow_o\ \langle \sigma :: \delta, m', h', \psi\rangle}$$

$$\text{FO-Ret-E}\ \frac{\langle e\ ,\ \sigma\rangle \Downarrow_E n}{\langle \texttt{return}\ e,\ \langle \sigma :: \delta, m, h\rangle\rangle\ \Downarrow_o\ \langle\langle \sigma[\mathbf{ret} \mapsto n] :: \delta, m, h\rangle, \cdot, return\rangle}$$

$$\text{FO-Ret}\ \frac{}{\langle \texttt{return},\ \varepsilon\rangle\ \Downarrow_o\ \langle \varepsilon, \cdot, return\rangle} \qquad \text{FO-Seq-Ret}\ \frac{\langle s_1,\ \varepsilon\rangle\ \Downarrow_o\ \langle \varepsilon', \psi_1, return\rangle}{\langle s_1\ ;\ s_2,\ \varepsilon\rangle\ \Downarrow_o\ \langle \varepsilon', \psi_1, return\rangle}$$

$$\text{FO-While-Ret}\ \frac{\langle b\ ,\ \sigma\rangle \Downarrow_B true \quad \langle s,\ \sigma\rangle\ \Downarrow_o\ \langle \sigma', \psi_1, return\rangle}{\langle \texttt{while}\ (b)\ \{s\},\ \sigma\rangle\ \Downarrow_o\ \langle \sigma'', \psi_1, return\rangle}$$

Figure 3-5: Dynamic Original Semantics of Procedures

- **Structured Types.** The rule [FO-FIELD-LOAD] specifies the semantics of loading a value of a field f from a structured type that is allocated in the heap and pointed to by p. The rule fetches the address of the structure pointed to by p from the current frame ($a = \sigma(p)$). The rule also fetches the descriptor for the structure, d, from the descriptor store ($d = m(a)$). Using the descriptor to provide the offset of the field ($d(f)$), the rule fetches the value of the field from the heap $n = h(a + d(f))$. The rule concludes by updating the value of x in the current frame to have the value n.

- **Arrays.** The rule [FO-ARRAY-LOAD] specifies the semantics of loading the contents of an array y at the indices $e_1, ..., e_k$ into the local variable x. The rule borrows much of its logic from that of the rule for array stores. The rule evaluates each of the indices e_i, producing a value n_i. The rule then checks that each subindex is within the bounds of its corresponding dimension. Finally, the rule computes n_o, the offset from the base address of the index element, and records the value of the element from the heap as the value of x in the local frame.

Procedure Calls. The rules [FO-CALL-E] and [FO-CALL-VOID] specify the semantics of procedure calls with and without a return value (respectively).The rule [FO-CALL-E] specifies that evaluation of each of the function call's parameters, e_i, evaluates to a value n_i. The function **param** $\in ID \times N \rightarrow Vars$ returns for each function *fn* the variable associated with its i-th parameter. Using this function, the rule also specifies a fresh frame σ_c where the variable for the i-th parameter of the function is bound to the value of the i-th argument, n_i.

The function **code** returns the statements corresponding to the body of *fn*. Using this function, the rule specifies an execution of the body of *fn* with the new frame σ_c as the top frame on the stack. The results of the rule are 1) the descriptor store and heap from the body's execution and 2) the value of **ret** (which is the variable designated for the return value of a function). The rule records the value of **ret** for x in the caller's frame.

The semantics specified by the rule [FO-CALL] is similar to that of [FO-CALL-RETURN] except that the rule does not fetch the value of **ret** from the callee's frame.

Procedure Returns. The rules [FO-RETURN-E] and [FO-RETURN] specify the semantics of `return` statements. [FO-RETURN-E] applies to `return` statements that evaluate and return the result of an expression e. The rule evaluates e and assigns its value to **ret** in the local frame. Note that the output configuration uses the value *return*, which denotes that any statements that may follow this statement should be skipped. The rule [FO-RETURN] specifies the semantics of `return` statements that do not return a value; the semantics simply returns the current environment.

Short-circuiting Execution. The rules [FO-SEQ-RET] and [FO-WHILE-RET] specify the semantics for skipping statements within a sequential composition or while loop in cases for which execution of a nested statement reaches a return statement.

3.2.3 Dynamic Relaxed Semantics.

Figures 3-6, 3-7, and 3-8 present the rules of the dynamic relaxed semantics of the full language. The dynamic relaxed semantics for the full language is conceptually similar to that of the dynamic relaxed semantics of the core calculus: the dynamic relaxed semantics differs from the dynamic original semantics only in that `relax` statements modify the state of the program (where as in the dynamic original semantics they have no effect).

$$\boxed{\langle s,\ \sigma\rangle\ \Downarrow_r\ \bar{\phi}}$$

$$\text{FR-Assign}\frac{\langle e\ ,\ \sigma\rangle \Downarrow_E n}{\langle x = e,\ \langle \sigma :: \delta, m, h\rangle\rangle\ \Downarrow_r\ \langle\langle \sigma[x \mapsto n] :: \delta, m, h\rangle, \cdot\rangle} \qquad \text{FR-Decl}\frac{\langle x = e,\ \varepsilon\rangle\ \Downarrow_r\ \phi}{\langle pt\, x = e,\ \varepsilon\rangle\ \Downarrow_r\ \phi}$$

$$\text{FR-Alloc}\frac{d = \mathbf{desc}(t) \qquad \langle a, h'\rangle = \mathbf{new}(h, \mathbf{size}(d))}{\langle t* x = \texttt{new}\, t,\ \langle \sigma :: \delta, m, h\rangle\rangle\ \Downarrow_r\ \langle\langle \sigma[x \mapsto a] :: \delta, m[a \mapsto d], h'\rangle, \cdot\rangle}$$

$$\text{FR-Alloc-Array}\frac{\begin{array}{cc}\forall i.\ \langle e_i\ ,\ \sigma\rangle \Downarrow_E n_i & \langle a, h'\rangle = \mathbf{new}(h, \mathbf{size}(\langle n_1, \ldots, n_k\rangle)) \\ \sigma' = \sigma[x \mapsto a] & m' = m[a \mapsto \langle n_1, \ldots, n_k\rangle]\end{array}}{\langle at\{n\}\ x = \texttt{new}\ at[e_1, \ldots, e_k],\ \langle \sigma :: \delta, m, h\rangle\rangle\ \Downarrow_r\ \langle\langle \sigma' :: \delta, m', h'\rangle, \cdot\rangle}$$

$$\text{FR-Load}\frac{a = \sigma(p) \qquad \langle 1\rangle = m(a) \qquad n = h(a)}{\langle x = *p,\ \langle \sigma :: \delta, m, h\rangle\rangle\ \Downarrow_r\ \langle\langle \sigma[x \mapsto n] :: \delta, m, h\rangle, \cdot\rangle} \qquad \text{FR-Store}\frac{\langle e\ ,\ \sigma\rangle \Downarrow_E n \qquad a = \sigma(p) \qquad \langle 1\rangle = m(a)}{\langle *p = e,\ \langle \sigma :: \delta, m, h\rangle\rangle\ \Downarrow_r\ \langle\langle \sigma :: \delta, m, h[a \mapsto n]\rangle, \cdot\rangle}$$

Figure 3-6: Dynamic Relaxed Semantics of Heap-manipulating Statements

$$\text{FR-Field-Load}\ \frac{a = \sigma(p) \qquad d = m(a) \qquad n = h(a + d(f))}{\langle x = p.f,\ \langle \sigma :: \delta, m, h \rangle \rangle \ \Downarrow_r \ \langle \langle \sigma[x \mapsto n] :: \delta, m, h \rangle, \cdot \rangle}$$

$$\text{FR-Field-Store}\ \frac{\langle e\ ,\ \sigma \rangle \Downarrow_E n \qquad a = \sigma(p) \qquad d = m(a)}{\langle p.f = e,\ \langle \sigma :: \delta, m, h \rangle \rangle \ \Downarrow_r \ \langle \langle \sigma :: \delta, m, h[(a + d(f)) \mapsto n] \rangle, \cdot \rangle}$$

$$\text{FR-Array-Load}\ \frac{\begin{array}{c} \forall i.\ \langle e_i\ ,\ \sigma \rangle \Downarrow_E n_i \qquad a = \sigma(y) \qquad \langle l_1, ..., l_k \rangle = m(a) \\ \forall i.\ 0 \leq n_i < l_i \qquad n_o = l_k + \sum_{i=0}^{k-1} n_i \cdot l_i \qquad n = h(a + n_o) \end{array}}{\langle x = y[e_1, ..., e_k],\ \langle \sigma :: \delta, m, h \rangle \rangle \ \Downarrow_r \ \langle \langle \sigma[x \mapsto n] :: \delta, m, h \rangle, \cdot \rangle}$$

$$\text{FR-Array-Store}\ \frac{\begin{array}{c} \forall i.\ \langle e_i\ ,\ \sigma \rangle \Downarrow_E n_i \qquad a = \sigma(x) \qquad \langle l_1, ..., l_k \rangle = m(a) \\ \forall i.\ 0 \leq n_i < l_i \qquad \langle e\ ,\ \sigma \rangle \Downarrow_E n_e \qquad n_o = l_k + \sum_{i=0}^{k-1} n_i \cdot l_i \end{array}}{\langle a[e_1, ..., e_k] = e,\ \langle \sigma :: \delta, m, h \rangle \rangle \ \Downarrow_r \ \langle \langle \sigma :: \delta, m, h[(a + n_o) \mapsto n_e \rangle, \cdot \rangle}$$

$$\text{FR-Havoc}\ \frac{\begin{array}{c} [\![\dot{B}]\!](\varepsilon, \varepsilon') = \mathit{true} \qquad \forall x \notin X.\ \pi_{\text{frame}}(\varepsilon)(x) = \pi_{\text{frame}}(\varepsilon')(x) \\ \mathbf{dom}(\pi_{\text{heap}}(\varepsilon)) = \mathbf{dom}(\pi_{\text{heap}}(\varepsilon')) \qquad \forall a \notin A\ \pi_{\text{heap}}(\varepsilon)(a) = \pi_{\text{heap}}(\varepsilon')(a) \end{array}}{\langle \texttt{havoc}\ (A, X)\ \texttt{st}\ ([\![b]\!]),\ \varepsilon \rangle \ \Downarrow_r \ \langle \varepsilon', \cdot \rangle} \qquad \text{FR-Relax}\ \frac{\langle \texttt{havoc}\ (X)\ \texttt{st}\ (e),\ \sigma \rangle \ \Downarrow_r \ \bar{\phi}}{\langle \texttt{relax}\ (X)\ \texttt{st}\ (e),\ \sigma \rangle \ \Downarrow_r \ \bar{\phi}}$$

Figure 3-7: Dynamic Relaxed Semantics of Heap-manipulating Statements (Continued)

$$\text{FR-Call-E}\frac{\begin{array}{c}\forall i.\ \langle e_i\ ,\ \sigma\rangle \Downarrow_E n_i \wedge \sigma_c(\mathbf{param}(fn,i)) = n_i \\ \langle \mathbf{code}(fn),\ \langle \sigma_c :: \sigma :: \delta\rangle\rangle \Downarrow_r \langle \sigma_c' :: \sigma :: \delta, m', h', \psi\rangle \qquad n = \sigma_c(\mathbf{ret})\end{array}}{\langle x = fn(e_1,...,e_k),\ \langle \sigma :: \delta, m, h\rangle\rangle \Downarrow_r \langle \sigma[x \mapsto n] :: \delta, m', h', \psi\rangle}$$

$$\boxed{\langle s,\ \sigma\rangle \Downarrow_r \bar{\phi}} \qquad \text{FR-Call}\frac{\begin{array}{c}\forall i.\ \langle e_i\ ,\ \sigma\rangle \Downarrow_E n_i \wedge \sigma_c(\mathbf{param}(fn,i)) = n_i \\ \langle \mathbf{code}(fn),\ \langle \sigma_c :: \sigma :: \delta, m, h\rangle\rangle \Downarrow_r \langle \sigma_c' :: \sigma :: \delta, m', h', \psi\rangle\end{array}}{\langle fn(e_1,...,e_k),\ \langle \sigma :: \delta, m, h\rangle\rangle \Downarrow_r \langle \sigma :: \delta, m', h', \psi\rangle}$$

$$\text{FR-Ret-E}\frac{\langle e\ ,\ \sigma\rangle \Downarrow_E n}{\langle \texttt{return}\ e,\ \langle \sigma :: \delta, m, h\rangle\rangle \Downarrow_r \langle\langle \sigma[\mathbf{ret} \mapsto n] :: \delta, m, h\rangle, \cdot, return\rangle}$$

$$\text{FR-Ret}\frac{}{\langle \texttt{return},\ \varepsilon\rangle \Downarrow_r \langle \varepsilon, \cdot, return\rangle} \qquad \text{FR-Seq-Ret}\frac{\langle s_1,\ \varepsilon\rangle \Downarrow_r \langle \varepsilon', \psi_1, return\rangle}{\langle s_1\ ;\ s_2,\ \varepsilon\rangle \Downarrow_r \langle \varepsilon', \psi_1, return\rangle}$$

$$\text{FR-While-Ret}\frac{\langle b\ ,\ \sigma\rangle \Downarrow_B true \quad \langle s,\ \sigma\rangle \Downarrow_r \langle \sigma', \psi_1, return\rangle}{\langle \texttt{while}\ (b)\ \{s\},\ \sigma\rangle \Downarrow_r \langle \sigma'', \psi_1, return\rangle}$$

Figure 3-8: Dynamic Relaxed Semantics of Procedures

$$\begin{array}{l} x, y, p, ret \in Vars \\ \quad E ::= n \mid x \mid E\ \texttt{iop}\ E \\ \quad R ::= x \mid x.f \mid x[E,...,E] \\ \quad P ::= \texttt{true} \mid \texttt{false} \mid E\ \texttt{cmp}\ E \mid P\ \texttt{lop}\ P \mid \neg P \mid \forall x\ .\ P \mid \exists x\ .\ P \\ \qquad \mid \texttt{emp} \mid R \mapsto E \mid P * P \mid P \mathrel{-\!\!*} P \end{array}$$

Figure 3-9: Assertion Logic Syntax

3.3 Original Axiomatic Semantics

I next present the original axiomatic semantics. This section includes presentations of the original axiomatic semantics's proof rules along with its underlying assertion logic.

3.3.1 Assertion Logic

Figure 3-9 presents the syntax of the assertion logic for the axiomatic original semantics of the full programming language. The logic's primary additions are the logical connectors of *separation logic* that enable the assertion logic to express properties of heap-allocated data and, moreover, enable proofs that reason about the heap in a modular way [75].

Expressions. The language of expressions, E, is the same as that from the core calculus; expressions include constants, variable references, and integer operations on expressions.

Reference Expressions. To enable the assertion logic to refer to the contents of heap-allocated data, a *reference expression* $r \in R$ denotes an address of a data item allocated in the heap. A reference expression x denotes the value of the variable x, which is a pointer to a primitive type allocated in the heap. An expression $x.f$ denotes the address of the field f of the structure pointed to by x. A reference $x[E,...,E]$ denotes the address of the element at the specified index in the array x. The syntax of reference expressions is similar to that of non-hygienic expressions (Section 3.1) with the primary difference being that reference expressions denote an address whereas non-hygienic expressions denote the value stored in the heap at an address.

Predicates. The predicates of the assertion logic include many of the predicates included in the assertion logic for the core caclulus's axiomatic original semantics. The predicate constants `true` and `false` have the same meaning as in the core calculus. As in the core calculus, a predicate can also be a comparison between expressions, a logical composition of predicates, a negation of a predicate, a universally quantified predicate, or an existentially quantified predicate. The full programming language's assertion logic differs from that of the core calculus with the inclusion of predicates that express an *empty heap* (`emp`), a *singleton heap* ($R \mapsto E$), *separating conjunction* $P * P$, and *separating implication* ($P \mathrel{-\!\!*} P$). These four predicates and connectors are the main tools introduced in separation logic to enable modular reasoning about the heap.

3.3.2 Semantics

Expression Semantics. Figure 3-10 presents the semantics of expressions in the assertion logic. The denotation of expressions in the assertion logic, E, are the same as that for the expressions in the language (Section 3.2.1). The semantics of reference expressions is given by the function $[\![R]\!] \in \Sigma \times M \to N$ that given a frame and a descriptor store computes the address that corresponds to the reference expression. The denotation of a pointer-

$$
\begin{aligned}
[\![R]\!] &\in \Sigma \times M \to N \\
[\![\mathtt{x}]\!](\sigma, m) &= \sigma(x) \\
[\![\mathtt{x.f}]\!](\sigma, m) &= \sigma(x) + m(\sigma(x))(f) \\
[\![\mathtt{x[}E_1, \ldots, E_2\mathtt{]}]\!](\sigma, m) &= \mathrm{let}\langle l_1, \ldots, l_k\rangle = m(\sigma(x)) \text{ in } \sigma(x) + [\![E_k]\!](\sigma) + \sum_{i=1}^{k-1} [\![E_i]\!](\sigma) \cdot l_i
\end{aligned}
$$

Figure 3-10: Reference Expression Semantics

$$
\begin{aligned}
[\![P]\!] &\in \mathcal{P}(\mathrm{E}) \\
[\![\mathtt{true}]\!] &= \mathrm{E} \quad [\![\mathtt{false}]\!] = \emptyset \\
[\![E_1 \ \mathtt{cmp}\ E_2]\!] &= \{\varepsilon \mid [\![E_1]\!](\pi_{\text{frame}}(\varepsilon)) \ cmp \ [\![E_2]\!](\pi_{\text{frame}}(\varepsilon))\} \\
[\![P_1 \ \mathtt{lop}\ P_2\ ]\!] &= \{\varepsilon \mid \varepsilon \in [\![P_1]\!] \ lop \ \varepsilon \in [\![P_2]\!]\} \\
[\![\neg P]\!] &= [\![\mathtt{true}]\!] \setminus [\![P]\!] \\
[\![\forall x \,.\, P]\!] &= \{\varepsilon \mid \forall n \in \mathbb{Z}.\ \varepsilon \in [\![P[n/x]]\!]\} \\
[\![\exists x \,.\, P]\!] &= \{\varepsilon \mid \exists n \in \mathbb{Z}.\ \varepsilon \in [\![P[n/x]]\!]\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathtt{emp}]\!] &= \{\varepsilon \mid \mathrm{dom}(\pi_{\text{heap}}(\varepsilon)) = \emptyset\} \\
[\![R \mapsto E]\!] &= \{\langle \sigma :: \delta, m, h\rangle \mid \mathrm{dom}(h) = \{[\![R]\!](\sigma, m)\} \wedge h([\![R]\!](\sigma, m)) = [\![E]\!](\sigma)\} \\
[\![P_1 * P_2]\!] &= \{\langle \sigma :: \delta, m, h\rangle \mid \exists_{h_1, h_2} \cdot h_1 \perp h_2 \wedge h = h_1 \cdot h_2 \wedge \\
&\qquad\qquad \langle \sigma :: \delta, m, h_1\rangle \in [\![P_1]\!] \wedge \langle \sigma :: \delta, m, h_2\rangle \in [\![P_2]\!]\} \\
[\![P \mathrel{-\!\!*} Q]\!] &= \{\langle \sigma :: \delta, m, h_2\rangle \mid \forall_{h_1} \cdot h_1 \perp h_2 \wedge \\
&\qquad\qquad \langle \sigma :: \delta, m, h_1\rangle \in [\![P]\!] \to \langle \sigma :: \delta, m, h_1 \cdot h_2\rangle \in [\![Q]\!]\}
\end{aligned}
$$

Figure 3-11: Predicate Semantics for Assertion Logic

typed variable x is the value of x as specified in the given frame. The denotation of a field reference $(x.f)$ is the address of the data structure in the heap $(\sigma(x))$ plus the offset from the structure's base address of the field itself $(m(\sigma(x))(f))$. The descriptor for the address of the structure gives this offset $(m(\sigma(x)))$. The denotation of an array reference $x[E_1, \ldots, E_k]$ is the base address of the array plus the offset of the array element as computed by fetching the array's length descriptor from the descriptor store and computing the offset of the element given the dimension indices $E_1, \ldots, E_k$. These address calculations are the same as the address calculations that underlie non-hygienic expressions.

Predicate Semantics Figure 3-11 presents the denotational semantics of the assertion logic's predicates. The semantics of `true`, `false`, and the other standard connectives of the assertion logic (i.e., expression comparison, logical connectives, negation, and predicate quantification) are functionally same as that of the core calculus with the modification that the denotation is a set of environments $\mathcal{P}(\mathrm{E})$ (which now includes a heap) instead of a set of frames $\mathcal{P}(\Sigma)$ (Section 2.3.1). The primary point of a departure in the semantics of the logic is the semantics of the separation logic connectors for reasoning about heaps. To explain the semantics of these connectors, I first define the following notations that, respectively, define *disjoint heaps* (heaps that have no overlapping allocated addresses) and *heap union* (the union of the address mappings of two heaps):

$$h_1 \perp h_2 \equiv \mathbf{dom}(h_1) \cap \mathbf{dom}(h_2) = \emptyset$$

$$h_1 \cdot h_2 \equiv \{\langle a,n\rangle | \langle a,n\rangle \in h_1 \vee \langle a,n\rangle \in h_2\}$$

Given these preliminary definitions, heap predicates have one of four forms:

- **Empty Heap.** The predicate `emp` asserts that the heap is empty. The denotion of the predicate is therefore a heap h such that $\mathbf{dom}(h) = \emptyset$.

- **Singleton Heap.** The predicate $R \mapsto E$ asserts that the heap has a single allocated address at $[\![R]\!]$ with a value of $[\![E]\!]$. The condition that $\mathbf{dom}(h) = \{[\![R]\!]\}$ restricts the domain of the heap to be the single referenced address.

- **Separating Conjunction.** The predicate $P * Q$ asserts that the heap can be partitioned into two heaps h_1 and h_2 such that 1) h_1 and h_2 have disjoint domains (denoted by $h_1 \perp h_2$), 2) P holds for h_1, and 3) Q holds for h_2.

- **Separating Implication.** The predicate $P \mathrel{-\!\!*} Q$ asserts that the current heap h_2, when extended with a disjoint heap h_1 that satisfies P, satisfies Q.

The separation logic connectives can be difficult to reason about without context. In the next section a present a more detailed explanation of their semantics and purpose in the context of the axiomatic original semantics' proof rules.

3.3.3 Proof rules

Figures 3-12, 3.3.3, 3-14, and 3-15 present the proof rules for the axiomatic original semantics as adapted to the full programming language. The axiomatic original semantics of the core calculus and that of the full programming language share similar rules for many constructs. However, where the two sets of rules differ the most is for the rules that reason about modifications of the program state – particularly rules for allocating, accessing, and modifying heap state – and procedure call and returns.

Judgment. Each figure presents a set of rules for the judgment $\Gamma \vdash_o \{P\}\ s\ \{Q\}$. This judgment has a similar meaning to the original axiomatic judgment for the core calculus: for an environment that satisfies P, if execution of s halts then the resulting environment satisfies Q. The *function context* $\Gamma \in P$ extends the standard axiomatic definition to incorporate the *postcondition* of the encapsulating procedure in which the rules are used to perform verification. This postcondition specifies a predicate over the heap and the function's return value that must be true at the end of the execution of the function, whether it be by reaching the end of the function, or by executing a `return` statement. The rules for procedure calls make the purpose of the function context clear.

Assignment. The rule for assignment statements ([FOA-ASSIGN]) $x = e$ is the same as that for the core calculus. Because x is allocated in the frame, and e is a hygienic expression composed of only references to constants and frame-allocated variables, the rule does not interact with the heap. Because there is no heap interaction in the statement, the rule is the same as in the core calculus.

Declarations. The rule [FOA-DECL] specifies the semantics of declarations of primitive type variables. The dynamic semantics of the statement dictates that the statement dynamically reduces to an assignment statement. The rule therefore reuses the rule for assignment to specify the statement's semantics.

$$\text{FOA-Assign}\ \frac{}{\Gamma \vdash_o \{Q[e/x]\}\ x = e\ \{Q\}} \qquad \text{FOA-Decl}\ \frac{\Gamma \vdash_o \{P\}\ x = e\ \{Q\}}{\Gamma \vdash_o \{P\}\ pt\ x = e\ \{Q\}}$$

$$\text{FOA-Alloc}\ \frac{}{\Gamma \vdash_o \{\forall x' \,.\, (x' \mapsto 0) \mathrel{-\!\!*} Q[x'/x]\}\ pt*\ x = \texttt{new}\ pt\ \{Q\}}$$

$$\text{FOA-Alloc-Struct}\ \frac{}{\Gamma \vdash_o \{\forall x' \,.\, (\bigodot_{f}^{\mathbf{fields}(t)} x'.f \mapsto 0) \mathrel{-\!\!*} Q[x'/x]\}\ t*\ x = \texttt{new}\ t\ \{Q\}}$$

$$\text{FOA-Alloc-Array}\ \frac{}{\Gamma \vdash_o \{\forall x' \,.\, (\bigodot_{\langle i_1,\ldots,i_k\rangle}^{\text{idx}(\langle e_1,\ldots,e_k\rangle)} x'[i_1,\ldots,i_k] \mapsto 0) \mathrel{-\!\!*} Q[x'/x]\}\ at\{k\}\ x = \texttt{new}\ at[e_1,\ldots,e_k]\ \{Q\}}$$

$$\text{FOA-Load}\ \frac{}{\Gamma \vdash_o \{\exists v' \,.\, (p \mapsto v') * ((p \mapsto v') \mathrel{-\!\!*} Q[v'/x])\}\ x = *p\ \{Q\}}$$

$$\text{FOA-Field-Load}\ \frac{}{\Gamma \vdash_o \{\exists v' \,.\, (p.f \mapsto v') * ((p.f \mapsto v') \mathrel{-\!\!*} Q[v'/x])\}\ x = p.f\ \{Q\}}$$

$$\text{FOA-Array-Load}\ \frac{}{\Gamma \vdash_o \{\exists v' \,.\, (y[e_1,\ldots,e_k] \mapsto v') * ((y[e_1,\ldots,e_k] \mapsto v') \mathrel{-\!\!*} Q[v'/x])\}\ x = y[e_1,\ldots,e_k]\ \{Q\}}$$

Figure 3-12: Axiomatic Original Semantics for State Modifications

$$\text{FOA-Store}\ \frac{}{\Gamma \vdash_o \{(p \mapsto -) * ((p \mapsto e) \mathrel{-\!\!*} Q)\}\ *p = e\ \{Q\}} \qquad \text{FOA-Field-Store}\ \frac{}{\Gamma \vdash_o \{(p.f \mapsto -) * ((p.f \mapsto e) \mathrel{-\!\!*} Q)\}\ p.f = e\ \{Q\}}$$

$$\text{FOA-Array-Store}\ \frac{}{\Gamma \vdash_o \{(y[e_1,\ldots,e_k] \mapsto -) * ((y[e_1,\ldots,e_k] \mapsto e) \mathrel{-\!\!*} Q)\}\ y[e_1,\ldots,e_k] = e\ \{Q\}}$$

$$\text{FOA-Havoc}\ \frac{P(V,V',V'') \equiv (\bigodot_{a,v'}^{A,V'} a \mapsto v') * \dot{b}[V/\texttt{old}(A)][V'/A][V''/\texttt{old}(X)] \qquad [\![\exists V,V',V''.\ P(V,V',V'')]\!] \neq \emptyset}{\Gamma \vdash_o \{\exists V,V',V''.\ (\bigodot_{a,v}^{A,V} a \mapsto v) * (P(V,V',V'') \mathrel{-\!\!*} Q[V''/X])\}\ \texttt{havoc}\ (A,X)\ \texttt{st}\ (\dot{b})\ \{Q\}}$$

Figure 3-13: Axiomatic Original Semantics for State Modifications (Continued)

Allocation. The programming languages supports allocating primitive types, structured types, and arrays in the heap. Each data structure type requires a different proof rule:

- **Primitive Types.** The rule [FOA-ALLOC] provides the basic general structure for reasoning about allocating data in the heap. A statement *pt x** = `new` *pt* allocates a block of memory of type *pt* and stores its value into the local variable x. The rule specifies the semantics of the operation in a *backwards-style* reasoning form. Namely, for an assertion Q to hold after the execution of the statement, then the assertion $\forall x' \,.\, (x' \mapsto 0) \mathrel{-\!\!*} Q[x'/x]$ must hold before execution of the statement. This predicate uses separating implication, denoting that Q holds after the execution of the statement if the statement executes from any environment where $Q[x'/x]$ is still true when the environment's heap is extended with a disjoint heap in which the address contained in x' is allocated and has the value 0. The variable x' in the predicate represents the address returned by the allocation procedure as the location of the allocated data structure. Note that the rule universally quantifies x', denoting that the allocation procedure non-deterministically returns an address and, therefore, Q must be true for all possible addresses.

- **Structured Types.** The rule [FOA-ALLOC-STRUCT] specifies the semantics of allocating structured types in the heap. The rule has the same general structure as that for allocating primitive types in that it implements a backwards reasoning style in which the precondition is a separating implication $P \mathrel{-\!\!*} Q[x'/x]$, where $Q[x'/x]$ denotes the substitution of the non-deterministically allocated address of the data structure (x') for x. The predicate P in this case is the *iterated separating conjunction*

$$\bigodot_{f}^{\mathbf{fields}(t)} x \,.\, f \mapsto 0.$$

  An iterated separating conjunction represents the finite conjunction of a family of predicates indexed over a domain of iteration [75]. This iterated separating conjunction uses f as an index (or free variable) inside the predicate $x \,.\, f \mapsto 0$. The domain of iteration, which is the set of values f may take, is in this case the set of fields for the type t of the allocated structured type. The function $\mathbf{fields} \rightarrow T \in \mathcal{P}(F)$ returns

the set of fields for a given type. For example, if the program declares type T by the declaration `struct T { int x; int y; }`, then $\textbf{fields}(T) = \{x, y\}$.

Conceptually, an iterated separating conjunction is the same as a finite expansion of separating conjunctions over the domain of iterations. In the case of a structure with k fields, the iterated separating conjunction in this rule expands to $x.f_1 \mapsto 0 * ... * x.f_k \mapsto 0$. It is also possible to generalize this form to enable axiomatic reasoning for domains of iteration that are not necessarily statically determined. Let I be a symbolically represented domain of iteration, then the following axioms characterize the semantics of iterated separating conjunction:

$$\frac{i_1 \in I \qquad \left(\bigodot_{i}^{I} P(i)\right) * Q}{P(i_1) * Q} \qquad \frac{I_1 \subseteq I \qquad I_2 \subseteq I \qquad I_1 \cap I_2 = \emptyset \qquad \left(\bigodot_{i}^{I} P(i)\right)}{\left(\bigodot_{i}^{I_1} P(i)\right) * \left(\bigodot_{i}^{I_2} P(i)\right)}$$

.

The first rule specifies that if an iterated separating conjunction with an indexed predicate P holds for a domain, then P holds for each element of the domain. Additionally, if the iterated separating conjunction is separated from a predicate Q, then P is also separated from Q (implying that P has a heap footprint at most as large as the entire iterated separated conjunction).

The second rule specifies that if an iterated separating conjunction holds for a domain I, then for any two disjoint subsets of I, iterated separating conjunctions over those domains are mutually disjoint.

Returning to the discussion of structured types, these axioms ensure that the iterated separating conjunction used in [FOA-ALLOC-STRUCT] dictate that the address of each field in the structured type is unique with respect to other fields in the type. Note that the descriptor store determines the exact offset and address of each field and, therefore, while the predicate asserts that each address is unique it does not, for example, assert that the fields are contiguously allocated. Instead, the relationship

between the construction of the descriptor store's field mappings and the allocation strategy of the **new** determine the exact layout in memory of structured types.

- **Arrays.** The rule [FOA-ALLOC-ARRAY] specifies the semantics of allocating arrays in the heap. The structure and connectives of the rule are similar to that of the rule for allocating structured types. The primary difference is that the iterated separating conjunction enumerates over all indices of the array rather than a set of fields. The function $\mathbf{idx} \in N^k \rightarrow \mathcal{P}(N^K)$ maps a descriptor $\langle l_1, ..., l_k \rangle$ describing the lengths of each dimension of the array to a set of tuples where each tuples has the form $\langle i_1, ..., i_k \rangle$ and $\forall_\iota \cdot 0 \leq i_\iota < l_k$. A key difference between the iterated separating conjunction form used here versus the form used in the rule for allocating structured types is that the extent of this iterated separating conjunction is not statically determined. Specifically, while the iterated separating conjunction used for structured types iterates over a finite, statically determined set of fields (and can therefore be statically expanded into individual separating conjunctions), the iterated conjunction used here depends on the dynamically computed lengths of the arrays. The previously stated axioms for iterated separating conjunction are therefore critical for reasoning about arrays.

Stores. As with allocations, stores may access either heap-allocated primitive types, structured types, or arrays.

- **Primitives Types.** The rule [FOA-STORE] provides the semantics for reasoning about stores to heap-allocated primitive types. The rule has a similar structure to that of the rule for loads of primitives. The rule presents a backwards-style reasoning form in which for a predicate Q to hold at the end of the execution of a statement $*p = e$, then the predicate $(p \mapsto -) * (p \mapsto e \mathrel{-\!\!*} Q)$ must hold before. This predicate asserts that 1) p must point to an allocated address $(p \mapsto -)$ and 2) that the environment must satisfy the separating implication that Q holds when the environment's heap is extended so that p has the value e. This rule's general structure also translates to structured types and arrays.

- **Structured Types.** The rule [FOA-FIELD-STORE] provides the semantics for reasoning about stores to the fields of heap-allocated structured types. The rule's structure is similar to that of the rule for stores to primitives types with the only difference being that each singleton heap predicate refers to a field ($p.f \mapsto e$) instead of a primitive type ($p \mapsto e$).

- **Arrays.** The rule [FOA-ARRAY-STORE] provides the semantics for reasoning about stores to heap-allocated arrays. As the with the rule for structured types, the rule is similar to that for primitive types with the exception that each singleton heap predicate refers to an array element ($x[e_1,...,e_k] \mapsto e$) instead of a primitive.

Loads. Just as with allocation and stores, loads from data allocated in the heap access either a primitive data type, a structured data type, or an array.

- **Primitive Types.** The rule [FOA-LOAD] provides the semantics for reasoning about loads from primitive data types allocated in the heap. The general structure of the rule also holds for structured types and arrays. The rule has a backwards-style reasoning form such that for a predicate Q to hold a after execution of a statement $x = *p$, then the predicate $\exists v' \,.\, (p \mapsto v' * (p \mapsto v' \mathrel{-\!\!*} Q[v'/x]))$ must hold before. In this predicate, the variable v' is a fresh variable that does not appear any where in either the statement or Q. Given this definition, the new predicate expresses that an environment must satisfy the property that there exists some value for v' such that 1) the address given by p has that value and (by separating conjunction) 2) the environment satisfies the a separating implication that states that when the environment's heap is extended with an address p that has the value v', then $Q[v'/x]$ holds.

- **Structured Data Types.** The rule [FOA-FIELD-LOAD] presents the rule for loading from the fields of structured data types allocated in the heap. For a statement $x = p.f$, the rule has a similar form to that for primitive loads. Specifically, for Q to hold after the execution of the statement, the predicate $\exists v' \,.\, ((p.f \mapsto v') * ((p.f \mapsto v') \mathrel{-\!\!*} Q[v'/x]))$ must hold before. As in the rule for loads, v' is a fresh variable that does not appear any where in the rule.

- **Arrays.** The rule [FOA-ARRAY-LOAD] provides the semantics for loading from an array allocated in the heap. As in the rules for primitive and structured data types, the rule has a backwards form in which for a predicate Q to be true after the execution of a statement $x = x[e_1, ..., e_k]$, then the predicate $\exists v' \,.\, ((x[e_1, ..., e_k] \mapsto v') * ((x[e_1, ..., e_k] \mapsto v') \mathrel{-\!\!*} Q[v'/x]))$ must hold after. The rule's semantics are similar to that of the previous two rules.

Havoc. The rule [FOA-HAVOC] provides the semantics for reasoning about `havoc` statements. The `havoc` statement of the full language departs from that of the core calculus (and also from other constructs in the full language) in that it incorporates non-hygienic expressions that mix both references to local variables and heap-allocated data.

To ensure that a `havoc` statement executes reliably and is soundly captured by the semantics, the rule specifies three main properties. In a backwards-style reasoning similar to that for stores to heap-allocated data, for a predicate Q to hold after the execution of the statement 1) each heap-allocated address referenced in the expression must be allocated before the execution of the statement, 2) the initial environment must satisfy the separating implication that if the environment is extended with a disjoint heap satisfying the statement's condition e, then the resulting environment satisfies Q, and 3) it must be the case that e is satisfiable, therefore ensuring that there exists a satisfying assignment of values to both the addresses in V and the local variables in X.

The rule specifies the first property via an iterated separating conjunction that enumerates over the set of reference expressions referenced in the `havoc` statement's condition, A, paired with a set of fresh variables V, one for each reference expression in A. Each conjunct asserts that there exists a value v to which the reference expression points to.

The rule specifies the second property via a separating implication, with the predicate $P(V, V', V'')$ specifying the exact condition of a heap that satisfies the `havoc` statements' condition. The predicate specifically states that each address referenced in the expression is allocated and also has a corresponding value v', from the set V' which stand as placeholders for the value of each address after assignment. In addition, the set of values V' and the set of values V'' – which are placeholders for the old values of the variables in X within e – satisfy e when substituted for the corresponding items within e.

$$\text{FOA-Proc}\frac{\langle P_{fn}, Q_{fn}\rangle = \gamma(fn) \qquad Q_{fn} \vdash_o \{P_{fn}\}\ s\ \{Q_{fn}\}}{\vdash fn(Params)\{S\}}$$

$$\text{FOA-Call}\frac{\langle P_{fn}, Q_{fn}\rangle = \gamma(fn) \qquad \forall 0 \leq i < k.\ p_i = \mathbf{param}(fn, i)}{\Gamma \vdash_o \{P_{fn}[\langle e_1, ..., e_k\rangle / \langle p_1, ..., p_k\rangle]\}\ fn(e_1, ..., e_k)\ \{Q_{fn}\}}$$

$$\text{FOA-Call-Ret}\frac{\langle P_{fn}, Q_{fn}\rangle = \gamma(fn) \qquad \forall 0 \leq i < k.\ p_i\mathbf{params}(fn, i)}{\Gamma \vdash_o \{P_{fn}[\langle e_1, ..., e_k\rangle / \langle p_1, ..., p_k\rangle] * P\}\ x = fn(e_1, ..., e_k)\ \{\exists_{x'} \cdot Q_{fn}[x/ret] * P[x'/x]\}}$$

$$\text{FOA-Return-E}\frac{}{\Gamma \vdash_o \{\Gamma[e/ret]\}\ \texttt{return}\ e\ \{false\}} \qquad \text{FOA-Return}\frac{}{\Gamma \vdash_o \{\Gamma\}\ \texttt{return}\ \{false\}}$$

Figure 3-14: Axiomatic Original Semantics for Procedures

The rule specifies the third property via a premise of the rule that asserts that the denotation of P is not empty, therefore indicating that the `havoc` is satisfiable.

Procedures. Figure 3-14 presents the rule for reasoning about procedures, including procedure calls, returns, and verification of whole procedures.

To facilitate verifying procedures, I use a *condition* map $\gamma \in \Gamma = F \rightarrow P \times P$ that maps a procedure name $fn \in F$ to a pair of predicates, $\langle P_{fn}, Q_{fn}\rangle$, that specify the procedure's *precondition* and *postcondition*, respectively. As in their conventional usage, a precondition specifies a predicate that must hold before execution of the procedure whereas the postcondition is a predicate that must (given verification) hold after the execution of the procedure. Preconditions and postconditions enable modular assume-guarantee style reasoning.

- **Verifying Procedures.** Verification of each block of code in a program proceeds in the context of an encapsulating procedure. The rule [FOA-PROC] specifies the semantics for verifying each procedure. The rule works by using the precondition of the function (P_{fn}) as an assumed predicate for the verification of s such that if s terminates, then it satisfies the postcondition (Q_{fn}). Note that this rule also additional adds the function's postcondition to the context of the rule, to the left of the turnstile, therefore enabling nested `return` statements to access the appropriate postcondition.

- **Procedure Calls.** The rules [FOA-CALL] and [FOA-CALL-RET] present the semantics for reasoning about procedures that do and do not return a value, respectively. The rule for calls that do not return a value has a simple semantics. Given the precondition P_{fn} from the condition map, if the predicate $P_{fn}[\langle e_1, ..., e_k \rangle / \langle p_1, ..., p_k \rangle]$ holds before execution of the function, then the postcondition Q_{fn} holds after. The substitution of each of the parameters in the precondition with its actual arguments adapts the precondition to specifically constrain the values of the arguments. The semantics of the rule as a whole represents the modular assume-guarantee style reasoning of the approach in that the proof obligation in the callee's context is used to verify that the procedure call's precondition is valid. The proof obligation for each procedure body is to, therefore, demonstrate – assuming that the precondition is valid – that the procedure establishes its postcondition.

  The rule for calls that return a value adapts the semantics of calls without a return value. Specifically, the rule uses the precondition from the condition map to assert that 1) the substituted precondition $P_{fn}[\langle e_1, ..., e_k \rangle / \langle p_1, ..., p_k \rangle]$ holds before execution of the call and 2) the precondition holds separately from another (potentially vacuous) predicate P. By design, the additional predicate P may refer to the value of x, which will be overwritten by the value of the function after it returns. If the conjunction of the two predicates holds before execution of the call, then, via a forward-style of reasoning, there exists some value for the variable v' such that 1) the postcondition $Q_{fn}[ret/x]$ holds and 2) P still holds when x' is substituted for the old value of x within P. Note that in a similar fashion as for parameters and actual arguments, the substitution on the postcondition replaces the distinguished placeholder for the return value (**ret**) with the variable (x) in which the `return` statement stores the return value.

- **Procedure Returns.** The rules [FOA-RETURN] and [FOA-RETURN-E] specify the semantics of `return` statements that do and do not return a value, respectively. The rule for statements that do not return a value follows directly from the meaning and purpose of using a postcondition to enable assume guarantee style reasoning. Specifically, Γ is the postcondition of the enclosing procedure and if the execution's

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

$$\text{FOA-Skip}\frac{}{\Gamma \vdash_o \{P\}\ \texttt{skip}\ \{P\}}$$

$$\text{FOA-Relax}\frac{\langle \mathcal{R}, \phi, X, X', P' \rangle = \text{convert}(\mathcal{L}, e)}{\Gamma \vdash_o \{\exists_{\text{codom}(\phi)} \cdot P'\}\ \texttt{relax}\ (X)\ \texttt{st}\ (e)\ \{\exists_{\text{codom}(\phi)} \cdot P'\}}$$

$$\text{FOA-Relate}\frac{}{\Gamma \vdash_o \{P\}\ \texttt{relate}\ l\colon (\hat{e})\ \{P\}} \qquad \text{FOA-Assert}\frac{}{\Gamma \vdash_o \{P \wedge e\}\ \texttt{assert}\ (e)\ \{P \wedge e\}}$$

$$\text{FOA-Assume}\frac{}{\Gamma \vdash_o \{P\}\ \texttt{assume}\ (e)\ \{P \wedge e\}}$$

$$\text{FOA-If}\frac{\Gamma \vdash_o \{P \wedge b\}\ s_1\ \{Q\} \quad \Gamma \vdash_o \{P \wedge \neg b\}\ s_2\ \{Q\}}{\Gamma \vdash_o \{P\}\ \texttt{if}\ (b)\ \{s_1\}\ \texttt{else}\ \{s_2\}\ \{Q\}}$$

$$\text{FOA-While}\frac{\Gamma \vdash_o \{P \wedge b\}\ s\ \{P\}}{\Gamma \vdash_o \{P\}\ \texttt{while}\ (b)\ \{s\}\ \{P \wedge \neg b\}}$$

$$\text{FOA-Seq}\frac{\Gamma \vdash_o \{P\}\ s_1\ \{R\} \quad \Gamma \vdash_o \{R\}\ s_2\ \{Q\}}{\Gamma \vdash_o \{P\}\ s_1; s_2\ \{Q\}}$$

$$\text{FOA-Conseq}\frac{\models P \Rightarrow P' \quad \Gamma \vdash_o \{P'\}\ s\ \{Q'\} \quad \models Q' \Rightarrow Q}{\Gamma \vdash_o \{P\}\ s\ \{Q\}}$$

$$\text{FOA-Frame}\frac{\vdash_o \{P\}\ s\ \{Q\}}{\Gamma \vdash_o \{P * R\}\ s\ \{Q * R\}}$$

Figure 3-15: Full Language Axiomatic Original Semantics (Shared)

environment satisfies that postcondition at the site of the `return` statement, then the procedure satisfies its postcondition for executions that end at that statement. The rule reifies this intuition by asserting that Γ must hold before the execution of the `return` statement.

The rule for `return` statements that return a value — `return` e — has a similar form as that for `return` statements that do not return a value. The key difference between the two rules is that the rule for a `return` statement with a returned expression substitutes the statement's expression e for the return value placeholder (**ret**) in the active postcondition. This substitution constrains the return value of the `return` statement in that context.

Shared Statements. While the core calculus and full programming language differ in that the full programming language includes more expressive types and heap-allocated data, the two languages still share many of the same proof rules. Figure 3-15 presents the proof rules that the two languages share. Specifically, the two languages share rules for `skip` statements ([FOA-SKIP]), `assert` statements ([FOA-ASSERT]), `relate` statements ([FOA-RELATE]), `assume` statements ([FOA-ASSUME]), `if` conditionals ([FOA-IF]), `while` statements ([FOA-WHILE]), sequential composition ([FOA-SEQ]), and consequence ([FOA-CONSEQ]).

Frame Rule. The original axiomatic semantics includes a rule of constancy [FO-CONST] that enables local reasoning about the effects of modifications of the program state. As discussed in Section 2.3.2, the rule of constancy states that if the precondition of the statement s can be decomposed into a conjunction $P \wedge R$ and the free variables in R are disjoint from the set of variables modified by s, then given a proof that Q holds after the execution of s, then the conjunction $Q \wedge R$ also holds. The rule of constancy therefore enables proofs to preserve properties that are not modified by the semantics of a statement.

The full programming language has similar a rule, [FOA-FRAME], that implements the standard *frame rule*. By adding pointers to the language, the rule of constancy is only valid for *pure* predicates (predicates that refer only to scalar variables allocated in the local frame [75]) because the rule does not consider the effects of aliasing (when two pointers point to the same address in memory). For predicates that refer to the value of heap-allocated data, the frame rule enables proofs that reason locally about heap modifications.

The rule has a similar structure to that of the rule of constancy. The primary difference is that the rule uses separating conjunction instead of a standard conjunction. Therefore, if the precondition of a statement s can be decomposed into a conjunction $P * R$, then given a proof that starting from an environment satisfying P, Q holds after the execution of s, then the conjunction $Q * R$ holds after. This rule is sound in that P and R hold over disjoint heaps and therefore modifications to the heap encapsulated by P do not effect the heap encapsulated by R.

$$\hat{E} ::= \mathtt{n} \mid \mathtt{x}\langle \mathtt{o} \rangle \mid \mathtt{x}\langle \mathtt{r} \rangle \mid \hat{E}\ \mathtt{iop}\ \hat{E}$$
$$\hat{R} := \mathtt{x}\langle \mathtt{o} \rangle \mid \mathtt{x}\langle \mathtt{r} \rangle \mid \mathtt{x}\langle \mathtt{o} \rangle.\mathtt{f} \mid \mathtt{x}\langle \mathtt{r} \rangle.\mathtt{f} \mid \mathtt{x}\langle \mathtt{o} \rangle[\hat{E},...,\hat{E}] \mid \mathtt{x}\langle \mathtt{r} \rangle[\hat{E},...,\hat{E}]$$
$$\hat{P} ::= \mathtt{true} \mid \mathtt{false} \mid \hat{E}\ \mathtt{cmp}\ \hat{E} \mid \hat{P}\ \mathtt{lop}\ \hat{P} \mid \neg\hat{P} \mid \exists \mathtt{x}\langle \mathtt{o} \rangle\ .\ \hat{P} \mid \exists \mathtt{x}\langle \mathtt{r} \rangle\ .\ \hat{P}$$
$$\mid \mathtt{emp}_o \mid \mathtt{emp}_r \mid R \mapsto_o E \mid R \mapsto_r E \mid P_1 * P_2 \mid P_1 \mathrel{-\!\!*} P_2$$

Figure 3-16: Relational Assertion Logic Syntax

3.4 Relaxed Axiomatic Semantics

As in the development for the core calculus, the full programming language also has an axiomatic semantics for the relaxed semantics of the program. The axiomatic semantics is relational in that it relates values (of both local variables and heap-allocated data) between the original program and the relaxed program. A core challenge in designing the relaxed semantics is augmenting separation logic to refer to two heaps and adapting procedure verification to also include specifications of relational preconditions and postconditions.

3.4.1 Relational Assertion Logic

Syntax. Figure 3-16 presents the syntax of the relational assertion logic that underpins the relaxed axiomatic semantics. The syntax of the logic is similar to the structure of the assertion logic for the original axiomatic semantics, including relational expressions, $\hat{E}$, relational reference expressions, $\hat{R}$, and relational predicates for both the standard first-order connectives and the separation logic connectives. Where the logic's syntax differs from that of the unary assertion logic is by introducing references to variables in both the original semantics ($\mathtt{x}\langle \mathtt{o} \rangle$) of the program and the relaxed semantics of the program ($\mathtt{x}\langle \mathtt{r} \rangle$). The logic also includes separation logic predicates and connectives that assert properties on the heaps of both the original and relaxed program.

The predicate $\mathtt{emp}_o$ asserts that the heap of the original program is empty – independently of the state of the heap of the relaxed program – whereas $\mathtt{emp}_r$ asserts that the heap of the relaxed program is empty – independently of the state of the heap of the original program. The connective $\hat{R} \mapsto_o \hat{E}$ asserts the heap of the original program is a singleton

$$
\begin{aligned}
[\![\hat{E}]\!] &\in \Sigma \times \Sigma \to \mathbb{Z} \\
[\![\mathtt{n}]\!](\sigma_o, \sigma_r) &= n \\
[\![\mathtt{x}\langle\mathtt{o}\rangle]\!](\sigma_o, \sigma_r) &= \sigma_o(x) \\
[\![\mathtt{x}\langle\mathtt{r}\rangle]\!](\sigma_o, \sigma_r) &= \sigma_r(x) \\
[\![\hat{E}_1 \ \mathtt{iop}\ \hat{E}_2]\!](\sigma_o, \sigma_r) &= [\![\hat{E}_1]\!](\sigma_o, \sigma_r)\ iop\ [\![\hat{E}_2]\!](\sigma_o, \sigma_r)
\end{aligned}
$$

$$
\begin{aligned}
[\![\hat{R}]\!] &\in \Sigma \times M \times \Sigma \times M \to N \\
[\![\mathtt{x}\langle\mathtt{o}\rangle]\!](\sigma_o, m_o, \sigma_r, m_r) &= \sigma_o(x) \\
[\![\mathtt{x}\langle\mathtt{r}\rangle]\!](\sigma_o, m_o, \sigma_r, m_r) &= \sigma_r(x) \\
[\![\mathtt{x}\langle\mathtt{o}\rangle.\mathtt{f}]\!](\sigma_o, m_o, \sigma_r, m_r) &= \text{let } a = [\![x\langle\mathtt{o}\rangle]\!](\sigma_o, m_o, \sigma_r, m_r) \text{ in } v + m_o(a)(f) \\
[\![\mathtt{x}\langle\mathtt{r}\rangle.\mathtt{f}]\!](\sigma_o, m_o, \sigma_r, m_r) &= \text{let } a = [\![x\langle\mathtt{r}\rangle]\!](\sigma_o, m_o, \sigma_r, m_r) \text{ in } v + m_r(a)(f) \\
[\![\mathtt{x}\langle\mathtt{o}\rangle\,[\hat{E}_1, ..., \hat{E}_2]]\!](\sigma_o, m_o, \sigma_r, m_r) &= \text{let } a = [\![x\langle\mathtt{o}\rangle]\!](\sigma_o, m_o, \sigma_r, m_r) \text{ in} \\
&\quad \text{let}\langle l_1, ..., l_k\rangle = m_o(a) \text{ in} \\
&\quad\quad a + [\![E_k]\!](\sigma_o, \sigma_r) + \sum_{i=1}^{k-1} [\![\hat{E}_i]\!](\sigma_o, \sigma_r) \cdot l_i \\
[\![\mathtt{x}\langle\mathtt{r}\rangle\,[\hat{E}_1, ..., \hat{E}_2]]\!](\sigma_o, m_o, \sigma_r, m_r) &= \text{let } a = [\![x\langle\mathtt{r}\rangle]\!](\sigma_o, m_o, \sigma_r, m_r) \text{ in} \\
&\quad \text{let}\langle l_1, ..., l_k\rangle = m_r(a) \text{ in} \\
&\quad\quad a + [\![E_k]\!](\sigma_o, \sigma_r) + \sum_{i=1}^{k-1} [\![\hat{E}_i]\!](\sigma_o, \sigma_r) \cdot l_i
\end{aligned}
$$

Figure 3-17: Semantics of Relational Expressions and Reference Expressions

heap in which the address $\hat{R}$ has the value $\hat{E}$ (again, independently of the state of the heap of the relaxed program). Similarly, $\hat{R} \mapsto_r \hat{E}$ asserts that the heap of the relaxed program is a singleton heap with $\hat{R}$ containing the value of $\hat{E}$. These independent connectives enable the logic to specify properties of the heap of each program individually, yet still relate values in the heap through relations expressed over local variables. For example, the predicate $\exists \mathtt{c}\langle\mathtt{r}\rangle\ .\ \mathtt{x}\langle\mathtt{o}\rangle \mapsto_o \mathtt{c}\langle\mathtt{r}\rangle \wedge \mathtt{x}\langle\mathtt{r}\rangle \mapsto_r \mathtt{c}\langle\mathtt{r}\rangle$ states that there exists some value, here captured by an auxiliary variable $\mathtt{c}\langle\mathtt{r}\rangle$, such that both the value of the address $\mathtt{x}\langle\mathtt{o}\rangle$ in original program's heap and the value of the address $\mathtt{x}\langle\mathtt{r}\rangle$ in relaxed program's heap have that same value. Note that these predicates enable proofs to relate the values of heap-allocated data between the original and relaxed program even through the data may be allocated at different addresses in the two programs.

$$
\begin{aligned}
[\![\hat{P}]\!] &\in \mathcal{P}(\mathrm{E}\times\mathrm{E}) \\
[\![\texttt{true}]\!] &= \mathrm{E}\times\mathrm{E} \quad [\![\texttt{false}]\!] = \emptyset \\
[\![\hat{E}_1 \ \texttt{cmp}\ \hat{E}_2]\!] &= \{(\varepsilon_o,\varepsilon_r) \mid [\![\hat{E}_1]\!](\pi_{\text{frame}}(\varepsilon_o),\pi_{\text{frame}}(\varepsilon_r))\ cmp\ [\![\hat{E}_2]\!](\pi_{\text{frame}}(\varepsilon_o),\pi_{\text{frame}}(\varepsilon_r))\} \\
[\![\hat{P}_1 \ \texttt{lop}\ \hat{P}_2\ ]\!] &= \{(\varepsilon_o,\varepsilon_r) \mid (\varepsilon_o,\varepsilon_r) \in [\![\hat{P}_1]\!]\ lop\ (\varepsilon_o,\varepsilon_r) \in [\![\hat{P}_2]\!]\} \\
[\![\neg\hat{P}]\!] &= [\![\texttt{true}]\!] \setminus [\![\hat{P}]\!] \\
[\![\exists \texttt{x}\langle\texttt{o}\rangle\ .\ \hat{P}]\!] &= \{(\varepsilon_o,\varepsilon_r) \mid n\in\mathbb{Z},\ (\varepsilon_o,\varepsilon_r) \in [\![\hat{P}[n/x\langle\texttt{o}\rangle]]\!]\} \\
[\![\exists \texttt{x}\langle\texttt{r}\rangle\ .\ \hat{P}]\!] &= \{(\varepsilon_o,\varepsilon_r) \mid n\in\mathbb{Z},\ (\varepsilon_o,\varepsilon_r) \in [\![\hat{P}[n/x\langle\texttt{r}\rangle]]\!]\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{emp}_o]\!] &= \{\langle\varepsilon_o,\varepsilon_r\rangle \mid \mathrm{dom}(\pi_{\text{heap}}(\varepsilon_o)) = \emptyset\} \\
[\![\texttt{emp}_r]\!] &= \{\langle\varepsilon_o,\varepsilon_r\rangle \mid \mathrm{dom}(\pi_{\text{heap}}(\varepsilon_r)) = \emptyset\} \\
[\![\hat{R} \mapsto_o \hat{E}]\!] &= \{\langle\langle\sigma_o :: \delta_o, m_o, h_o\rangle, \langle\sigma_r :: \delta_r, m_r, h_r\rangle\rangle \mid \mathrm{dom}(h_o) = \{[\![\hat{R}]\!](\sigma_o,m_o,\sigma_r,m_r)\} \\
&\quad h_o([\![\hat{R}]\!](\sigma_o,m_o,\sigma_r,m_r)) = [\![\hat{E}]\!](\sigma_o,\sigma_r)\} \\
[\![\hat{R} \mapsto_r E]\!] &= \{\langle\langle\sigma_o :: \delta_o, m_o, h_o\rangle, \langle\sigma_r :: \delta_r, m_r, h_r\rangle\rangle \mid \mathrm{dom}(h_r) = \{[\![\hat{R}]\!](\sigma_o,m_o,\sigma_r,m_r)\} \\
&\quad h_r([\![\hat{R}]\!](\sigma_o,m_o,\sigma_r,m_r)) = [\![\hat{E}]\!](\sigma_o,\sigma_r)\} \\
[\![\hat{P}_1 * \hat{P}_2]\!] &= \{\langle\langle\sigma_o :: \delta_o, m_o, h_o\rangle, \langle\sigma_r :: \delta_r, m_r, h_r\rangle\rangle \mid \exists h_{o1}, h_{o2}, h_{r1}, h_{r2}\cdot \\
&\quad h_{o1} \perp h_{o2} \wedge h_{r1} \perp h_{r2} \\
&\quad h_o = h_{o1}\cdot h_{o2} \wedge h_r = h_{r1}\cdot h_{r2} \\
&\quad \langle\langle\sigma_o :: \delta_o, m_o, h_{o1}\rangle, \langle\sigma_r :: \delta_r, m_r, h_{r1}\rangle\rangle \in [\![\hat{P}_1]\!] \\
&\quad \langle\langle\sigma_o :: \delta_o, m_o, h_{o2}\rangle, \langle\sigma_r :: \delta_r, m_r, h_{r2}\rangle\rangle \in [\![\hat{P}_2]\!]\} \\
[\![\hat{P} \mathrel{-\!\!*} \hat{Q}]\!] &= \{\langle\langle\sigma_o :: \delta_o, m_o, h_{o2}\rangle, \langle\sigma_r :: \delta_r, m_r, h_{r2}\rangle\rangle \mid \forall h_{o1}, h_{r1}\cdot (h_{o1} \perp h_{o2} \wedge h_{r1} \perp h_{r2} \wedge \\
&\quad \langle\langle\sigma_o :: \delta_o, m_o, h_{o1}\rangle, \langle\sigma_r :: \delta_r, m_r, h_{r1}\rangle\rangle \in [\![\hat{P}]\!]) \rightarrow \\
&\quad (\langle\langle\sigma :: \delta, m, h_{o1}\cdot h_{o2}\rangle, \langle\sigma :: \delta, m, h_{r1}\cdot h_{r2}\rangle\rangle \in [\![\hat{Q}]\!])\}
\end{aligned}
$$

Figure 3-18: Predicate Semantics for Relational Assertion Logic

Semantics. Figure 3-18 presents the semantics of the full language's relational assertion logic. The semantics of the standard predicates and connectives (i.e, `true`, `false`, comparison between expressions, logical connectives, negation, and quantifiers) are similar to that of the relational predicate semantics of Section 2.3.3 with the primary difference being that the denotation of a predicate is an element of $\mathcal{P}(\mathrm{E} \times \mathrm{E})$ versus $\mathcal{P}(\Sigma \times \Sigma)$. Where the two semantics differ, however, is with the addition of separation logic connectives to the relational assertion logic.

- **Empty Heap.** The predicates $\mathtt{emp}_o$ and $\mathtt{emp}_r$ assert that the heap of the original program and the heap of the relaxed program are empty, respectively. Each respective denotation asserts that the domain of each heap is the empty set. Note that it is possible to recover a predicate `emp` that specifies that both heaps are empty by noting that $\mathtt{emp} \equiv \mathtt{emp}_o \wedge \mathtt{emp}_r$.

- **Singleton Heap.** The predicates $\hat{R} \mapsto_o \hat{E}$ and $\hat{R} \mapsto_r \hat{E}$ assert that the heap of the original program and the heap of the relaxed program are singleton heaps, respectively.

  The denotation of $\hat{R} \mapsto_o \hat{E}$ ($\hat{R} \mapsto_r \hat{E}$) is the set of all pairs of environments for which the domain of the original (relaxed) program's heap consists only of the address denoted by $\hat{R}$. Further, the value of the original (relaxed) program's heap at $\hat{R}$ has the value $\hat{E}$.

  Note that $\hat{R}$ and $\hat{E}$ are relational and that their denotations take as inputs the frames from both the original and relaxed program. This enables these predicates to link and relate addresses and values in the their heaps using relationships between variables of both program's. For example, it is possible to specify that $\mathtt{x}\langle\mathtt{o}\rangle \mapsto_o 1 \wedge \mathtt{x}\langle\mathtt{o}\rangle \mapsto_r 1$: the address contained in x in the original program has the value 1 in both the original and relaxed programs' heaps. This can, for example, represent a scenario where the relaxed program has updated x to contain a different address from that of the original program, but the underlying address is still allocated in the relaxed program's heap and can be accessed using values of the original program.

  Note that for each predicate expressed on the original program or the relaxed program, the predicate does not assert any constraint on the other heap. For example,

asserting that the original program is a singleton heap does not impose any constraint on the relaxed program's heap (or vice-versa). This denotation enables the predicates to work in a natural way with other connectives. For example, it is the case that $(R_o \mapsto_o E_o * R_r \mapsto_r E_r) \Leftrightarrow (R_o \mapsto_o E_o \wedge R_r \mapsto_r E_r)$, meaning that singleton heap predicates between the two heaps do not interfere and, therefore, the denoted heaps are clearly separated.

- **Separating Conjunction.** The semantics of separating conjunction adapts the unary definition from Section 3.3.2 to assert the disjointness of original program's heap between predicates P_1 and P_2 and, independently, the disjointness of the relaxed program's heap between P_1 and P_2. The semantics achieves this by specifying the denotation of a separating conjunction as the set of environment pairs of the form $\langle\langle \sigma_o :: \delta_o, m_o, h_o\rangle, \langle \sigma_r :: \delta_r, m_r, h_r\rangle\rangle$. Each environment pair is an element in the denotation if there exists two heap pairs $\langle h_{o1}, h_{o2}\rangle$ and $\langle h_{o1}, h_{o2}\rangle$ such that:

  1. h_{o1} is disjoint from h_{o2}
  2. h_{r1} is disjoint from h_{r2}
  3. h_o is the union of h_{o1} and h_{o2}
  4. h_r is the union of h_{r1} and h_{r2}
  5. P_1 holds for the reconstructed environment pair:

     $$\langle\langle \sigma_o :: \delta_o, m_o, h_{o1}\rangle, \langle \sigma_r :: \delta_r, m_r, h_{r1}\rangle\rangle$$

     (where h_{o1} and h_{r1} replace h_o and h_r in the original environment pair).
  6. P_2 holds for the reconstructed environment pair

     $$\langle\langle \sigma_o :: \delta_o, m_o, h_{o2}\rangle, \langle \sigma_r :: \delta_r, m_r, h_{r2}\rangle\rangle.$$

- **Separating Implication.** As with separating conjunction, relational separating implication adapts its unary definition to the relational setting. Conceptually, the predicate $\hat{P} \mathrel{-\!\!*} \hat{Q}$ denotes the set of environment pairs for which extending both the original and relaxed program's heaps with new heaps that satisfy $\hat{P}$ results in a new pair of environments that satisfy $\hat{Q}$. The semantics specifies that an environment pair

$\langle\langle \sigma_o :: \delta_o, m_o, h_{o2}\rangle, \langle \sigma_r :: \delta_r, m_r, h_{r2}\rangle\rangle$ is an element of the denotation of $\hat{P} \twoheadrightarrow \hat{Q}$ when for all heaps h_{o1} and h_{r1}, if h_{o1} and h_{o2} are disjoint from h_{o2} and h_{r2} (respectively) and they (along with their respective stacks and metadata stores) satisfy $\hat{P}_1$, then the combined heaps $h_{o1} \cdot h_{o2}$ and $h_{r1} \cdot h_{r2}$ must satisfy $\hat{Q}$.

3.4.2 Proof Rules

The relational assertion logic gives an expressive underlying logic for specifying relations between both the local and heap-allocated data of the original and relaxed program. Figures 3-19 and 3-20 present the axiomatic relaxed semantics of the program, which enables the development of proofs for these relational specifications.

Judgment. The judgment $Q_{fn}, \hat{R}_{fn} \vdash_r \{\hat{P}\}\ s\ \{\hat{Q}\}$ specifies that within a *relational function context* $Q_{fn}, \hat{R}_{fn}$, for a pair of environments that satisfy $\hat{P}$, if the joint execution of the original program and the relaxed program from their respective environments terminates, then the resulting pair of environments satisfies $\hat{Q}$. A relational function context extends the unary function context from the axiomatic original semantics to include the relational postcondition specified in the `ensures-r` clause of the encapsulating function. Specifically, as in the original axiomatic semantics, Q_{fn}, is the unary postcondition of the encapsulating function whereas $\hat{R}_{fn}$ is the relational postcondition.

Assignment and Declarations. The rule for assignment to local variables follows from the rule presented in Section 2.3.3 for the relaxed semantics of the core calculus. Local variable declarations, which do not require heap allocated storage, reduce to the semantics of an assignment (as in the original axiomatic semantics of the full language – Section 3.3.3).

Allocations

Allocation statements in the relaxed semantics of the program adapt the rules from the original axiomatic semantics, including their use of separation logic connectives, to the relational domain. An important aspect of allocation in the relational domain is that the allocated address for each data structure may differ between the original and relaxed programs. The proof rules therefore need to be agnostic to the choice of the address.

$$\text{FRA-Assign}\ \frac{}{Q_{fn}, \hat{R}_{fn} \vdash_r \{\hat{Q}[inj_o(e)/x\langle \mathtt{o}\rangle][inj_r(e)/x\langle \mathtt{r}\rangle]\}\ x = e\ \{\hat{Q}\}} \qquad \text{FRA-Decl}\ \frac{Q_{fn}, \hat{R}_{fn} \vdash_r \{\hat{P}\}\ x = e\ \{\hat{Q}\}}{Q_{fn}, \hat{R}_{fn} \vdash_r \{\hat{P}\}\ pt\ x = e\ \{\hat{Q}\}}$$

$$\text{FRA-Alloc}\ \frac{}{Q_{fn}, \hat{R}_{fn} \vdash_r \{\forall x'\langle \mathtt{o}\rangle, x'\langle \mathtt{r}\rangle\ .\ (x'\langle \mathtt{o}\rangle \mapsto_o 0 \wedge x'\langle \mathtt{r}\rangle \mapsto_r 0) \mathrel{-\!\!*} \hat{Q}[x'\langle \mathtt{o}\rangle/x\langle \mathtt{o}\rangle][x'\langle \mathtt{r}\rangle/x\langle \mathtt{r}\rangle]\}\ pt* \ x = \mathtt{new}\ pt\ \{\hat{Q}\}}$$

$$\text{FRA-Alloc-Struct}\ \frac{\hat{P}_1 \equiv (\bigodot_{f}^{\text{fields}(t)} x'\langle \mathtt{o}\rangle\ .\ f \mapsto_o 0) \qquad \hat{P}_2 \equiv (\bigodot_{f}^{\text{fields}(t)} x'\langle \mathtt{r}\rangle\ .\ f \mapsto_r 0)}{Q_{fn}, \hat{R}_{fn} \vdash_r \{\forall x'\langle \mathtt{o}\rangle, x'\langle \mathtt{r}\rangle\ .\ (\hat{P}_1 \wedge \hat{P}_2) \mathrel{-\!\!*} \hat{Q}[x'\langle \mathtt{o}\rangle/x\langle \mathtt{o}\rangle][x'\langle \mathtt{r}\rangle/x\langle \mathtt{r}\rangle]\}\ t* \ x = \mathtt{new}\ t\ \{\hat{Q}\}}$$

$$\text{FRA-Alloc-Array}\ \frac{\hat{P}_1 \equiv \left(\bigodot_{\langle i_1\langle \mathtt{o}\rangle, \ldots, i_k\langle \mathtt{o}\rangle\rangle}^{inj_o(\langle e_1, \ldots, e_k\rangle)} x'\langle \mathtt{o}\rangle[i_1\langle \mathtt{o}\rangle, \ldots, i_k\langle \mathtt{o}\rangle] \mapsto_o 0\right) \qquad \hat{P}_2 \equiv \left(\bigodot_{\langle i_1\langle \mathtt{r}\rangle, \ldots, i_k\langle \mathtt{r}\rangle\rangle}^{inj_r(\langle e_1, \ldots, e_k\rangle)} x'\langle \mathtt{r}\rangle[i_1\langle \mathtt{r}\rangle, \ldots, i_k\langle \mathtt{r}\rangle] \mapsto_r 0\right)}{Q_{fn}, \hat{R}_{fn} \vdash_r \{\forall x'\langle \mathtt{o}\rangle, x'\langle \mathtt{r}\rangle\ .\ (\hat{P}_1 \wedge \hat{P}_2) \mathrel{-\!\!*} \hat{Q}[x'\langle \mathtt{o}\rangle/x\langle \mathtt{o}\rangle][x'\langle \mathtt{r}\rangle/x\langle \mathtt{r}\rangle]\}\ at\{k\}\ x = \mathtt{new}\ at[e_1, \ldots, e_k]\ \{\hat{Q}\}}$$

$$\text{FRA-Load}\ \frac{\hat{P} \equiv (p\langle \mathtt{o}\rangle \mapsto_o v\langle \mathtt{o}\rangle) \rightarrow (p\langle \mathtt{r}\rangle \mapsto_r v\langle \mathtt{r}\rangle)}{Q_{fn}, \hat{R}_{fn} \vdash_r \{\exists v\langle \mathtt{o}\rangle, v\langle \mathtt{r}\rangle\ .\ \hat{P} * (\hat{P} \mathrel{-\!\!*} \hat{Q}[v'\langle \mathtt{o}\rangle/x\langle \mathtt{o}\rangle][v'\langle \mathtt{r}\rangle/x\langle \mathtt{r}\rangle])\}\ x = *p\ \{\hat{Q}\}}$$

$$\text{FRA-Field-Load}\ \frac{\hat{P} \equiv (p\langle \mathtt{o}\rangle\ .\ f \mapsto_o v\langle \mathtt{o}\rangle) \rightarrow (p\langle \mathtt{r}\rangle\ .\ f \mapsto_r v\langle \mathtt{r}\rangle)}{Q_{fn}, \hat{R}_{fn} \vdash_r \{\exists v\langle \mathtt{o}\rangle, v\langle \mathtt{r}\rangle\ .\ \hat{P} * (\hat{P} \mathrel{-\!\!*} \hat{Q}[v'\langle \mathtt{o}\rangle/x\langle \mathtt{o}\rangle][v'\langle \mathtt{r}\rangle/x\langle \mathtt{r}\rangle])\}\ x = p\ .\ f\ \{\hat{Q}\}}$$

$$\text{FRA-Array-Load}\ \frac{\hat{P} \equiv (y\langle \mathtt{o}\rangle\ [inj_o(e_1), \ldots, inj_o(e_k)] \mapsto_o v'\langle \mathtt{o}\rangle) \rightarrow (y\langle \mathtt{r}\rangle\ [inj_r(e_1), \ldots, inj_r(e_k)] \mapsto_r v'\langle \mathtt{r}\rangle)}{Q_{fn}, \hat{R}_{fn} \vdash_r \{\exists v\langle \mathtt{o}\rangle, v\langle \mathtt{r}\rangle\ .\ \hat{P} * (\hat{P} \mathrel{-\!\!*} \hat{Q}[v'\langle \mathtt{o}\rangle/x\langle \mathtt{o}\rangle][v'\langle \mathtt{r}\rangle/x\langle \mathtt{r}\rangle])\}\ x = y[e_1, \ldots, e_k]\ \{\hat{Q}\}}$$

Figure 3-19: Axiomatic Relaxed Semantics for State Modifications

$$\text{FRA-Store}\ \frac{}{Q_{fn}, \hat{R}_{fn} \vdash_r \{((p\langle \mathtt{o}\rangle \mapsto_o -) \to (p\langle \mathtt{r}\rangle \mapsto_r -)) * (p\langle \mathtt{o}\rangle \mapsto_o inj_o(e) \wedge p\langle \mathtt{r}\rangle \mapsto_r inj_r(e)) \mathrel{-\!\!*} \hat{Q}\}\ *p = e\ \{\hat{Q}\}}$$

$$\text{FRA-Field-Store}\ \frac{}{Q_{fn}, \hat{R}_{fn} \vdash_r \{((p\langle \mathtt{o}\rangle . f \mapsto_o -) \to (p\langle \mathtt{r}\rangle . f \mapsto_r -)) * ((p\langle \mathtt{o}\rangle . f \mapsto_o inj_o(e) \wedge p\langle \mathtt{r}\rangle . f \mapsto_r inj_r(e)) \mathrel{-\!\!*} \hat{Q})\}\ p.f = e\ \{\hat{Q}\}}$$

$$\text{FRA-Array-Store}\ \frac{\begin{array}{c}\hat{P}_1 \equiv (y\langle \mathtt{o}\rangle\,[inj_o(e_1), \ldots, inj_o(e_k)] \mapsto_o -) \to (y\langle \mathtt{r}\rangle\,[inj_r(e_1), \ldots, inj_r(e_k)] \mapsto_r -) \\ \hat{P}_2 \equiv (y\langle \mathtt{o}\rangle\,[inj_o(e_1), \ldots, inj_o(e_k)] \mapsto_o e) \wedge (y\langle \mathtt{r}\rangle\,[inj_r(e_1), \ldots, inj_r(e_k)] \mapsto_r e)\end{array}}{Q_{fn}, \hat{R}_{fn} \vdash_r \{\hat{P}_1 * (\hat{P}_2 \mathrel{-\!\!*} \hat{Q})\}\ y[e_1, \ldots, e_k] = e\ \{\hat{Q}\}}$$

$$\hat{R}(A,V) \equiv (\bigodot_{a\langle \mathtt{o}\rangle, v\langle \mathtt{o}\rangle}^{A\langle \mathtt{o}\rangle, V\langle \mathtt{o}\rangle} a\langle \mathtt{o}\rangle \mapsto v\langle \mathtt{o}\rangle) \to (\bigodot_{a\langle \mathtt{r}\rangle, v\langle \mathtt{r}\rangle}^{A\langle \mathtt{r}\rangle, V\langle \mathtt{r}\rangle} a\langle \mathtt{r}\rangle \mapsto v\langle \mathtt{r}\rangle)$$

$$P(V,V',V'') \equiv (\bigodot_{a,v'}^{A,V'} a \mapsto v') * \dot{b}[V/\mathbf{old}(A)][V'/A][V''/\mathbf{old}(X)] \qquad \hat{Q}'(V'') \equiv Q[V''\langle \mathtt{o}\rangle/X\langle \mathtt{o}\rangle][V''\langle \mathtt{r}\rangle/X\langle \mathtt{r}\rangle]$$

$$\text{FRA-Havoc}\ \frac{\hat{S}(V,V',V'') = (inj_o(P(V,V',V'')) \wedge inj_r(P(V,V',V''))) \qquad [\![\exists V\langle \mathtt{r}\rangle, V'\langle \mathtt{r}\rangle, V''\langle \mathtt{r}\rangle.\ inj_r(P(V,V',V''))]\!] \neq \emptyset}{Q_{fn}, \hat{R}_{fn} \vdash_r \{\exists^{V\langle \mathtt{o}\rangle V'\langle \mathtt{o}\rangle V''\langle \mathtt{o}\rangle}_{V\langle \mathtt{r}\rangle, V'\langle \mathtt{r}\rangle, V''\langle \mathtt{r}\rangle} . \hat{R}(A,V) * (\hat{S} \mathrel{-\!\!*} \hat{Q}')\}\ \texttt{havoc (}A\texttt{,}X\texttt{) st (}\dot{b}\texttt{) }\{\hat{Q}\}}$$

$$\hat{R}(A,V) \equiv (\bigodot_{a\langle \mathtt{r}\rangle, v\langle \mathtt{r}\rangle}^{A\langle \mathtt{r}\rangle, V\langle \mathtt{r}\rangle} a\langle \mathtt{r}\rangle \mapsto v\langle \mathtt{r}\rangle)$$

$$\text{FRA-Relax}\ \frac{P(V,V',V'') \equiv (\bigodot_{a,v'}^{A,V'} a \mapsto v') * \dot{b}[V/\mathbf{old}(A)][V'/A][V''/\mathbf{old}(X)] \qquad [\![\exists V\langle \mathtt{r}\rangle, V'\langle \mathtt{r}\rangle, V''\langle \mathtt{r}\rangle.\ inj_r(P(V,V',V''))]\!] \neq \emptyset}{Q_{fn}, \hat{R}_{fn} \vdash_r \{\exists V\langle \mathtt{r}\rangle, V'\langle \mathtt{r}\rangle, V''\langle \mathtt{r}\rangle.\ \hat{R}(A,V) * (inj_r(P(V,V',V'')) \mathrel{-\!\!*} \hat{Q}[V''\langle \mathtt{r}\rangle/X\langle \mathtt{r}\rangle])\}\ \texttt{relax (}A\texttt{,}X\texttt{) st (}\dot{b}\texttt{) }\{\hat{Q}\}}$$

Figure 3-20: Axiomatic Relaxed Semantics (Continued)

Primitive Types. The rule [FRA-ALLOC] specifies the semantics of allocating primitive types in the heap. The rule uses a backwards-style formulation to state that for $\hat{Q}$ to hold after the joint execution of the original and relaxed program, then for all possible addresses of the allocated type in the original program, $x'\langle o \rangle$, and the relaxed program, $x'\langle r \rangle$, the initial environment pair must satisfy $\hat{Q}$ (with appropriate substitutions of the new addresses for the original and relaxed values of x) when their respective heaps are extended with the new addresses.

Structured Types. The rule [FRA-ALLOC-STRUCT] specifies the semantics of allocating structured types in the heap. As in the rule for primitive types, the rule also uses a backwards-style formulation and a similar structure. The primary difference is that instead of allocating and initializing a single address, the rule uses iterated separating conjunction to allocate and initialize the entire structured type.

Arrays. The rule [FRA-ALLOC-ARRAY] specifies the semantics of allocating an array in the heap. The rule has a similar structure to that for structured types, using iterated separating conjunction to allocate and initialize the entirety of the array.

Loads

Loading from heap-allocated data structures in the relaxed axiomatic semantics has a similar basic structure as that in the original axiomatic semantics. A key point, however, is that loads require in the precondition that each data structure is allocated. This ensures that the program does not access memory that has not been allocated or is outside of the bounds of an array. While this check is explicitly required (via a proof) in the axiomatic original semantics, the principles of transferring properties from the original program over to verify properties of the relaxed program enables a relational verification of this proof obligation just as was done for verifying assertions in the axiomatic relaxed semantics of the core calculus.

Primitive Types. The rule [FRA-LOAD] specifies the semantics of loads from heap-allocated primitive data types. The rule uses a backwards-style formulation to specify that

for a predicate $\hat{Q}$ to hold after the joint execution of the both the original and relaxed program, then the heaps of the two programs must be able to be decomposed into two parts: 1) a part for which x in the original program points to a value $v\langle \mathtt{o} \rangle$ and x in the relaxed program points to a value $v\langle \mathtt{r} \rangle$ and 2) a part that when extended with the given values for both x in the original and relaxed program, satisfies $\hat{Q}$ (when substituted with the appropriate values for x). Note that $\hat{P}_1$ represents this first part of the heap. However, note that $\hat{P}_1$ relaxes the requirement that x must point to a value by using an implication. As with standard assertions in the relaxed axiomatic semantics of the core calculus, my relational verification approach enables proofs to use properties about the original program as assumptions.

Structured Types. The rule [FRA-LOAD-STRUCT] specifies the semantics of loads from the fields of heap-allocated structured types. As with loads from primitives types, the rule uses a backwards-style form and also incorporates the ability of the relational verification approach to assume that the structured type pointed to by x (and the associated field f) are allocated in the original program.

Arrays. The rule [FRA-LOAD-ARRAY] specifies the semantics of loads of the elements of heap-allocated arrays. This rule's form is similar to that of loads from heap-allocated structured types, with the primary difference being that the rule reasons about an element indexed by $\langle e_1, ..., e_k \rangle$ versus a field f.

Stores

As with loads from heap-allocated data, the axiomatic semantics of stores to the heap captures two aspects of their behavior 1) the address to be written must have been previously allocated in the program and 2) the postcondition after the statement must reflect the updated values of the heap.

Primitive Types. The rule [FRA-STORE] specifies the semantics of stores to primitive types allocated in the heap. These stores have a similar backwards-style reasoning form as

those of the axiomatic original semantics. For a property $\hat{Q}$ to hold after the joint original and relaxed execution of the statement $p* = e$, p must be allocated before the execution of the statement in the relaxed program. Note that as in the rule for the loads, the rule specifies this requirement with the predicate $(p\langle \mathtt{o} \rangle \mapsto -) \rightarrow p\langle \mathtt{r} \rangle \mapsto -$, which enables a relational verification approach that assumes that p is allocated in the original program to assist with verifying that p is allocated in the relaxed program. In addition to the allocation requirement, the initial environments must separately satisfy the separating implication that states that if the heaps of the environments are extended such that the address of p in both the original and relaxed program contain the value of e (when evaluated in the respective original and relaxed environments), then $\hat{Q}$ holds.

Structured Types and Arrays. The rules [FRA-FIELD-STORE] and [FRA-ARRAY-STORE] specify the semantics of stores to the fields of heap-allocated structured types and the elements of arrays, respectively. The semantics for these operations is similar to that of stores to primitive types with the main exception being that the rules either access a field, $p.f$, or an array element, $y[e_1, ..., e_k]$

Havoc and Relax

The rules [FRA-HAVOC] and [FRA-RELAX] present the semantics of `havoc` statements and `relax` statements, respectively. The relaxed semantics of the `havoc` statement adapts that of the core calculus in a similar fashion as the original semantics of `havoc` in the full language adapts that of the core calculus. Specifically, the rule asserts that the addresses referenced by the `havoc` are allocated before execution of the statement. The pair of environments satisfies the separating implication that asserts that the pair satisfies $\hat{Q}$ when each environment is extended with a disjoint heap that satisfies the `havoc` statement's condition. And, finally, the statement's condition must be satisfiable. Note that as in other rules, the rule leverages the assumed properties of the original program. Specifically, the rule leverages the assumption that each referenced address is allocated in the original program to verify that each address is allocated in the relaxed.

$$Q_{fn}, \hat{R}_{fn} \vdash_r \{\hat{P} \wedge inj_o(b) \wedge inj_r(b)\}\ s_1\ \{\hat{Q}\}$$
$$Q_{fn}, \hat{R}_{fn} \vdash_r \{\hat{P} \wedge \neg inj_o(b) \wedge \neg inj_r(b)\}\ s_2\ \{\hat{Q}\}$$

$$\text{FRA-If}\frac{\begin{array}{cc}\vdash_{\text{r}\langle\text{o}\rangle} \{\hat{P} \wedge inj_o(b)\}\ s_1\ \{\hat{R}_1\} & \vdash_{\text{r}\langle\text{r}\rangle} \{\hat{R}_1 \wedge \neg inj_o(b)\}\ s_2\ \{\hat{Q}\} \\ \vdash_{\text{r}\langle\text{o}\rangle} \{\hat{P} \wedge \neg inj_o(b)\}\ s_2\ \{\hat{R}_2\} & \vdash_{\text{r}\langle\text{r}\rangle} \{\hat{R}_2 \wedge inj_o(b)\}\ s_1\ \{\hat{Q}\}\end{array}}{Q_{fn}, \hat{R}_{fn} \vdash_r \{\hat{P}\}\ \texttt{if}\ (b)\ \{s_1\}\ \texttt{else}\ \{s_2\}\ \{\hat{Q}\}}$$

$$Q_{fn}, \hat{R}_{fn} \vdash_r \{\hat{P} \wedge inj_o(b) \wedge inj_r(b)\}\ s_1\ \{\hat{P}\}$$

$$\text{FRA-While}\frac{\vdash_{\text{r}\langle\text{o}\rangle} \{\hat{P} \wedge inj_o(b) \wedge \neg inj_r(b)\}\ s_1\ \{\hat{P}\} \qquad \vdash_{\text{r}\langle\text{r}\rangle} \{\hat{P} \wedge \neg inj_o(b) \wedge inj_r(b)\}\ s_1\ \{\hat{P}\}}{Q_{fn}, \hat{R}_{fn} \vdash_r \{\hat{P}\}\ \texttt{while}\ (b)\ \{s\}\ \{\hat{P} \wedge \neg inj_o(b) \wedge \neg inj_r(b)\}}$$

Figure 3-21: Relaxed Axiomatic Semantics (Control Flow)

The rule for `relax` statements has largely the same structure as that for `havoc` statements with the exception being that a `relax` statement only modifies the state of the relaxed program, whereas the state of the original program is unmodified.

Control Flow

Figure 3-21 presents the rules for control flow (`if` statements and `while` statements) in the relaxed axiomatic semantics. Specifically, in the core calculus, if the control flow of the original and relaxed program diverge at a control flow statement, then the proof rules use the rule [diverge] to project the relational properties between the original and relaxed program down to unary properties that hold individually for the original and relaxed program (and do not relate the semantics between the two programs). The rules presented here instead use a *loosed* axiomatic semantics that preserves the relations between an original (relaxed) program when the relaxed (original) program modifies its own state while the original (relaxed) program's execution is paused.

Loosed Axiomatic Semantics. Figure 3-22 presents the rules of the loosed axiomatic relaxed semantics judgment $\vdash_{\text{r}\langle\text{r}\rangle} \{\hat{P}\}\ s\ \{\hat{Q}\}$. The intended meaning of the judgment is the semantic judgment $\models_{\text{r}\langle\text{r}\rangle} \{\hat{P}\}\ s\ \{\hat{Q}\}$, which states that for all pairs of environments $(\varepsilon, \varepsilon_r)$ that satisfy $\hat{P}$, if evaluation of s from ε_r under the dynamic relaxed semantics produces an environment ε_r', then the pair of environments $\langle \varepsilon, \varepsilon_r' \rangle$ satisfies $\hat{Q}$.

$$\text{FLAR-Assign}\ \frac{}{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle\mathrm{r}\rangle} \{\hat{Q}[inj_r(e)/x\langle\mathrm{r}\rangle]\}\ x = e\ \{\hat{Q}\}} \qquad \text{FLAR-Decl}\ \frac{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle\mathrm{r}\rangle} \{\hat{P}\}\ x = e\ \{\hat{Q}\}}{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle\mathrm{r}\rangle} \{\hat{P}\}\ pt\ x = e\ \{\hat{Q}\}}$$

$$\text{FLAR-Alloc}\ \frac{}{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle\mathrm{r}\rangle} \{\forall x'\langle\mathrm{r}\rangle\ .\ (x'\langle\mathrm{r}\rangle \mapsto_r 0) \mathrel{-\!\!*} \hat{Q}[x'\langle\mathrm{r}\rangle/x\langle\mathrm{r}\rangle]\}\ pt{*}\ x = \texttt{new}\ pt\ \{\hat{Q}\}}$$

$$\text{FLAR-Alloc-Struct}\ \frac{\hat{P} \equiv \left(\bigodot_{f}^{\mathrm{fields}(t)} x'\langle\mathrm{r}\rangle\,.\,f \mapsto_r 0\right)}{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle\mathrm{r}\rangle} \{\forall x'\langle\mathrm{r}\rangle\ .\ \hat{P} \mathrel{-\!\!*} \hat{Q}[x'\langle\mathrm{r}\rangle/x\langle\mathrm{r}\rangle]\}\ t{*}\ x = \texttt{new}\ t\ \{\hat{Q}\}}$$

$$\text{FLAR-Alloc-Array}\ \frac{\hat{P} \equiv \left(\bigodot_{\langle i_1\langle\mathrm{r}\rangle,\ldots,i_k\langle\mathrm{r}\rangle\rangle}^{inj_r(\langle e_1,\ldots,e_k\rangle)} x'\langle\mathrm{r}\rangle[i_1\langle\mathrm{r}\rangle,\ldots,i_k\langle\mathrm{r}\rangle] \mapsto_r 0\right)}{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle\mathrm{r}\rangle} \{\forall x'\langle\mathrm{r}\rangle\ .\ \hat{P} \mathrel{-\!\!*} \hat{Q}[x'\langle\mathrm{r}\rangle/x\langle\mathrm{r}\rangle]\}\ at\{k\}\ x = \texttt{new}\ at[e_1,\ldots,e_k]\ \{\hat{Q}\}}$$

$$\text{FLAR-Load}\ \frac{\hat{P} \equiv p\langle\mathrm{r}\rangle \mapsto_r v\langle\mathrm{r}\rangle}{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle\mathrm{r}\rangle} \{\exists v\langle\mathrm{r}\rangle\ .\ \hat{P} * (\hat{P} \mathrel{-\!\!*} \hat{Q}[v\langle\mathrm{r}\rangle/x\langle\mathrm{r}\rangle])\}\ x = {*}p\ \{\hat{Q}\}}$$

$$\text{FLAR-Field-Load}\ \frac{\hat{P} \equiv p\langle\mathrm{r}\rangle\,.\,f \mapsto_r v\langle\mathrm{r}\rangle}{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle\mathrm{r}\rangle} \{\exists v\langle\mathrm{r}\rangle\ .\ \hat{P} * (\hat{P} \mathrel{-\!\!*} \hat{Q}[v\langle\mathrm{r}\rangle/x\langle\mathrm{r}\rangle])\}\ x = p\,.\,f\ \{\hat{Q}\}}$$

$$\text{FLAR-Array-Load}\ \frac{\hat{P} \equiv y\langle\mathrm{r}\rangle\,[inj_r(e_1),\ldots,inj_r(e_k)] \mapsto_r v\langle\mathrm{r}\rangle}{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle\mathrm{r}\rangle} \{\exists v\langle\mathrm{r}\rangle\ .\ \hat{P} * (\hat{P} \mathrel{-\!\!*} \hat{Q}[v\langle\mathrm{r}\rangle/x\langle\mathrm{r}\rangle])\}\ x = y[e_1,\ldots,e_k]\ \{\hat{Q}\}}$$

Figure 3-22: Loosed Axiomatic Relaxed Semantics

$$\text{FLAR-Store}\ \frac{}{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle \mathrm{r} \rangle} \{(p\langle \mathtt{r} \rangle \mapsto_r -) * ((p\langle \mathtt{r} \rangle \mapsto_r inj_r(e)) \mathrel{-\!\!*} \hat{Q})\} * p = e \{\hat{Q}\}}$$

$$\text{FLAR-Field-Store}\ \frac{}{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle \mathrm{r} \rangle} \{(p\langle \mathtt{r} \rangle . f \mapsto_r -) * ((p\langle \mathtt{r} \rangle . f \mapsto_r inj_r(e)) \mathrel{-\!\!*} \hat{Q})\}\ p.f = e \{\hat{Q}\}}$$

$$\text{FLAR-Array-Store}\ \frac{\begin{array}{c}\hat{P}_1 \equiv (y\langle \mathtt{r} \rangle\, [inj_r(e_1), \ldots, inj_r(e_k)] \mapsto_r -) \\ \hat{P}_2 \equiv (y\langle \mathtt{r} \rangle\, [inj_r(e_1), \ldots, inj_r(e_k)] \mapsto_r e)\end{array}}{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle \mathrm{r} \rangle} \{\hat{P}_1 * (\hat{P}_2 \mathrel{-\!\!*} \hat{Q})\}\ y[e_1, \ldots, e_k] = e \{\hat{Q}\}}$$

$$\hat{R}(A,V) \equiv \bigodot_{a\langle \mathtt{r} \rangle, v\langle \mathtt{r} \rangle}^{A\langle \mathtt{r} \rangle, V\langle \mathtt{r} \rangle} a\langle \mathtt{r} \rangle \mapsto v\langle \mathtt{r} \rangle$$

$$P(V,V',V'') \equiv (\bigodot_{a,v'}^{A,V'} a \mapsto v') * \dot{b}[V/\mathbf{old}(A)][V'/A][V''/\mathbf{old}(X)] \qquad \hat{Q}'(V'') \equiv Q[V''\langle \mathtt{r} \rangle / X\langle \mathtt{r} \rangle]$$

$$\text{FLAR-Havoc}\ \frac{\hat{S}(V,V',V'') = inj_r(P(V,V',V'')) \qquad [\![\exists V\langle \mathtt{r} \rangle, V'\langle \mathtt{r} \rangle, V''\langle \mathtt{r} \rangle.\ inj_r(P(V,V',V''))]\!] \neq \emptyset}{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle \mathrm{r} \rangle} \{\exists V\langle \mathtt{r} \rangle, V'\langle \mathtt{r} \rangle, V''\langle \mathtt{r} \rangle.\ \hat{R}(A,V) * (\hat{S} \mathrel{-\!\!*} \hat{Q}')\}\ \mathtt{havoc}\ (A,X)\ \mathtt{st}\ (\dot{b})\ \{\hat{Q}\}}$$

$$\text{FLAR-Relax}\ \frac{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle \mathrm{r} \rangle} \{P\}\ \mathtt{havoc}\ (A,X)\ \mathtt{st}\ (\dot{b})\ \{\hat{Q}\}}{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle \mathrm{r} \rangle} \{P\}\ \mathtt{relax}\ (A,X)\ \mathtt{st}\ (\dot{b})\ \{\hat{Q}\}}$$

$$\text{FLAR-If}\ \frac{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle \mathrm{r} \rangle} \{\hat{P} \wedge inj_r(b)\}\ s_1\ \{\hat{Q}\} \qquad Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle \mathrm{r} \rangle} \{\hat{P} \wedge \neg inj_r(b)\}\ s_2\ \{\hat{Q}\}}{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle \mathrm{r} \rangle} \{\hat{P}\}\ \mathtt{if}\ (b)\ \{s_1\}\ \mathtt{else}\ \{s_2\}\ \{\hat{Q}\}}$$

$$\text{FLAR-While}\ \frac{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle \mathrm{r} \rangle} \{\hat{P} \wedge inj_r(b)\}\ s\ \{\hat{P}\}}{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle \mathrm{r} \rangle} \{\hat{P}\}\ \mathtt{while}\ (b)\ \{s\}\ \{\hat{P} \wedge \neg inj_r(b)\}}$$

$$\text{FLAR-Seq}\ \frac{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle \mathrm{r} \rangle} \{\hat{P}\}\ s_1\ \{\hat{Q}\} \qquad Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle \mathrm{r} \rangle} \{\hat{Q}\}\ s_2\ \{\hat{R}\}}{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle \mathrm{r} \rangle} \{\hat{P}\}\ s_1\ ;\ s_2\ \{\hat{R}\}}$$

$$\text{FLAR-Conseq}\ \frac{\models \hat{P} \Rightarrow \hat{Q} \qquad Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle \mathrm{r} \rangle} \{\hat{Q}\}\ s\ \{\hat{R}\} \qquad \models \hat{R} \Rightarrow \hat{S}}{Q_{fn}, \hat{R}_{fn} \vdash_{\mathrm{r}\langle \mathrm{r} \rangle} \{\hat{P}\}\ s\ \{\hat{S}\}}$$

Figure 3-23: Loosed Axiomatic Relaxed Semantics (Continued)

This definition of the loosed axiomatic relaxed semantics is *semi-relational* in that it relates the dynamic relaxed semantics to another *fixed* environment ε. For the purposes of defining the meaning of well-behaved relaxed programs independently of the original evaluation, this extra environment is not immediately necessary. However, this definition enables us to also use this semantics when reasoning about cases where the relaxed evaluation of the program diverges from the original evaluation. In these cases, ε refers to the state of an original evaluation of the program at the point at which they diverged.

There is a one-to-one correspondence between the rules of the core calculus's intermediate original semantics (Section 2.3.3, Figure 2-11) and the loosed axiomatic relaxed semantics. Specifically, each rule in the loosed axiomatic relaxed semantics is a relational variant of a corresponding rule in the axiomatic intermediate semantics For example, consider the following statements:

- **Assert.** The rule for the `assert` (b) statement [FLA-ASSERT] mimics that of the axiomatic intermediate semantics. If a pair of environments satisfies $\hat{P}$ before evaluation of the statement, then if evaluation under the dynamic relaxed semantics does not yield wr, then the pair of environments satisfies $\hat{P} \wedge inj_r(b)$. The condition $inj_r(b)$ must also hold in the precondition of the rule to prevent evaluation from yielding wr.

- **Assume.** The rule for the `assume` (b) statement [FLA-ASSUME] also mimics that of the axiomatic intermediate semantics. When a developer writes an `assume` (b) statement in the relaxed programming model, it is assumed to be valid for the original program, before considering relaxation. After relaxing the program, such an assumption may no longer be valid and, therefore, a well-defined relaxed program must establish that it does not interfere with the validity of b. To establish this, the rule requires that $inj_r(b)$ hold in the precondition of the rule, preventing evaluation from yielding ba.

- **Relate.** By design, the loosed axiomatic relaxed semantics does not contain a rule for the `relate` : $(\hat{b})$ statement. The role of the `relate` statement is to note and verify a relational assertion between the dynamic original and relaxed evaluations at the point at which it appears. Because the loosed axiomatic relaxed semantics does not

relate the current point of the relaxed evaluation to the current point in the original evaluation, the `relate` statement does not have a meaning under this semantics. As with the intermediate semantics, the restriction prohibits `relate` statements from appearing under diverged control flow.

If statements. The rule for `if` statements ([FRA-IF]) uses the loosed axiomatic semantics to provide a semantics. The splits reasoning about this semantics into four cases:

1. The `if` statement's condition evaluates to *true* in both the original and relaxed executions of the program — In this case the rule uses the standard judgment for the relaxed axiomatic semantics to establish that the statement's postcondition, $\hat{Q}$, holds at the end of the original and relaxed execution.

2. The `if` statement's condition evaluates to *false* in both the original and relaxed executions of the program — In this case the rule again uses the standard judgment for the relaxed axiomatic semantics to establish that $\hat{Q}$ holds at the end of the executions.

3. The `if` statement's condition evaluates to *true* in the original execution of the program but evaluates to *false* in the relaxed execution — In this case, the control of the two executions diverges. The rule therefore relies on the loosed axiomatic semantics to characterize the two program's behavior. Specifically, it uses the loosed axiomatic original semantics to specify that the original execution of s_1 satisfies an intermediate condition $\hat{R}_1$ and then uses the loosed axiomatic relaxed semantics to specify that from a pair of environments satisfying $\hat{R}_1$, the relaxed execution of s_2 yields a pair of environments that satisfy Q.

4. The `if` statement's condition evaluates to *false* in the original execution of the program but evaluates to *true* in the relaxed execution — This case mimics the previous case, except that the original program executes s_2 and the relaxed program executes s_1.

While statement. The rule for `while` statements ([FRA-WHILE]) uses a similar approach as the rule for `if` statements to decompose the reasoning about control divergences.

The rule factors the reasoning into three cases:

- The `while` statement's condition evaluates to *true* in both the original and relaxed executions — In this case, the control flow between the two executions is convergent and, therefore, the rule uses the standard relaxed axiomatic judgment to verify the lockstep execution of the original and relaxed program preserves the loop invariant, $\hat{P}$.

- The `while` statement's condition evaluates to *true* in the original program and evaluates to *false* in the relaxed program — In this case the control flow of the original and relaxed executions diverge. Specifically, the relaxed execution has finished executing the loop. To handle this case, the rule uses the loosed axiomatic original semantics to establish the continuation of the relaxed execution preserves the loop invariant.

- The `while` statements' condition evaluates to *false* in the original program and evaluates to *true* in the relaxed program — In this case the control flow of the original and relaxed executions again diverge. As with the previous case, the rule uses the loosed axiomatic rules to provide a semantics. Specifically, the rule uses the loosed axiomatic relaxed semantics to establish that the continued execution of the relaxed program preserves the loop invariant.

Loosed Axiomatic Original Semantics. I also introduce a corresponding loosed axiomatic original semantics $\vdash_{\mathrm{r}\langle\mathrm{o}\rangle} \{\hat{P}\}\ s\ \{\hat{Q}\}$. Similar to the loosed axiomatic relaxed semantics, the intended meaning of the judgment is the semantic judgment $\models_{\mathrm{r}\langle\mathrm{o}\rangle} \{\hat{P}\}\ s\ \{\hat{Q}\}$, which states that for all pairs of environments $(\varepsilon_o, \varepsilon)$ that satisfy $\hat{P}$, if evaluation of s from ε_o under the dynamic original semantics yields an environment ε_o', then the pair of environments $\langle \varepsilon_o', \varepsilon \rangle$ satisfies $\hat{Q}$.

Procedures

Procedures introduce several more key points at which a relational reasoning approach can assist in developing proofs for the relaxed semantics of the program. Specifically, modular

$$\text{FRA-Call}\ \frac{\langle P_{fn}, \hat{P}_{fn}, Q_{fn}, \hat{Q}_{fn}\rangle = \hat{\gamma}(fn) \qquad P'_{fn} \equiv P_{fn}[\langle e_1,...,e_k\rangle/\langle p_1,...,p_k\rangle]}{Q_{pc}, \hat{R}_{pc} \vdash_r \{(inj_o(P'_{fn}) \to inj_r(P'_{fn})) \wedge \hat{P}_{fn}\}\ fn(e_1,...,e_k)\ \{(inj_o(Q_{fn}) \to inj_r(Q_{fn})) \wedge \hat{Q}_{fn}\}}$$

$$\text{FRA-Call-Ret}\ \frac{\begin{array}{c}\langle P_{fn}, \hat{P}_{fn}, Q_{fn}, \hat{Q}_{fn}\rangle = \hat{\gamma}(fn) \qquad P'_{fn} \equiv P_{fn}[\langle e_1,...,e_k\rangle/\langle p_1,...,p_k\rangle] \qquad Q'_{fn} = Q_{fn}[x/ret] \\ T \equiv inj_o(Q'_{fn}) \to inj_r(Q'_{fn}) \wedge \hat{Q}_{fn}[x\langle \mathtt{o}\rangle/\mathrm{ret}\langle \mathtt{o}\rangle][x\langle \mathtt{r}\rangle/\mathrm{ret}\langle \mathtt{r}\rangle]\end{array}}{Q_{pc}, \hat{R}_{pc} \vdash_r \{(inj_o(P'_{fn}) \to inj_r(P'_{fn})) \wedge \hat{P}_{fn}) * \hat{P}\}\ x = fn(e_1,...,e_k)\ \{\exists x'\langle \mathtt{o}\rangle, x'\langle \mathtt{r}\rangle\ .\ \hat{T} * \hat{P}[x'\langle \mathtt{o}\rangle/x\langle \mathtt{o}\rangle][x'\langle \mathtt{r}\rangle/x\langle \mathtt{r}\rangle]\}}$$

$$\text{FRA-Return}\ \frac{}{Q_{pc}, \hat{R}_{pc} \vdash_r \{(inj_o(Q_{pc}) \to inj_r(Q_{pc})) \wedge \hat{R}_{pc}\}\ \mathtt{return}\ \{\mathtt{false}\}}$$

$$\text{FRA-Return-E}\ \frac{Q'_{pc} \equiv Q_{pc}[e/ret] \qquad \hat{R}'_{pc} \equiv \hat{R}_{pc}[inj_o(e)/ret\langle \mathtt{o}\rangle][inj_r(e)/ret\langle \mathtt{r}\rangle]}{Q_{pc}, \hat{R}_{pc} \vdash_r \{(inj_o(Q'_{pc}) \to inj_r(Q'_{pc})) \wedge \hat{R}'_{pc}\}\ \mathtt{return}\ e\ \{\mathtt{false}\}}$$

$$\text{FRA-Proc}\ \frac{\langle P_{fn}, \hat{R}_{fn}, Q_{fn}, \hat{Q}_{fn}\rangle = \gamma(fn) \qquad Q_{fn}; \hat{Q}_{fn} \vdash_r \{\langle P_{fn} \cdot P_{fn}\rangle \wedge \hat{R}_{fn}\}\ s\ \{(inj_o(Q_{fn}) \to inj_r(Q_{fn})) \wedge \hat{Q}_{fn}\}}{\vdash fn(Params)\{S\}}$$

Figure 3-24: Relaxed Axiomatic Semantics (Procedures)

$$\text{FLAR-Call}\ \frac{\langle P_{fn}, \hat{P}_{fn}, Q_{fn}, \hat{Q}_{fn}\rangle = \hat{\gamma}(fn) \qquad P'_{fn} \equiv P_{fn}[\langle e_1,...,e_k\rangle/\langle p_1,...,p_k\rangle]}{Q_{pc}, \hat{R}_{pc} \vdash_{\mathrm{r}\langle\mathrm{r}\rangle} \{inj_r(P'_{fn})\}\, fn(e_1,...,e_k)\ \{inj_r(Q_{fn})\}}$$

$$\text{FLAR-Call-Ret}\ \frac{\begin{array}{c}\langle P_{fn}, \hat{P}_{fn}, Q_{fn}, \hat{Q}_{fn}\rangle = \hat{\gamma}(fn) \\ P'_{fn} \equiv P_{fn}[\langle e_1,...,e_k\rangle/\langle p_1,...,p_k\rangle] \qquad Q'_{fn} = Q_{fn}[x/ret]\end{array}}{Q_{pc}, \hat{R}_{pc} \vdash_{\mathrm{r}\langle\mathrm{r}\rangle} \{inj_r(P'_{fn}) * \hat{R}\}\ x = fn(e_1,...,e_k)\ \{\exists x'\langle\mathtt{r}\rangle\ .\ Q'_{fn} * \hat{R}[x'\langle\mathtt{r}\rangle/x\langle\mathtt{r}\rangle]\}}$$

$$\text{FLAR-Return}\ \frac{\mathtt{true} \Rightarrow \hat{R}_{fn}}{Q_{pc}, \hat{R}_{pc} \vdash_{\mathrm{r}\langle\mathrm{r}\rangle} \{inj_r(Q_{fn})\}\ \mathtt{return}\ \{\mathtt{false}\}}$$

$$\text{FLAR-Return-E}\ \frac{\hat{R}'_{fn} \equiv \hat{R}_{fn}[inj_o(e)/ret\langle\mathtt{o}\rangle][inj_r(e)/ret\langle\mathtt{r}\rangle] \qquad \mathtt{true} \Rightarrow \hat{R}'_{fn}}{Q_{pc}, \hat{R}_{pc} \vdash_{\mathrm{r}\langle\mathrm{r}\rangle} \{inj_r(Q_{fn}[e/ret])\}\ \mathtt{return}\ e\ \{\mathtt{false}\}}$$

Figure 3-25: Loosed Relaxed Axiomatic Semantics (Procedures)

verification of procedures may require additional *relational* preconditions and postconditions that are verified on entry to the function. Second, verifying that the relaxed execution of the program satisfies a function's precondition is another point at which a proof may assume that the original program has been verified and, therefore satisfies its specification. Third, modular relational verification of a procedure dictates that the body of an invoked function is verified using the same proof methodology. Therefore, while a traditional verification approach ensures that a function satisfies its postcondition, the relational verification approach ensures that the function satisfies its postcondition – assuming that the original program also satisfies its postcondition. These key properties are apparent in the rules for verifying procedures.

Procedure Calls. The rules [FRA-CALL] and [FRA-Call-Ret] specify the semantics of procedure calls without a return value and procedure calls with a return value, respectively. [FRA-CALL] illustrates how the three key concepts of verifying procedures work together.

The *relational* specification map $\hat{\gamma} \in F \to P \times \hat{P} \times P \times \hat{P}$ gives the set of specifications for a procedure, which includes its standard unary precondition P_{fn}, its *relational* precondition $\hat{P}_{fn}$, its standard unary postcondition Q_{fn}, and its *relational* precondition $\hat{Q}_{fn}$.

The standard unary precondition corresponds to the precondition used in the axiomatic

original semantics and therefore (given a verification using the axiomatic original semantics) holds in the original program. Mapping this precondition to the relational domain using a relational verification approach means that the precondition is *assumed* to hold for the original program and, using this assumption, is verified to hold for the relaxed program. [FRA-CALL] specifies this property by including in the precondition for the statement the predicate $inj_o(P'_{fn}) \rightarrow inj_r(P'_{fn})$.

The relational precondition $\hat{P}_{fn}$ specifies a precondition for the relation between values in the original program and the relaxed program on entry to the function. This relation may for example, express that the values between the two programs are the same, or, bounded by small a quantifiable distance. These relations can then enable relational verification inside the body of the function or enable verifying relational properties about the output of the function. For example, the relational precondition may state that the difference in value of an input x to a function differs by at most 10% between the original and relaxed program and, therefore, enable a proof that the outputs of function's computation differ between the two programs by at most 15%. The relational precondition must hold on entry to the function and therefore [FRA-CALL] specifies this requirement by adding it to the precondition for the rule.

Similar to the standard unary precondition, the unary postcondition corresponds to the postcondition used and verified in the axiomatic original semantics and therefore (given a verification using the axiomatic original semantics) holds in the original program. Relying on the relational reasoning approach to provide a semantics, this postcondition is assumed to be true after the execution of the function in the original program, whereas the postcondition is verified to be true in the relaxed program. [FRA-CALL] specifies this semantics by including in the postcondition of the statement the predicate $inj_o(Q'_{fn}) \rightarrow inj_r(Q'_{fn})$.

The relational postcondition $\hat{Q}_{fn}$ specifies a constraint on the relation between values in the original program and relaxed program after the function's execution. As with the relational precondition, this postcondition may for example specify a quantified difference between a variable in the original program and the relaxed after execution of the function. The relational postcondition holds for all joint executions of the body of the function and therefore also holds after the execution of the function call.

The rule for calls that return a value, [FRA-CALL-RET], extends the semantics of [FRA-CALL] to include additional substitutions on the function's two postconditions to syntactically replace the symbolic name of the function's return value in its specification (**ret**) with the concrete name of the variable that receives the function's result.

Returns. The rules [FRA-RET] and [FRA-RET-E] specify the semantics of return statements. As outlined in the discussion of the properties that hold after calling a function, verifying a return statement has two verification obligations: 1) verify that the relaxed program satisfies the function's unary postcondition, assuming the postcondition is true in the original program, and 2) verify that the joint execution of the original and relaxed program satisfy the function's relational postcondition.

The rule [FRA-RET] specifies the first condition by including in the precondition of the statement the predicate $inj_o(Q_{fn}) \rightarrow inj_r(Q_{fn})$. The symbol Q_{fn} denotes the unary postcondition of the function as has been added into the verification context by the proof rules for the logic.

[FRA-RET] specifies the second condition by including the relational predicate $\hat{R}_{fn}$ in the precondition of the statement. The symbol $\hat{R}$ denotes the relational postcondition of the statement as has been added into the verification context by the proof rules for the logic.

Similar to relationship between the rules for function calls with and without return values, [FRA-RET-E] extends the semantics specified in [FRA-RET] for `return` statements that return a value. [FRA-RET-E] specifically substitutes each occurrence of the symbolic name of the function's return value for the `return` statement's expression.

Procedure Verification. The rule [FRA-PROC] specifies the overall semantics for verifying a procedure declaration. The first acquires the function's preconditions and postconditions from the specification map. The rule then requires a verification of the function's body within a context $Q_{fn}; \hat{Q}_{fn}$ (which provides the necessary postconditions for verifying `return` statements within the function's body). The verification specifically requires that from a pair of environments such that 1) P_{fn} holds independently for both the original and relaxed program's environments and 2) the pair of environments satisfies the relationship $\hat{P}_{fn}$, then after execution of the function, 1) Q_{fn} holds independently for both of the resulting original and relaxed environments and 2) the pair of environments satisfies $\hat{Q}_{fn}$.

Loosed Axiomatic Semantics. Figure 3-25 presents the rules for the loosed axiomatic semantics of procedure calls and returns. The rules share the same logic as that for the relaxed semantics except for two major points because the semantics captures the behavior of the relaxed program after its control flow has diverged from that of the original program: 1) the rules do not use properties of the original program as assumptions (properties must hold outright) and 2) relational postconditions for the procedure must be tautologies. The second condition captures the fact that the because the relaxed program and original program are not at the same program point, the relation specified as the procedure's postcondition is not well-defined. The restriction that the postcondition must be a tautology therefore means that `return` statements for procedures with relational postconditions must not appear under diverging control flow constructs (unless the postconditions are trivial).

3.5 Properties

The proof rules for the full language provide many of the same properties and guarantees that hold for the core calculus. In this section, I present these definitions as adapted to the semantics of the full language.

3.5.1 Axiomatic Original Semantics

The proof rules of the axiomatic original semantics are sound with respect to the dynamic original semantics and also guarantee that verified programs do not fail assertions or access invalid memory.

Lemma 10 (Soundness).

$$\textit{If } \gamma \vdash_o \mathit{fn}\ \{\ s\ \},\ \textit{then } \gamma \models_o \mathit{fn}\ \{\ s\ \}$$

This lemma states that given a proof of the correctness of a procedure in the program, then execution of that procedure from any state that satisfies the procedure's precondition results in a final state that satisfies the procedure's postcondition. Note that that semantic guarantee also encompasses executions the reach `return` statements and therefore may not necessarily execute the entirety of s.

Lemma 11 (Original Progress Modulo Assumptions).

$$\textit{If } \gamma \vdash_o fn\{\ s\ \}, \textit{ and } \varepsilon \models P, \textit{ and } \langle s,\ \varepsilon\rangle \Downarrow_o \bar{\phi}, \textit{ then } \bar{\phi} \neq wr$$

This lemma states that given a proof for a procedure *fn*, then from all environments that satisfy the procedure's precondition, if execution of the procedure's body terminates, yielding an output configuration $\bar{\phi}$, then the execution does not encounter an error ($\bar{\phi} \neq wr$). Note that this definition does not preclude the execution from violating an assumption. The additional reasoning required to establish this judgment – when compared to the core calculus – are the steps that reason about heap accesses. Each rule for a heap access includes a guard in the precondition the ensures that the address is allocated and therefore the program does not encounter an error.

3.5.2 Loosed Axiomatic Relaxed Semantics

The loosed axiomatic relaxed semantics enjoys the same properties as the intermediate axiomatic semantics from the core calculus: soundness and progress.

Lemma 12 (Loosed Progress).

$$\textit{If } Q_{pc}, \hat{R}_{pc} \vdash_{\mathrm{r}\langle\mathrm{r}\rangle} \{\hat{P}\}\ s\ \{\hat{Q}\}, \textit{ then } Q_{pc}, \hat{R}_{pc} \models_{\mathrm{r}\langle\mathrm{r}\rangle} \{\hat{P}\}\ s\ \{\hat{Q}\}$$

This lemma states that the loosed axiomatic relaxed semantics is sound. The reasoning behind this lemma is similar to that of the intermediate axiomatic semantics of the core calculus with th primary exception being that the assertion logic is relational.

Lemma 13 (Loosed Progress).

$$\textit{If } Q_{pc}, \hat{R}_{pc} \vdash_{\mathrm{r}\langle\mathrm{r}\rangle} \{\hat{P}\}\ s\ \{\hat{Q}\}, \textit{ and } (\varepsilon_o, \varepsilon_r) \models \hat{P}, \textit{ and } \langle s,\ \varepsilon_r\rangle \Downarrow_r \bar{\phi}_r, \textit{ then } \neg err(\bar{\phi}_r)$$

This lemma states that the loosed axiomatic relaxed semantics establishes full progress for the relaxed program. The reasoning behind this lemma is also similar to that of the intermediate semantics in that the proof rules assert that each assertion in the program must be valid for the relaxed program (without the ability to assume that they are valid in the original program).

3.5.3 Relaxed Axiomatic Semantics

The relaxed axiomatic semantics of the program is sound (with respect to the joint execution of the original and relaxed program), ensures the soundness of `relate` statements, and – in combination with the original axiomatic semantics – ensures that the relaxed program does not violate any assertions or access invalid memory.

Lemma 14 (Soundness).

$$\textit{If } \gamma \vdash_r \mathit{fn}\ \{\ s\ \},\ \textit{then } \gamma \models_r \mathit{fn}\ \{\ s\ \}$$

This lemma states that given a proof for a procedure, for all pairs of environments that satisfy the precondition of the procedure, if the executions of the original program and the relaxed program from their respective environments in the pair terminate, then the resulting environments satisfy the procedure's postcondition.

Theorem 15 (Soundness of Relational Assertions).

$$\gamma \vdash_r \mathit{fn}\ \{\ s\ \} \textit{ and } (\varepsilon_o, \varepsilon_r) \models \hat{P},\ \textit{and } \langle s,\ \varepsilon_o\rangle \Downarrow_o \langle \varepsilon_o',\ \psi_1\rangle,\ \textit{and } \langle s,\ \varepsilon_r\rangle \Downarrow_r \langle \varepsilon_r',\ \psi_2\rangle,$$
$$\textit{then } \Gamma \vdash \psi_1 \sim \psi_2$$

This lemma states that given a proof for a procedure, then the pair of observation lists generated by executions of the original and relaxed program from environments that satisfy the procedure's postcondition satisfy the program's relational assertions. This lemma extends the relational soundness of the core calculus to include relations over the heaps of the original and relaxed program.

Theorem 16 (Relative Relaxed Progress).

$$\textit{If } \gamma \vdash \mathit{fn}\ \{\ s\ \} \textit{ and } (\varepsilon_o, \varepsilon_r) \models \langle P \cdot P\rangle,\ \textit{and } (\varepsilon_o, \varepsilon_r) \models \hat{P},\ \textit{and } \langle s,\ \varepsilon_o\rangle \Downarrow_o \bar{\phi}_o,\ \textit{and } \neg \mathit{err}(\bar{\phi}_o),$$
$$\textit{and } \langle s,\ \varepsilon_r\rangle \Downarrow_r \bar{\phi}_r,\ \textit{then } \neg \mathit{err}(\bar{\phi}_r)$$

A main theorem for the relaxed program is it an enjoys a relative progress property. Namely, for all pairs of environments that satisfy a procedure's precondition, if execution

of the original program from its respective environment terminates not in error, then the relaxed program also does not encounter an error. This theorem follows directly from the soundness of the logic and the soundness of the relational verification approach adopted for each assertion in the relaxed program, including both `assert` statements and internal assertions, such as ensuring that each access to heap-allocated memory is valid.

Theorem 17 (Relaxed Progress).

$$\textit{If } \gamma \vdash_o \mathit{fn}\ \{\ s\ \},\ \textit{and } \gamma \vdash_r \mathit{fn}\ \{\ s\ \},\ (\varepsilon_o, \varepsilon_r) \models \langle P \cdot P \rangle,\ \textit{and } (\varepsilon_o, \varepsilon_r) \models \hat{P},\ \textit{and } \langle s,\ \varepsilon_o \rangle \Downarrow_o \bar{\phi}_o,$$
$$\textit{and } \bar{\phi}_o \neq ba,\ \textit{and } \langle s,\ \varepsilon_r \rangle \Downarrow_r \bar{\phi}_r,\ \textit{then } \neg err(\bar{\phi}_r).$$

The relaxed progress theorem combines the proof rules of the original axiomatic semantics and the relaxed axiomatic to give an overall guarantee of the development. Specifically, given a proof in the original axiomatic semantics and a proof in the relaxed axiomatic semantics, then for all pairs of environments that satisfy the procedure's precondition, if execution of the original program terminates and does not violate an assumption, then if the relaxed program terminates it does encounter an error.

Corollary 18 (Relaxed Progress Modulo Original Assumptions).

$$\textit{If } \gamma \vdash_o \mathit{fn}\ \{\ s\ \},\ \textit{and } \gamma \vdash_r \mathit{fn}\ \{\ s\ \},\ (\varepsilon_o, \varepsilon_r) \models \langle P \cdot P \rangle,\ \textit{and } (\varepsilon_o, \varepsilon_r) \models \hat{P},\ \textit{and } \langle s,\ \varepsilon_r \rangle \Downarrow_r \bar{\phi}_r,$$
$$\textit{and } err(\bar{\phi}_r),\ \textit{If } \langle s,\ \varepsilon_o \rangle \Downarrow_o \bar{\phi}_o,\ \textit{then } \bar{\phi}_o = ba$$

A corollary of the Relaxed Progress Theorem is if the original program is not verified then if the relaxed program violates and assertion or assumption, the original program also violates and assertion or assumption. As with the core calculus, the rules of the full language provide a non-interference guarantee such that in an environment where the behavior of the original program is not formally guaranteed then errors in the relaxed program correspond to errors in the original.

3.6 Case Studies

I next present a set of case studies that demonstrate how to reason about extended control flow, arrays, heap-allocated data, and procedures.

3.6.1 Adaptive Loop Perforation

Bodytrack is an implementation of a computer vision system that identifies and tracks the position and pose of a human body in a video. The relaxation I focus on uses *adaptive* loop perforation to optimize a computation that uses a sampling-based approach to cast points on a shape that represents a limb. This computation uses the results of the samples to estimate how well the shape fits the image as a representation of the limb. Loop perforation in this case reduces the number of samples that the sampling-based computation uses with the aim of increasing performance while still enabling the computation to return an acceptably accurate result [58].

Relaxation. Bodytrack computes a pose of a human body as represented by a connected set of 3-dimensional cylinders, where each cylinder corresponds to body part (e.g., a fore-arm, head, or torso). The transformation targets a loop that enumerates over a generated set of 2-dimensional points that the loop lays across a 2-dimensional projection of the cylinder that represents a limb:

```
float *err = new float;
int num_samples = 0
int acc = 0;
for (int i = 0; i < n; ++i) {
   int *x = new int;
   int *y = new int;

   GetCoord(i, x, y);

   num_samples = num_samples + Sample(x, y, err);
   relax (i) with ((i == old(i)) ||
                   (!(num_samples > 0) || i == old(i) + 1)};
}
float err_v = *err;
assert (0 < num_samples);
float avg_error = err_v / num_samples;
```

The loop enumerates over coordinates of each point (given by pointers `x` and `y`) and takes a sample of the image features that underlie that point with the function `Sample`. The function takes a sample of the image features to determine if the point corresponds to the image's foreground (indicating part of the body) or background (indicating an error because the cylinder isn't over the body). Using this sample, it computes an error term, increments `err` with the value of the error term and returns whether or not it was able to successfully sample the point.

A key challenge to perforating this loop is avoiding the divide-by-zero error that may occur during division of `err` by `samples` at the end of the function. The function `Sample` may in some cases (specifically if the sample point falls outside of the bounds of the image) not take a sample and therefore return 0 as a return value. Loop perforation therefore needs to preserve the property that if there is one valid sample, then `num_samples` must be greater than zero.

Naive loop perforation will violate this property because loop perforation may cause the loop to skip the only valid sample point, therefore introducing an error into the relaxed program that does not exist in the original. This act would violate the relaxed progress property we desire from relaxed programs.

The relaxation I instead present is *adaptive* loop perforation. Adaptive loop perforation waits to perforate the loop until a condition over the local state of the program is satisfied [19]. The `relax` statement implements adaptive perforation of this loop by including the guard that `num_samples` must be greater than zero for the statement to consider incrementing `i` an additional time. Given this guard, adaptive perforation of this loop preserves the relational invariant that if `num_samples` is greater than zero in the original program, then `num_samples` is greater than zero in the relaxed. This invariant enables a proof that establishes that the relaxed program does not interfere with the `assume` statement that specifies the safety condition of the division.

Proof. The relaxed axiomatic proof establishes that the following invariant holds at the end of the loop:

$$(0 < \texttt{num\_samples<o>}) \rightarrow (0 < \texttt{num\_samples<r>}).$$

To establish this invariant, I work with the loop invariant

$$\begin{aligned}\mathtt{i}\langle\mathtt{o}\rangle = \mathtt{i}\langle\mathtt{r}\rangle \rightarrow ((\mathtt{0} < \mathtt{num\_samples}\langle\mathtt{o}\rangle) \rightarrow (\mathtt{0} < \mathtt{num\_samples}\langle\mathtt{r}\rangle)) \wedge \\ !(\mathtt{i}\langle\mathtt{o}\rangle = \mathtt{i}\langle\mathtt{r}\rangle) \rightarrow \mathtt{0} < \mathtt{num\_samples}\langle\mathtt{r}\rangle \wedge \\ \mathtt{n}\langle\mathtt{o}\rangle = \mathtt{n}\langle\mathtt{r}\rangle \wedge \mathtt{i}\langle\mathtt{o}\rangle \leq \mathtt{i}\langle\mathtt{r}\rangle \wedge \mathtt{i}\langle\mathtt{o}\rangle \leq n\langle\mathtt{o}\rangle \wedge \mathtt{i}\langle\mathtt{r}\rangle \leq n\langle\mathtt{r}\rangle\end{aligned}$$

Because this relaxation causes the control of the original and relaxed program to diverge, the proof of this loop invariant requires using the extended rules for reasoning about loops. Using those rules, there are four cases to reason about:

- **Initialization.** Before entering the loop, it is the case that `i`, `num_samples`, and `n` in both the original program and relaxed program have the same respective values. Therefore the loop invariant holds.

- **Both Original and Relaxed Program Execute Iteration.** Next, both the original and relaxed program execute iterations of the loop, enabling us to reason about the two executions in lockstep (even though the value of `i` may differ between the two programs). The goal of the proof at this point is to demonstrate that that the invariant is preserved at the end of an iteration where both the relaxed and original program take a step.

  For the first conjunct of the invariant, preservation needs to be established at two mains point. First at the call to `Sample`, preservation follows from specifications of determinism. Specifically, `Sample` must have a specification that states that two calls of the procedure with the same arguments return the same values. This is can be provided as a relational specification.

  The second statement that may violate the first conjunct of the invariant is the relax statement. Specifically, the statement may violate the constraint that `i` has the same value between the original and relaxed program. However, because of the condition of the relaxation, if this is the case, then the second conjunct of the invariant ($!(\mathtt{i}\langle\mathtt{o}\rangle = \mathtt{i}\langle\mathtt{r}\rangle) \rightarrow \mathtt{0} < \mathtt{num\_samples}\langle\mathtt{r}\rangle$) holds.

For the second conjunct of the invariant, the primary reasoning happens at the call to `Sample`, the shared (between the original and relaxed program) update to `i`, and the `relax` statement. At the call to sample, it is possible to specify and use a specification for `Sample` that states that its result is non-negative, therefore ensuring that the resulting update to `num_samples` does not decrease its value (potentially making it less than one). The shared update to `i` preserves the invariant as `i` is still different between the two programs. At the `relax` statement, `i` may be updated in the relaxed program but not in the original program. Because of the invariant that $\texttt{i}\langle\texttt{o}\rangle \leq \texttt{i}\langle\texttt{r}\rangle$, if `i` is updated in the relaxed program, then its value will still be different from that of the original program.

- **Original Program Executes Iteration, Relaxed Program Has Terminated.** For this case to occur, it must be true that $\texttt{i}\langle\texttt{o}\rangle < \texttt{n}\langle\texttt{o}\rangle$ and $!(\texttt{i}\langle\texttt{r}\rangle < \texttt{n}\langle\texttt{r}\rangle)$. Because $\texttt{n}\langle\texttt{o}\rangle = \texttt{n}\langle\texttt{r}\rangle$, it must be the case that $!(\texttt{i}\langle\texttt{o}\rangle = \texttt{i}\langle\texttt{r}\rangle)$ and therefore $0 < \texttt{num\_samples<r>}$. Because the original program does not modify `num_samples<r>`, the execution at the end of the loop's body will still satisfy the loop invariant.

- **Original Program Has Terminated, Relaxed Program Executes Iteration.** For this case to occur, it must be true that $!\texttt{i}\langle\texttt{o}\rangle < \texttt{n}\langle\texttt{o}\rangle$ and $(\texttt{i}\langle\texttt{r}\rangle < \texttt{n}\langle\texttt{r}\rangle)$. Because of the invariants that $\texttt{i}\langle\texttt{o}\rangle \leq \texttt{i}\langle\texttt{r}\rangle$ and $\texttt{n}\langle\texttt{o}\rangle = \texttt{n}\langle\texttt{r}\rangle$, this case infeasible and can therefore be discharged without any additional obligations.

At the end of the loop, using the invariants $\texttt{i}\langle\texttt{o}\rangle \leq n\langle\texttt{o}\rangle$ and $\texttt{i}\langle\texttt{r}\rangle \leq n\langle\texttt{r}\rangle$, along with the facts that $!(\texttt{i}\langle\texttt{o}\rangle < n\langle\texttt{o}\rangle)$ and $!(\texttt{i}\langle\texttt{r}\rangle \leq n\langle\texttt{r}\rangle)$, we can conclude that $\texttt{i}\langle\texttt{o}\rangle = \texttt{i}\langle\texttt{r}\rangle$. Given the loop invariant, we can therefore prove that it is the case that (`0 < num_samples<o>`) $\rightarrow$ (`0 < num_samples<r>`). This property enables us to demonstrate discharge the proof obligation for the `assume`, establishing the relative progress of the relaxed program: if the original program satisfies the `assume` statement, then the relaxed program does also.

3.6.2 Separation

Many of my examples deal directly with verifying the effects of relaxation on the semantics of the program. Verification works hand in hand with the specifications and assertions that

```
struct Body {
  float x, float y;
  Body *next;
};

void step_system(Body *bodies, int nsteps) {
  int i = 0;
  while (i < nsteps) {
    Tree *tree = build_tree(bodies);
    update_bodies<2>(tree);
    ++i;
  }
}

struct SPT {
   Body *body;
   SPT left;
   SPT Right;
};

SPT *tree build_tree(Body *bodies) :
   requires List(bodies)
   ensures Tree(ret)
{ ... }

void update_bodies(SPT *tree) :
   requires Tree(tree)
{ ... }
```

Figure 3-26: Structure of Barnes-Hut Simulation

a developer has placed in a program in that they constrain the set of valid, verifiable relaxations. In addition to reasoning about how relaxations modify data, in many cases it is also important to constrain which data is modified. For example, developers can use relational assertions, such as `relax` (x) `st` $(\texttt{x}\langle\texttt{o}\rangle = \texttt{x}\langle\texttt{r}\rangle)$ or `relax` (x) `st` $(*\texttt{x}\langle\texttt{o}\rangle = *\texttt{x}\langle\texttt{r}\rangle)$ to assert that relaxation must not modify a local or heap-allocated variable. However, in the case larger data structures, it can be productive to rely on separation logic to constrain the footprint of a relaxed computation. For example, consider a version of the Barnes-Hut N-body simulation computation from the Olden Benchmark Suite [22] presented in Figure 3-26.

This code snippet provides the outline of the basic structure of the computation. The computation keeps a list of bodies (e.g., particles) which here have coordinates `x` and `y` that specify their position[1]. The computation maintains the list of bodies through the `next` field of each body in a list.

The computation's main computation, `step_system`, iterates for `n` time steps. In each step, the computation uses the list of bodies to build a space partitioning tree (`build_tree`) that hierarchically partitions the space in which the bodies exist, enabling efficient computation of their interactions (`update_bodies`).

The `update_bodies` computation is amenable to relaxations, such as statistical parallelization [56]. However, one challenge for verifying a relaxation of `update_bodies`, is ensuring that the list of bodies is not modified in a way that may jeopardize the safety of `build_tree` when executed in the next iteration of the loop.

To enable this verification, we specify as `build_tree`'s precondition that the list of bodies needs to have an appropriate shape. We specify shape of the list by introducing into the logic an uninterpreted predicate `List`:

$$\texttt{List}(b) \leftrightarrow b = 0 \vee (\exists b'.\ b.\texttt{next} \mapsto b' * \texttt{List}(b'))$$

This predicate specifies that a `Body` pointer b is a list if it is either 0 (indicating an empty list) or there exists another address b' such that the `next` field of b points to another *disjoint* list.

The specification for `build_tree` is that it produces a valid `Tree`:

$$\texttt{Tree}(t) = t = 0 \vee (\exists b, v_1 v_2.\ n.\texttt{body} \mapsto b * b.\texttt{x} \mapsto v_1 * b.\texttt{y} \mapsto v_2) \vee$$
$$(\exists n_1.\ n.\texttt{left} \mapsto n_1 * \texttt{Tree}(n_1) * \exists n_2.\ n.\texttt{right} \mapsto n_2 * \texttt{Tree}(n_2)$$

A valid tree is a `Tree` pointer t that is either 0 (indicating an empty tree), a leaf node with a `Body` pointer in its `body` field that has values for both x and y, or it is an interior node for which both of its children are also trees.

The goal of the specification for `update_bodies` is to ensure that the list structure of *bodies* is preserved. To achieve this, we rely on the separation properties of separation logic

[1] In the full implementation these particles also include fields for other properties, such as mass and velocity

to specify `update_bodies`'s footprint in its precondition. Specifically, `update_bodies`'s precondition specifies that the tree it receives is a tree. Because of the properties of separation logic, the `Tree` predicate explicitly enumerates the heap-allocated data that is available to `update_bodies`. Namely, `Tree` specifies that all nodes that are reachable from the tree's root by following each node's `left` and `right` pointers are available. Moreover, the procedure may manipulate the `x` and `y` coordinates of each body. Note that this specification does not include the `next` fields of each body. Because these are not accessible to the function, they cannot be modified by the function's relaxed implementation. Therefore, via the logic's frame rule, the property `List`(bodies) (which is expressed only over the `next` fields of each body) will hold after `update_bodies`'s execution and therefore before `build_tree` on the next iteration.

3.7 Related Work

The core foundation of my work in this chapter is the work I presented in Chapter 2. Building upon this foundation, I have added the capability for relational reasoning for programs that have divergent control flow, procedures, and heap-allocated data structures and arrays.

Separation Logic. Reynold's work lays out the basic connectives for separation logic, including separating conjunction, separating implication, and iterated separating conjunction [75]. My work adapts Reynold's general separation logic, which includes arbitrary address calculations to a more structured Java-like context in which accessed address an offset from an allocated object array.

Relational Separation Logic. Yang proposes a relational separation logic that builds upon Reynold's original work [92]. His primary contribution is the syntax and semantics of relational Hoare quadruples, which relate values between two programs. The base connectives of his logic and my logic are shared with that of Reynold's work. A primary difference between Relational Separation Logic and my work is I have designed my logic to enable the relational reasoning approach inherent in relaxed programs: a proof for a re-

laxed program may assume and transfer properties of the original program over to relaxed program to lower the verification burden. Relational separation logic, in contrast, does not support an asymmetric distinction between an original program (that is assumed to be correct) and a relaxed program (that one desires to build a proof for given the assumptions over the original program).

Control Flow. My extended rules for control flow are inspired by rules (of a variety of different forms) that have been proposed for reasoning about control flow. The loosed axiomatic original and relaxed semantics are inspired by the left and right rules of self-composition approaches to relational verification [8]. A key distinction in my work is loosed axiomatic original and relaxed semantics have asymmetric guarantees for the behavior of their respective programs. The loosed axiomatic original semantics is sound, but does not provide a progress property for the original program as it simply collects the semantics of the original program. The loosed axiomatic relaxed semantics provides both soundness and progress for the relaxed program.

3.8 Conclusion

The relaxed programming framework enables developers to precisely characterize the effects of changes to the semantics of programs along with the acceptability properties that these augmented programs must satisfy. My work in this chapter extends the relaxed programming framework to enable more precise reasoning about programs that have divergent control flow and use traditional programming structures, such as arrays, heap-allocated data structures, and procedures. Handling these additional programming structures, enables relaxed programming for a wide variety of applications and domains, proving a sound reasoning framework for both manual and automated development of relaxed programs as the research community continues to uncover the potential of approximate hardware and software computing systems.

Chapter 4

Verifying Quantitative Reliability

Relaxed programs enable worst-case reasoning about the safety and accuracy of approximate computations. However, there are new opportunities for approximation in which probalistic reasoning can provide additional confidence to developers about the behavior of their programs.

For example, system reliability is a major challenge in the design of emerging architectures. Energy efficiency and circuit scaling are becoming major goals when designing new devices. However, aggressively pursuing these design goals can often increase the frequency of *soft errors* – which occur non-deterministically with some probability – in small [87] and large systems [17] alike. Researchers have developed numerous techniques for detecting and masking soft errors in both hardware [31] and software [74, 69, 27, 84]. However, in emerging computational platforms, fully detecting and masking soft errors may be infeasiable or substantially hinder overall system performance because these techniques typically come at the price of increased execution time, increased energy consumption, or both.

Building on the same insights that movitated the development of relaxed programs – namely, that the semantics of many computations can be relaxed and still provide useful and acceptable functionality – researchers have proposed executing computations without (or with at most selectively applied) mechanisms that detect and mask soft errors in the persuit of 1) fast and energy efficient execution that 2) delivers acceptably accurate results often enough to satisfy the needs of their users despite the presence of unmasked soft errors.

In this environment the probilistic semantics of the errors themselves and the probalistic impact on program behavior is as important as worst-case reasoning about the acceptability of the computation.

To meet this challenge, this chapter presents a new programming language, Rely, and an associated program analysis that computes the *quantitative reliability* of the computation — i.e., the probability with which the computation produces a correct result when parts of the computation execute on unreliable hardware. Specifically, given a hardware specification and a Rely program, the analysis computes, for each value that the computation produces, a conservative probability that the value is computed correctly despite the possibility of soft errors.

Rely supports and is specifically designed to enable partitioning a program into *critical* regions (which must execute without error) and *approximate* regions (which can execute acceptably even in the presence of occasional errors) [77, 21]. In contrast to existing approaches, which support only a binary distinction between critical and approximate regions, quantitative reliability can provide precise static probabilistic acceptability guarantees for computations that execute on unreliable hardware platforms.

4.1 Overview and Contributions

This section presents an overview of the quantitative reliability framework, including the Rely programming language and its corresponding reliability analysis, along with an overview of the framework's primary contributions.

4.1.1 Rely

Rely is an imperative language that enables developers to specify and verify quantitative reliability specifications for programs that allocate data in unreliable memory regions and incorporate unreliable arithmetic/logical operations.

Quantitative Reliability Specifications. Rely supports quantitative reliability specifications for the results that functions produce. For example, a developer can declare a function

signature `int<0.99*R(x, y)> f(int x, int y, int z)`, where `0.99*R(x, y)` is the reliability specification for `f`'s return value. The symbolic expression `R(x, y)` is the *joint reliability* of `x` and `y` – namely, the probability that they both simultaneously have the correct value on entry to the function. This specification states that the reliability of the return value of `f` must be *at least* 99% of `x` and `y`'s reliability when the function was called.

Joint reliabilities serve as an abstraction of a function's input distribution, which enables Rely's analysis to be both modular and oblivious to the exact shape of the distributions. This is important because 1) such exact shapes can be difficult for developers to identify and specify and 2) known tractable classes of probability distributions are not closed under many operations found in standard programming languages, which can complicate attempts to develop compositional analyses that work with such exact shapes [57, 94, 35, 85].

Machine Model. Rely assumes a simple machine model that consists of a processor (with a register file and an arithmetic/logic unit) and a main memory. The model includes unreliable arithmetic/logical operations (which return an incorrect value with non-negligible probability [27, 84, 32, 33]) and unreliable physical memories (in which data may be written or read incorrectly with non-negligible probability [49, 84, 32]). Rely works with a *hardware reliability specification* that lists the probability with which each operation in the machine model executes correctly.

Language. Rely is an imperative language with integer, logical, and floating point expressions, arrays, conditionals, while loops, and function calls. In addition to these standard language features, Rely also allows a developer to allocate data in unreliable memories and write code that uses unreliable arithmetic/logical operations. For example, the declaration `int x in urel` allocates the variable `x` in an unreliable memory named `urel` where both reads and writes of `x` may fail with some probability. A developer can also write an expression `a +. b`, which is an unreliable addition of the values `a` and `b` that may produce an incorrect result.

Semantics. I have designed the semantics of Rely to exploit the full availability of unreliable computation in an application. Rely only requires reliable computation at points where doing so ensures that programs are memory safe and exhibit control flow integrity.

Rely's semantics models an abstract machine that consists of a heap and a stack. The stack consists of frames that contain references to the locations of each invoked function's variables (which are allocated in the heap). To protect references from corruption, the stack is allocated in a reliable memory region and stack operations – i.e., pushing and popping frames – execute reliably. To prevent out-of-bounds memory accesses that may occur as a consequence of an unreliable array index computation, Rely requires that each array read and write includes a bounds check; these bounds check computations also execute reliably. Rely does not require a specific underlying mechanism to execute these operations reliably; one can use any applicable software or hardware technique [74, 69, 27, 34, 67, 86, 37, 90].

To prevent the execution from taking control flow edges that are not in the program's static control flow graph, Rely assumes that 1) instructions are stored, fetched, and decoded reliably (as is supported by existing unreliable processor architectures [84, 32]) and 2) control flow branch targets are computed reliably.

4.1.2 Quantitative Reliability Analysis

Given a Rely program and a hardware reliability specification, Rely's analysis uses a precondition generation approach to generate a symbolic *reliability precondition* for each function. A reliability precondition captures a set of constraints that is sufficient to ensure that a function satisfies its reliability specification when executed on the underlying unreliable hardware platform. The reliability precondition is a conjunction of predicates of the form $A_{out} \leq r \cdot \mathcal{R}(X)$, where A_{out} is a placeholder for a developer-provided reliability specification for an output named *out*, r is a real number between 0 and 1, and the term $\mathcal{R}(X)$ is the joint reliability of a set of parameters X.

Conceptually, each predicate specifies that the reliability given in the specification (given by A_{out}) should be less than or equal to the reliability of a path that the program may take to compute the result (given by $r \cdot \mathcal{R}(X)$). The analysis computes the reliability of a path from the probability that all operations along the path execute reliably.

The specification is valid if the probabilities for all paths to computing a result exceed that of the result's specification. To avoid the inherent intractability of considering all

possible paths, Rely uses a simplification procedure to reduce the precondition to one that characterizes the least reliable path(s) through the function.

Loops. One of the core challenges in designing Rely and its analysis is dealing with unreliable computation within loops. The reliability of variables updated within a loop may depend on the number of iterations that the loop executes. Specifically, if a variable has a loop-carried dependence and updates to that variable involve unreliable operations, then the variable's reliability is a monotonically decreasing function of the number of iterations of the loop – on each loop iteration the reliability of the variable degrades relative to its previous reliability. If a loop does not have a compile-time bound on the maximum number of iterations, then the reliability of such a variable can, in principle, degrade arbitrarily, and the only conservative approximation of the reliability of such a variable is zero.

To provide specification and verification flexibility, Rely provides two types of loop constructs: statically *bounded* `while` loops and statically *unbounded* `while` loops. Statically bounded `while` loops allow a developer to provide a static bound on the *maximum* number of iterations of a loop. The dynamic semantics of such a loop is to exit if the number of executed iterations reaches this bound. This bound allows Rely's analysis to soundly construct constraints on the reliability of variables modified within the loop by unrolling the loop for its maximum bound.

Statically unbounded `while` loops have the same dynamic semantics as standard `while` loops. In the absence of a static bound on the number of executed loop iterations, however, Rely's analysis constructs a dependence graph of the loop's body to identify variables that are *reliably updated* – specifically, all operations that influence these variables' values are reliable. Because the execution of the loop does not decrease the reliability of these variables, the analysis identifies that their reliabilities are unchanged. For the remaining, unreliably updated variables, Rely's analysis conservatively sets their reliability to zero.

Specification Checking. In the last step of the analysis of a function, Rely verifies that the function's specifications are consistent with its reliability precondition. Because reliability specifications are also of the form $r \cdot \mathcal{R}(X)$, the final precondition is a conjunction of

predicates of the form $r_1 \cdot \mathcal{R}(X_1) \leq r_2 \cdot \mathcal{R}(X_2)$, where $r_1 \cdot \mathcal{R}(X_1)$ is a reliability specification and $r_2 \cdot \mathcal{R}(X_2)$ is a path reliability. If these predicates are valid, then the reliability of each computed output is greater than that given by its specification.

The validity problem for these predicates has a sound mapping to the conjunction of two simple constraint validity problems: inequalities between real numbers ($r_1 \leq r_2$) and set inclusion constraints over finite sets ($X_2 \subseteq X_1$). Checking the validity of a reliability precondition is therefore decidable and efficiently checkable.

4.1.3 Case Studies

I have used Rely to build unreliable versions of six building block computations for media processing, machine learning, and data analytics applications. My case studies illustrate how quantitative reliability enables a developer to use principled reasoning to relax the semantics of both *approximate computations* and *checkable computations*.

An approximate computation (including many multimedia, financial, machine learning, and big data analytics applications) can often acceptably tolerate occasional errors in its execution and/or the data that it manipulates [77, 58, 21].

A checkable computation can be augmented with an efficient checker that verifies the acceptability of the computation's results [14, 46, 15, 71]. If the checker does detect an error, it can reexecute the computation to obtain an acceptable result.

For approximate computations, quantitative reliability allows a developer to reify and verify the results of the fault injection and accuracy explorations that are typically used to identify the minimum acceptable reliability of a computation [58, 57, 88, 94]. For checkable computations, quantitative reliability allows a developer to use the performance specifications of both the computation and its checker to determine the computation's overall performance given that – with some probability – it may produce an incorrect answer and therefore needs to be reexecuted.

4.1.4 Contributions

This chapter presents the following contributions:

Quantitative Reliability Specifications. I present quantitative reliability specifications, which characterize the probability that a program executed on unreliable hardware produces the correct result, as a constructive method for developing applications. Quantitative reliability specifications enable developers who build applications for unreliable hardware architectures to perform sound and verified reliability engineering.

Language and Semantics. I present Rely, a language that allows developers to specify reliability requirements for programs that allocate data in unreliable memory regions and use unreliable arithmetic/logical operations. I also present a dynamic semantics for Rely via a probabilistic small-step operational semantics. This semantics is parameterized by a hardware reliability specification that characterizes the probability that an unreliable arithmetic/logical or memory read/write operation executes correctly.

Semantics of Quantitative Reliability. I formalize the semantics of quantitative reliability as it relates to the probabilistic dynamic semantics of a Rely program. Specifically, I define the quantitative reliability of a variable as the probability that its value in an unreliable execution of the program is the same as that in a fully reliable execution. I also define the semantics of a logical predicate language that can characterize the reliability of variables in a program.

Quantitative Reliability Analysis. I present a program analysis that verifies that the dynamic semantics of a Rely program satisfies its quantitative reliability specifications. For each function in the program, the analysis computes a symbolic reliability precondition that characterizes the set of valid specifications for the function. The analysis then verifies that the developer-provided specifications are valid according to the reliability precondition.

Case Studies. I have used Rely's implementation to develop unreliable versions of six building block computations for media processing, machine learning, and data analytics applications. These case studies illustrate how to use quantitative reliability to develop and reason about both approximate and checkable computations in a principled way.

$$
\begin{array}{rcl}
n & \in & \text{Int}_M \\
r & \in & \mathbb{R} \\
x, \ell & \in & \text{Var} \\
a & \in & \text{ArrVar}
\end{array}
\qquad
\begin{array}{rcl}
e \in Exp & \rightarrow & n \mid x \mid (Exp) \mid Exp\ iop\ Exp \\
b \in BExp & \rightarrow & \texttt{true} \mid \texttt{false} \mid Exp\ cmp\ Exp \mid (BExp) \mid \\
 & & BExp\ lop\ BExp \mid \texttt{!}BExp \mid \texttt{!.}BExp \\
CExp & \rightarrow & e \mid a
\end{array}
$$

$$
\begin{array}{rcl}
m & \in & \text{MVar} \\
V & \rightarrow & x \mid a \mid V,x \mid V,a \\
RSpec & \rightarrow & r \mid \texttt{R}(V) \mid r\texttt{*R}(V) \\
T & \rightarrow & \texttt{int} \mid \texttt{int<}RSpec\texttt{>}
\end{array}
\qquad
\begin{array}{rcl}
F & \rightarrow & (T \mid \texttt{void})\ ID\ (P^*)\ \texttt{\{}\ S\ \texttt{\}} \\
P & \rightarrow & P_0\ [\texttt{in}\ m] \\
P_0 & \rightarrow & \texttt{int}\ x \mid T\ a(n) \\
S & \rightarrow & D^*\ S_s\ S_r^?
\end{array}
$$

$$
\begin{array}{rcl}
D & \rightarrow & D_0\ [\texttt{in}\ m] \\
D_0 & \rightarrow & \texttt{int}\ x\ [= Exp] \mid \texttt{int}\ a\texttt{[}n^+\texttt{]} \\
S_s & \rightarrow & \texttt{skip} \mid x = Exp \mid\ x = a\texttt{[}Exp^+\texttt{]} \mid a\texttt{[}Exp^+\texttt{]} = Exp \mid \\
 & & ID\texttt{(}CExp^*\texttt{)} \mid x = ID\texttt{(}CExp^*\texttt{)} \mid \texttt{if}_\ell\ BExp\ S\ S \mid S\ ;\ S \\
 & & \texttt{while}_\ell\ BExp\ [:\ n]\ S\ \mid\ \texttt{repeat}_\ell\ n\ S \\
S_r & \rightarrow & \texttt{return}\ Exp
\end{array}
$$

Figure 4-1: Rely's Language Syntax

4.2 Example

Figure 4-1 presents the syntax of the Rely language. Rely is an imperative language for computations over integers, floats (not presented), and multidimensional arrays. To illustrate how a developer can use Rely, Figure 4-2 presents a Rely-based implementation of a pixel block search algorithm derived from that in the x264 video encoder [91].

The function `search_ref` searches a region (`pblocks`) of a previously encoded video frame to find the block of pixels that is most similar to a given block of pixels (`cblock`) in the current frame. The motion estimation algorithm uses the results of `search_ref` to encode `cblock` as a function of the identified block.

This is an approximate computation that can trade correctness for more efficient execution by approximating the search to find a block. If `search_ref` returns a block that is not the most similar, then the encoder may require more bits to encode `cblock`, potentially decreasing the video's peak signal-to-noise ratio or increasing its size. However, previous studies on soft error injection [27] and more aggressive transformations like loop perforation [58, 88] have demonstrated that the quality of x264's final result is only slightly affected by perturbations of this computation.

```
1  #define nblocks 20
2  #define height  16
3  #define width   16
4
5  int<0.99*R(pblocks, cblock)> search_ref (
6      int<R(pblocks)> pblocks(3) in urel,
7      int<R(cblock)>  cblock(2) in urel)
8  {
9    int minssd = INT_MAX,
10       minblock = -1 in urel;
11   int ssd, t, t1, t2 in urel;
12   int i = 0, j, k;
13
14   repeat nblocks  {
15     ssd = 0;
16     j = 0;
17     repeat height {
18       k = 0;
19       repeat width {
20         t1 = pblocks[i,j,k];
21         t2 = cblock[j,k];
22         t = t1 -. t2;
23         ssd = ssd +. t *. t;
24         k = k + 1;
25       }
26       j = j + 1;
27     }
28
29     if (ssd <. minssd) {
30       minssd = ssd;
31       minblock = i;
32     }
33
34     i = i + 1;
35   }
36   return minblock;
37 }
```

Figure 4-2: Rely Code for Motion Estimation Computation

4.2.1 Reliability Specifications

The function declaration on Line 5 specifies the types and reliabilities of `search_ref`'s parameters and return value. The parameters of the function are `pblocks(3)`, a three-dimensional array of pixels, and `cblock(2)`, a two-dimensional array of pixels. In addition to the standard signature, the function declaration contains *reliability specifications* for each result that the function produces.

Rely's reliability specifications express the reliability of a function's results – when executed on an unreliable hardware platform – as a function of the reliabilities of its inputs. The specification for the reliability of `search_ref`'s result is `int<0.99*R(pblocks,cblock)>`. This states that the return value is an integer with a reliability that is at least 99% of the *joint reliability* of the parameters `pblocks` and `cblock` (denoted by `R(pblocks, cblock)`). The joint reliability of a set of parameters is the probability that they all have the correct value when passed in from the caller. This specification holds for all possible values of the joint reliability of `pblocks` and `cblock`. For instance, if the contents of the arrays `pblocks` and `cblock` are fully reliable (correct with probability one), then the return value is correct with probability 0.99.

In Rely, arrays are passed by reference and the execution of a function can, as a side effect, modify an array's contents. The reliability specification of an array therefore allows a developer to constrain the *reliability degradation* of its contents. Here `pblocks` has an output reliability specification of `R(pblocks)` (and similarly for `cblock`), meaning that all of `pblock`'s elements are at least as reliable when the function exits as they were on entry to the function.

4.2.2 Unreliable Computation

Rely targets hardware architectures that expose both reliable operations (which always execute correctly) and more energy-efficient unreliable operations (which execute correctly with only some probability). Specifically, Rely supports reasoning about reads and writes of unreliable memory regions and unreliable arithmetic/logical operations.

Memory Region Specification. Each parameter declaration also specifies the memory region in which the data of the parameter is allocated. Memory regions correspond to the physical partitioning of memory at the hardware level into regions of varying reliability. Here `pblocks` and `cblock` are allocated in an unreliable memory region named `urel`.

Lines 9-12 declare the local variables of the function. By default, variables in Rely are allocated in a default, fully reliable memory region. However, a developer can also optionally specify a memory region for each local variable. For example, the variables declared on Lines 9-11 reside in `urel`.

Unreliable Operations. The operations on Lines 22, 23, and 29 are unreliable arithmetic/logical operations. In Rely, every arithmetic/logical operation has an unreliable counterpart that is denoted by suffixing a period after the operation symbol. For example, "`-.`" denotes unreliable subtraction and "`<.`" denotes unreliable comparison.

Using these operations, `search_ref`'s implementation *approximately* computes the index (`minblock`) of the most similar block, i.e. the block with the minimum distance from `cblock`. The `repeat` statement on line 14, iterates a constant `nblock` number of times, enumerating over all previously encoded blocks. For each encoded block, the `repeat` statements on lines 17 and 19 iterate over the `height*width` pixels of the block and compute the sum of the squared differences (`ssd`) between each pixel value and the corresponding pixel value in the current block `cblock`. Finally, the computation on lines 29 through 32 selects the block that is – approximately – the most similar to `cblock`.

4.2.3 Hardware Semantics

Figure 4-3 illustrates the conceptual machine model behind Rely's reliable and unreliable operations; the model consists of a CPU and a memory.

CPU. The CPU consists of 1) a register file, 2) arithmetic logical units that perform operations on data in registers, and 3) a control unit that manages the program's execution.

The arithmetic-logical unit can execute reliably or unreliably. I have represented this in Figure 4-3 by physically separate reliable and unreliable functional units, but this distinction can be achieved through other mechanisms, such as dual-voltage architectures [32].

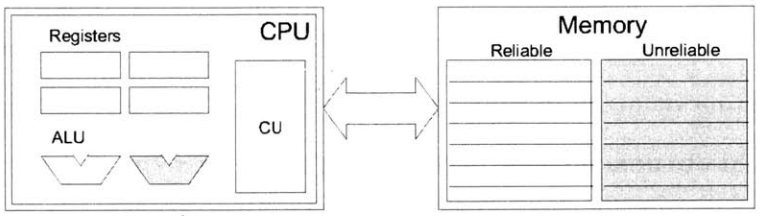

Figure 4-3: Machine Model Illustration. Gray boxes represent unreliable components

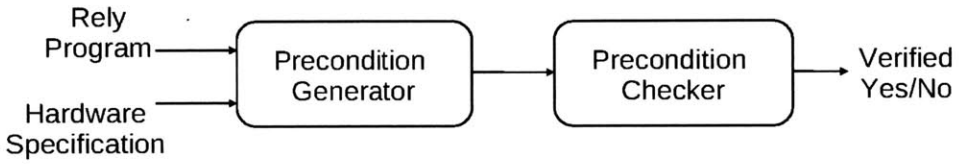

Figure 4-4: Rely's Analysis Overview

Unreliable functional units may omit additional checking logic, enabling the unit to execute more efficiently but also allowing for soft errors that may occur due to, for example, power variations within the ALU's combinatorial circuits or particle strikes. As is provided by existing computer architecture proposals [84, 32], the control unit of the CPU reliably fetches, decodes, and schedules instructions; given a virtual address in the application, the control unit correctly computes a physical address and operates only on that address.

Memory. Rely supports machines with memories that consist of an arbitrary number of memory partitions (each potentially of different reliability), but for simplicity Figure 4-3 partitions memory into two regions: reliable and unreliable. Unreliable memories can, for example, use decreased DRAM refresh rates to reduce power consumption at the expense of increased soft error rates [49, 84].

4.2.4 Reliability Analysis

Given a Rely program, Rely's reliability analysis verifies that each function in the program satisfies its reliability specification when executed on unreliable hardware. Figure 4-4 presents an overview of Rely's analysis. It takes as input a Rely program and a *hardware reliability specification.*

The analysis consists of two components: the *precondition generator* and the *precondition checker*. For each function, the precondition generator produces a precondition that characterizes the reliability of the function's results given a hardware reliability specification that characterizes the reliability of each unreliable operation. The precondition checker then determines if the function's specifications satisfy the constraint. If so, then the function satisfies its reliability specification when executed on the underlying unreliable hardware in that the reliability of its results exceed their specifications.

Design. As a key design point, the analysis generates preconditions according to a conservative approximation of the semantics of the function. Specifically, it characterizes the reliability of a function's result according to the probability that the function computes that result fully reliably.

To illustrate the intuition behind this design point, consider the evaluation of an integer expression e. The reliability of e is the probability that it evaluates to the same value n in an unreliable evaluation as in the fully reliable evaluation. There are two ways that an unreliable evaluation can return n: 1) the unreliable evaluation of e encounters no faults and 2) the unreliable evaluation possibly encounters faults, but still returns n by chance.

Rely's analysis conservatively approximates the reliability of a computation by only considering the first scenario. This design point simplifies the reasoning to the task of computing the probability that a result is reliably computed as opposed to reasoning about a computation's input distribution and the probabilities of all executions that produce the correct result. As a consequence, the analysis requires as input only a hardware reliability specification that gives the probability with which each arithmetic/logical operation and memory operation executes correctly. The analysis is therefore oblivious to a computation's input distribution and does not require a full model of how soft errors affect its result.

Hardware Reliability Specification

Rely's analysis works with a hardware reliability specification that specifies the reliability of arithmetic/logical and memory operations. Figure 4-5 presents a hardware reliability specification that is inspired by the results from existing computer architecture litera-

```
reliability spec {
  operator (+.) = 1 - 10^-7;
  operator (-.) = 1 - 10^-7;
  operator (*.) = 1 - 10^-7;
  operator (<.) = 1 - 10^-7;
  memory rel {rd = 1, wr = 1};
  memory urel {rd = 1 - 10^-7, wr = 1};
}
```

Figure 4-5: Hardware Reliability Specification

ture [31, 49]. Each entry specifies the reliability – the probability of a correct execution – of arithmetic operations (e.g., `+.`) and memory read/write operations.

For ALU operations, the presented reliability specification uses the reliability of an unreliable multiplication operation from [31, Figure 9]. For memory operations, the specification uses the probability of a bit flip in a memory cell from [49, Figure 4] with extrapolation to the probability of a bit flip within a 32-bit word. Note that a memory region specification includes two reliabilities: the reliability of a read (`rd`) and the reliability of a write (`wr`).

Precondition Generator

For each function, Rely's analysis generates a *reliability precondition* that conservatively bounds the set of valid specifications for the function. A reliability precondition is a conjunction of predicates of the form $A_{out} \leq r \cdot \mathcal{R}(X)$, where A_{out} is a placeholder for a developer-provided reliability specification for an output with name *out*, r is a numerical value between 0 and 1, and the term $\mathcal{R}(X)$ is the joint reliability of the set of variables X on entry to the function.

The analysis starts at the end of the function from a postcondition that must be true when the function returns and then works backward to produce a precondition such that if the precondition holds before execution of the function, then the postcondition holds at the end of the function.

Postcondition. The postcondition for a function is the constraint that the reliability of each array argument exceeds that given in its specification. For `search_ref`, the postcondition Q_0 is

$$Q_0 = A_{\texttt{pblocks}} \leq \mathcal{R}(\texttt{pblocks}) \wedge A_{\texttt{cblock}} \leq \mathcal{R}(\texttt{cblock}),$$

which specifies that the reliability of the arrays `pblocks` and `cblock` – $\mathcal{R}(\texttt{pblocks})$ and $\mathcal{R}(\texttt{cblock})$ – should be at least that specified by the developer – $A_{\texttt{pblocks}}$ and $A_{\texttt{cblock}}$.

Precondition Generation. The analysis of the body of the `search_ref` function starts at the `return` statement. Given the postcondition Q_0, the analysis creates a new precondition Q_1 by conjoining to Q_0 a predicate that states that the reliability of the return value ($r_0 \cdot \mathcal{R}(\texttt{ minblock})$) is at least that of its specification (A_{ret}):

$$Q_1 = Q_0 \wedge A_{ret} \leq r_0 \cdot \mathcal{R}(\texttt{minblock}).$$

The reliability of the return value comes from my design principle for reliability approximation. Specifically, this reliability is the probability of correctly reading `minblock` from unreliable memory – which is $r_0 = 1 - 10^{-7}$ according to the hardware reliability specification – multiplied by $\mathcal{R}(\texttt{minblock})$, the probability that the preceding computation correctly computed and stored `minblock`.

Loops. The statement that precedes the `return` statement is the `repeat` statement on Line 14. A key difficulty with reasoning about the reliability of variables modified within a loop is that if a variable is updated unreliably and has a loop-carried dependence then its reliability monotonically decreases as a function of the number of loop iterations. Because the reliability of such variables can, in principle, decrease arbitrarily in an unbounded loop, Rely provides both an unbounded loop statement (with an associated analysis) and an alternative *bounded loop* statement that lets a developer specify a compile-time bound on the maximum number of its iterations that therefore bounds the reliability degradation of modified variables. The loop on Line 14 iterates `nblocks` times and therefore decreases the reliability of any modified variables `nblocks` times. Because the reliability degradation is bounded, Rely's analysis uses unrolling to reason about the effects of a bounded loop.

Conditionals. The analysis of the body of the loop on Line 14 encounters the `if` statement on Line 29.[1] This `if` statement uses an unreliable comparison operation on `ssd` and `minssd`, both of which reside in unreliable memory. The reliability of `minblock` when

[1]This happens after encountering the increment of `i` on Line 34, which does not modify the current precondition because it does not reference `i`.

(3) $\{Q_0 \wedge A_{ret} \leq r_0^4 \cdot \mathcal{R}(\texttt{i}, \texttt{ssd}, \texttt{minssd}) \wedge A_{ret} \leq r_0^4 \cdot \mathcal{R}(\texttt{ minblock}, \texttt{ssd}, \texttt{minssd})\}$

```
    if (ssd <.  minssd) {
```

(2) $\{Q_0 \wedge A_{ret} \leq r_0 \cdot \mathcal{R}(\texttt{i}, \ell_{29})\}$

```
      minssd = ssd;
```

$\{Q_0 \wedge A_{ret} \leq r_0 \cdot \mathcal{R}(\texttt{i}, \ell_{29})\}$

```
      minblock = i;
```

$\{Q_0 \wedge A_{ret} \leq r_0 \cdot \mathcal{R}(\texttt{minblock}, \ell_{29})\}$

```
    } else {
```

(2) $\{Q_0 \wedge A_{ret} \leq r_0 \cdot \mathcal{R}(\texttt{minblock}, \ell_{29})\}$

```
      skip;
```

$\{Q_0 \wedge A_{ret} \leq r_0 \cdot \mathcal{R}(\texttt{minblock}, \ell_{29})\}$

```
    }
```

(1) $\{Q_0 \wedge A_{ret} \leq r_0 \cdot \mathcal{R}(\texttt{minblock}, \ell_{29})\}$

Figure 4-6: `if` Statement Analysis in the Last Loop Iteration

modified on Line 31 therefore also depends on the reliability of this expression because faults may force the execution down a different path.

Figure 4-6 presents a Hoare logic style presentation of the analysis of the conditional statement. The analysis works in three steps; the preconditions generated by each step are numbered with the corresponding step.

Step 1. To capture the implicit dependence of a variable on an unreliable condition, Rely's analysis first uses latent *control flow variables* to make these dependencies explicit. A control flow variable is a unique program variable (one for each statement) that records whether the conditional evaluated to *true* or *false*. I denote the control flow variable for the `if` statement on Line 29 by ℓ_{29}.

To make the control flow dependence explicit, the analysis adds the control flow variable to all joint reliability terms in Q_1 that contain variables modified within the body of the `if` conditional (`minssd` and `minblock`).

Step 2. The analysis next recursively analyses both the "then" and "else" branches of the conditional, producing one precondition for each branch. As in a standard precondition generator (e.g., weakest-preconditions) the assignment of `i` to `minblock` in the "then" branch replaces `minblock` with `i` in the precondition. Because reads from `i` and writes to `minblock` are reliable (according to the specification) the analysis does not introduce any new r_0 factors.

Step 3. In the final step, the analysis leaves the scope of the conditional and conjoins the two preconditions for its branches after transforming them to include the direct dependence of the control flow variable on the reliability of the `if` statement's condition expression.

The reliability of the `if` statement's expression is greater than or equal to the product of 1) the reliability of the `<.` operator (r_0), 2) the reliability of reading both `ssd` and `minssd` from unreliable memory (r_0^2), and 3) the reliability of the computation that produced `ssd` and `minssd` ($\mathcal{R}(\texttt{ssd},\texttt{minssd})$). The analysis therefore transforms each predicate that contains the variable ℓ_{29}, by multiplying the right-hand side of the inequality with r_0^3 and replacing the variable ℓ_{29} with `ssd` and `minssd`. This produces the precondition Q_2:

$$Q_2 = Q_0 \wedge A_{ret} \leq r_0^4 \cdot \mathcal{R}(\texttt{i},\texttt{ssd},\texttt{minssd}) \wedge A_{ret} \leq r_0^4 \cdot \mathcal{R}(\texttt{minblock},\texttt{ssd},\texttt{minssd}).$$

Simplification. After unrolling a single iteration of the loop that begins at Line 14, the analysis produces $Q_0 \wedge A_{ret} \leq r_0^{2564} \cdot \mathcal{R}(\texttt{pblocks},\texttt{cblock},\texttt{i},\texttt{ssd},\texttt{minssd})$ as the precondition for a single iteration of the loop's body. The constant 2564 represents the number of unreliable operations within a single loop iteration.

Note that there is one less predicate in this precondition than in Q_2. As the analysis works backwards through the program, it uses a simplification technique that identifies that a predicate $A_{ret} \leq r_1 \cdot \mathcal{R}(X_1)$ *subsumes* another predicate $A_{ret} \leq r_2 \cdot \mathcal{R}(X_2)$. Specifically, the analysis identifies that $r_1 \leq r_2$ and $X_2 \subseteq X_1$, which together mean that the second predicate is a weaker constraint on A_{ret} than the first and can therefore be removed. This follows from the fact that the joint reliability of a set of variables is less than or equal to the joint reliability of any subset of the variables – regardless of the distribution of their values.

This simplification is how Rely's analysis achieves scalability when there are multiple paths in the program; specifically a simplified precondition characterizes the least reliable path(s) through the program.

Final Precondition. When the analysis reaches the beginning of the function after fully unrolling the loop on Line 14, it has a precondition that bounds the set of valid specifications as a function of the reliability of the parameters of the function. For `search_ref`, the analysis generates the precondition $A_{ret} \leq 0.994885 \cdot \mathcal{R}(\texttt{pblocks},\texttt{cblock}) \wedge A_{\texttt{pblocks}} \leq \mathcal{R}(\texttt{pblocks}) \wedge A_{\texttt{cblock}} \leq \mathcal{R}(\texttt{cblock})$.

Precondition Checker

The final precondition is a conjunction of predicates of the form $A_{out} \leq r \cdot \mathcal{R}(X)$, where A_{out} is a placeholder for the reliability specification of an output. Because reliability specifications are all of the form $r \cdot \mathcal{R}(X)$ (Figure 4-1), each predicate in the final precondition (where each A_{out} is replaced with its specification) is of the form form $r_1 \cdot \mathcal{R}(X_1) \leq r_2 \cdot \mathcal{R}(X_2)$, where $r_1 \cdot \mathcal{R}(X_1)$ is a reliability specification and $r_2 \cdot \mathcal{R}(X_2)$ is computed by the analysis. Similar to the analysis's simplifier (Section 4.2.4), the precondition checker verifies the validity of each predicate by checking that 1) r_1 is less than r_2 and 2) $X_2 \subseteq X_1$.

For `search_ref`, the analysis computes the predicates $0.99 \cdot \mathcal{R}(\texttt{pblocks}, \texttt{cblock}) \leq 0.994885 \cdot \mathcal{R}(\texttt{pblocks}, \texttt{cblock})$, $\mathcal{R}(\texttt{pblocks}) \leq \mathcal{R}(\texttt{ pblocks})$, and also $\mathcal{R}(\texttt{cblock}) \leq \mathcal{R}(\texttt{cblock})$. Because these predicates are valid according to the checking procedure, `search_ref` satisfies its reliability specification when executed.

4.3 Language Semantics

Because soft errors may probabilistically change the execution path of a program, I model the semantics of a Rely program with a probabilistic, non-deterministic transition system. Specifically, the dynamic semantics defines probabilistic transition rules for each arithmetic/logical operation and each read/write on an unreliable memory region.

Over the next several sections, I develop a small-step semantics that specifies the probability of each individual transition of an execution. In Section 4.3.6, I provide big-step definitions that specify the probability of an entire execution.

4.3.1 Preliminaries

Rely's semantics models an abstract machine that consists of a heap and a stack. The heap is an abstraction over the physical memory of the concrete machine, including its various reliable and unreliable memory regions. Each variable (both scalar and array) is allocated in the heap. The stack consists of frames – one for each function invocation – which contain references to the locations of each allocated variable. This conceptual model of

local variables does not need to be concretized in the compilation model. For example, placing local variables in a reliable stack can achieve competitive performance [54].

Hardware Reliability Specification. A hardware reliability specification $\psi \in \Psi = (iop + cmp + lop + M_{op}) \to \mathbb{R}$ is a finite map from arithmetic/logical operations (iop, cmp, lop) and *memory region operations* (M_{op}) to reliabilities (i.e., the probability that the operation executes correctly).

Arithmetic/logical operations *iop*, *cmp*, and *lop* include both reliable and unreliable versions of each integer, comparison, and logical operation. The reliability of each reliable operation is 1 and the reliability of an unreliable operation is as provided by a specification (Section 4.2.4).

The finite maps $rd \in M \to M_{op}$ and $wr \in M \to M_{op}$ define memory region operations as reads and writes (respectively) on memory regions $m \in M$, where M is the set of all memory regions in the reliability specification.

The hardware reliability specification 1_ψ denotes the specification for fully reliable hardware in which all arithmetic/logical and memory operations have reliability 1.

References. A *reference* is a tuple $\langle n_b, \langle n_1, \ldots, n_k \rangle, m \rangle \in \text{Ref}$ consisting of a base address $n_b \in \text{Loc}$, a dimension descriptor $\langle n_1, \ldots, n_k \rangle$, and a memory region m. The address space Loc is finite. A base address and the components of a dimension descriptor are machine integers $n \in \text{Int}_\text{M}$, which have finite bit width and therefore create a finite set.

References describe the location, dimensions, and memory region of variables in the heap. For scalars, the dimension descriptor is the single-dimension, single-element descriptor $\langle 1 \rangle$. The projections π_{base} and π_{dim} select the base address and the dimension descriptor of a reference, respectively.

Frames, Stacks, Heaps, and Environments. A *frame* $\sigma \in \Sigma = \text{Var} \to \text{Ref}$ is a finite map from variables to references. A *stack* $\delta \in \Delta ::= \sigma \mid \sigma :: \Delta$ is a non-empty list of frames. A *heap* $h \in H = \text{Loc} \to \text{Int}_\text{M}$ is a finite map from addresses to machine integers. An *environment* $\varepsilon \in \text{E} = \Delta \times H$ is a stack and heap pair, $\langle \delta, h \rangle$.

Memory Allocator. The abstract memory allocator *new* is a potentially non-deterministic partial function that executes reliably. It takes a heap h, a memory region m, and a dimension descriptor and returns a fresh address n_b that resides in memory region m and a new heap h' that reflects updates to the internal memory allocation data structures.

Auxiliary Probability Distributions. Each nondeterministic choice in Rely's semantics must have an underlying probability distribution so that the set of possible transitions at any given small step of an execution creates a probability distribution – i.e., the sum of the probabilities of each possibility is one. In Rely, there are two points at which an execution can make a nondeterministic choice: 1) the result of an incorrect execution of an unreliable operation and 2) the result of allocating a new variable in the heap.

The discrete probability distribution $P_f(n_f \mid op, n_1, ..., n_k)$ models the manifestation of a soft error during an incorrect execution of an operation. Specifically, it gives the probability that an incorrect execution of an operation op on operands $n_1, \ldots, n_k$ produces a value n_f that is different from the correct result of the operation. This distribution is inherently tied to the properties of the underlying hardware.

The discrete probability distribution $P_m(n_b, h' \mid h, m, d)$ models the semantics of a non-deterministic memory allocator. It gives the probability that a memory allocator returns a fresh address n_b and an updated heap h' given an initial heap h, a memory region m, and a dimension descriptor d.

I define these distributions only to support a precise formalization of the dynamic semantics of a program; they do not need to be specified for a given hardware platform or a given memory allocator to use Rely's reliability analysis.

4.3.2 Semantics of Expressions

Figure 4-7 presents a selection of the rules for the dynamic semantics of integer expressions. The labeled probabilistic small-step evaluation relation $\langle e, \sigma, h\rangle \xrightarrow{\theta, p}_{\psi} e'$ states that from a frame σ and a heap h, an expression e evaluates in one step with probability p to an expression e' given a hardware reliability specification ψ. The label $\theta \in \{\mathrm{C}, \langle \mathrm{C}, n\rangle, \langle \mathrm{F}, n_f\rangle\}$ denotes whether the transition corresponds to a correct (C or $\langle \mathrm{C}, n\rangle$) or a faulty ($\langle \mathrm{F}, n_f\rangle$)

$$\text{E-VAR-C}\quad \frac{\langle n_b, \langle 1\rangle, m\rangle = \sigma(x)}{\langle x, \sigma, h\rangle \xrightarrow{\mathrm{C},\, \psi(rd(m))}_{\psi} h(n_b)}$$

$$\text{E-VAR-F}\quad \frac{\langle n_b, \langle 1\rangle, m\rangle = \sigma(x) \qquad p = (1-\psi(rd(m)))\cdot P_f(n_f \mid rd(m), h(n_b))}{\langle x, \sigma, h\rangle \xrightarrow{\langle \mathrm{F}, n_f\rangle, p}_{\psi} n_f}$$

$$\text{E-IOP-R1}\quad \frac{\langle e_1, \sigma, h\rangle \xrightarrow{\theta, p}_{\psi} e_1'}{\langle e_1\ iop\ e_2, \sigma, h\rangle \xrightarrow{\theta, p}_{\psi} e_1'\ iop\ e_2} \qquad \text{E-IOP-R2}\quad \frac{\langle e_2, \sigma, h\rangle \xrightarrow{\theta, p}_{\psi} e_2'}{\langle n\ iop\ e_2, \sigma, h\rangle \xrightarrow{\theta, p}_{\psi} n\ iop\ e_2'}$$

$$\text{E-IOP-C}\quad \frac{}{\langle n_1\ iop\ n_2, \sigma, h\rangle \xrightarrow{\mathrm{C},\, \psi(iop)}_{\psi} iop(n_1, n_2)} \qquad \text{E-IOP-F}\quad \frac{p = (1-\psi(iop))\cdot P_f(n_f \mid iop, n_1, n_2)}{\langle n_1\ iop\ n_2, \sigma, h\rangle \xrightarrow{\langle \mathrm{F}, n_f\rangle, p}_{\psi} n_f}$$

Figure 4-7: Dynamic Semantics of Integer Expressions

evaluation of that step. For a correct transition $\langle \mathrm{C}, n\rangle$, $n \in \mathrm{Int_M}$ records a nondeterministic choice made for that step. For a faulty transition $\langle \mathrm{F}, n_f\rangle$, $n_f \in \mathrm{Int_M}$ represents the value that the fault introduced in the semantics of the operation.

To illustrate the meaning of the rules, consider the rules for variable reference expressions. A variable reference x reads the value stored in the memory address for x. The are two possibilities for the evaluation of a variable reference:

- Correct [E-VAR-C]. The variable reference evaluates correctly and successfully returns the integer stored in x. This happens with probability $\psi(rd(m))$, where m is the memory region in which x allocated. This probability is the reliability of reading from x's memory region.
- Faulty [E-VAR-F]. The variable reference experiences a fault and returns another integer n_f. The probability that the faulty execution returns a specific integer n_f is $(1-\psi(rd(m)))\cdot P_f(n_f \mid rd(m), h(n_b))$. P_f is the distribution that gives the probability that a failed memory read operation returns a value n_f instead of the true stored value $h(n_b)$ (Section 4.3.1).

4.3.3 Semantics of Statements

Figure 4-8 presents the scalar and control flow fragment of Rely. The labeled probabilistic small-step execution relation $\langle s, \varepsilon \rangle \xrightarrow{\theta, p}_{\psi} \langle s', \varepsilon' \rangle$ states that execution of the statement s in the environment ε takes one step yielding a statement s' and an environment ε' with probability p under the hardware reliability specification ψ. As in the dynamic semantics for expressions, a label θ denotes whether the transition evaluated correctly (C or $\langle \mathrm{C}, n \rangle$) or experienced a fault ($\langle \mathrm{F}, n_f \rangle$). The semantics of the statements in the language is largely similar to that of traditional presentations except that the statements have the ability to encounter faults during execution.

The semantics I present here is designed to allow unreliable computation at all points in the application – subject to the constraint that the application is still memory safe and exhibits control flow integrity.

Memory Safety. To protect references that point to memory locations from corruption, the stack is allocated in a reliable memory region and stack operations – i.e., pushing and popping frames – execute reliably (Section 4.3.5). To prevent out-of-bounds memory accesses that may occur due to an unreliable array index computation, Rely requires that each array read and write include a bounds check. These bounds check computations execute reliably (Section 4.3.4).

Control Flow Integrity. To prevent execution from taking control flow edges that do not exist in the program's static control flow graph, Rely assumes that 1) instructions are stored, fetched, and decoded reliably (as supported by existing unreliable processor architectures [84, 32]) and 2) targets of control flow branches are reliably computed. These two properties allow for the control flow transfers in the rules [E-IF-TRUE], [E-IF-FALSE], and [E-SEQ-R2] to execute reliably with probability 1.

Note that the semantics does not require a specific underlying mechanism to achieve reliable execution and, therefore, an implementation can use any applicable software or hardware technique [74, 69, 27, 34, 67, 86, 37, 90].

E-DECL-R

$$\frac{\langle e,\sigma,h\rangle \xrightarrow{\theta,p}_{\psi} e'}{\langle \texttt{int}\ x = e\ \texttt{in}\ m, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\theta,p}_{\psi} \langle \texttt{int}\ x = e'\ \texttt{in}\ m, \langle \sigma :: \delta, h\rangle\rangle}$$

E-DECL

$$\frac{\langle n_b, h'\rangle = new(h,m,\langle 1\rangle) \qquad p_m = P_m(n_b, h' \mid h,m,\langle 1\rangle)}{\langle \texttt{int}\ x = n\ \texttt{in}\ m, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\langle \mathrm{C}, n_b\rangle, p_m}_{\psi} \langle x = n, \langle \sigma[x \mapsto \langle n_b, \langle 1\rangle, m\rangle] :: \delta, h'\rangle\rangle}$$

E-ASSIGN-R

$$\frac{\langle e,\sigma,h\rangle \xrightarrow{\theta,p}_{\psi} e'}{\langle x = e, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\theta,p}_{\psi} \langle x = e', \langle \sigma :: \delta, h\rangle\rangle}$$

E-ASSIGN-C

$$\frac{\langle n_b, \langle 1\rangle, m\rangle = \sigma(x) \qquad p = \psi(wr(m))}{\langle x = n, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\mathrm{C},p}_{\psi} \langle \texttt{skip}, \langle \sigma :: \delta, h[n_b \mapsto n]\rangle\rangle}$$

E-ASSIGN-F

$$\frac{\langle n_b, \langle 1\rangle, m\rangle = \sigma(x) \qquad p = (1 - \psi(wr(m))) \cdot P_f(n_f \mid wr(m), h(n_b), n)}{\langle x = n, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\langle \mathrm{F}, n_f\rangle, p}_{\psi} \langle \texttt{skip}, \langle \sigma :: \delta, h[n_b \mapsto n_f]\rangle\rangle}$$

E-IF

$$\frac{\langle b,\sigma,h\rangle \xrightarrow{\theta,p}_{\psi} b'}{\langle \texttt{if}_\ell\ b\ s_1\ s_2, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\theta,p}_{\psi} \langle \texttt{if}_\ell\ b'\ s_1\ s_2, \langle \sigma :: \delta, h\rangle\rangle}$$

E-IF-TRUE

$$\frac{}{\langle \texttt{if}_\ell\ \mathit{true}\ s_1\ s_2, \varepsilon\rangle \xrightarrow{\mathrm{C},1}_{\psi} \langle s_1, \varepsilon\rangle}$$

E-IF-FALSE

$$\frac{}{\langle \texttt{if}_\ell\ \mathit{false}\ s_1\ s_2, \varepsilon\rangle \xrightarrow{\mathrm{C},1}_{\psi} \langle s_2, \varepsilon\rangle}$$

E-SEQ-R1

$$\frac{\langle s_1, \varepsilon\rangle \xrightarrow{\theta,p}_{\psi} \langle s_1', \varepsilon'\rangle}{\langle s_1\ ;\ s_2, \varepsilon\rangle \xrightarrow{\theta,p}_{\psi} \langle s_1'\ ;\ s_2, \varepsilon'\rangle}$$

E-SEQ-R2

$$\frac{}{\langle \texttt{skip}\ ;\ s_2, \varepsilon\rangle \xrightarrow{\mathrm{C},1}_{\psi} \langle s_2, \varepsilon\rangle}$$

E-WHILE

$$\frac{}{\langle \texttt{while}_\ell\ b\ s, \varepsilon\rangle \xrightarrow{\mathrm{C},1}_{\psi} \langle \texttt{if}_\ell\ b\ \{s\ ;\ \texttt{while}_\ell\ b\ s\}\ \{\texttt{skip}\}, \varepsilon\rangle}$$

E-WHILE-BOUNDED

$$\frac{}{\langle \texttt{while}_\ell\ b : n\ s, \varepsilon\rangle \xrightarrow{\mathrm{C},1}_{\psi} \langle \texttt{if}_\ell\ b\ \{s\ ;\ \texttt{while}_\ell\ b : (n-1)\ s\}\ \{\texttt{skip}\}, \varepsilon\rangle}$$

Figure 4-8: Dynamic Semantics of Statements

E-ARRAY-DECL-R

$$\frac{\langle e_i, \sigma\rangle \xrightarrow{\theta,p}_{\psi} e_i'}{\langle \texttt{int}\ a[n_1,\ldots,e_i,\ldots,e_k]\ \texttt{in}\ m, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\theta,p}_{\psi} \langle \texttt{int}\ a[n_1,\ldots,e_i',\ldots,e_k]\ \texttt{in}\ m, \langle \sigma :: \delta, h\rangle\rangle}$$

E-ARRAY-DECL

$$\frac{\forall i.0 < n_i \qquad \langle n_b, h'\rangle = new(h, m, \langle n_1,\ldots,n_k\rangle) \qquad \sigma' = \sigma[a \mapsto \langle n_b, \langle n_1,\ldots,n_k\rangle, m\rangle]}{\langle \texttt{int}\ a[n_1,\ldots,n_k]\ \texttt{in}\ m, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\mathrm{C},1}_{\psi} \langle \texttt{skip}, \langle \sigma' :: \delta, h'\rangle\rangle}$$

E-ARRAY-LOAD-IDX

$$\frac{\langle e_i, \sigma\rangle \xrightarrow{\theta,p}_{\psi} e_i'}{\langle x = a[n_1,\ldots,e_i,\ldots,e_k], \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\theta,p}_{\psi} \langle x = a[n_1,\ldots,e_i',\ldots,e_k], \langle \sigma :: \delta, h\rangle\rangle}$$

E-ARRAY-LOAD-C

$$\frac{\sigma(a) = \langle n_b, \langle l_1,\ldots,l_k\rangle, m\rangle \qquad n_o = l_k + \sum_{i=0}^{k-1} n_i \cdot l_i \qquad n = h(n_b + n_o)}{\langle x = a[n_1,\ldots,n_k], \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\mathrm{C},\psi(rd(m))}_{\psi} \langle x = n, \langle \sigma :: \delta, h\rangle\rangle}$$

E-ARRAY-LOAD-F

$$\frac{\sigma(a) = \langle n_b, \langle l_1,\ldots,l_k\rangle, m\rangle \qquad n_o = l_k + \sum_{i=0}^{k-1} n_i \cdot l_i \qquad p = (1 - \psi(rd(m))) * P(n_f \mid rd(m))}{\langle x = a[n_1,\ldots,n_k], \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\langle \mathrm{F}, n_f\rangle, p}_{\psi} \langle x = n_f, \langle \sigma :: \delta, h\rangle\rangle}$$

Figure 4-9: Dynamic Semantics of Arrays

E-CALLX-EXPR-ARG

$$\frac{\langle e_i, \sigma\rangle \xrightarrow{\theta,p}_{\psi} e_i'}{\langle x = f(v_1,\ldots,e_i,\ldots,e_k), \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\theta,p}_{\psi} \langle x = f(v_1,\ldots,e_i',\ldots,e_k), \langle \sigma :: \delta, h\rangle\rangle}$$

E-CALLX-INT-ARG

$$\frac{\langle e_i, \sigma\rangle \xrightarrow{\theta,p}_{\psi} n \qquad v_i = \langle \mathbb{N}, n\rangle}{\langle x = f(v_1,\ldots,e_i,\ldots,e_k), \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\theta,p}_{\psi} \langle x = f(v_1,\ldots,v_i\ldots,e_k), \langle \sigma :: \delta, h\rangle\rangle}$$

E-CALLX-ARR-ARG

$$\frac{v_i = \langle \text{Ref}, (\sigma(a_i))\rangle}{\langle x = f(v_1,\ldots,a_i,\ldots,e_k), \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\text{C},1}_{\psi} \langle x = f(v_1,\ldots,v_i,\ldots,e_k), \langle \sigma :: \delta, h\rangle\rangle}$$

E-CALLX-UNFOLD

$$\frac{\forall (a_i, i) \in \mathit{array\_params}(f)\ .\ \sigma'(a_i) = v_i \qquad s_{\mathit{int\_init}} \equiv \texttt{int}\ x_{i_1} = v_{i_1}\ \texttt{in}\ m_{i_1}\ ;\ \ldots\ ;\ \texttt{int}\ x_{i_k} = v_{i_k}\ \texttt{in}\ m_{i_k} \text{ where } (x_{i_j}, m_{i_j}, i_j) \in \mathit{int\_params}(f)}{\langle x = f(v_1,\ldots,v_k), \langle \delta, h\rangle\rangle \xrightarrow{\text{C},1}_{\psi} \langle x = f(v_1,\ldots,v_k)\ s_{\mathit{int\_init}}\ ;\ \mathit{code}(f)\ , \langle \sigma' :: \delta, h\rangle\rangle}$$

E-CALLX-BODY

$$\frac{\langle s, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\theta,p}_{\psi_f} \langle s', \langle \sigma' :: \delta', h'\rangle\rangle}{\langle x = f(v_1,\ldots,v_k)\ s, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\theta,p}_{\psi_c} \langle x = f(v_1,\ldots,v_k)\ \ s', \langle \sigma' :: \delta', h'\rangle\rangle}$$

E-RETURN-R

$$\frac{\langle e, \sigma\rangle \xrightarrow{\theta,p}_{\psi} \langle e', \sigma\rangle}{\langle \texttt{return}\ e, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\theta,p}_{\psi} \langle \texttt{return}\ e', \langle \sigma :: \delta, h\rangle\rangle}$$

E-CALLX-RETURN

$$\frac{}{\langle x = f(v_1,\ldots,v_k)\ \texttt{return}\ n, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\text{C},1}_{\psi} \langle x = n, \langle \delta, h\rangle\rangle}$$

E-SEQ-RETURN

$$\frac{s_1 \in \{\texttt{return}, \texttt{return}\ n\}}{\langle s_1\ ;\ s_2, \langle \delta, h\rangle\rangle \xrightarrow{\text{C},1}_{\psi} \langle s_1, \langle \delta, h\rangle\rangle}$$

Figure 4-10: Dynamic Semantics of Function Calls and Returns

4.3.4 Semantics of Arrays

Figure 4-9 presents the dynamic semantics of array operations.

Declarations. An array declaration allocates a new array in the heap. The boundaries of an array are given by a sequence of expressions, each of which can evaluate unreliably [E-ARRAY-DECL-R]. Given these boundaries, a new array is allocated using the function *new*, which returns the base address of an array that has been freshly allocated in the memory region m [E-ARRAY-DECL]. The function *new* executes reliably. To guard against ill-defined behavior, the given semantics also reliably checks that the length of each dimension is non-negative.

Loads. Executing an array load entails multiple steps. In the first step, the program reduces the expressions for each index in left-to-right order [E-ARRAY-LOAD-IDX]. Note that reduction of the index expressions may encounter faults, producing incorrect indices. Because the dynamic semantics incorporates bounds checks to prevent incorrect indices from yielding ill-defined, out-of-bounds behaviors, the developer is free to choose the reliability of the index expressions.

Given the array reference and the reduced indices of each dimension, the program then checks to see if each index is within bounds of the allocated dimensions of the array and also calculates the offset of the element to be accessed [E-ARRAY-LOAD-C, E-ARRAY-LOAD-F]. If the index check fails, the computation terminates. In the final step, the array load proceeds by attempting to fetch the corresponding value from the given memory region. With probability $\psi(rd(m))$, this step executes correctly and returns the value from memory [E-ARRAY-LOAD-C]. With probability $1 - \psi(rd(m))$, the memory read fails, producing an alternative value n_f with probability $P(n_f)$ [E-ARRAY-LOAD-F].

Stores. The semantics of stores are similar to that for loads except with the reliability of writes to the array's memory region ($\psi(wr(m))$) substituted for the reliability of reads.

Note that the store operation may fail in two ways: 1) the index computation produces a wrong index (the rule is similar to [E-ARRAY-LOAD-IDX]) or 2) the write operation

may fail, with probability $1 - \psi(wr(m))$. If the index computation fails and the computed index is outside of the loop bounds, then the computation skips the write operation. If the index computation fails and the computed index is within the bound, then the write will modify another location within the array. If the write operation fails, the semantics stores an alternative value n_f with probability $P(n_f)$.

4.3.5 Semantics of Functions

Figure 4-10 presents the dynamic semantics of function calls and returns. A function call and return sequence executes via the following procedure:

Call Argument Evaluation. A function call first evaluates its arguments in left-to-right order [E-CALLX-ARGS]. Note that evaluation of the arguments may encounter faults.

Call Body Unfolding. After the arguments to a call have been fully evaluated, control then transfers to the body of function [E-CALLX-UNFOLD]. The rule transfers control by fetching the code for the body of the function via the utility function $code(f)$.

The rule also creates a new state σ' and pushes the old state σ onto the program stack δ. The rule initializes the new state with the appropriate values for its parameters by prepending a set of declarations s_{int_init} to allocate and initialize the integer parameters of the function. The function *array_params* returns the set of names and position of the formal array parameters of the function f. The function *int_params* returns the set of names, memory locations, and positions of integer formal parameters of the function f.

Note that this rule executes correctly with probability 1; this implies that both control transfers and manipulations of the program stack are performed fully reliably. While reliable control transfers are given by Rely's machine model, reliable program stack manipulations require that a compiler allocate the program stack in a reliable memory region.

Return Value Evaluation. A `return` e statement fully evaluates e under the hardware reliability model, yielding a value n [E-RETURN-E].

Call Return Execution. Once the body of the call executes and reaches a `return` statement, execution proceeds by restoring the old state σ' from the program stack, assigning the return value to the destination variable, and transferring control back to the caller [E-CALL-RETURN-E]. Note that as with [E-CALLX-UNFOLD], this step executes correctly with probability 1 and therefore both the control transfer and stack manipulation are fully reliable.

4.3.6 Big-step Notations

I use the following big-step execution relations in the remainder of this chapter.

Definition 2 (Big-step Trace Semantics).

$$\langle s, \varepsilon \rangle \xrightarrow{\tau, p}_{\psi} \varepsilon' \equiv \langle s, \varepsilon \rangle \xrightarrow{\theta_1, p_1}_{\psi} \ldots \xrightarrow{\theta_n, p_n}_{\psi} \langle \texttt{skip}, \varepsilon' \rangle$$
$$\textit{where } \tau = \theta_1, \ldots, \theta_n \textit{ and } p = \prod_i p_i$$

The big-step trace semantics is a reflexive transitive closure of the small-step execution relation that records a trace of the execution. A *trace* $\tau \in \mathrm{T} ::= \cdot \mid \theta :: \mathrm{T}$ is a sequence of small-step transition labels. The *probability of a trace*, p, is the product of the probabilities of each transition.

Definition 3 (Big-step Aggregate Semantics).

$$\langle s, \varepsilon \rangle \xRightarrow{p}_{\psi} \varepsilon' \textit{ where } p = \sum_{\tau \in \mathrm{T}} p_\tau \textit{ such that } \langle s, \varepsilon \rangle \xrightarrow{\tau, p_\tau}_{\psi} \varepsilon'$$

The big-step aggregate semantics enumerates over the set of all finite length traces and collects the aggregate probability that a statement s evaluates to an environment ε' from an environment ε given a hardware reliability specification ψ. The big-step aggregate semantics therefore gives the total probability that a statement s starts from an environment ε and terminates in an environment ε'.[2]

[2]The inductive (versus co-inductive) interpretation of T yields a countable set of finite-length traces and therefore the sum over T is well-defined.

Termination and Errors. An unreliable execution of a statement may experience a run-time error (due to an out-of-bounds array access) or not terminate at all. The big-step aggregate semantics does not collect such executions. Therefore, the sum of the probabilities of the big-step transitions from an environment ε may not equal to 1. Specifically, let $p \in \mathrm{E} \to \mathbb{R}$ be a measure for the set of environments reachable from ε, i.e., $\forall \varepsilon'. \langle s, \varepsilon \rangle \overset{p(\varepsilon')}{\Longrightarrow}_{\psi} \varepsilon'$. Then p is *subprobability* measure, i.e., $0 \leq \sum_{\varepsilon' \in \mathrm{E}} p(\varepsilon') \leq 1$ [43].

4.4 Semantics of Quantitative Reliability

I next present definitions that give a semantic meaning to the reliability of a Rely program.

4.4.1 Paired Execution

The *paired execution* semantics is the primary execution relation that enables one to reason about the reliability of a program. Specifically, the relation pairs the semantics of the program when executed reliably with its semantics when executed unreliably.

Definition 4 (Paired Execution). $\varphi \in \Phi = \mathrm{E} \to \mathbb{R}$

$$\langle s, \langle \varepsilon, \varphi \rangle \rangle \Downarrow_{\psi} \langle \varepsilon', \varphi' \rangle \textit{ such that } \langle s, \varepsilon \rangle \overset{\tau, p_r}{\Longrightarrow}_{1_{\psi}} \varepsilon' \textit{ and } \varphi'(\varepsilon'_u) = \sum_{\varepsilon_u \in \mathrm{E}} \varphi(\varepsilon_u) \cdot p_u \textit{ where } \langle s, \varepsilon_u \rangle \overset{p_u}{\Longrightarrow}_{\psi} \varepsilon'_u$$

The relation states that from a *configuration* $\langle \varepsilon, \varphi \rangle$ consisting of an environment ε and an *unreliable environment distribution* φ, the paired execution of a statement s yields a new configuration $\langle \varepsilon', \varphi' \rangle$.

The environments ε and ε' are related by the fully reliable execution of s. Namely, an execution of s from an environment ε yields ε' under the fully reliable hardware model 1_{ψ}.

The unreliable environment distributions φ and φ' are probability mass functions that map an environment to the probability that the unreliable execution of the program is in that environment. In particular, φ is a distribution on environments before the unreliable execution of s whereas φ' is the distribution on environments after executing s. These distributions specify the probability of reaching a specific environment as a result of faults during the execution.

$$[\![P]\!] \in \mathcal{P}(\mathrm{E} \times \Phi) \qquad [\![\texttt{true}]\!] = \mathrm{E} \times \Phi \qquad [\![\texttt{false}]\!] = \varnothing \qquad [\![P_1 \wedge P_2]\!] = [\![P_1]\!] \cap [\![]\!] P_2$$

$$[\![R_1 \leq R_2]\!] = \{\langle \varepsilon, \varphi \rangle \mid [\![R_1]\!](\varepsilon, \varphi) \leq [\![R_2]\!](\varepsilon, \varphi)\}$$

$$[\![R]\!] \in \mathrm{E} \times \Phi \to \mathbb{R} \qquad [\![r]\!](\varepsilon, \varphi) = r \qquad [\![R_1 \cdot R_2]\!](\varepsilon, \varphi) = [\![R_1]\!](\varepsilon, \varphi) \cdot [\![R_2]\!](\varepsilon, \varphi)$$

$$[\![\mathcal{R}(X)]\!](\varepsilon, \varphi) = \sum_{\varepsilon_u \in \mathcal{E}(X, \varepsilon)} \varphi(\varepsilon_u) \qquad \mathcal{E} \in \mathcal{P}(\mathrm{Var} + \mathrm{ArrVar}) \times \mathrm{E} \to \mathcal{P}(\mathrm{E})$$

$$\mathcal{E}(X, \varepsilon) = \{\varepsilon' \mid \varepsilon' \in \mathrm{E} \wedge \forall v.\, v \in X \Rightarrow \mathrm{equiv}(\varepsilon', \varepsilon, v)\})$$

$$\mathrm{equiv}(\langle \sigma' :: \delta', h' \rangle, \langle \sigma :: \delta, h \rangle, v) = \forall i.\, 0 \leq i < \mathrm{len}(v, \sigma) \Rightarrow h'(\pi_{\mathrm{base}}(\sigma'(v)) + i) = h(\pi_{\mathrm{base}}(\sigma(v)) + i)$$

$$\mathrm{len}(v, \sigma) = \mathrm{let}\ \langle n_0, \ldots, n_k \rangle = \pi_{\mathrm{dim}}(\sigma(v))\ \mathrm{in} \prod_{0 \leq i \leq k} n_i$$

Figure 4-11: Predicate Semantics

The unreliable environment distributions are discrete because E is a countable set. Therefore, φ' can be defined pointwise: for any environment $\varepsilon'_u \in \mathrm{E}$, the value of $\varphi'(\varepsilon'_u)$ is the probability that the unreliable execution of the statement s results in the environment ε'_u given the distribution on possible starting environments, φ, and the aggregate probability p_u of reaching ε'_u from any starting environment $\varepsilon_u \in \mathrm{E}$ according to the big-step aggregate semantics. In general, φ' is a subprobability measure because it is defined using the big-step aggregate semantics, which is also a subprobability measure (Section 4.3.6).

4.4.2 Reliability Predicates and Transformers

The paired execution semantics enables a definition of the semantics of statements as transformers on *reliability predicates* that bound the reliability of program variables. A reliability predicate P is a predicate of the form:

$$\begin{array}{lcl} P & \to & \texttt{true} \mid \texttt{false} \mid R \leq R \mid P \wedge P \\ R & \to & r \mid \mathcal{R}(X) \mid R \cdot R \end{array}$$

A predicate can either be the constant `true`, the constant `false`, a comparison between *reliability factors* (R), or a conjunction of predicates. A reliability factor is real-valued quantity that is either a constant r in the range $[0, 1]$; a joint reliability factor $\mathcal{R}(X)$ that gives the probability that all program variables in the set X have the same value in the unreliable execution as they have in the reliable execution; or a product of reliability factors, $R \cdot R$.

This combination of predicates and reliability factors enables a developer to specify bounds on the reliability of variables in the program, such as $0.99999 \leq \mathcal{R}(\{x\})$, which states that the probability that x has the correct value in an unreliable execution is at least 0.99999.

Semantics of Reliability Predicates.

Figure 4-11 presents the denotational semantics of reliability predicates via the semantic function $[\![P]\!]$. The denotation of a reliability predicate is the set configurations that satisfy the predicate. A key new element in the semantics of this predicate language is the semantics of joint reliability factors.

Joint Reliability Factor. A joint reliability factor $\mathcal{R}(X)$ represents the probability that an unreliable environment ε_u sampled from the unreliable environment distribution φ has the same values for all variables in the set X as that in the reliable environment ε. To define this probability, I use the function $\mathcal{E}(X, \varepsilon)$, which gives the set of environments that have the same values for all variables in X as in the environment ε. The denotation of a joint reliability factor is then the sum of the probabilities of each of these environments according to φ.

Auxiliary Definitions. I define predicate satisfaction and validity as follows:

$$\begin{aligned} \langle \varepsilon, \varphi \rangle \models P &\equiv \langle \varepsilon, \varphi \rangle \in [\![P]\!] \\ \models P &\equiv \forall \varepsilon. \forall \varphi. \langle \varepsilon, \varphi \rangle \models P \end{aligned}$$

Reliability Transformer

Given a semantics for predicates, it is now possible to view the paired execution of a program as a *reliability transformer* – namely, a transformer on reliability predicates that is reminiscent of Dijkstra's Predicate Transformer Semantics [30].

Definition 5 (Reliability Transformer).

$$\psi \models \{P\}\ s\ \{Q\} \equiv$$
$$\forall \varepsilon. \forall \varphi. \forall \varepsilon'. \forall \varphi'. (\langle \varepsilon, \varphi \rangle \models P \wedge \langle s, \langle \varepsilon, \varphi \rangle \rangle \Downarrow_{\psi} \langle \varepsilon', \varphi' \rangle) \Rightarrow \langle \varepsilon', \varphi' \rangle \models Q$$

The paired execution of a statement s is a transformer on reliability predicates, denoted $\psi \models \{P\}\ s\ \{Q\}$. Specifically, the paired execution of s *transforms* P to Q if for all $\langle \varepsilon, \varphi \rangle$ that satisfy P and for all $\langle \varepsilon', \varphi' \rangle$ yielded by the paired execution of s from $\langle \varepsilon, \varphi \rangle$, $\langle \varepsilon', \varphi' \rangle$ satisfies Q. The paired execution of s transforms P to Q for any P and Q where this relationship holds.

Reliability predicates and reliability transformers enable Rely to use symbolic predicates to characterize and constrain the shape of the unreliable environment distributions before and after execution of a statement. This approach provides a well-defined domain in which to express Rely's reliability analysis as a generator of constraints on the shape of the unreliable environment distributions for which a function still satisfies its specification.

4.5 Reliability Analysis

For each function in a program, Rely's reliability analysis generates a symbolic *reliability precondition* with a precondition generator style analysis. The reliability precondition is a reliability predicate that constrains the set of specifications that are valid for the function. Specifically, the reliability precondition is of the form $\bigwedge_{i,j} R_i \leq R_j$ where R_i is the reliability factor for a developer-provided specification of a function output and R_j is a reliability factor that gives a conservative lower bound on the reliability of that output. If the reliability precondition is valid, then the developer-provided specifications are valid for the function.

4.5.1 Preliminaries

Transformed Semantics. I formalize Rely's analysis over a transformed semantics of the program that is produced via a source-to-source transformation function $\mathcal{T}$ that performs two transformations:

- **Conditional Flattening.** Each conditional has a unique *control flow variable* ℓ associated with it that $\mathcal{T}$ uses to flatten a conditional of the form $\texttt{if}_\ell\ (b)\ \{s_1\}\ \{s_2\}$ to the sequence $\ell = b\ ;\ \texttt{if}_\ell\ (\ell)\ \{s_1\}\ \{s_2\}$. This transformation reifies the control flow variable as an explicit program variable that records the value of the conditional.

- **SSA.** The transformation function also transforms a Rely program to a SSA renamed version of the program. The ϕ-nodes for a conditional include a reference to the control flow variable for the conditional. For example, $\mathcal{T}$ transforms a sequence of statements of the form $\ell = b$; $\texttt{if}_\ell\ (\ell)\ \{x = 1\}\ \{x = 2\}$ to the sequence of statements $\ell = b$; $\texttt{if}_\ell\ (\ell)\ \{x_1 = 1\}\ \{x_2 = 2\}$; $x = \phi(\ell, x_1, x_2)$. I rely on standard treatments for the semantics of ϕ-nodes [9] and arrays [42].

  I also note that $\mathcal{T}$ applies the SSA transformation such that a reference of a parameter at any point in the body of the function refers to its initial value on entry to the function. This property naturally gives a function's reliability specifications a semantics that refers to the reliability of variables on entry to the function.

These two transformations together make explicit the dependence between the reliability of a conditional's control flow variable and the reliability of variables modified within.

Auxiliary Maps. The map $\Lambda \in \text{Var} \to M$ is a map from program variables to their declared memory regions. I compute this map by inspecting the parameter and variable declarations in the function. The map $\Gamma \in \text{Var} \to R$ is a unique map from the outputs of a function – namely, the return value and arrays passed as parameters – to the reliability factors (Section 4.4.2) for the developer-provided specification of each output. I allocate a fresh variable named ret that represents the return value of the program.

Substitution. A substitution $e_0[e_2/e_1]$ replaces all occurrences of the expression e_1 with the expression e_2 within the expression e_0. Multiple substitution operations are applied from left to right. The substitution matches set patterns. For instance, the pattern $\mathcal{R}(\{x\} \cup X)$ represents a joint reliability factor that contains the variable x, alongside with the remaining variables in the set X. Then, the result of the substitution $r_1 \cdot \mathcal{R}(\{x,z\})[r_2 \cdot \mathcal{R}(\{y\} \cup X)/\mathcal{R}(\{x\} \cup X)]$ is the expression $r_1 \cdot r_2 \cdot \mathcal{R}(\{y,z\})$.

4.5.2 Precondition Generation

The analysis generates preconditions according to a conservative approximation of the paired execution semantics. Specifically, it characterizes the reliability of a value in a

$$\rho \in (Exp + BExp) \to \mathbb{R} \times \mathcal{P}(\text{Var}) \qquad \rho(n) = (1, \varnothing) \qquad \rho(x) = (\psi(rd(\Lambda(x))), \{x\})$$

$$\rho(e_1\ iop\ e_2) = (\rho_1(e_1) \cdot \rho_1(e_2) \cdot \psi(iop),\ \rho_2(e_1) \cup \rho_2(e_2)) \qquad \rho_1(e) = \pi_1(\rho(e))$$

$$\rho_2(e) = \pi_2(\rho(e))$$

$$\begin{aligned}
RP_\psi &\in S \times P \to P \\
RP_\psi(\texttt{return}\ e, Q) &= Q \wedge \Gamma(\text{ret}) \leq \rho_1(e) \cdot \mathcal{R}(\rho_2(e)) \\
RP_\psi(x = e, Q) &= Q\,[(\rho_1(e) \cdot \psi(wr(\Lambda(x))) \cdot \\
&\qquad \mathcal{R}(\rho_2(e) \cup X))/\mathcal{R}(\{x\} \cup X)] \\
RP_\psi(x = a[e_1, \ldots, e_n], Q) &= Q\,[((\textstyle\prod_i \rho_1(e_i)) \cdot \psi(rd(\Lambda(a))) \cdot \psi(wr(\Lambda(x))) \cdot \\
&\qquad \mathcal{R}(\{a\} \cup (\textstyle\bigcup_i \rho_2(e_i)) \cup X))/\mathcal{R}(\{x\} \cup X)] \\
RP_\psi(a[e_1, \ldots, e_n] = e, Q) &= Q\,[(\rho_1(e) \cdot (\textstyle\prod_i \rho_1(e_i)) \cdot \psi(wr(\Lambda(a))) \cdot \\
&\qquad \mathcal{R}(\rho_2(e) \cup (\textstyle\bigcup_i \rho_2(e_i)) \cup \{a\} \cup X))/\mathcal{R}(\{a\} \cup X)] \\
RP_\psi(\texttt{skip}, Q) &= Q \\
RP_\psi(s_1\ ;\ s_2, Q) &= RP_\psi(s_1, RP_\psi(s_2, Q)) \\
RP_\psi(\texttt{if}_\ell\ \ell\ s_1\ s_2, Q) &= RP_\psi(s_1, Q) \wedge RP_\psi(s_2, Q) \\
RP_\psi(x = \phi(\ell, x_1, x_2), Q) &= Q\,[\mathcal{R}(\{\ell, x_1\} \cup X)/\mathcal{R}(\{x\} \cup X)] \wedge \\
&\qquad Q[\mathcal{R}(\{\ell, x_2\} \cup X)/\mathcal{R}(\{x\} \cup X)] \\
\\
RP_\psi(\texttt{while}_\ell\ b : 0\ s, Q) &= Q \\
RP_\psi(\texttt{while}_\ell\ b : n\ s, Q) &= RP_\psi(\mathcal{T}(\texttt{if}_{\ell_n}\ b\ \{s\ ;\ \texttt{while}_\ell\ b : (n-1)\ s\}\ \texttt{skip}), Q) \\
\\
RP_\psi(\texttt{int}\ x = e\ \texttt{in}\ m, Q) &= RP_\psi(x = e, Q) \\
RP_\psi(\texttt{int}\ a[n_0, \ldots, n_k]\ \texttt{in}\ m, Q) &= Q\,[\mathcal{R}(X)/\mathcal{R}(\{a\} \cup X)]
\end{aligned}$$

Figure 4-12: Reliability Precondition Generation

function according to the probability that the function computes that value – including its dependencies – fully reliably given a hardware specification.

Figure 4-12 presents a selection of Rely's reliability precondition generation rules. The generator takes as input a statement s, a postcondition Q, and (implicitly) the maps Λ and Γ. The generator produces as output a precondition P, such that if P holds before the paired execution of s, then Q holds after.

I have designed the analysis so that Q is the constraint over the developer-provided specifications that must hold at the end of execution of a function. Because arrays are passed by reference in Rely and can therefore be modified, one property that must hold at

the end of execution of a function is that each array must be at least as reliable as implied by its specification. The analysis captures this property by setting the initial Q for the body of a function to

$$\bigwedge_{a_i} \Gamma(a_i) \leq \mathcal{R}(a_i')$$

where a_i is the ith array parameter of the function and a_i' is an SSA renamed version of the array that contains the appropriate value of a_i at the end of the function. This constraint therefore states that the reliability implied by the specifications must be less than or equal to the actual reliability of each input array at the end of the function. As the precondition generator works backwards through the function, it generates a new precondition that – if valid at the beginning of the function – ensures that Q holds at the end.

Reasoning about Expressions

The topmost part of Figure 4-12 first presents the rules for reasoning about the reliability of evaluating an expression. The reliability of evaluating an expression depends on two factors: 1) the reliability of the operations in the expression and 2) the reliability of the variables referenced in the expression. The function $\rho \in (Exp + BExp) \rightarrow \mathbb{R} \times \mathcal{P}(\text{Var})$ computes the core components of these two factors. It returns a pair consisting of 1) the probability of correctly executing all operations in the expression and 2) the set of variables referenced by the expression. The projections ρ_1 and ρ_2 return each component, respectively. Using these projections, the reliability of an expression e – given any reliable environment and unreliable environment distribution – is therefore *at least* $\rho_1(e) \cdot \mathcal{R}(\rho_2(e))$, where $\mathcal{R}(\rho_2(e))$ is the joint reliability of all the variables referenced in e. The rules for boolean and relational operations are defined analogously.

Generation Rules for Statements

As in a precondition generator, the analysis works backwards from the end of the program to the beginning. I have therefore structured the discussion of the statements starting with function returns.

Function Returns. When execution reaches a function return, `return` e, the analysis must verify that the reliability of the return value is greater than the reliability that the developer specified. To verify this, the analysis rule generates the additional constraint $\Gamma(\text{ret}) \leq \rho_1(e) \cdot \mathcal{R}(\rho_2(e))$. This constrains the reliability of the return value, where $\Gamma(\text{ret})$ is the reliability specification for the return value.

Assignment. For the program to satisfy a predicate Q after the execution of an assignment statement $x = e$, then Q must hold given a substitution of the reliability of the expression e for the reliability of x. The substitution $Q[(\rho_1(e) \cdot \psi(wr(\Lambda(x))) \cdot \mathcal{R}(\rho_2(e) \cup X))/\mathcal{R}(\{x\} \cup X)]$ binds each reliability factor in which x occurs – $\mathcal{R}(\{x\} \cup X)$ – and replaces the factor with a new reliability factor $\mathcal{R}(\rho_2(e) \cup X)$ where $\rho_2(e)$ is the set of variables referenced by e.

The substitution also multiplies the reliability factor by $\rho_1(e) \cdot \psi(wr(\Lambda(x)))$, which is the probability that e evaluates fully reliably and its value is reliably written to the memory location for x.

Array loads and stores. The reliability of a load statement $x = a[e_1, \ldots, e_n]$ depends on the reliability of the indices $e_1, \ldots, e_n$, the reliability of the values stored in a, and the reliability of reading from a's memory region. The rule's implementation is similar to that for assignment.

The reliability of an array store $a[e_1, \ldots, e_n] = e$ depends on the reliability of the source expression e, the reliability of the indices $e_1, \ldots, e_n$, and the reliability of writing to a. Note that the rule preserves the presence of a within the reliability term. By doing so, the rule ensures that it tracks the full reliability of all the elements within a.

Conditional. For the program to satisfy a predicate Q after a conditional statement of the form $\texttt{if}_\ell\ b\ s_1\ s_2$, each branch must satisfy Q. The rule therefore generates a precondition that is a conjunction of the results of the analysis of each branch.

Phi-nodes. The rule for a ϕ-node $x = \phi(\ell, x_1, x_2)$ captures the implicit dependence of the effects of control flow on the value of a variable x. For the merged value x, the rule estab-

$$
\begin{aligned}
RP_{\psi}(x = f(e_1,\ldots,e_n),Q,C) &= WPI(f,array\_par\_num(f),Q,C)\\
\\
WPI(f,0,Q,C) &= Q\,[\psi(wr(\Lambda(x))\cdot Relspec\_act(f,0,X\cup C)/\mathcal{R}(\{x\}\cup X)]\\
WPI(f,j,Q,C) &= \text{let } Q' = WPI(f,j-1,Q,C) \text{ in}\\
&\quad Q'\,[Relspec\_act(f,array\_par(j),X)/\mathcal{R}(\{array\_par(j)\}\cup X)]\ \wedge\ Q'\\
\\
Relspec\_act(f,a,X) &= Relspec\_form(f,a)\\
&\quad [\rho_1(act\_par(f,1))\mathcal{R}(\rho_2(act\_par(f,1))\cup X\cup Y)/\mathcal{R}(\{form\_par(f,1)\}\cup Y)]\\
&\quad [\rho_1(act\_par(f,i))\mathcal{R}(\rho_2(act\_par(f,i))\cup X\cup Y)/\mathcal{R}(\{form\_par(f,i)\}\cup Y)]\\
&\quad [\rho_1(act\_par(f,n))\mathcal{R}(\rho_2(act\_par(f,n))\cup X\cup Y)/\mathcal{R}(\{form\_par(f,n)\}\cup Y)]
\end{aligned}
$$

Figure 4-13: Constraint Generation for Function Calls

lishes Q by generating a precondition that ensures that Q holds independently for both x_1 and x_2, given an appropriate substitution. Note that the rule also includes ℓ in the substitution; this explicitly captures x's dependence on ℓ. The flattening statement inserted before a conditional (Section 4.5.1), later replaces the reliability of ℓ with that of its dependencies.

Bounded `while` and `repeat`. Bounded `while` loops, $\texttt{while}_\ell\ b : n\ s$, and `repeat` loops, $\texttt{repeat}\ n\ s$, execute their bodies at most n times. Execution of such a loop therefore satisfies Q if P holds beforehand, where P is the result of invoking the analysis on n sequential copies of the body. The rule implements this approach via a sequence of bounded recursive calls to transformed versions of itself.

Unbounded `while`. I present the analysis for unbounded `while` loops in the section that follows.

Function calls. Figure 4-13 presents the constraint generation rule for function calls. The analysis for functions is modular and takes the reliability specification from the function declaration and substitutes the reliabilities of the function's formal arguments with the reliabilities of the expressions that represent the function's actual arguments.

Specifically, for a function call $x = f(e_1,\ldots,e_n)$ the constraint generator performs two tasks. First, it substitutes the declared reliabilities of the function's parameters with the reliability expressions for the actual parameters of the function known at the call site. Second, to update the reliability expressions for multiple modified variables (i.e., the modified array variables), it constructs a constraint that abstracts the reliability of a function call as

the reliability of multiple assignment statements – one statement represents the assignment of the final value of the function to the variable x, the remaining statements represent the assignment of each potentially modified array parameter of the function.

The constraint generator uses several helper functions to specify the constraint transformation. The helper function *Relspec_form*(f, j) obtains the *declared reliability specification* of the return value of the function if $j = 0$ or the j-th array parameter if $j \geq 1$. The total number of array parameters can be obtained using the function *fpar_arrvarnum*(f). The total number of parameters of the function (array and scalar) is n.

The function *Relspec_act*(f, j, X) returns the *actual* reliability of the corresponding parameter – it takes the declared reliability and substitutes the names of all n formal arguments of the function with the reliability expressions of the corresponding actual arguments. The function *act_par*(f, i) returns the reliability of the expression for the i-th actual parameter of the function f. The function *form_par*(f, i) returns the name of the i-th parameter of the function f.

The function *WPI*(f, j, Q, C) constructs the new reliability constraint for a function call with a scalar return value and multiple array parameters. If $j = 0$ the function constructs the reliability constraint for a scalar assignment of the return vale. It substitutes the variable x with an expression that denotes the reliability of the function's return value.

If $j > 0$, the function constructs the constraint for each array parameter. The function *array_par_num*(f) returns the number of the array parameters of a function f. The function *array_par*(j) returns the name of the j-th array parameter. The function creates the new constraint in a similar way as the array store statement. Specifically, it constructs a predicate that is a conjunction of two predicates: 1) a predicate that characterizes the reliability of the array when the call decreases the reliability of the array and 2) a predicate that characterizes the reliability of the array when the array already has a reliability lower than the one specified by the function's reliability specification.

Note also that this modular approach supports reasoning about recursion. When analyzing a function, if the analysis assumes that the specification of a recursive invocation is valid, then the result of the recursive call is no more reliable then the specification the analysis is trying to verify. If there is any unreliable computation on that result, then it is

less reliable than the specification and therefore cannot be verified unless the given specification is zero. This is consistent with the analysis of unbounded `while` loops, which I present in the following section.

Unbounded `while` Loops.

An unbounded loop, $\texttt{while}_\ell\ b\ s$, may execute for a number of iterations that is not bounded statically. The reliability of a variable that is modified unreliably within a loop and has a loop-carried dependence is a monotonically decreasing function of the number of loop iterations. The only sound approximation of the reliability of such a variable is therefore zero. However, unbounded loops may also update a variable reliably. In this case, the reliability of the variable is the joint reliability of its dependencies. I have designed an analysis for unbounded `while` loops to distinguish these two cases as follows:

Dependence Graph. The analysis first constructs a dependence graph for the loop. Each node in the dependence graph corresponds to a variable that is read or written within the condition or body of the loop. There is a directed edge from the node for a variable x to the node for a variable y if the value of y depends on the value of x. The analysis additionally classifies each edge as reliable or unreliable meaning that a reliable or unreliable operation creates the dependence.

There is an edge from the node for a variable x to the node for the variable y if one of the following holds:

- **Assignment:** there is an assignment to y where x occurs in the expression on the right hand side of the assignment; this condition captures direct data dependencies. The analysis classifies such an edge as reliable if every operation in the assignment (i.e., the operations in the expression and the write to memory itself) are reliable. Otherwise, the analysis marks the edge as unreliable. The rules for array load and store statements are similar, and include dependencies induced by the computation of array indices.
- **Control Flow Side Effects:** y is assigned within an `if` statement and the `if` statement's control flow variable is named x; this condition captures control dependencies. The analysis classifies each such edge as reliable.

The analysis uses the dependence graph to identify the set of variables in the loop that are *reliably updated.* A variable x is reliably updated if all simple paths (and simple cycles) to x in the dependence graph contain only reliable edges.

Fixpoint Analysis. Given a set of reliably updated variables X_r, the analysis next splits the postcondition Q into two parts. For each predicate $R_i \leq r \cdot \mathcal{R}(X)$ in Q (where R_i is a developer-provided specification), the analysis checks if the property $\forall x \in X . x \in \text{modset}(s)$ $\Rightarrow x \in X_r$ holds, where $\text{modset}(s)$ computes the set of variables that may be modified by s. If this holds, then all the variables in X are either modified reliably or not modified at all within the body of the loop. The analysis conjoins the set of predicates that satisfy this property to create the postcondition Q_r and conjoins the remaining predicates to create Q_u.

The analysis next iterates the function $F(A)$ starting from `true`, where $F(A) = Q_r \wedge$ $RP_\psi(\mathcal{T}(\texttt{if}_\ell\ b\ s\ \texttt{skip}), A)$, until it reaches a fixpoint. The resulting predicate Q_r' is a translation of Q_r such the joint reliability of a set of variables is replaced by the joint reliability of its dependencies.

Lemma 19 (Termination). *Iteration of $F(A)$ terminates.*

This follows by induction on iterations, the monotonicity of RP and the fact that the range of $F(A)$ (given a simplifier that removes redundant predicates, which I present in the following section) is finite (together, finite descending chains). The key intuition is that the set of real-valued constants in the precondition before and after an iteration does not change (because all variables are reliably updated) and the set of variables that can occur in a joint reliability factor is finite. Therefore, there are a finite number of unique preconditions in the range of $F(A)$.

Final Precondition. In the last step, the analysis produces a final precondition that preserves the reliability of variables that are reliably updated by conjoining Q_r' with the predicate $Q_u[(R_i \leq 0)/(R_i \leq R_j)]$, where R_i and R_j are joint reliability factors. The substitution on Q_u sets the joint reliability factors that contain unreliably updated variables to zero.

Properties

Rely's analysis is sound with respect to the transformer semantics presented in Section 4.4.

Theorem 20 (Soundness). $\psi \models \{RP_{\psi}(s, Q)\}\ s\ \{Q\}$

This theorem states that if a configuration $\langle \varepsilon, \varphi \rangle$ satisfies a generated precondition and the paired execution of s yields a configuration $\langle \varepsilon', \varphi' \rangle$, then $\langle \varepsilon', \varphi' \rangle$ satisfies Q. Alternatively, s transforms the precondition generated by the analysis to Q.

4.5.3 Specification Checking

As the last step of the analysis for a function, the analysis checks the developer-provided reliability specifications for the function's outputs as captured by the precondition generator's final precondition. Because each specification has the form $r \cdot \mathcal{R}(X)$ (Figure 4-1) the precondition is a conjunction of predicates of the form $r_1 \cdot \mathcal{R}(X_1) \leq r_2 \cdot \mathcal{R}(X_2)$. While these joint reliability factors represent arbitrary and potentially complex distributions of the values of X_1 and X_2, there is simple and sound (though not complete) procedure to check the validity of each predicate in a precondition that follows from the *ordering* of joint reliability factors.

Proposition 1 (Ordering). *For two sets of variables X and Y, if $X \subseteq Y$ then $\mathcal{R}(Y) \leq \mathcal{R}(X)$.*

This follows from the fact that the joint reliability of a set of variables Y is less than or equal to the joint reliability of any subset of the variables – regardless of the distribution of their values. As a consequence of the ordering of joint reliability factors, there is a simple and sound method to check the validity of a predicate.

Corollary 21 (Predicate Validity). *If $r_1 \leq r_2$ and $X_2 \subseteq X_1$ then $\models r_1 \cdot \mathcal{R}(X_1) \leq r_2 \cdot \mathcal{R}(X_2)$.*

The constraint $r_1 \leq r_2$ is a comparison of two real numbers and the constraint $X_2 \subseteq X_1$ is an inclusion of finite sets. Note that both types of constraints are decidable and efficiently checkable.

Checking. Because the predicates in the precondition generator's output are mutually independent, it is possible to use Corollary 21 to check the validity of the full precondition by checking the validity of each predicate in turn.

4.5.4 Implementation

The parser for the Rely language, the precondition generator, and the precondition checker are implemented in OCaml. The implementation consists of 2500 lines of code. The analysis can operate on numerical or symbolic hardware reliability specifications. The implementation performs simplification transformations after each precondition generator step to simplify numerical expressions and remove predicates that are trivially valid or *subsumed* by another predicate.

Proposition 2 (Predicate Subsumption). *A reliability predicate* $r_1 \cdot \mathcal{R}(X_1) \leq r_2 \cdot \mathcal{R}(X_2)$ *subsumes (i.e., soundly replaces) a predicate* $r_1' \cdot \mathcal{R}(X_1') \leq r_2' \cdot \mathcal{R}(X_2')$ *if* $r_1' \cdot \mathcal{R}(X_1') \leq r_1 \cdot \mathcal{R}(X_1)$ *and* $r_2 \cdot \mathcal{R}(X_2) \leq r_2' \cdot \mathcal{R}(X_2')$

This property follows directly from the ordering of joint reliability factors.

4.6 Case Studies

I next discuss six computations (three checkable, three approximate) that I implemented and analyzed with Rely.

4.6.1 Benchmarks

Newton's Method. Figure 4-14 presents an unreliable Rely implementation of Newton's method. Newton's method searches for a root of a function; given a differentiable function $f(x)$, its derivative $f'(x)$, and a starting point x_s, it computes a value x_0 such that $f(x_0) = 0$.

This is an example of a fixed-point iteration computation that executes a `while` loop at most `maxstep` steps. The computation within each loop iteration can execute unreliably: each iteration updates the estimate of the root x by computing the value of the

```
1  #define tolerance 0.000001
2  #define maxsteps 40
3
4  float<0.9999*R(x)> F(float x in urel);
5  float<0.9999*R(x)> dF(float x in urel);
6
7  float <0.99*R(xs)> newton(float xs in urel){
8     float x, xprim in urel;
9     float t1, t2 in urel;
10
11    x = xs;
12    xprim = xs +. 2*.tolerance;
13
14    while ((x -. xprim >=. tolerance)
15           ||. (x -. xprim <=. -tolerance)
16          ) : maxsteps {
17       xprim = x;
18       t1 = F(x);
19       t2 = dF(x);
20       x = x -. t1 /. t2;
21    }
22
23    if (!.((x -. xprim <=. tolerance)
24         &&. (x -. xprim >=. -tolerance))) {
25        x = INFTY;
26    }
27    return x;
28 }
```

Figure 4-14: Newton's Method Implementation

function f and the derivative f'. If the computation converges in the maximum number of steps, the function returns the produced value. Otherwise it returns the error value (infinity). The reliability of the computation depends on the reliability of the starting value x_s and the reliability of the functions f and f'. If the reliability specification of f is `float<0.9999*R(x)> F(float x)` (and similar for f'), then the analysis verifies that the reliability of the whole computation is at least `0.99*R(xs)`.

Secant Method. Figure 4-15 presents an implementation of this computation; the computation also searches for a root of a function f. It takes as input two points x_a and x_b for which the function has the opposite sign and returns the value x_0 within the interval $[x_a, x_b]$ for which the function f evaluates to zero.

This is the fixed point computation that at each step computes the middle point x_c and its function $f(x_c)$. Based on the sign of $f(x_c)$ the algorithm divides the search interval

```
1  #define tolerance 0.000001
2  #define maxsteps 40
3
4  float<0.9999*R(x)> F(float x in unrel);
5
6  float<0.9999*R(x)> dF(float x in unrel);
7
8  float <0.995*R(xa, xb)> secant
9    (float xa in unrel, float xb in unrel) {
10    float a in unrel;
11    float b in unrel;
12    float c in unrel;
13    float fa in unrel;
14    float fb in unrel;
15    float fc in unrel;
16    bool converged in unrel;
17
18    a = xa;
19    b = xb;
20
21    fa = F(a);
22    fb = F(b);
23
24    while ((a -. b >=. tolerance) ||.
25           (a -. b <=. 0.0-.tolerance))
26          : maxsteps {
27       c = (a +. b) /.2;
28       fc = F(c);
29
30      if ((fa >= 0) &&. (fc >= 0)) {
31         a = c;
32         fa = fc;
33      } else {
34        b = c;
35        fb = fc;
36      }
37    }
38    if (!.((a -. b <=. tolerance) &&.
39        (a -. b >=. 0.0-.tolerance))) {
40       x = INFTY;
41    }
42    return x;
43 }
```

Figure 4-15: Secant Method Implementation

(using a conditional statement). The whole loop can execute unreliably. If the function f has the reliability specification `float<0.9999*R(x)> F(float x)`, then the analysis verifies that the reliability of the computation is at least `.995*R(xa, xb)`.

Note that the reliability of this computation is higher than the reliability of Newton's method. This is because the Secant method makes a single call to the function f, whereas Newton's method makes calls to both f and f' in every iteration. Given multiple options (such as Newton's method and Secant), verified reliability specifications may help developers select the option that best satisfies their combined reliability and efficiency goals.

Coordinate Conversion. This computation converts polar coordinates of a point to Cartesian coordinates. Given a coordinate (R, θ), where R is a radius and θ is an angle, it computes the coordinates $x = R\cos(\theta)$ and $y = R\sin(\theta)$. This is also a checkable computation: the square of the diameter is equal to the sum of the squares of Cartesian coordinates.

Figure 4-16 presents the implementation of this computation. The function takes the inputs `r` and `theta` and stores the result in the output array `xy`. The body of the computation can execute unreliably. The analysis first verifies the implementation of the trigonometric functions (which are evaluations of appropriate Chebyshev interpolating polynomials). If the reliability of the input x is `R(x)`, then the the analysis verifies that the functions $\sin(x)$ and $\cos(x)$ have the reliability at least `0.99999*R(x)`. The analysis uses this result to verify that if the reliability of the parameters `r` and `theta` is `R(r,theta)` , then the reliability of the coordinate conversion computation is at least `0.99995*R(r,theta,xy)`.

Motion Estimation. This is the computation presented in Section 4.2.

Matrix-vector Multiplication This computation calculates the product of a matrix M and a vector v. The dimensions of the matrix are $w \times h$, the length of v is h, and the length of the resulting vector u is w. The computation takes `M`, `v`, and `u` as inputs, and computes the values of the elements of the vector `u`.

Figure 4-17 presents the implementation of this computation. All operations on the input data can execute unreliably. The output of the function is the vector u. Assuming the maximum size of the square matrix to be 64x64 (as in some signal processing applications),

```
1  //Array with sine polynomial coefficients
2  #define nsin 20
3  const csin (1) = { /*...*/ };
4
5  //Array with cosine polynomial coefficients
6  #define ncos 19
7  const ccos (1) = { /*...*/ };
8
9  float<0.99999*R(x)> usin(float x in unrel) {
10       float res in unrel; float t in unrel;
11       int i; i = 1;
12
13       res = csin[0];
14       while (true) : nsin {
15          t = csin[i];
16          res = res *. x +. t;
17          i = i + 1;
18       }
19       return res;
20 }
21
22 float<0.99999*R(x)> ucos(float x in unrel) {
23       float res in unrel; float t in unrel;
24       int i; i = 1;
25
26       res = ccos[0];
27       while (true) : ncos {
28          t = ccos[i];
29          res = res *. x +. t;
30          i = i + 1;
31       }
32       return res;
33 }
34
35 void main(float r in unrel, float theta in unrel,
36      float<0.99995*R(r, theta, xy)> xy) {
37      float x in unrel; float y in unrel;
38      float t in unrel;
39
40      t = ucos(theta);
41      xy(1) = r *. t;
42      t = usin(theta);
43      xy(2) = r *. t;
44 }
```

Figure 4-16: Coordinate Conversion Implementation

```
1  const num matx = 64;
2  const num maty = 64;
3
4  void matvec (float mat(2) in urel,
5               float v(1) in urel,
6               num<.997*R(mat, vec, outvec)>
7                    u(1) in urel)
8  {
9    num t1 in urel;
10   num t2 in urel;
11   num t3 in urel;
12   int i, j, k;
13
14   i = 0;
15   repeat matx  {
16     j = 0;
17     t3 = 0;
18     repeat maty {
19             t1 = mat[j, i];
20             t2 = v[j];
21             t3 = t3 +. t1 *. t2;
22     }
23
24     u[i] = t3;
25     i = i + 1;
26   }
27 }
```

Figure 4-17: Matrix-Vector multiplication Implementation

the specified output reliability for the vector u is `0.997*R(M, v, u)`. The analysis result states that the reliability of *any* element of *u* is greater than this specified output reliability. Note that the specification needs to account for the possible unreliability of assignments to the vector *u* before entering the function, because it does not track the array indices to determine that every element of u is modified inside the function.

```
1  int <0.9999995 * R(val)> abs(int val in unrel) {
2   int t in unrel = val;
3   if (t <=. 0) {
4     t = 0 -. t;
5   }
6   return t;
7  }
8
9  int<0.99995 * R(bA, bB, satdstart)>  hadamarddiff
10    (int<R(bA)> bA(2) in unrel,
11     int<R(bB)> bB(2) in unrel,
12     int satdstart in unrel)
13 {
14   int isatd in unrel;
15   int tmp00 in unrel; int tmp01 in unrel;
16   int tmp02 in unrel; int tmp03 in unrel;
17   int tmp10 in unrel; int tmp11 in unrel;
18   int tmp12 in unrel; int tmp13 in unrel;
19   int tmp20 in unrel; int tmp21 in unrel;
20   int tmp22 in unrel; int tmp23 in unrel;
21   int tmp30 in unrel; int tmp31 in unrel;
22   int tmp32 in unrel; int tmp33 in unrel;
23   int t1 in unrel; int t2 in unrel;
24   int t3 in unrel; int t4 in unrel;
25   int tt in unrel;
26
27   t1 = bA[0,0]; tt = bB[0,0]; t1 = t1 -. tt;
28   t2 = bA[0,1]; tt = bB[0,1]; t2 = t2 -. tt;
29   t3 = bA[0,2]; tt = bB[0,2]; t3 = t3 -. tt;
30   t4 = bA[0,3]; tt = bB[0,3]; t4 = t4 -. tt;
31   tmp00 =  t1 +. t2 +. t3 +. t4;
32   tmp01 =  t1 +. t2 -. t3 -. t4;
33   tmp02 =  t1 -. t2 -. t3 +. t4;
34   tmp03 =  t1 -. t2 +. t3 -. t4;
35
36   t1 = bA[1,0]; tt = bB[1,0]; t1 = t1 -. tt;
37   t2 = bA[1,1]; tt = bB[1,1]; t2 = t2 -. tt;
38   t3 = bA[1,2]; tt = bB[1,2]; t3 = t3 -. tt;
39   t4 = bA[1,3]; tt = bB[1,3]; t4 = t4 -. tt;
40   tmp10 =  t1 +. t2 +. t3 +. t4;
41   tmp11 =  t1 +. t2 -. t3 -. t4;
42   tmp12 =  t1 -. t2 -. t3 +. t4 ;
43   tmp13 =  t1 -. t2 +. t3 -. t4 ;
```

Figure 4-18: Hadamard Transform Implementation (Part 1)

```
45   t1 = bA[2,0]; tt = bB[2,0]; t1 = t1 -. tt;
46   t2 = bA[2,1]; tt = bB[2,1]; t2 = t2 -. tt;
47   t3 = bA[2,2]; tt = bB[2,2]; t3 = t3 -. tt;
48   t4 = bA[2,3]; tt = bB[2,3]; t4 = t4 -. tt;
49   tmp20 =  t1 +. t2 +. t3  +. t4;
50   tmp21 =  t1 +. t2 -. t3  -. t4;
51   tmp22 =  t1 -. t2 -. t3  +. t4;
52   tmp23 =  t1 -. t2 +. t3  -. t4;
53
54   t1 = bA[3,0]; tt = bB[3,0]; t1 = t1 -. tt;
55   t2 = bA[3,1]; tt = bB[3,1]; t2 = t2 -. tt;
56   t3 = bA[3,2]; tt = bB[3,2]; t3 = t3 -. tt;
57   t4 = bA[3,3]; tt = bB[3,3]; t4 = t4 -. tt;
58   tmp30 =  t1 +. t2 +. t3 +. t4;
59   tmp31 =  t1 +. t2 -. t3 -. t4;
60   tmp32 =  t1 -. t2 -. t3 +. t4;
61   tmp33 =  t1 -. t2 +. t3 -. t4;
62   isatd = satdstart;
63
64   t1 = abs(tmp00 +. tmp10 +. tmp20 +. tmp30);
65   t2 = abs(tmp00 +. tmp10 -. tmp20 -. tmp30);
66   t3 = abs(tmp00 -. tmp10 -. tmp20 +. tmp30);
67   t4 = abs(tmp00 -. tmp10 +. tmp20 -. tmp30);
68   isatd = isatd +. t1 +. t2 +. t3 +. t4;
69
70   t1 = abs(tmp01 +. tmp11 +. tmp21 +. tmp31);
71   t2 = abs(tmp01 +. tmp11 -. tmp21 -. tmp31);
72   t3 = abs(tmp01 -. tmp11 -. tmp21 +. tmp31);
73   t4 = abs(tmp01 -. tmp11 +. tmp21 -. tmp31);
74   isatd = isatd +. t1 +. t2 +. t3 +. t4;
75
76   t1 = abs(tmp02 +. tmp12 +. tmp22 +. tmp32);
77   t2 = abs(tmp02 +. tmp12 -. tmp22 -. tmp32);
78   t3 = abs(tmp02 -. tmp12 -. tmp22 +. tmp32);
79   t4 = abs(tmp02 -. tmp12 +. tmp22 -. tmp32);
80   isatd = isatd +. t1 +. t2 +. t3 +. t4;
81
82   t1 = abs(tmp03 +. tmp13 +. tmp23 +. tmp33);
83   t2 = abs(tmp03 +. tmp13 -. tmp23 -. tmp33);
84   t3 = abs(tmp03 -. tmp13 -. tmp23 +. tmp33);
85   t4 = abs(tmp03 -. tmp13 +. tmp23 -. tmp33);
86   return isatd +. t1 +. t2 +. t3 +. t4;
87 }
```

Figure 4-19: Hadamard Transform Implementation (Part 2)

Benchmark	Type	LOC	Time (ms)	Predicates
newton	Checkable	21	8	1
secant	Checkable	30	7	2
coord	Checkable	36	19	1
search_ref	Approximate	37	348	3
matvec	Approximate	32	110	4
hadamard	Approximate	87	18	3

Figure 4-20: Benchmark Analysis Summary

Hadamard Transform This computation takes as input two blocks of 4x4 pixels and computes the sum of differences between the pixels in the frequency domain. This computation is used in digital signal processing applications (including motion estimation).

Figures 4-18 and 4-19 present the implementation of this computation. The computation calculates the intermediate distances between the elements stored in the arrays and computes the sum of the absolute differences of combinations of these elements. The video can be stored in unreliable memory and the entire computation can execute unreliably. The analysis verifies that if the reliability of the two input blocks `bA` and `bB` is `R(bA,bB)`, then the reliability of the computation is greater than `0.99995*R(bA, bB)`.

4.6.2 Analysis Summary

Table 4-20 presents Rely's analysis results on the benchmark computations. For each benchmark, the table presents the type of the computation (checkable or approximate), its length in lines of code (LOC), the execution time of the analysis, and the number of inequality predicates in the final precondition produced by the precondition generator.

Analysis Time. The analysis times for all benchmarks are under one second when executed on an Intel Xeon E5520 machine with 16 GB of main memory.

Number of Predicates. Using the hardware reliability specification from Figure 4-5 to generate a reliability precondition for each benchmark, the number of predicates in each benchmark's precondition is small (all less than five). The reason why the number of predicates is small is because simplification removes most of the additional predicates introduced by the rules for conditionals. Specifically, these predicates are often subsumed by another predicate.

4.6.3 Reliability and Accuracy

A developer's choice of reliability specifications is typically influenced by the perceived effect that the unreliable execution of the computation may have on the accuracy of its result and its execution time and energy consumption. I present two case studies that illustrate how developers can use Rely to reason about the tradeoffs between accuracy and performance that are available for checkable computations and approximate computations.

Checkable Computations

Checkable computations are those that can be augmented with an efficient checker that dynamically verifies the correctness of the computation's result. If the checker detects an error, then it reexecutes the computation or executes an alternative reliable implementation. I next present how a developer can use Rely to build and reason about the performance of a *checked* implementation of Newton's method.

Newton's method. A developer can build a checked implementation of `newton` with the following code:

```
float root = newton(xs);
float ezero = f(root);
if (ezero < -tolerance || ezero > tolerance)
  root = newton_r(xs);
```

To check the reliability of the root x_0 that `newton` produces, it is sufficient to evaluate the function $f(x_0)$ and check if the result is zero (within some tolerance). If the checker detects that the result is not a zero, then the computation calls `newton_r`, a fully reliable alternative implementation.

Reliability/Execution Time Tradeoff. Quantitative reliability enables a developer to model the performance of this checked implementation of Newton's method. Let τ_u be the expected execution time of `newton`, τ_{um} the expected execution time of `newton` when executed for its maximum number of internal loop iterations, τ_c the expected execution time of the checker, and τ_r the expected execution time of `newton_r`.

The expected execution time of the checked computation when `newton` produces a correct result is $T_1 = \tau_u + \tau_c$. In the case when `newton` produces an incorrect result, the expected execution time is at most $T_2 = \tau_{um} + \tau_c + \tau_r$ (i.e., the maximum expected execution time of `newton` plus the expected execution time of both the checker and the reexecution via `newton_r`). This formula is conservative because it assumes that a fault causes `newton` to execute for its maximum number of iterations.

If r denotes the reliability of `newton`, then the expected execution time of the checked computation as a whole is $T' = r \cdot T_1 + (1 - r) \cdot T_2$, which produces a projected speedup s, where $s = \tau_r / T'$. These formulas allow a developer to find the reliability r that meets the developer's performance improvement goal and can be analogously applied for alternative resource usage measures, such as energy consumption and throughput.

Example. As an illustration, let us assume that the computation executes on unreliable hardware and its reliable version is obtained using software-level replication. Using the replication approach proposed in [74], the replicated implementation is 40% slower than the unreliable version – i.e, $\tau_r = 1.4\tau_u$. Furthermore, let the reliable Newton's method computation converge on average in a half of the maximum number of steps (i.e., $\tau_u = \tau_{um}/2$) and let the execution time of the checker be half of the time of a single iteration of Newton's method. For a projected speedup of 1.32, the developer can use the previous formulas to compute the target reliability $r = 0.99$. Rely can verify that `newton_u` executes with this reliability (given the hardware reliability specification from Figure 4-5) when the input x_s is fully reliable.

Approximate Computations

Many applications can tolerate inaccuracies in the results of their internal approximate computations, which are computations that can produce a range of results of different quality. The approximate computations in these applications can be transformed to trade accuracy of their results for increased performance by producing lower quality results.

In building and transforming approximate computations, there are two correctness properties that a developer or transformation system must consider: *integrity* – which ensures

that the computation produces a valid result that the remaining computation can consume without failing – and *accuracy* – which determines the quality of the result itself (Chapter 2). In this case study, I present how a developer can use Rely in combination with other techniques to reason about the integrity and accuracy of `search_ref` (Section 4.2).

Integrity. Recall that `search_ref` searches for the index of the block within the array of pixel blocks `pblocks` that is the minimum distance from the block of pixels `cblock`. An important integrity property for `search_ref` is therefore that it returns a valid index: an index that is at least `0` and at most `nblocks - 1`. However, this property may not hold in an unreliable execution because `search_ref`'s unreliable operations may cause it to return an arbitrary result. To guard against this, a developer has several options.

One of the developer's options is to modify `search_ref` to dynamically check that its result is valid and reexecute itself if the check fails. This case is analogous to that for checkable computations (Section 4.6.3), with the distinction that the checker implements a partial correctness specification.

Another option is to modify `search_ref` to dynamically check that its result is valid and instead of reexecuting itself if the check fails, it *rectifies* [76, 50] its result by returning a valid index at random. This transformation enables `search_ref` to produce a valid – though approximate – result.

Alternatively, the developer can place `minblock` in reliable memory and set its initial value to a fixed, valid index (e.g., `0`); this implements a fixed rectification approach. Because `i` is also stored in reliable memory, `minblock` will always contain a valid index despite `search_ref`'s other unreliable operations. The developer can establish this fact either informally or formally with relational verification [18].

Accuracy. A computation's reliability bound states how often the computation returns a correct result and therefore also states a conservative bound on the computation's accuracy. To determine an acceptable reliability bound (and therefore an acceptable accuracy), the developer can perform local or end-to-end accuracy experiments [58, 88, 39]. As an illustration of an end-to-end experiment, I present a simulation approach for `search_ref`.

To estimate the effects of the unreliable execution, I modified a fully reliable version of `search_ref` (one without unreliable operations) to produce the correct minimum distance block with probability p and produce the maximum distance block with probability $1 - p$. This modification provides a conservative estimate of the bound on `search_ref`'s accuracy loss given a reliability p (when inputs to the computation are reliable) and the assumption that a fault causes `search_ref` to return the worst possible result.

I then implemented two versions of the x264 video encoder [91]: one with the reliable version of `search_ref` and one with the modified version. For several values of p, I then compared the videos produced by the reliable and modified encoders on 17 HD video sequences (each 200 frames in length) from the Xiph.org Video Test Media repository [51]. I then quantified the difference between the quality of the resulting videos via the Quality Loss Metric (QLM), previously used in [58]. This metric computes a relative difference between the quality scores of the two videos, each of which is computed as a weighted sum of the peak signal to noise ratio and the encoded video size.

p	0.90	0.95	0.97	0.98	0.99
QLM	0.041	0.021	0.012	0.009	0.004

Figure 4-21: `search_ref` Simulation Result

Table 4-21 presents the average QLM as a function of the reliability of `search_ref`. A developer can use the results of the simulation to identify an acceptable amount of quality loss for the encoded video. For example, if the developer is willing to accept at most 1% quality loss (which corresponds to the QLM value 0.01), then the developer can select 0.98 from Table 4-21 as the target reliability for an unreliable version of `search_ref`. The reliability specification that the developer then writes for an unreliable version is `0.98*R(pblocks,cblock)`. As demonstrated in Section 4.2, Rely's analysis can verify that the presented unreliable implementation satisfies an even tighter reliability (i.e., 0.99).

4.7 Related Work

In this section, I present an overview of the other work that intersects with Rely and its contributions to modelling approximating computing, probabilistic and relational verification, and verification of approximate programs.

4.7.1 Critical and Approximate Regions

Almost all approximate computations have critical regions that must execute without error for the computation as a whole to execute acceptably.

Dynamic Criticality Analysis. One way to identify critical and approximate regions is to change different regions of the computation or data in some way and observe the effect. To the best of my knowledge, Rinard was the first to present a technique (task skipping) designed to identify critical and approximate regions in approximate computations [77, 78]. Carbin and Rinard subsequently presented a technique that uses directed fuzzing to identify critical and approximate computations, program data, and input data [21]. Other techniques use program transformations [58] and input fuzzing [6].

Static Criticality Analysis. Researchers have also developed several specification-based static analyses that let the developer identify and separate critical and approximate regions. Flikker [49] is a set of language extensions with a runtime and hardware support to enable more energy efficient execution of programs on inherently unreliable memories. It allows a developer to partition data into critical and approximate regions (but does not enforce full separation between the regions). Based on these annotations, the Flikker runtime allocates and stores data in a reliable or unreliable memory. Sampson et al. [84] present EnerJ, a programming language with an information-flow type system that allows a developer to partition program's data into approximate and critical data and ensures that operations on approximate data do not affect critical data or memory safety of programs.

All of this prior research focuses on the binary distinction between reliable and approximate computations. In contrast, the research presented in this thesis allows a developer to specify and verify that even approximate computations produce the correct result most of the time. Overall, this additional information can help developers better understand the effects of deploying their computations on unreliable hardware and exploit the benefits that unreliable hardware offers.

4.7.2 Relational Reasoning for Approximate Programs

In Chapters 2 and 3 I presented a verification system for relaxed approximate programs based on a relational Hoare logic. The system enables rigorous reasoning about the integrity and worst-case accuracy properties of a program's approximate regions. My work in those chapters builds upon the relational verification techniques originated in Rinard et al.'s Credible Compilation [82], Pnueli et al.'s Translation Validation [70], and later by Benton's Relational Hoare Logic [12].

Rely differs from these approaches because of its probabilistic relational reasoning: specifically, the probability that an unreliable implementation returns the correct result. However, these non-probabilistic approaches are complementary to Rely in that they enable reasoning about the non-probabilistic properties of an approximate computation.

4.7.3 Accuracy Analysis

In addition to reasoning about how often a computation may produce a correct result, it may also be desirable to reason about the accuracy of the result that the computation produces. Dynamic techniques observe the accuracy impact of program transformations [77, 78, 81, 58, 88, 5, 39, 4, 52, 56], or injected soft errors [48, 27, 49, 84, 90]. Researchers have developed static techniques that use probabilistic reasoning to characterize the accuracy impact of various phenomena [57, 24, 73, 94, 11, 62]. And of course, the accuracy impact of the floating point approximation to real arithmetic has been extensively studied by numerical analysts [23].

4.7.4 Probabilistic Program Analysis

Kozen's work [43] was the first to propose the analysis of probabilistic programs as *transformers* of discrete probability distributions. Researchers have since developed a number of program analyses for probabilistic programs, including those based on axiomatic reasoning [61, 10, 11], abstract interpretation [59, 29, 89, 25], and symbolic execution [35, 85].

The language features that introduce probabilistic nondeterminism in programs that this previous research studied include probabilistic sampling, `x = random()` [43, 59, 10, 11], probabilistic choice between statements, `s1` $\oplus_p$ `s2` [61, 29, 25], and specifications of

the distributions of computation's inputs [89]. Rely refines the probabilistic operators by defining a set of unreliable arithmetic and memory read/write operations that model faults in the underlying hardware.

Morgan et al. [61] propose a weakest-precondition style analysis for probabilistic programs that treats the programs as *expectation transformers*. Preconditions and postconditions are defined as bounds on probabilities that particular logical predicates hold at a specified location in the program. Rely's analysis, like [61], constructs precondition predicates for program statements. However, Rely's predicates are relational, relating the states of the reliable and unreliable executions of the program. Moreover, Rely's analysis is more precise as it uses direct multiplicative lower bounds on reliability as opposed to additive upper bounds on error.

Barthe et al. [10] define a probabilistic relational Hoare logic for a simple probabilistic imperative language that is similar to Kozen's. The relational predicates are arbitrary conjunctions or disjunctions of relational expressions over program variables, each of which is endowed with a probability of being true. While general, this approach requires manual proofs or an SMT solver to verify the validity of predicates. In comparison, Rely presents a semantics for reliability predicates that incorporates joint reliability factors, which create a simple and efficient checking procedure.

Reliability Analysis. Analyzing the reliability of complex physical and software systems is a classical research problem [47]. More recently researchers have presented symbolic execution techniques for checking complex probabilistic assertions. Filieri et al. [35] present an analysis for finite-state Java programs. Sankaranarayanan et al. [85] present an analysis for computations with linear expressions and potentially unbounded loops. These techniques require knowledge of the distributions of the inputs to the computation. Rely's analysis requires only the probability with which each operation in the computation executes correctly.

4.7.5 Fault Tolerance and Resilience

Researchers have developed various software, hardware, or mixed approaches for detection and recovery from specific types of soft errors that guarantee a reliable program execu-

tion [74, 69, 27, 34, 67, 86, 37, 90, 80, 68, 20, 41]. For example, Reis et al. [74] present a compiler that replicates a computation to detect and recover from single event upset errors.

These techniques are complementary to Rely in that each can provide implementations of operations that need to be reliable, as either specified by the developer or as required by Rely, to preserve memory safety and control flow integrity.

4.7.6 Emerging Hardware Architectures

Recently researchers have proposed multiple hardware architectures to trade reliability for additional energy or performance savings. Some of the recent research efforts include probabilistic CMOS chips [66], stochastic processors [64], error resilient architecture [45], unreliable memories [44, 49, 84, 65], and the Truffle architecture [32]. These techniques typically use voltage scaling at different granularities.

This previous research demonstrated that for specific classes of applications, such as multimedia processing and machine learning, the proposed architectures provided energy or time savings profitable to the user. Rely aims to help developers better understand and control the behavior of their applications on such platforms.

4.8 Conclusion

Driven by hardware technology trends, future computational platforms are projected to contain unreliable hardware components. To safely exploit the benefits (such as reduced energy consumption) that such unreliable components may provide, developers need to understand the effect that these components may have on the overall reliability of the approximate computations that execute on them.

I present a language, Rely, for exploiting unreliable hardware and an associated analysis that provides probabilistic reliability guarantees for Rely computations executing on unreliable hardware. By enabling developers to better understand the probabilities with which this hardware enables approximate computations to produce correct results, these guarantees can help developers safely exploit the significant benefits that unreliable hardware platforms offer.

Chapter 5

Thesis Summary and Conclusion

Many application domains on the forefront of computing have an inherent trade-off between the quality of the result that a computation produces and the time or resources needed to compute the result. This trade-off enables a variety of novel methodologies for developing new programs and/or changing the semantics of existing programs to unlock opportunities for a computation to navigate its trade-offs given the quantity and quality of the computation's available resources.

A core component of navigating these trade-offs is guaranteeing that the resulting computation satisfies the acceptability properties that ensure that the computation has a well-defined execution and returns a result that is acceptable for the computation's user.

5.1 Summary

This thesis presents several program logics and program analyses that 1) enable developers to specify acceptability properties of computations, 2) enable both developers and programming systems to change and transform the semantics of computations, and 3) enable both developers and programming systems to verify that the resulting computation satisfies its acceptability properties.

5.1.1 Relaxed Programming

This thesis presents the relaxed programming development model, along with a programming language and program logic that enables specifying and verifying relaxations of a program's semantics. A core motivation for my development of the relaxed programming model is that many transformations that change the semantics of programs make localized modifications of the parts of a program's state that control the structure of its execution.

Building upon this observation, the relaxed programming framework codifies three basic principles into its programming language and program logic:

Acceptability. A core challenge in changing the semantics of the program is determining what makes the program's original implementation and (therefore its relaxed implementation) acceptable for its anticipated use. To this end, I have defined the concept of acceptability properties and provided language support for specifying acceptability properties. Acceptability properties include, for example, standard unary assertions that assert the integrity of the program (e.g., no out-of-bounds memory access). Acceptability properties also include *relational* properties that relate the original program to all acceptable relaxations. For example, an acceptability property may state that relaxation cannot interfere with a portion of the program's state or dictate that relaxation can only change a portion of the program state by at most some determined amount.

Relaxation. The relaxed programming model introduces into the programming language the `relax (`X`) st (`b`)` statement, which non-deterministically modifies a set of variables, X, in the program state subject to a boolean constraint b. The `relax` statement enables a developer or compilation system to change the semantics of the program in a well-defined and constrained way.

Relational Reasoning. Relaxing the semantics of the program may alter the behavior of the program in a way that jeopardizes the validity of the program's acceptability properties. A key concern is therefore verifying that the resulting program still satisfies its acceptability properties. My program logic addresses this concern with a novel verification approach that relies on the observation that relaxations often preserve the acceptability of the relaxed program because they often do not inference with the acceptability original program.

By relating the semantics of the original and relaxed program, my program logic enables a relational reasoning approach in which developers can use the logic to demonstrate the independence of the program's acceptability properties from relaxation and, therefore, transfer much of the reasoning that was done for the original program over to verify the relaxed program.

These basic concepts work together to provide a core calculus for specifying, relaxing, and verifying programs.

5.1.2 Extended Relaxed Programming

This thesis also extends the core principles of relaxed programming from a simple imperative language to a larger language that includes standard programming constructs, such as procedures, arrays, and heap-allocated linked data structures.

A core realization of extending the programming language is that many standard programming language constructs have embedded assertions that are also amenable to relational verification. For example, data structures allocated in the heap have implicit assertions that specify that pointers into the heap are valid. Accesses to elements in arrays also require assertions that check the validity of indices used in the access. Also, verifying programs that contain procedures in a modular assume-guarantee style typically requires the use of preconditions and postconditions, which express properties that must hold and the beginning and end of a function, respectively.

These built-in assertions are additional points at which 1) relaxation may jeopardize the validity of the assertion and 2) relational reasoning can be used to leverage the reasoning done for the original program to verify the relaxed.

5.1.3 Verifying Quantitative Reliability

Relaxed programming provides an effective framework for reasoning about the worse-case behaviors of the program. However, for many approximate computing applications, developers are also interested in acceptability properties that are probabilistic in nature.

To this end, this thesis presents, Rely, a semantic framework and program analysis for

specifying and verifying the reliability of applications that execute on hardware substrates that have been augmented to produce approximate results.

Rely includes four key contributions:

- **Specifying Reliability.** Rely's language enables developers to specify the required reliability of a computation – the probability that the computation produces the correct result.
- **Specifying Unreliable Computation.** Rely's language includes a methodology for 1) providing specifications of the reliability of an approximate hardware system and 2) exposing exposure unreliable hardware primitives to the developer or compilation system via the programming language.
- **Semantics of Reliability.** The Rely framework includes semantic foundations that give a well-defined meaning to the reliability of a program when its execution on unreliable hardware is interpreted as the execution of a probabilistic transition system.
- **Reliability Analysis.** The Rely framework also provides an reliability analysis that automatically verifies if a given program when executed on a specified unreliable hardware system satisfies its reliability specification.

These contributions position Rely as a robust platform for exploring not only the reliability of approximate hardware/software systems, but also the trade-offs between their integrity, accuracy, and performance.

5.2 Future Directions

Approximate Computing Frameworks. In traditional compiler frameworks, each optimization pass provides the modular guarantee that it will preserve the semantics of the program. This guarantee enables a compiler to sequentially compose optimization passes and still produce a semantically equivalent program at the end of the optimization process. For optimizations that change the behavior of programs, however, this compositional reasoning no longer holds. For example, optimizations that target unreliable hardware can deliver a range of trade-offs between the reliability and performance of the program. An open question is then how to automate the exploration of this trade-off space and preserve sufficient structure throughout the process to perform effective end-to-end verification.

Compiling programs to unreliable hardware can also produce a broader understanding of program approximation in general. The numerical analysis, theory, and database communities have proposed a variety of different program approximate mechanisms. A open challenge is therefore how to integrate techniques from these communities (such as replacing code with sublinear algorithms and representing program data with sparse, multiresolution data representations) into a general approximate computing framework that operates at multiple levels in the computational stack.

Emerging Software Development Concerns. The programming language community's goal of making software easier to write has been partially realized through communities like GitHub and Stack Overflow. By making it easy to access and adapt software components, these communities have brought about an ecosystem in which software components move rapidly from project to project and from domain to domain. As these forces continue to push software development, the vast majority of the components of future software systems will have uncertain provenance and operation. Software developers will therefore understand less of their systems' overall behavior than they currently do today.

Understanding software is currently one of the primary ways we gain confidence that our software provides some guarantee, such as security or functional correctness. As we continue to use collaborative software communities, building confidence in software will become even more important because these communities will face the same epidemiological challenges as human communities: bugs and security vulnerabilities will spread among programs. A critical research question will therefore become how to build confidence in these poorly understood systems.

Semi-automated program verification will solve this problem for core components of the software stack (e.g., compilers, operating systems, and standard libraries and data structures). These components have strong logical characterizations of correctness that are amenable to verification. For these systems, developers will specify interfaces for individual components and then verify that their implementations respect these specifications. Verified implementations will enable developers to reuse components with confidence – and without understanding their exact provenance or operation.

On the expansive periphery of the software ecosystem, however, where software is one-off, quickly developed, and often user-facing, correctness is less well-defined and resilience is a primary objective. For this software, we will need new techniques that build confidence through resilience. To this end, relaxed programs can serve as a platform for exposing and manipulating global system behaviors to create resilience. The principles of relaxation can make it possible to connect these behaviors with developers' limited understanding in a way that enables developers to distinguish between the set of behaviors they have built into their software and the set of behaviors that emerge from resiliency mechanisms. This approach has the potential to make poorly understood software systems fast, secure, and usable.

5.3 Conclusion

The software and hardware communities have grown accustom to the digital abstraction of computing: the computing substrate is designed to either faithfully execute an operation or detect and report that an error has occurred. This abstraction has enabled a process whereby increased performance capability in the substrate enables the development of increasingly larger and more complicated computing systems that are composed of less complicated, modularly-specified components.

Emerging trends in the scalability of existing hardware design techniques, however, jeopardize the hope that future gains in computing performance will still be accompanied by a digital abstraction. Instead, future high-performance computing platforms may produce uncertain results and, therefore, it may no longer be possible to use traditional techniques to modularly compose components to execute on these platforms.

While there is an immediate opportunity for my work enable the reasoning needed to reliably achieve better performance in the face of uncertainty, the true motivation for my work is that the nature of computing itself has changed. Emerging applications, such as machine learning, multimedia, and data analytics are inherently uncertain computations that operate over uncertain inputs. Moreover, emerging uncertain computational substrates, such as intermittently powered devices, biological devices, and quantum computing, create new possibilities for where computation can take place and even what can be computed itself.

Going forward this work will enable the software and hardware communities to discard the notion that they must rely on the digital abstraction to build computing systems. Instead, emerging computing systems will use abstractions of acceptability that will enable these systems to exploit not only the performance benefits of uncertain substrates, but also the new possibilities that these platforms offer for what can be accomplished with computation.

Appendix A

This appendix presents the full Coq development of the relaxed programming model from Chapter 2, including the programming language, program logic, and the proofs of the properties discussed in Section 2.4. The appendix has the following organization:

Auxiliary Definitions. Section A.1 presents auxiliary definitions that are used in the remainder of the development. These definitions include, for example, properties of the basic data structures (such as lists) used in the formalization.

Expressions. Section A.2 presents definitions of the abstract syntax and semantics of environments and expressions (as presented in Section 2.2). The presentation includes the syntax and semantics of both unary expressions and relational expressions. Along with these definitions, the section also formalizes and proves properties of the semantics necessary for later parts of the development. For example, the section defines the equivalence relation for environments (Definition `env_equiv`) along with a proof that the relation is symmetric (Lemma `env_equiv_sym`) and reflexive (Lemma `env_equiv_refl`).

Statements. Section A.3 presents the abstract syntax of the statements of the programming language. The following two sections give the statements their dynamic semantics.

Dynamic Original Semantics. Section A.4 presents the definitions and proofs for the dynamic original semantics presented in Section 2.2.2. The section presents the basic definition of the evaluation relation. This section also specifies 1) how to compute the set of variables modified by a statement and 2) proves that execution of a statement does

not modify any variables except for those returned by the specified computation (Lemma `original_eval_not_in_mods_constant`).

Dynamic Relaxed Semantics. Section A.6 presents the definitions and proofs for the dynamic relaxed semantics presented in Section 2.2.3. Similar to the section for the dynamic original semantics, this section presents the dynamic relaxed semantics's evaluation relation. This section also specifies how to compute the set of variables modified by the relaxed semantics of a statement and proves that computed the set is consistent with the semantics (Lemma `relaxed_eval_not_in_mods_constant`).

Assertion Logic. Section A.7 presents the relational assertion logic presented in Section 2.3.1. This section presents the abstract syntax of the logic, its denotational semantics, and the majority of the semantic properties required of the logic by the remainder of the development. For example, the section presents definitions that tie the syntax and semantics of the program's unary expressions to that of their relational counterparts. The section also presents the basic definitions for *grafting*, which is the basic mechanism I've used to implement capture-avoiding substitution. Grafting implements capture-avoiding substitution by mandating that variables that occur in binding positions come from a set that is disjoint from the set of variables that occur in expressions. These definitions set the stage for the following section on the substitution properties of the logic.

Substitution. Section A.8 presents the definitions and proofs for the relational assertion logic's substitution properties. The definitions and proofs here are the bulk of the development. A key lemma and proof is the soundness of substitution (Lemma `pred_denote_sub`), which states that the effects of an assignment statement on the environment of a program can be soundly characterized (with backwards-style Hoare reasoning) by substituting the assignment statement's expression for all occurrences of the variable in a predicate that describes the statement's postcondition. This lemma is the basis of the soundness proofs for assignment rules in the development's multiple axiomatic semantics.

Unary Assertion Logic. Section A.9 presents the abstract syntax and semantics of the unary assertion logic used in the axiomatic original semantics presented in Section 2.3.1. While the logic presented in that section as independent of the relational assertion logic, my implementation embeds the semantics of the unary assertion logic into that of the relational assertion logic. Specifically, the unary assertion logic constrains properties only of one environment in the pair of environments that underpin the relational assertion logic's semantics. This reuse enables a more efficient proof development in that properties of the unary logic can be straightforwardly derived from their counterparts in the relational assertion logic.

Axiomatic Original Semantics. Section A.10 presents the proof rules for the axiomatic original semantics along with proofs of the rules' properties (as defined in Section 2.4.1). The primary proofs are that the axiomatic original semantics 1) is sound with respect to the dynamic original semantics (Lemma `original_axiomatic_soundness`) and 2) establishes progress for an original program (Lemma `original_axiomatic_progress`).

Axiomatic Intermediate Semantics. Section A.11 presents the proof rules for the axiomatic intermediate semantics along with proofs of the rules' properties (as defined in Section 2.4.2). The primary proofs are that the axiomatic intermediate semantics 1) is sound w.r.t. the relaxed semantics (Lemma `intermediate_axiomatic_soundness`) and 2) establishes progress for a relaxed program (Lemma `intermediate_axiomatic_progress`).

Axiomatic Relaxed Semantics. Section A.12 presents the proof rules for the axiomatic relaxed semantics along with proofs of the rules' properties (presented in Section 2.4.3). These theorems are the main theorems of the development. This section specifically presents proofs that the axiomatic relaxed semantics 1) is sound with respect to the dynamic relaxed semantics (Lemma `relaxed_axiomatic_soundness`), 2) establishes that all relational assertions in the relaxed program are valid (Lemma `relational_assertion_soundness`), and 3) establishes progress (in its multiple forms) for the relaxed program (Lemmas `relaxed_axiomatic_relative_progress`, `relaxed_axiomatic_progress`).

A.1 Util

```
Require Export List.

Require Import Classical.

Require Import CpdtTactics.

Lemma not_in_filter : ∀ (A : Type) (l : list A) x f, ¬ In x l → ¬ In x (filter f l).

Proof.
  induction l. auto.

   intros. simpl in H. apply not_or_and in H. inversion H.

   unfold filter. fold (filter f l). destruct (f a).

     simpl. apply and_not_or. split.

       auto. apply IHl. auto.

     apply IHl. auto.

Qed.

Lemma filter_app : ∀ (A : Type) (x : A) f (l1 l2 : list A), (filter f l1) ++ (filter f l2) = filter
f (l1 ++ l2).

Proof.
  induction l1.

    intros. simpl. trivial.

    simpl. intros. destruct (f a).

      simpl. rewrite → IHl1. trivial.

      apply IHl1.

Qed.

Lemma filter_filter : ∀ (A : Type) (x : A) f (l : list A), filter f (filter f l) = filter f l.

Proof.
  induction l.

    simpl. trivial.

     simpl. destruct (f a) as [] _eqn.

       simpl. destruct (f a) as [] _eqn. rewrite → IHl. trivial.
```

```
        inversion Heqb.
        apply IHl.
Qed.

Lemma filter_comm : ∀ (A : Type) (x : A) f g (l : list A), filter f (filter g l) = filter g (filter f
l).
Proof.
  induction l; simpl; auto.
  destruct (g a) as [] _eqn. simpl. destruct (f a) as [] _eqn. simpl. rewrite →
Heqb. rewrite → IHl. trivial.
    apply IHl.
   destruct (f a) as [] _eqn. simpl. rewrite → Heqb. apply IHl.
  apply IHl.
Qed.

Lemma not_in_app : ∀ (A : Type) (a : A) (m n: list A), ¬In a (m ++ n) ↔ ¬In a m ∧ ¬ In a
n.
Proof.
      induction m. simpl. auto. simpl. split; intros.
       auto.
       inversion H. auto.
       split; intros.
         simpl in H. apply not_or_and in H. inversion H.
         apply IHm in H1. inversion H1.
         simpl. split.
           apply and_not_or. auto.
           auto.
         inversion H. simpl in H0. apply not_or_and in H0. inversion H0.
         simpl. apply and_not_or. split.
           auto.
           apply IHm. auto.
```

```
Qed.

Definition and_not_in_app (A : Type) (a : A) (m n : list A) := (proj2 (not_in_app _ a m
n)).

Definition not_in_app_and (A : Type) (a : A) (m n : list A) := (proj1 (not_in_app _ a m
n)).

Hint Resolve and_not_in_app not_in_app_and : datatypes.

Lemma in_not_in_not_eq : ∀ (A : Type) (xs : list A) x a, In x xs → ¬In a xs → x ≠ a.
Proof.
  induction xs. intros. contradict H.
  simpl. intros. apply not_or_and in H0. inversion H0.
  inversion H.
  subst x. assumption.
  apply IHxs; assumption.
Qed.

Section two_list_ind.

  Variable A : Set.
  Variable B : Set.
  Variable P : list A → list B → Prop.
  Hypothesis nil_case : P nil nil.
  Hypothesis left_nil : ∀ (b : B) (lb : list B), P nil (b::lb).
  Hypothesis right_nil : ∀ (a : A) (la : list A), P (a::la) nil.
  Hypothesis app_case : ∀ (a : A) (b : B) (la : list A) (lb : list B), P la lb → P (a::la)
(b::lb).
  Program Fixpoint two_list_ind (la : list A) (lb : list B) :=
    match la with
     | a::arest ⇒
       match lb with
       | b::brest ⇒ app_case a b arest brest (two_list_ind arest brest)
       | nil ⇒ right_nil a arest
```

```
      end
    | nil ⇒
      match lb with
      | b::brest ⇒ left_nil b brest
      | nil ⇒ nil_case
      end
    end.
End two_list_ind.

Lemma list_nil :
  ∀ (A : Type) (l : list A),
    (∀ a, In a l → ¬ In a l) → l = nil.
Proof.
  induction l. simpl. auto.
   simpl. crush; exfalso; eauto.
Qed.

Section remove.
Lemma not_In_not_In_remove :
    ∀ (A: Type) (l : list A) (x : A), ¬ In x l → ∀ y eq, ¬ In x (remove eq y l).
Proof.
  induction l; simpl ; intros.
   auto.
   destruct (eq y a). eauto.
     crush. eauto.
Qed.
Lemma not_eq_not_In_remove_no_In :
  ∀ (A : Type) (l : list A) (x : A),
    ∀ y, x ≠ y → ∀ eq, ¬ In x (remove eq y l) → ¬ In x l.
Proof.
  Hint Resolve and_not_or.
  induction l; auto.
```

```
  simpl; intros. destruct (eq y a). subst; eauto.
  simpl in ×. apply not_or_and in H0. destruct H0.
  eauto.
Qed.

Lemma In_remove :
  ∀ (A : Type) (l : list A) x y eq,
    In x l → y ≠ x → In x (remove eq y l).
Proof.
  induction l; simpl; intros. auto.
   destruct H. subst.
   destruct (eq y x). tauto. simpl; auto.
   destruct (eq y a); simpl; eauto.
Qed.

Lemma In_remove_In :
 ∀ (A : Type) (l : list A) x y eq,
   In x (remove eq y l) → In x l .
Proof.
   induction l; simpl; intros. auto.
   destruct (eq y a); simpl; eauto.
   simpl in H. destruct H; eauto.
Qed.

Lemma remove_app :
    ∀ (A : Set) (m n : list A) (x : A) eq,
      remove eq x (m ++ n) = (remove eq x m) ++ (remove eq x n).
Proof.
   induction m;
      crush.
      destruct (eq x a); crush.
Qed.
```

```
Lemma remove_remove:
  ∀ (A : Set) (l : list A) (x : A) eq,
    remove eq x (remove eq x l) = remove eq x l.
Proof.
  induction l. auto.
    simpl. intros. destruct (eq x a).
     subst. eauto.
     simpl. destruct (eq x a); (congruence || auto).
Qed.

Lemma remove_commute :
  ∀ (A : Set) (l : list A) (x y : A) eq,
    remove eq x (remove eq y l) = remove eq y (remove eq x l) .
Proof.
  induction l.
    auto...
    simpl; intros. destruct (eq y a); destruct (eq x a); simpl;
      auto.
      destruct (eq y a) ; congruence...
      destruct (eq x a) ; congruence...
      destruct (eq y a); destruct (eq x a); congruence.
Qed.

End remove.
```

A.2 Expressions

```
Require Import util.
Require Import Coq.Bool.Bool.
Require Import Classical.
Require Import List.
Open Scope list_scope.
```

```
Require Export Coq.Arith.Compare_dec.
Require Export Coq.Arith.Peano_dec.
Require Import Tactics.
Require Import CpdtTactics.
Require Import Coq.Program.Tactics.

Inductive var : Set :=
 | Id : nat → var.

Inductive arr_var : Set :=
  | ArrId : nat → arr_var.

Inductive rel_var : Set :=
  | Org : var → rel_var
  | Rel : var → rel_var
.

Inductive rel_arr_var : Set :=
  | OrgArr : arr_var → rel_arr_var
  | RelArr : arr_var → rel_arr_var
.

Definition state := var → nat.
Definition arr_lengths := arr_var → nat.

Section Heap.

Record heap : Set :=
  mkHeap { array_contents : arr_var → nat → nat}.

Definition empty_state : state := fun _ ⇒ 0.
Definition empty_heap : heap :=
   mkHeap (fun _ _ ⇒ 0).

End Heap.

Record environment : Type :=
  mkEnv {st : state ; he : heap}.
```

```
Definition empty_env := mkEnv empty_state empty_heap.

Definition stack_equiv (s s' : state) vars :=
  (∀ v : var, In (inl arr_var v) vars → s v = s' v).

Definition heap_equiv he he' vars :=
  (∀ av : arr_var, In (inr var av) vars →
    ∀ i, array_contents he av i = array_contents he' av i).

Definition env_equiv env env' vars :=
  stack_equiv (st env) (st env') vars ∧
  heap_equiv (he env) (he env') vars .

Ltac unfold_equivs :=
  repeat (match goal with
  | [ H : context[env_equiv _ _ _ ] ⊢ _ ] ⇒ unfold env_equiv in H
  | [ H : context[stack_equiv _ _ _ ] ⊢ _ ] ⇒ unfold stack_equiv in H
  | [ H : context[heap_equiv _ _ _ ] ⊢ _ ] ⇒ unfold heap_equiv in H

  | [ ⊢ context[env_equiv _ _ _ ] ] ⇒ unfold env_equiv

  | [ ⊢ context[stack_equiv _ _ _ ] ] ⇒ unfold stack_equiv

  | [ ⊢ context[heap_equiv _ _ _ ] ] ⇒ unfold heap_equiv

  end).

Lemma env_equiv_left :
  ∀ env env' v1 v2, env_equiv env env' (v1 ++ v2) → env_equiv env env' v1 .
Proof.
  unfold_equivs; crush.
Qed.

Lemma env_equiv_right :
  ∀ env env' v1 v2, env_equiv env env' (v1 ++ v2) → env_equiv env env' v2 .
```

```
Proof.
  unfold_equivs ; crush.
Qed.

Lemma env_equiv_sym :
  ∀ env env' v, env_equiv env env' v ↔ env_equiv env' env v.
Proof.
  unfold_equivs; crush.
Qed.

Lemma env_equiv_refl :
  ∀ env vars, env_equiv env env vars.
Proof.
  unfold_equivs; crush.
Qed.

Lemma env_equiv_trans :
  ∀ env1 env2 vars, env_equiv env1 env2 vars →
    ∀ env3, env_equiv env2 env3 vars → env_equiv env1 env3 vars.
Proof.
  unfold_equivs; crush.
Qed.

Inductive cmp : Set := LT | EQ | LEQ.
Inductive lop : Set := And | Or.
Inductive iop : Set := Plus | Minus | Times.

Inductive exp : Set :=
  | Const : nat → exp
  | Var : var → exp
  | Arr : arr_var → exp → exp
  | Iop : iop → exp → exp → exp
.

Inductive rexp : Set :=
```

| RConst : nat → **rexp**

| RVar : **rel_var** → **rexp**

| RArr : **rel_arr_var** → **rexp** → **rexp**

| RIop : **iop** → **rexp** → **rexp** → **rexp**

.

`Inductive` **rbexp** : `Set` :=

| RBConst : bool → **rbexp**

| RCmp : **cmp** → **rexp** → **rexp** → **rbexp**

| RLop : **lop** → **rbexp** → **rbexp** → **rbexp**

| RNeg : **rbexp** → **rbexp**.

Notation "E1 '-_e' E2" := (lop Minus *E1 E2*) (at *level* 80) : *accept_scope*.

Notation "E1 '+_e' E2" := (lop Plus *E1 E2*) (at *level* 80) : *accept_scope*.

`Inductive` **bexp** : `Set` :=

| BConst : bool → **bexp**

| Cmp : **cmp** → **exp** → **exp** → **bexp**

| Lop : **lop** → **bexp** → **bexp** → **bexp**

| Neg : **bexp** → **bexp**

.

`Section` language.

`Definition` var_eq_dec: ∀ (*v1 v2*: **var**), {*v1*=*v2*} + {*v1*≠*v2*}.

`Proof`.

decide equality.

decide equality.

`Qed`.

`Definition` arr_var_eq_dec : ∀ *av1 av2* : **arr_var**, {*av1*=*av2*} + {*av1*≠*av2*}.

`Proof`.

decide equality. *decide equality*.

`Qed`.

`Definition` var_arr_var_eq_dec :

```
  ∀ (av1 av2 : var+arr_var),
    {av1 = av2} + {av1 ≠ av2}.
Proof.
  repeat decide equality.
Qed.

Definition rel_var_eq (rv1 rv2 : rel_var) : bool :=
  match (rv1, rv2) with
  | (Org v1, Org v2) | (Rel v1, Rel v2) ⇒ if var_eq_dec v1 v2 then true else false
  | _ ⇒ false
  end.

Definition rel_var_eq_dec : ∀ rv1 rv2 : rel_var, {rv1=rv2} + {rv1≠rv2}.
Proof.
  decide equality; apply var_eq_dec.
Qed.

Definition rel_arr_var_eq_dec : ∀ rv1 rv2 : rel_arr_var, {rv1=rv2} + {rv1≠rv2}.
Proof.
  decide equality; apply arr_var_eq_dec.
Qed.

End language.

Ltac red_exprs := simpl ; intros; repeat (match goal with
    | [ H : ¬ In ?v (?m ++ ?n) ⊢ _ ] ⇒ destruct (not_in_app_and _ _ _ _ H); clear H
    | [ H : In ?v (?m ++ ?n) ⊢ _ ] ⇒ apply in_app_or in H

    | [ H : context[eq_nat_dec ?x ?y] ⊢ _ ] ⇒ destruct (eq_nat_dec x y) in H; simpl

    | [ ⊢ context[eq_nat_dec ?x ?y] ] ⇒ destruct (eq_nat_dec x y) ; simpl

    | [ H : context[var_eq_dec ?x ?y] ⊢ _ ] ⇒ destruct (var_eq_dec x y) in H ; simpl
    | [ H : context[arr_var_eq_dec ?x ?y] ⊢ _ ] ⇒ destruct (arr_var_eq_dec x y) in H ;
```

```
simpl
    | [ H : context[var_arr_var_eq_dec ?x ?y] ⊢ _ ] ⇒ destruct (var_arr_var_eq_dec x y)
in H ; simpl



    | [ ⊢ context[var_eq_dec ?x ?y] ] ⇒ destruct (var_eq_dec x y) ; simpl
    | [ ⊢ context[var_arr_var_eq_dec ?x ?y] ] ⇒ destruct (var_arr_var_eq_dec x y) ; simpl
    | [ ⊢ context[arr_var_eq_dec ?x ?y] ] ⇒ destruct (arr_var_eq_dec x y) ; simpl
 end).

Definition assign_state (s : state) (v : var) (n : nat) : state :=
    fun (x : var) ⇒ if (var_eq_dec x v) then n else (s x).

Fixpoint assign_vars (s : state) (vars : list var) (vals : list nat) : state :=
    match (vars, vals) with
    | (v::vs, val::vals) ⇒ assign_vars (assign_state s v val) vs vals
    | _ ⇒ s
    end
.

Lemma assign_state_imd : ∀ s v n, (assign_state s v n) v = n.
Proof.
  intros. unfold assign_state. destruct var_eq_dec; crush.
Qed.

Lemma assign_state_same : ∀ v v' s n, v ≠ v' → (assign_state s v n) v' = s v'.
Proof.
   intros. unfold assign_state. destruct var_eq_dec; crush.
Qed.

Definition update_variable (e : environment) (v : var) (n : nat) : environment :=
   mkEnv (assign_state (st e) v n) (he e).

Lemma update_variable_correct :
   ∀ env v n, st (update_variable env v n) v = n.
```

```
Proof.
  Hint Resolve assign_state_imd.
  crush.
Qed.

Section update_variable_equiv.

Hint Rewrite assign_state_same : cpdt.

Lemma update_variable_equiv :
  ∀ env v n vars,
    ¬ In (inl _ v) vars →
      env_equiv env (update_variable env v n) vars.
Proof.
  unfold_equivs; crush.
Qed.

End update_variable_equiv.

Definition update_array_contents (env : environment) (av: arr_var) (i : nat) (n : nat)
: environment :=
  let h := he env in
  let new_contents :=
      fun x idx ⇒
        if (arr_var_eq_dec x av) then
          (if eq_nat_dec i idx then n else array_contents h x idx)
        else
          array_contents h x idx
    in
      mkEnv (st env) (mkHeap new_contents).

Lemma update_array_contents_correct :
  ∀ env av i n,
    array_contents (he (update_array_contents env av i n)) av i = n.
Proof.
```

```
    intros; unfold update_array_contents; simpl.
    red_exprs; crush.
Qed.

Lemma update_array_contents_same :
  ∀ env av1 i1 av2 i2 n,
    av1 ≠ av2 → array_contents (he (update_array_contents env av1 i1 n)) av2 i2 = ar-
ray_contents (he env) av2 i2.
Proof.
    intros; unfold update_array_contents; simpl.
    red_exprs ; crush.
Qed.

Lemma update_array_contents_stack :
  ∀ env idx n av v,
      st (update_array_contents env av idx n) v = st env v.
Proof.
 intros.
 unfold update_array_contents. crush.
Qed.

Hint Rewrite update_array_contents_stack : cpdt.
Hint Rewrite update_array_contents_same : cpdt.

Lemma update_array_contents_equiv :
  ∀ env v vars,
    ¬ In (inr _ v) vars →
      ∀ idx n, env_equiv env (update_array_contents env v idx n) vars.
Proof.
  unfold_equivs; crush ; red_exprs; crush.
Qed.

Definition iop_denote (i : iop) : nat → nat → nat :=
  match i with
```

```
    | Plus ⇒ plus
    | Minus ⇒ minus
    | Times ⇒ mult
  end.

Fixpoint exp_eval (e : exp) (env : environment) : nat :=
  match e with
  | Const n ⇒ n
  | Var v ⇒ st env v
  | Arr av e ⇒
    let idx := (exp_eval e env) in
      array_contents (he env) av idx
  | Iop op e1 e2 ⇒ (iop_denote op) (exp_eval e1 env) (exp_eval e2 env)
  end.

Lemma exp_eval_determ : ∀ e s n m, exp_eval e s = n → exp_eval e s = m → n = m.
Proof.
  destruct e; simpl; congruence.
Qed.

Lemma env_equiv_update :
  ∀ env env' a vars ,
    env_equiv env env' (remove var_arr_var_eq_dec (inl _ a) vars) →
    env_equiv env (update_variable env' a (st env a)) vars.
Proof.
  intros.
  unfold_equivs. simpl.
  destruct_conjs; split.
    intros. destruct (var_eq_dec a v).
      subst; rewrite assign_state_imd; crush.
      rewrite assign_state_same; crush; apply H; crush; apply In_remove; crush.
      crush ; apply H0; apply In_remove; crush.
```

```
Qed.

Require Import Coq.Bool.Sumbool.

Require Import Coq.Arith.EqNat.

Program Definition cmp_denote (op : cmp) : nat → nat → bool :=
  match op with
    | LT ⇒ fun x y ⇒ bool_of_sumbool (lt_dec x y)
    | LEQ ⇒ fun x y ⇒ bool_of_sumbool (le_dec x y)
    | EQ ⇒ fun x y ⇒ bool_of_sumbool (eq_nat_dec x y)
  end.

Definition cmp_reflect (op : cmp) : nat → nat → Prop :=
 match op with
    | LT ⇒ lt
    | LEQ ⇒ le
    | EQ ⇒ eq
 end.

Definition cmp_reflect_neg (op : cmp) : nat → nat → Prop :=
 match op with
    | LT ⇒ ge
    | LEQ ⇒ gt
    | EQ ⇒ fun x y ⇒ x ≠ y
 end.

Definition lop_denote (op : lop) : bool → bool → bool :=
  match op with
    | And ⇒ andb
    | Or ⇒ orb
  end.

Definition lop_reflect (op : lop) : Prop → Prop → Prop :=
  match op with
  | And ⇒ and
```

```
  | Or ⇒ or
  end
.

Definition lop_reflect_neg (op : lop) : Prop → Prop → Prop :=
  match op with
  | And ⇒ or
  | Or ⇒ and
  end.

Fixpoint bexp_eval (be : bexp) (environment : environment) : bool :=
  match be with
  | BConst b ⇒ b
  | Cmp op e1 e2 ⇒ (cmp_denote op) (exp_eval e1 environment) (exp_eval e2 environment)
  | Lop op be1 be2 ⇒ (lop_denote op) (bexp_eval be1 environment) (bexp_eval be2 environment)
  | Neg be ⇒ negb (bexp_eval be environment)
  end
.

Lemma exp_eval_dec : ∀ (e : exp) en (n : nat), {exp_eval e en = n} + {exp_eval e en ≠ n}.
Proof.
  decide equality.
Qed.

Lemma bexp_eval_dec : ∀ (be : bexp) en ( b : bool), {bexp_eval be en = b} + {bexp_eval be en ≠ b}.
Proof.
  decide equality.
Qed.

Fixpoint rexp_eval (re : rexp) (s_o s_r : environment): nat :=
  match re with
```

```
  | RConst n ⇒ n
  | RVar rv ⇒
    match rv with
    | Org v ⇒ (st s_o v)
    | Rel v ⇒ (st s_r v)
    end
  | RIop op re1 re2 ⇒ (iop_denote op) (rexp_eval re1 s_o s_r) (rexp_eval re2 s_o s_r)
  | RArr (OrgArr av) e ⇒ array_contents (he s_o) av (rexp_eval e s_o s_r)
  | RArr (RelArr av) e ⇒ array_contents (he s_r) av (rexp_eval e s_o s_r)
  end.

Fixpoint rbexp_eval (rbe : rbexp) (s_o s_r : environment) : bool :=
  match rbe with
  | RBConst b ⇒ b
  | RCmp op re1 re2 ⇒ (cmp_denote op) (rexp_eval re1 s_o s_r) (rexp_eval re2 s_o s_r)
  | RLop op rbe1 rbe2 ⇒ (lop_denote op) (rbexp_eval rbe1 s_o s_r) (rbexp_eval rbe2 s_o
s_r)
  | RNeg rbe ⇒ negb (rbexp_eval rbe s_o s_r)
  end
.


Section language2.

End language2.
```

A.3 Statements

```
Require Import Expressions.

Definition accept_id := nat.

Inductive statement : Type :=
  | Skip : statement
```

| Assign : **var** → **exp** → **statement**

| AssignArr : **arr_var** → **exp** → **exp** → **statement**

| Havoc : list **var** → **bexp** → **statement**

| Relax : list **var** → **bexp** → **statement**

| If : **bexp** → **statement** → **statement** → **statement**

| While : **bexp** → **statement** → **statement**

| Assume : list **var** → **bexp** → **statement**

| Assert : **bexp** → **statement**

| Accept : accept_id → **rbexp** → **statement**

| Seq : **statement** → **statement** → **statement**

.

```
Require Import List.
```

```
Fixpoint accept_free (st : statement) : Prop :=
  match st with
  | Skip ⇒ True
  | Assign _ _ ⇒ True
  | AssignArr _ _ _ ⇒ True
  | Havoc _ _ ⇒ True
  | Relax _ _ ⇒ True
  | If _ st1 st2 ⇒ (accept_free st1) ∧ (accept_free st2)
  | While _ st ⇒ accept_free st
  | Assume _ _ ⇒ True
  | Assert _ ⇒ True
  | Accept _ _ ⇒ False
  | Seq st1 st2 ⇒ (accept_free st1) ∧ (accept_free st2)
  end.
```

A.4 OriginalDynamic

A.5 Dynamic Original Semantcs

```
Require Import Language.
Require Import List.
Require Import Coq.Program.Equality.
Require Import Coq.Relations.Relation_Operators.
Require Import Coq.Relations.Operators_Properties.
Require Import CpdtTactics.
Definition obs_list := list (accept_id * environment).
```

An Input Configuration

```
Inductive iconfig :=
 iconfig_intro : statement → environment → iconfig.
Notation "<I st , env I>" := (iconfig_intro st env).
```

An Output Configuration `Inductive` **oconfig** :=

```
  | oconfig_good : environment → obs_list → oconfig
```

Wrong configuration `| wr : oconfig`

Bad assume configuration `| ba : oconfig`

.

```
Notation "<# env , ol #>" := (oconfig_good env ol).
```

Definition of error predicate `Inductive` **error** : **oconfig** → `Prop` :=

```
  | error_wr : error wr
  | error_ba : error ba
.
Lemma error_valid_config_false :
  ∀ s ol, error <# s, ol #> → False .
Proof.
    inversion 1.
```

```
Qed.

Ltac error_good_config :=
  match goal with
      | [ H : error <# _ , _ #> ⊢ _ ] ⇒ inversion H
      end.
```

A subsequent state of is valid if for all unmentioned variables, the values are the same and the new state satifies the havoc condition `Definition` havoc_sat (*vars* : list **var**) *be env env'* :=

(∀ *vars'*, (∀ *v*, ln *v vars* → ¬ ln (inl _ *v*) *vars'*) → env_equiv *env env' vars'*) ∧ (bexp_eval *be env'* = true).

```
Inductive original_eval_big : iconfig → oconfig → Prop :=

  | oeb_skip : ∀ s, original_eval_big <| Skip, s |> <# s, nil #>

  | oeb_assign : ∀ s v e n,
      exp_eval e s = n →
      original_eval_big <| Assign v e, s |> <# update_variable s v n, nil #>

  | oeb_assign_arr : ∀ env av e1 e2 n idx,
      exp_eval e2 env = n →
      exp_eval e1 env = idx →
      original_eval_big <| AssignArr av e1 e2 , env |> <# update_array_contents env
av idx n, nil #>

  | oeb_havoc_true : ∀ vars be s s',
      havoc_sat vars be s s' →
      original_eval_big <| Havoc vars be, s |> <# s', nil #>

  | oeb_havoc_false : ∀ vars be s,
```

(¬ ∃ *s'*, havoc_sat *vars be s s'*) →

original_eval_big <| Havoc *vars be*, *s* |> wr

| obe_relax : ∀ *s vars be o*,

original_eval_big <| Assert *be*, *s* |> *o* →

original_eval_big <| Relax *vars be*, *s* |> *o*

| oeb_assert_true : ∀ *be s*,

bexp_eval *be s* = true →

original_eval_big <| Assert *be*, *s* |> <# *s*, nil #>

| oeb_assert_false : ∀ *be s*,

bexp_eval *be s* = false →

original_eval_big <| Assert *be*, *s* |> wr

| oeb_assume_true : ∀ *be vars s*,

bexp_eval *be s* = true →

original_eval_big <| Assume *vars be*, *s* |> <# *s*, nil #>

| oeb_assume_false : ∀ *be vars s*,

bexp_eval *be s* = false →

original_eval_big <| Assume *vars be*, *s* |> ba

| oeb_accept : ∀ *re aid env*,

original_eval_big <| Accept *aid re*, *env* |> <# *env*, (*aid*, *env*) : : nil #>

| oeb_if_true : ∀ *s be st1 st2 o*,

bexp_eval *be s* = true →

original_eval_big <| *st1*, *s* |> *o* →

original_eval_big <| If *be st1 st2*, *s* |> *o*

| oeb_if_false : ∀ *s be st1 st2 o*,

bexp_eval *be s* = false →

original_eval_big <| *st2* , *s* |> *o* →

original_eval_big <| If *be st1 st2* , *s* |> *o*

| oeb_seq : ∀ *st1 st2 s s' s" ol_1 ol_2*,

original_eval_big <| *st1* , *s* |> <# *s'* , *ol_1* #> →

original_eval_big <| *st2* , *s'* |> <# *s"* , *ol_2* #> →

original_eval_big <| Seq *st1 st2* , *s* |> <# *s"* , *ol_2* ++ *ol_1* #>

| oeb_seq_bad1 : ∀ *st1 st2 s o*,

original_eval_big <| *st1* , *s* |> *o* →

error *o* →

original_eval_big <| Seq *st1 st2* , *s* |> *o*

| oeb_seq_bad2 : ∀ *st1 st2 s s' ol_1 o*,

original_eval_big <| *st1* , *s* |> <# *s'* , *ol_1* #> →

original_eval_big <| *st2* , *s'* |> *o* →

error *o* →

original_eval_big <| Seq *st1 st2* , *s* |> *o*

| oeb_while_false : ∀ *be st s*,

bexp_eval *be s* = false →

original_eval_big <| While *be st* , *s* |> <# *s* , nil #>

| oeb_while_true : ∀ *be st s s' s" ol_1 ol_2*,

bexp_eval *be s* = true →

original_eval_big <| *st* , *s* |> <# *s'* , *ol_1* #> →

original_eval_big <| While *be st* , *s'* |> <# *s"* , *ol_2* #> →

original_eval_big <| While *be st*, *s* |> <# *s''*, *ol_2* ++ *ol_1* #>

| oeb_while_bad1 : ∀ *be st s o*,
bexp_eval *be s* = true →
original_eval_big <| *st*, *s* |> *o* →
error *o* →
original_eval_big <| While *be st*, *s* |> *o*

| oeb_while_bad2 : ∀ *be st s s' ol o*,
bexp_eval *be s* = true →
original_eval_big <| *st*, *s*|> <# *s'*, *ol*#> →
original_eval_big <| While *be st*, *s'* |> *o* →
error *o* →
original_eval_big <| While *be st* , *s* |> *o*

.

```
Ltac invert_original_big :=
  match goal with
  | [ H : original_eval_big _ _ ⊢ _ ] ⇒ inversion H; subst; clear H
  end.

Fixpoint original_mods (st : statement) : list (var + arr_var) :=
  match st with
  | Skip ⇒ nil
  | Assign v _ ⇒ (inl _ v) : : nil
  | AssignArr av e1 e2 ⇒ (inr _ av) : : nil
  | Havoc vars _ ⇒ map (fun v ⇒ inl _ v) vars
  | Relax _ _ ⇒ nil
  | If be st1 st2 ⇒ (original_mods st1) ++ (original_mods st2)
  | While be st ⇒ (original_mods st)
  | Assume _ _ ⇒ nil
  | Assert _ ⇒ nil
```

```
  | Accept _ _ ⇒ nil
  | Seq st1 st2 ⇒ (original_mods st1) ++ (original_mods st2)
  end.

Hint Resolve update_variable_equiv.

Hint Resolve update_array_contents_equiv.

Hint Resolve env_equiv_refl.

Hint Resolve error_valid_config_false.

Hint Resolve in_map.

Hint Resolve in_or_app.

Lemma original_eval_not_in_mods_constant :
  ∀ st env env' ol', original_eval_big <| st, env |> <#env', ol' #> →
    ∀ vars, (∀ v, In v vars → ¬ In v (original_mods st)) →
      env_equiv env env' vars.

Proof.
  induction st;
  try solve[intros; invert_original_big ; unfold havoc_sat in *; crush; eauto].

  crush; invert_original_big; invert_original_big; crush.

 simpl; intros; dependent induction H; red_exprs; crush; try error_good_config;
   eapply env_equiv_trans; [ eapply IHst; eauto | eapply IHoriginal_eval_big2; eauto].

 inversion 1; crush; try error_good_config ; eapply env_equiv_trans; [ eapply IHst1;
eauto | eapply IHst2; eauto].

Qed.
```

A.6 RelaxedDynamic

```
Require Import Language.

Require Import List.

Require Export OriginalDynamic.

Require Import Classical.
```

```
Require Import Coq.Program.Tactics.
Require Import util.
Require Import Coq.Program.Equality.
Require Import CpdtTactics.

Inductive relaxed_eval_big : iconfig → oconfig → Prop :=

  | rd_skip : ∀ s o,
    original_eval_big <| Skip, s |> o →
    relaxed_eval_big <| Skip, s |> o

  | rd_assign : ∀ s v e o,
      original_eval_big <| Assign v e, s |> o →
      relaxed_eval_big <| Assign v e, s |> o

  | rd_assign_arr : ∀ env av e1 e2 o,

      original_eval_big <| AssignArr av e1 e2 , env |> o →
      relaxed_eval_big <| AssignArr av e1 e2 , env |> o

  | rd_havoc : ∀ vars be s o,
      original_eval_big <| Havoc vars be, s |> o →
      relaxed_eval_big <| Havoc vars be, s |> o

  | rd_relax : ∀ s vars be o,
      relaxed_eval_big <| Havoc vars be, s |> o →
      relaxed_eval_big <| Relax vars be, s |> o

  | rd_assert : ∀ be s o,
      original_eval_big <| Assert be, s |> o →
      relaxed_eval_big <| Assert be, s |> o
```

| rd_assume : ∀ *be vars s o*,

original_eval_big <| Assume *vars be*, *s* |> *o* →

relaxed_eval_big <| Assume *vars be*, *s* |> *o*

| rd_accept : ∀ *re aid env o*,

original_eval_big <| Accept *aid re*, *env* |> *o* →

relaxed_eval_big <| Accept *aid re*, *env* |> *o*

| rd_if_true : ∀ *s be st1 st2 o*,

bexp_eval *be s* = true →

relaxed_eval_big <| *st1*, *s* |> *o* →

relaxed_eval_big <| If *be st1 st2*, *s* |> *o*

| rd_if_false : ∀ *s be st1 st2 o*,

bexp_eval *be s* = false →

relaxed_eval_big <| *st2*, *s* |> *o* →

relaxed_eval_big <| If *be st1 st2*, *s* |> *o*

| rd_seq : ∀ *st1 st2 s s' s'' ol_1 ol_2*,

relaxed_eval_big <| *st1*, *s* |> <# *s'*, *ol_1* #> →

relaxed_eval_big <| *st2*, *s'* |> <# *s''*, *ol_2* #> →

relaxed_eval_big <| Seq *st1 st2*, *s* |> <# *s''*, *ol_2* ++ *ol_1* #>

| rd_seq_bad1 : ∀ *st1 st2 s o*,

relaxed_eval_big <| *st1*, *s* |> *o* →

error *o* →

relaxed_eval_big <| Seq *st1 st2*, *s* |> *o*

| rd_seq_bad2 : ∀ *st1 st2 s s' ol_1 o*,

relaxed_eval_big <| *st1*, *s* |> <# *s'*, *ol_1* #> →

relaxed_eval_big <| *st2*, *s'* |> *o* →

error *o* →

relaxed_eval_big <| Seq *st1 st2*, *s* |> *o*

| rd_while_false : ∀ *be st s*,

bexp_eval *be s* = false →

relaxed_eval_big <| While *be st*, *s* |> <# *s*, nil #>

| rd_while_true : ∀ *be st s s' s" ol_1 ol_2*,

bexp_eval *be s* = true →

relaxed_eval_big <| *st*, *s* |> <# *s'*, *ol_1* #> →

relaxed_eval_big <| While *be st*, *s'* |> <# *s"*, *ol_2* #> →

relaxed_eval_big <| While *be st*, *s* |> <# *s"*, *ol_2* ++ *ol_1* #>

| rd_while_bad1 : ∀ *be st s o*,

bexp_eval *be s* = true →

relaxed_eval_big <| *st*, *s* |> *o* →

error *o* →

relaxed_eval_big <| While *be st*, *s* |> *o*

| rd_while_bad2 : ∀ *be st s s' ol o*,

bexp_eval *be s* = true →

relaxed_eval_big <| *st*, *s*|> <# *s'*, *ol*#> →

relaxed_eval_big <| While *be st*, *s'* |> *o* →

error *o* →

relaxed_eval_big <| While *be st* , *s* |> *o*

.

`Hint` *Constructors relaxed_eval_big.*

`Hint` *Constructors original_eval_big.*

```
Fixpoint relaxed_mods (st : statement) : list (var + arr_var) :=
  match st with
  | Skip ⇒ nil
  | Assign v _ ⇒ (inl _ v) : : nil
  | AssignArr av e1 e2 ⇒ (inr _ av) : : nil
  | Havoc vars _ ⇒ map (fun v ⇒ inl _ v) vars
  | Relax vars _ ⇒ map (fun v ⇒ inl _ v) vars
  | If be st1 st2 ⇒ (relaxed_mods st1) ++ (relaxed_mods st2)
  | While be st ⇒ (relaxed_mods st)
  | Assume _ _ ⇒ nil
  | Assert _ ⇒ nil
  | Accept _ _ ⇒ nil
  | Seq st1 st2 ⇒ (relaxed_mods st1) ++ (relaxed_mods st2)
  end.

Ltac invert_relaxed :=
  match goal with
  | [ H : relaxed_eval_big _ _ ⊢ _] ⇒ inversion H; subst
  end.

Section relaxed_eval.

Hint Resolve env_equiv_refl.
Hint Resolve update_variable_equiv.
Hint Resolve update_array_contents_equiv.
Hint Resolve in_map.
Hint Resolve in_or_app.

Lemma relaxed_eval_not_in_mods_constant :
  ∀ st env env' ol', relaxed_eval_big <| st, env |> <#env', ol' #> →
    ∀ vars, (∀ v, In v vars → ¬ In v (relaxed_mods st)) →
      env_equiv env env' vars.
Proof.
```

induction *st*;

try *solve*[intros; *invert_relaxed* ; try *invert_original_big* ; unfold *havoc_sat* in *; *crush*; eauto].

crush; *invert_relaxed*. inversion *H5*; *crush*. *invert_original_big*. unfold *havoc_sat* in *; *crush*. eauto.

simpl; intros; dependent induction *H*; *crush*; try *error_good_config*;

eapply env_equiv_trans; [eapply *IHst*; eauto | eapply *IHrelaxed_eval_big2*; eauto].

inversion 1; *crush*; try *error_good_config* ; eapply env_equiv_trans; [eapply *IHst1*; eauto | eapply *IHst2*; eauto].

Qed.

End relaxed_eval.

A.7 AssertionLogic

Require Export Language.

Require Import util.

Require Import Classical.

Require Import Coq.Bool.Bool.

Require Import Coq.Program.Tactics.

Require Import Tactics.

Require Import CpdtTactics.

Require Import Coq.Logic.FunctionalExtensionality.

Inductive **bvar** : Set :=

| BId : nat → **bvar**.

Definition bvar_eq_dec: ∀ (*v1 v2*: **bvar**), {*v1*=*v2*} + {*v1*≠*v2*}.

Proof.

decide equality.

decide equality.

```
Qed.

Inductive exp : Set :=
  | Const : nat → exp
  | BLVar : bvar → exp
  | BRVar : bvar → exp
  | LVar : var → exp
  | RVar : var → exp
  | Arr : ref_exp → exp → exp
  | Iop : iop → exp → exp → exp

with ref_exp : Set :=
  | LArrVar : arr_var → ref_exp
  | RArrVar : arr_var → ref_exp
  | ArrUpdate : ref_exp → exp → exp → ref_exp
.

Scheme exp_ind' := Induction for exp Sort Prop
  with ref_exp_ind' := Induction for ref_exp Sort Prop.

Combined Scheme exp_ref_exp_ind from exp_ind', ref_exp_ind'.

Inductive selector : Set :=
  | Left : selector
  | Right : selector
.

Fixpoint exp_inj (e : Expressions.exp) (s : selector) : exp :=
  match e with
  | Expressions.Var v ⇒
    match s with
    | Left ⇒ LVar v
    | Right ⇒ RVar v
    end
```

```
  | Expressions.Const n ⇒ Const n
  | Expressions.Arr av e1 ⇒
      match s with
      | Left ⇒ Arr (LArrVar av) (exp_inj e1 s)
      | Right ⇒ Arr (RArrVar av) (exp_inj e1 s)
      end
  | Expressions.Iop op e1 e2 ⇒ Iop op (exp_inj e1 s) (exp_inj e2 s)
end.

Notation "@ e '<o>'" := (exp_inj e Left) (at level 0).

Section assertion_logic.

Fixpoint free_vars (e : exp) (s : selector) : list (var + arr_var) :=
  match e with
  | Const _ ⇒ nil
  | BLVar _ ⇒ nil
  | BRVar v ⇒ nil
  | LVar v ⇒
      match s with
      | Left ⇒ (inl _ v) : : nil
      | Right ⇒ nil
      end
  | RVar v ⇒
    match s with
    | Left ⇒ nil
    | Right ⇒ (inl _ v) : : nil
    end
  | Arr ref e ⇒ (ref_exp_free_vars ref s) ++ (free_vars e s)
  | Iop op e1 e2 ⇒ (free_vars e1 s) ++ (free_vars e2 s)
  end
with ref_exp_free_vars ref (s : selector) : list (var + arr_var) :=
```

```
  match ref with
  | LArrVar av ⇒ match s with Left ⇒ (inr _ av) : : nil | Right ⇒ nil end
  | RArrVar av ⇒ match s with Right ⇒ (inr _ av) : : nil | Left ⇒ nil end
  | ArrUpdate ref' e1 e2 ⇒ (ref_exp_free_vars ref' s) ++ (free_vars e1 s) ++ (free_vars
e2 s)
  end
.

Fixpoint exp_graft (e : exp) (x : var) (s : selector) (e' : exp) : exp :=
  match e with
    | Const _ ⇒ e

    | LVar v ⇒
      match s with
      | Left ⇒ if var_eq_dec v x then e' else e
      | Right ⇒ e
      end

   | RVar v ⇒
      match s with
      | Left ⇒ e
      | Right ⇒ if var_eq_dec v x then e' else e
      end

    | BLVar v ⇒ e
    | BRVar v ⇒ e
    | Arr ref ei ⇒ Arr (ref_exp_graft ref x s e') (exp_graft ei x s e')
    | Iop i e1 e2 ⇒ Iop i (exp_graft e1 x s e') (exp_graft e2 x s e')
  end
with ref_exp_graft (ref : ref_exp) (x : var) (s : selector) (e : exp) : ref_exp :=
  match ref with
```

```
  | LArrVar _ ⇒ ref
  | RArrVar _ ⇒ ref
  | ArrUpdate ref' e1 e2 ⇒ ArrUpdate (ref_exp_graft ref' x s e) (exp_graft e1 x s e)
(exp_graft e2 x s e)
  end
.

Fixpoint exp_graft_arr (e : exp) (x : arr_var) (s : selector) (ref : ref_exp) : exp :=
  match e with
    | Const _ ⇒ e
    | LVar v ⇒ e
    | RVar v ⇒ e
    | BLVar v ⇒ e
    | BRVar v ⇒ e
    | Arr r ei ⇒ Arr (ref_exp_graft_arr r x s ref) (exp_graft_arr ei x s ref)
    | Iop i e1 e2 ⇒ Iop i (exp_graft_arr e1 x s ref) (exp_graft_arr e2 x s ref)
  end
with ref_exp_graft_arr (ref : ref_exp) (x : arr_var) (s : selector) (ref' : ref_exp) :
ref_exp :=
  match ref with
  | LArrVar av ⇒
    match s with
    | Left ⇒ if arr_var_eq_dec av x then ref' else ref
    | Right ⇒ ref
    end
  | RArrVar av ⇒
    match s with
    | Left ⇒ ref
    | Right ⇒ if arr_var_eq_dec av x then ref' else ref
    end
  | ArrUpdate ref1 e1 e2 ⇒ ArrUpdate (ref_exp_graft_arr ref1 x s ref') (exp_graft_arr e1
```

```
x s ref') (exp_graft_arr e2 x s ref')
  end.

Fixpoint exp_subst (e : exp) (x : bvar) (s : selector) (n : nat) : exp :=
  match e with
  | Const _ ⇒ e
  | LVar _ ⇒ e
  | RVar _ ⇒ e
  | BLVar v ⇒
    match s with
    | Left ⇒ if bvar_eq_dec v x then (Const n) else e
    | Right ⇒ e
    end
  | BRVar v ⇒
    match s with
    | Left ⇒ e
    | Right ⇒ if bvar_eq_dec v x then (Const n) else e
    end
  | Arr ref ei ⇒ Arr (ref_exp_subst ref x s n) (exp_subst ei x s n)
  | Iop i e1 e2 ⇒ Iop i (exp_subst e1 x s n) (exp_subst e2 x s n)
  end
with ref_exp_subst (ref : ref_exp) (x : bvar) (s : selector) (n : nat) : ref_exp :=
  match ref with
  | LArrVar _ ⇒ ref
  | RArrVar _ ⇒ ref
  | ArrUpdate ref' e1 e2 ⇒ ArrUpdate (ref_exp_subst ref' x s n) (exp_subst e1 x s n)
(exp_subst e2 x s n)
  end.

Fixpoint ref_exp_subst_arr (ref : ref_exp) (x : arr_var) (s : selector) (ref' : ref_exp)
: ref_exp :=
  match ref with
```

```
  | LArrVar _ ⇒ ref
  | RArrVar _ ⇒ ref
  | ArrUpdate r e1 e2 ⇒ ArrUpdate (ref_exp_subst_arr r x s ref') (exp_subst_arr e1 x s
ref') (exp_subst_arr e2 x s ref')
  end

with exp_subst_arr (e : exp) (x : arr_var) (s : selector) (ref : ref_exp) : exp :=
  match e with
  | Const _ ⇒ e
  | LVar v ⇒ e
  | RVar v ⇒ e
  | BLVar v ⇒ e
  | BRVar v ⇒ e
  | Arr r ei ⇒ Arr (ref_exp_subst_arr r x s ref) (exp_subst_arr ei x s ref)
  | Iop i e1 e2 ⇒ Iop i (exp_subst_arr e1 x s ref) (exp_subst_arr e2 x s ref)
  end.
Definition bvar_to_var (b : bvar) : var :=
  match b with
  | (BId n) ⇒ (Id n)
 end.
Inductive exp_denote : exp → environment → environment → nat → Prop :=

  | exp_denote_const : ∀ n l_env r_env, exp_denote (Const n) l_env r_env n

  | exp_denote_lvar : ∀ v l_env n, st l_env v = n → ∀ r_env, exp_denote (LVar v) l_env
r_env n

  | exp_denote_rvar : ∀ v r_env n, st r_env v = n → ∀ l_env, exp_denote (RVar v) l_env
r_env n
```

| exp_denote_arr :

∀ *e l_env r_env idx f*, **exp_denote** *e l_env r_env idx* →

∀ *ref*, **ref_exp_denote** *ref l_env r_env f* →

exp_denote (Arr *ref e*) *l_env r_env* (*f idx*)

| exp_denote_iop : ∀ *l_env r_env*,

∀ *e1 n1*, **exp_denote** *e1 l_env r_env n1* →

∀ *e2 n2*, **exp_denote** *e2 l_env r_env n2* →

∀ *op*, **exp_denote** (lop *op e1 e2*) *l_env r_env* (iop_denote *op n1 n2*)

`with` **ref_exp_denote** : **ref_exp** → **environment** → **environment** → (nat → nat) → `Prop :=`

| ref_exp_denote_larrvar : ∀ *av l_env f*,

array_contents (he *l_env*) *av* = *f* → ∀ *r_env*, **ref_exp_denote** (LArrVar *av*) *l_env r_env f*

| ref_exp_denote_rarrvar : ∀ *av r_env f*,

array_contents (he *r_env*) *av* = *f* → ∀ *l_env*, **ref_exp_denote** (RArrVar *av*) *l_env r_env f*

| ref_exp_denote_rupdate : ∀ *ref l_env r_env f*,

ref_exp_denote *ref l_env r_env f* →

∀ *e1 n1*, **exp_denote** *e1 l_env r_env n1* →

∀ *e2 n2*, **exp_denote** *e2 l_env r_env n2* →

ref_exp_denote (ArrUpdate *ref e1 e2*) *l_env r_env* (`fun` *idx* ⇒ `if` eq_nat_dec *n1 idx* `then` *n2* `else` *f idx*).

`Scheme` *exp_denote_mut* := `Induction for` *exp_denote* `Sort Prop`

`with` *ref_exp_denote_mut* := `Induction for` *ref_exp_denote* `Sort Prop.`

Combined `Scheme` *exp_ref_exp_denote_ind from exp_denote_mut, ref_exp_denote_mut.*

```
Hint Constructors exp_denote.
Hint Constructors ref_exp_denote.
Lemma exp_denote_determ_combined :
  (∀ e l_env r_env,
    ∀ n1, exp_denote e l_env r_env n1 →
    ∀ n2, exp_denote e l_env r_env n2 → n1 = n2)
  ∧
  (∀ ref l_env r_env,
    ∀ f1, ref_exp_denote ref l_env r_env f1 →
    ∀ f2, ref_exp_denote ref l_env r_env f2 → f1 = f2)
.
Proof.
  apply (exp_ref_exp_ind _ _) ;
   try solve [
     inversion 1; inversion 1; crush].
    inversion 3. inversion 1. crush.
    cut (f = f0).
  crush. apply f_equal; eauto. eauto.
   inversion 3. inversion 1. crush. eauto.
  intros until 1. intros until 1. intros until 1.
  inversion 1. inversion 1. crush. cut (n0 = n1). cut (n2 = n3). cut (f = f0). crush.
   eauto. eauto. eauto.
Qed.
Definition exp_denote_determ := (proj1 exp_denote_determ_combined).
Definition ref_exp_denote_determ := (proj2 exp_denote_determ_combined).
Definition select (A : Type) (s : selector) (a1 a2 : A) :=
  match s with
  | Left ⇒ a1
  | Right ⇒ a2
```

```
  end.

Lemma exp_denote_sound :
 ∀ (e : Expressions.exp) (s : selector) env1 env2 n,
    exp_denote (exp_inj e s) env1 env2 n →
      ∀ env, (select _ s env1 env2) = env →
        exp_eval e env = n.
Proof.
  induction e; destruct s;
   simpl; inversion 1; crush; eauto.

  inversion H6; crush; eauto.
  inversion H6; crush; eauto.
Qed.

Lemma exp_denote_sound_left (e : Expressions.exp) :
 ∀ (env1 env2 : environment) (n : nat),
      exp_denote @ e <o> env1 env2 n → exp_eval e env1 = n.
Proof.
  intros; eapply exp_denote_sound; eauto; crush.
Qed.

Lemma exp_denote_sound_right (e : Expressions.exp) :
 ∀ (env1 env2 : environment) (n : nat),
      exp_denote (exp_inj e Right) env1 env2 n → exp_eval e env2 = n.
Proof.
  intros; eapply exp_denote_sound; eauto; crush.
Qed.

Lemma exp_denote_complete_left :
  ∀ (e : Expressions.exp) env1 n,
    exp_eval e env1 = n → ∀ env2, exp_denote @e<o> env1 env2 n.
Proof.
  Hint Constructors exp_denote.
```

```
  Hint Constructors ref_exp_denote.
  induction e; crush; eauto.
Qed.

Lemma exp_denote_complete_right :
  ∀ (e : Expressions.exp) env2 n,
    exp_eval e env2 = n → ∀ env1, exp_denote (exp_inj e Right) env1 env2 n.
Proof.
  Hint Constructors exp_denote.
  Hint Constructors ref_exp_denote.
  induction e; crush; eauto.

Qed.

Inductive pred : Type :=
  | PTrue : pred
  | PFalse : pred
  | PCmp : cmp → exp → exp → pred
  | PLop : lop → pred → pred → pred
  | PNeg : pred → pred
  | PLExists : bvar → pred → pred
  | PRExists : bvar → pred → pred
  | PLForall : bvar→ pred → pred
  | PRForall : bvar→ pred → pred
.

Fixpoint bexp_inj (be : Expressions.bexp) (s : selector) : pred :=
  match be with
  | BConst true ⇒ PTrue
  | BConst false ⇒ PFalse
  | Cmp op e1 e2 ⇒ PCmp op (exp_inj e1 s) (exp_inj e2 s)
  | Lop op be1 be2 ⇒ PLop op (bexp_inj be1 s) (bexp_inj be2 s)
  | Neg be' ⇒ PNeg (bexp_inj be' s)
```

```
 end.

Program Fixpoint pred_free_vars (p : pred) (s : selector) : list (var+arr_var) :=
  match p with
  | PTrue | PFalse ⇒ nil
  | PCmp op e1 e2 ⇒ (free_vars e1 s) ++ (free_vars e2 s)
  | PLop op p1 p2 ⇒ (pred_free_vars p1 s) ++ (pred_free_vars p2 s)
  | PNeg p ⇒ pred_free_vars p s
  | PLExists x p' ⇒ pred_free_vars p' s
  | PRExists x p' ⇒ pred_free_vars p' s
  | PLForall x p' ⇒ pred_free_vars p' s
  | PRForall x p' ⇒ pred_free_vars p' s




  end
.

Fixpoint pred_graft (p : pred) (x : var) (s : selector) (e : exp) : pred :=
  match p with
  | PTrue | PFalse ⇒ p
  | PCmp op e1 e2 ⇒ PCmp op (exp_graft e1 x s e) (exp_graft e2 x s e)
  | PLop op p1 p2 ⇒ PLop op (pred_graft p1 x s e) (pred_graft p2 x s e)
  | PNeg p' ⇒ PNeg (pred_graft p' x s e)
  | PLExists z p' ⇒ PLExists z (pred_graft p' x s e)
  | PRExists z p' ⇒ PRExists z (pred_graft p' x s e)
  | PLForall z p' ⇒ PLForall z (pred_graft p' x s e)
  | PRForall z p' ⇒ PRForall z (pred_graft p' x s e)
   end.

Fixpoint pred_graft_arr (p : pred) (x : arr_var) (s : selector) (r : ref_exp) : pred :=
  match p with
```

```
  | PTrue | PFalse ⇒ p
  | PCmp op e1 e2 ⇒ PCmp op (exp_graft_arr e1 x s r) (exp_graft_arr e2 x s r)
  | PLop op p1 p2 ⇒ PLop op (pred_graft_arr p1 x s r) (pred_graft_arr p2 x s r)
  | PNeg p' ⇒ PNeg (pred_graft_arr p' x s r)
  | PLExists z p' ⇒ PLExists z (pred_graft_arr p' x s r)
  | PRExists z p' ⇒ PRExists z (pred_graft_arr p' x s r)
  | PLForall z p' ⇒ PLForall z (pred_graft_arr p' x s r)
  | PRForall z p' ⇒ PRForall z (pred_graft_arr p' x s r)
  end.

Fixpoint pred_subst_var (p : pred) (x : bvar) (s : selector) (n : nat) : pred :=
  match p with
  | PTrue | PFalse ⇒ p
  | PCmp op e1 e2 ⇒ PCmp op (exp_subst e1 x s n) (exp_subst e2 x s n)
  | PLop op p1 p2 ⇒ PLop op (pred_subst_var p1 x s n) (pred_subst_var p2 x s n)
  | PNeg p' ⇒ PNeg (pred_subst_var p' x s n)
  | PLExists z p' ⇒
    match s with
    | Left ⇒
        if bvar_eq_dec x z then
          p
        else
          PLExists z (pred_subst_var p' x s n)
    | Right ⇒ PLExists z (pred_subst_var p' x s n)
    end
  | PRExists z p' ⇒
    match s with
    | Left ⇒ PRExists z (pred_subst_var p' x s n)
    | Right ⇒
        if bvar_eq_dec x z then
          p
```

```
        else
          PRExists z (pred_subst_var p' x s n)
      end

| PLForall z p' ⇒
      match s with
      | Left ⇒
          if bvar_eq_dec x z then
            p
          else
            PLForall z (pred_subst_var p' x s n)
      | Right ⇒ PLForall z (pred_subst_var p' x s n)
      end
  | PRForall z p' ⇒
      match s with
      | Left ⇒ PRForall z (pred_subst_var p' x s n)
      | Right ⇒
          if bvar_eq_dec x z then
            p
          else
            PRForall z (pred_subst_var p' x s n)
      end
  end.

Inductive pred_denote : pred → environment → environment → Prop :=

  | pd_true : ∀ l_env r_env, pred_denote PTrue l_env r_env

  | pd_cmp : ∀ op e1 e2 l_env r_env,
      ∀ n1, exp_denote e1 l_env r_env n1 →
      ∀ n2, exp_denote e2 l_env r_env n2 → (cmp_reflect op) n1 n2 →
```

pred_denote (PCmp *op e1 e2*) *l_env r_env*

| pd_and : ∀ *p1 p2 l_env r_env*,
pred_denote *p1 l_env r_env* →
pred_denote *p2 l_env r_env* →
pred_denote (PLop And *p1 p2*) *l_env r_env*

| pd_or_left : ∀ *p1 p2 l_env r_env*,
pred_denote *p1 l_env r_env* →
pred_denote (PLop Or *p1 p2*) *l_env r_env*

| pd_or_right : ∀ *p1 p2 l_env r_env*,
pred_denote *p2 l_env r_env* →
pred_denote (PLop Or *p1 p2*) *l_env r_env*

| pd_not : ∀ *p l_env r_env*,
npred_denote *p l_env r_env* →
pred_denote (PNeg *p*) *l_env r_env*

| pd_lexists : ∀ *p x n l_env r_env*,
pred_denote (pred_subst_var *p x* Left *n*) *l_env r_env* → **pred_denote** (PLExists *x p*) *l_env r_env*

| pd_rexists : ∀ *p x n l_env r_env*,
pred_denote (pred_subst_var *p x* Right *n*) *l_env r_env* → **pred_denote** (PRExists *x p*) *l_env r_env*

| pd_lforall : ∀ *p x l_env r_env*,
(∀ *n*, **pred_denote** (pred_subst_var *p x* Left *n*) *l_env r_env*) → **pred_denote** (PLForall *x p*) *l_env r_env*

| pd_rforall : ∀ *p x l_env r_env*,

(∀ *n*, **pred_denote** (pred_subst_var *p x* Right *n*) *l_env r_env*) → **pred_denote** (PRForall *x p*) *l_env r_env*

`with` **npred_denote** : **pred** → **environment** → **environment** → `Prop` :=

| npd_false : ∀ *l_env r_env*, **npred_denote** PFalse *l_env r_env*

| npd_cmp : ∀ *op e1 e2 l_env r_env*,

∀ *n1*, **exp_denote** *e1 l_env r_env n1* →

∀ *n2*, **exp_denote** *e2 l_env r_env n2* →

¬ (cmp_reflect *op*) *n1 n2* → **npred_denote** (PCmp *op e1 e2*) *l_env r_env*

| npd_and_left : ∀ *p1 p2 l_env r_env*, **npred_denote** *p1 l_env r_env* → **npred_denote** (PLop And *p1 p2*) *l_env r_env*

| npd_and_right : ∀ *p1 p2 l_env r_env*, **npred_denote** *p2 l_env r_env* → **npred_denote** (PLop And *p1 p2*) *l_env r_env*

| npd_or : ∀ *p1 p2 l_env r_env*,

(**npred_denote** *p1 l_env r_env*) →

(**npred_denote** *p2 l_env r_env*) →

npred_denote (PLop Or *p1 p2*) *l_env r_env*

| npd_not : ∀ *p l_env r_env*, **pred_denote** *p l_env r_env* → **npred_denote** (PNeg *p*) *l_env r_env*

| npd_lexists : ∀ *p x l_env r_env*,

pred_denote (PLForall *x* (PNeg *p*)) *l_env r_env* → **npred_denote** (PLExists *x p*) *l_env r_env*

| npd_rexists : ∀ *p x l_env r_env*,

pred_denote (PRForall *x* (PNeg *p*)) *l_env r_env* → **npred_denote** (PRExists *x* *p*) *l_env r_env*

| npd_lforall : ∀ *p x l_env r_env*,

pred_denote (PLExists *x* (PNeg *p*)) *l_env r_env* → **npred_denote** (PLForall *x p*) *l_env r_env*

| npd_rforall : ∀ *p x l_env r_env*,

pred_denote (PRExists *x* (PNeg *p*)) *l_env r_env* → **npred_denote** (PRForall *x p*) *l_env r_env*.

Scheme *pred_npred_denote_ind* := Induction for *pred_denote* Sort Prop

with *npred_pred_denote_ind* := Induction for *npred_denote* Sort Prop.

Combined Scheme *pred_npred_denote_combined_ind*

from pred_npred_denote_ind, npred_pred_denote_ind.

Hint Resolve env_equiv_left env_equiv_right env_equiv_sym.

Hint *Constructors exp_denote ref_exp_denote.*

Lemma exp_denote_same_free_vars_combined:

(∀ *e env1 env1' env2 env2'*,

env_equiv *env1 env1'* (free_vars *e* Left) →

env_equiv *env2 env2'* (free_vars *e* Right) →

∀ *n*, **exp_denote** *e env1 env2 n* → **exp_denote** *e env1' env2' n*)

∧

(∀ *ref env1 env1' env2 env2'*,

env_equiv *env1 env1'* (ref_exp_free_vars *ref* Left) →

env_equiv *env2 env2'* (ref_exp_free_vars *ref* Right) →

∀ *f*, **ref_exp_denote** *ref env1 env2 f* → **ref_exp_denote** *ref env1' env2' f*).

Proof.

```
  apply (exp_ref_exp_ind _ _ );
    try solve [unfold_equivs; try inversion 3; crush] .
    intros. inversion H3; crush. econstructor; eauto.
    intros. inversion H3; crush; econstructor; eauto.
   intros. inversion H1; crush. econstructor.
    unfold_equivs; crush. apply functional_extensionality. crush.
   intros. inversion H1; crush. econstructor.
    unfold_equivs; crush. apply functional_extensionality. crush.
    intros. inversion H4; crush. econstructor; eauto.
    eapply H0; eauto.
    eapply H1; eauto.
Qed.

Definition exp_denote_same_free_vars := (proj1 exp_denote_same_free_vars_combined).
Definition ref_exp_denote_same_free_vars := (proj2 exp_denote_same_free_vars_combined).

Lemma exp_free_vars_subst_combined :
  (∀ e a x n s,
    In a (free_vars (exp_subst e x s n) s) → In a (free_vars e s)) ∧
  (∀ r a x n s,
    In a (ref_exp_free_vars (ref_exp_subst r x s n) s) → In a (ref_exp_free_vars r s))
.
Proof.
  Hint Resolve in_or_app.
  apply (exp_ref_exp_ind _ _ ); red_exprs ; crush; eauto.
    destruct s; crush.
    destruct bvar_eq_dec; crush.
    destruct s; crush.
    destruct bvar_eq_dec; crush.
     red_exprs. apply in_or_app. right.
```

```
    apply in_or_app. crush. left. eauto.
    right. eauto.
Qed.

Definition exp_free_vars_subst := (proj1 exp_free_vars_subst_combined).
Definition ref_exp_free_vars_subst := (proj2 exp_free_vars_subst_combined).

Lemma env_equiv_subset :
  ∀ env env' vars,
    env_equiv env env' vars →
    ∀ vars', (∀ v, In v vars' → In v vars ) → env_equiv env env' vars'.
Proof.
  unfold_equivs; crush.
Qed.

Lemma pred_free_vars_subst:
  ∀ p v x s n,
    In v (pred_free_vars (pred_subst_var p x s n) s) → In v (pred_free_vars p s)
.
Proof.
 Hint Resolve in_or_app exp_free_vars_subst.
  induction p;
    try solve [crush; red_exprs; crush; eauto];
    crush; destruct bvar_eq_dec; destruct s; crush; eauto.
Qed.

Lemma exp_free_vars_subst2_combined :
  (∀ e s s' x v n,
    In v (free_vars (exp_subst e x s' n) s) →
    In v (free_vars e s))
  ∧
  (∀ ref s s' x v n,
    In v (ref_exp_free_vars (ref_exp_subst ref x s' n) s) →
```

```
    In v (ref_exp_free_vars ref s)).
Proof.
  apply (exp_ref_exp_ind _ _ ); red_exprs ; crush; eauto.
   try (destruct s; destruct bvar_eq_dec; destruct s'; crush).
   try (destruct s; destruct bvar_eq_dec; destruct s'; crush).
   red_exprs; crush; apply in_or_app; eauto.
Qed.

Definition exp_free_vars_subst2 := (proj1 exp_free_vars_subst2_combined).

Lemma pred_free_vars_subst2 :
  ∀ (p : pred) s s' x v n,
    In v (pred_free_vars (pred_subst_var p x s' n) s) →
    In v (pred_free_vars p s).
Proof.
  Hint Resolve exp_free_vars_subst2.
  induction p; red_exprs ; crush; eauto;
  destruct s'; try destruct bvar_eq_dec; eauto.
Qed.

Hint Constructors pred_denote.
Hint Constructors npred_denote.
Hint Resolve env_equiv_subset.
Hint Resolve exp_denote_same_free_vars.
Hint Resolve env_equiv_left env_equiv_right.
Hint Resolve pred_free_vars_subst2.

Lemma pred_denote_same_free_vars_combined :
  (∀ P env1 env2, pred_denote P env1 env2 →
    ∀ env1', env_equiv env1 env1' (pred_free_vars P Left) →
    ∀ env2', env_equiv env2 env2' (pred_free_vars P Right) →
      pred_denote P env1' env2')
```

```
  ∧

  (∀ P env1 env2, npred_denote P env1 env2 →
    ∀ env1', env_equiv env1 env1' (pred_free_vars P Left) →
    ∀ env2', env_equiv env2 env2' (pred_free_vars P Right) →
      npred_denote P env1' env2')

.
Proof.

  apply (pred_npred_denote_combined_ind
    (fun P env1 env2 ⇒ fun pr ⇒ _) (fun P env1 env2 ⇒ fun pr ⇒ _));


    try match goal with

    | [ ⊢ context[PCmp _ _ _] ] ⇒ simpl; intros; econstructor; eauto
    | [ ⊢ context[pred_denote (PLop And _ _) _ _] ] ⇒ simpl; intros; econstructor;
eauto
    | [ ⊢ context[npred_denote (PLop Or _ _) _ _] ] ⇒ simpl; intros; econstructor;
eauto

    | [ ⊢ context[PLop _ _ _] ] ⇒ simpl; intros; eauto

    | _ ⇒ auto
    end;
    simpl; econstructor; intros; eapply H; eapply env_equiv_subset; eauto.
Qed.

Definition pred_denote_same_free_vars := (proj1 pred_denote_same_free_vars_combined).
Definition npred_denote_same_free_vars := (proj2 pred_denote_same_free_vars_combined).
```

```
Definition pred_satifies (p1 p2 : pred) :=
  ∀ (env1 env2 : environment), pred_denote p1 env1 env2 → pred_denote p2 env1
env2.

Lemma cmp_reflect_denote :
  ∀ c n1 n2, cmp_reflect c n1 n2 → cmp_denote c n1 n2 = true.
Proof.
  destruct c; crush.
    destruct lt_dec; crush.
    destruct eq_nat_dec; crush.
    destruct le_dec; crush.
Qed.

Lemma not_cmp_reflect_denote :
  ∀ c n1 n2, ¬ cmp_reflect c n1 n2 → cmp_denote c n1 n2 = false.
Proof.
  destruct c; crush.
    destruct lt_dec; crush.
    destruct eq_nat_dec; crush.
    destruct le_dec; crush.
Qed.

Lemma bexp_inj_soundness :
  ∀ be env1 env2 s,
    (pred_denote (bexp_inj be s) env1 env2 → bexp_eval be (select _ s env1 env2) =
true)
  ∧
    (npred_denote (bexp_inj be s) env1 env2 → bexp_eval be (select _ s env1 env2) =
false).
Proof.
  induction be; crush ;
    try (destruct b; try inversion H; destruct s; crush).
```

```
  Hint Resolve cmp_reflect_denote
exp_denote_sound
not_cmp_reflect_denote
exp_denote_sound.
  inversion H; crush.
    cut (exp_eval e (select _ s env1 env2) = n1) ; crush.
    cut (exp_eval e0 (select _ s env1 env2) = n2) ; crush.
      eauto.
      eauto.
   inversion H; crush.
    cut (exp_eval e (select _ s env1 env2) = n1) ; crush.
    cut (exp_eval e0 (select _ s env1 env2) = n2) ; crush.
      eauto.
      eauto.
  inversion H; crush.
    pose proof (IHbe1 env1 env2 s).
    pose proof (IHbe2 env1 env2 s). crush.
 inversion H; crush;
    pose proof (IHbe1 env1 env2 s) ;
    pose proof (IHbe2 env1 env2 s) ;
    crush.
    pose proof (IHbe1 env1 env2 s) ;
    pose proof (IHbe2 env1 env2 s) ;
    inversion H; crush; crush.
    pose proof (IHbe1 env1 env2 s) ;
    pose proof (IHbe2 env1 env2 s) ;
    inversion H; crush;
    apply andb_false_iff; crush.
   pose proof (IHbe env1 env2 s);
```

```
    inversion H; crush.
    pose proof (IHbe env1 env2 s);
    inversion H; crush.
Qed.

Theorem assertion_soundness :
  ∀ s P be,
    pred_satifies P (bexp_inj be s) →
      (∀ env1 env2, pred_denote P env1 env2 → bexp_eval be (select _ s env1 env2) =
true).
Proof.
    unfold pred_satifies.
    intros.
    apply bexp_inj_soundness; crush.
Qed.

Hint Constructors exp_denote.
Hint Resolve exp_denote_sound_left exp_denote_sound_right.

Ltac helper :=
  match goal with
  | [ ⊢ exp_denote (exp_inj ?e Left) _ _ _ ] ⇒ apply exp_denote_complete_left; apply
eq_refl
  | [ ⊢ exp_denote (exp_inj ?e Right) _ _ _ ] ⇒ apply exp_denote_complete_right;
apply eq_refl
  end.

Lemma bexp_inj_complete_combined :
  ∀ s be,
    (∀ env1 env2, bexp_eval be (select _ s env1 env2) = true → pred_denote (bexp_inj
be s) env1 env2)
    ∧
    (∀ env1 env2, bexp_eval be (select _ s env1 env2) = false → npred_denote (bexp_inj
```

```
be s) env1 env2).
Proof.
  destruct s; induction be; simpl; intros; crush ;

   try match goal with

   | [ ⊢ npred_denote (PCmp ?c ?e1 ?e2) _ _ ] ⇒
     destruct c ; crush;
       (destruct lt_dec || destruct eq_nat_dec || destruct le_dec);
       econstructor; try helper; crush
   | [ ⊢ pred_denote (PCmp ?c ?e1 ?e2) _ _ ] ⇒
     destruct c ; crush;
       (destruct lt_dec || destruct eq_nat_dec || destruct le_dec);
       econstructor; try helper; crush

   | [ ⊢ pred_denote (PLop ?l _ _) _ _ ] ⇒
       destruct l ;
        (apply andb_true_iff in H3 || apply orb_true_iff in H3); crush

   | [ ⊢ npred_denote (PLop ?l _ _) _ _ ] ⇒
     destruct l ;
       (apply andb_false_iff in H3 || apply orb_false_iff in H3); crush

   end.
   apply negb_true_iff in H1; crush.
   apply negb_false_iff in H1; crush.
   apply negb_true_iff in H1; crush.
   apply negb_false_iff in H1; crush.
Qed.
```

```
Definition bexp_inj_pred_denote_left (be : bexp) := proj1 (bexp_inj_complete_combined
Left be).
Definition bexp_inj_npred_denote_left (be : bexp) := proj2 (bexp_inj_complete_combined
Left be).
Fixpoint pred_mult_exists (P : pred) (vars: list (var * bvar)) (s : selector) : pred :=
  match vars with
  | nil ⇒ P
  | (v, v') :: rest ⇒
    match s with
    | Left ⇒ PLExists v' (pred_graft (pred_mult_exists P rest s) v s (BLVar v'))
    | Right ⇒ PRExists v' (pred_graft (pred_mult_exists P rest s) v s (BRVar v'))
    end
  end.
Inductive exp_array_free : exp → selector → Prop :=
| eaf_const : ∀ s n, exp_array_free (Const n) s
| eaf_blvar : ∀ s b, exp_array_free (BLVar b) s
| eaf_brvar : ∀ s b, exp_array_free (BRVar b) s
| eaf_lvar : ∀ s v, exp_array_free (LVar v) s
| eaf_rvar : ∀ s v, exp_array_free (RVar v) s
| eaf_arr : ∀ s r, ref_exp_array_free r s → ∀ e, exp_array_free e s → exp_array_free
(Arr r e) s
| eaf_iop : ∀ s e1, exp_array_free e1 s → ∀ e2, exp_array_free e2 s → ∀ op, exp_array_free
(Iop op e1 e2) s

with ref_exp_array_free : ref_exp → selector → Prop :=
| reaf_larrvar : ∀ av, ref_exp_array_free (LArrVar av) Right
| reaf_rarrvar : ∀ av, ref_exp_array_free (RArrVar av) Left
| reaf_arrupdate : ∀ s r, ref_exp_array_free r s →
    ∀ e1, exp_array_free e1 s → ∀ e2, exp_array_free e2 s →
      ref_exp_array_free (ArrUpdate r e1 e2) s.
```

```
Scheme exp_array_free_ind' := Induction for exp_array_free Sort Prop
  with ref_exp_array_free_ind' := Induction for ref_exp_array_free Sort Prop.

Combined Scheme exp_ref_exp_array_free_ind from exp_array_free_ind'
ref_exp_array_free_ind'.

Inductive pred_array_free : pred → selector → Prop :=
| paf_true : ∀ s, pred_array_free PTrue s
| paf_false : ∀ s, pred_array_free PFalse s
| paf_cmp : ∀ e1 s, exp_array_free e1 s → ∀ e2, exp_array_free e2 s → ∀ op, pred_array_free
(PCmp op e1 e2) s
| paf_lop : ∀ P1 s, pred_array_free P1 s → ∀ P2, pred_array_free P2 s → ∀ op,
pred_array_free (PLop op P1 P2) s
| paf_neg : ∀ P s, pred_array_free P s → pred_array_free (PNeg P) s
| paf_lexists : ∀ P s, pred_array_free P s → ∀ b, pred_array_free (PLExists b P) s
| paf_rexists : ∀ P s, pred_array_free P s→ ∀ b, pred_array_free (PRExists b P) s
| paf_lforall : ∀ P s, pred_array_free P s → ∀ b, pred_array_free (PLForall b P) s
| paf_rforall : ∀ P s, pred_array_free P s → ∀ b, pred_array_free (PRForall b P) s
.

Hint Constructors exp_array_free.
Hint Constructors ref_exp_array_free.
Hint Constructors pred_array_free.

Fixpoint exp_array_free_dec e s {struct e} :
{exp_array_free e s} + {¬ exp_array_free e s}
with ref_exp_array_free_dec r s {struct r} :
{ref_exp_array_free r s} + {¬ ref_exp_array_free r s}.
    induction e; try solve [ left; auto ].
    pose proof (ref_exp_array_free_dec r s).
    destruct IHe; destruct H.
    left. auto.
    right. unfold not; inversion 1; auto.
```

```
    right. unfold not; inversion 1; auto.
    right; unfold not; inversion 1; auto.
    destruct IHe1; destruct IHe2;
      try solve [left; auto];
      right; unfold not; inversion 1; auto.
    induction r.
      destruct s.
        right; unfold not; inversion 1; auto.
        left; auto.
      destruct s.
        left; auto.
        right; unfold not; inversion 1; auto.
      pose proof (exp_array_free_dec e s).
      pose proof (exp_array_free_dec e0 s).
      destruct IHr; destruct H; destruct H0;
        try solve [right; unfold not; inversion 1; auto].
        left; auto.
Defined.

Definition pred_array_free_dec P s : {pred_array_free P s} + {¬ pred_array_free P
s}.
  induction P. crush. crush.
    pose proof (exp_array_free_dec e s) as H1.
    pose proof (exp_array_free_dec e0 s) as H2.
    destruct H1; destruct H2; try solve [left; crush];
      right; unfold not; inversion 1; crush.
    destruct IHP1; destruct IHP2; try solve [left; crush];
      right; unfold not; inversion 1; crush.
    destruct IHP; try solve [left; crush];
      right; unfold not; inversion 1; crush.
```

```
    destruct IHP; try solve [left; crush];
      right; unfold not; inversion 1; crush.
    destruct IHP; try solve [left; crush];
      right; unfold not; inversion 1; crush.
    destruct IHP; try solve [left; crush];
      right; unfold not; inversion 1; crush.
    destruct IHP; try solve [left; crush];
      right; unfold not; inversion 1; crush.
Defined.

Lemma exp_array_free_not_in_combined :
  (∀ e s, exp_array_free e s → ∀ v, In (inr _ v) (free_vars e s) → False)
  ∧
  (∀ r s, ref_exp_array_free r s → ∀ v, In (inr _ v) (ref_exp_free_vars r s) → False).
Proof.
  apply exp_ref_exp_array_free_ind; simpl; intros; auto; red_exprs.
    destruct s; crush.
    destruct s; crush.
    crush; red_exprs; crush; eauto.
    crush; red_exprs; crush; eauto.
    crush; red_exprs; crush; eauto.
Qed.

Definition exp_array_free_not_in := (proj1 exp_array_free_not_in_combined).

Lemma pred_array_free_not_in :
  ∀ p s, pred_array_free p s → (∀ v, In (inr _ v) (pred_free_vars p s) → False).
Proof.
  Hint Resolve exp_array_free_not_in.
  induction 1; red_exprs; crush; eauto.
Qed.

Lemma pred_graft_free_vars :
```

∀ *P v e s*, ¬ In (inl **arr_var** *v*) (free_vars *e s*) → ¬ In (inl _ *v*) (pred_free_vars (pred_graft *P v s e*) *s*).

```
Proof.
  induction P; simpl; auto.
```

Lemma exp_graft_free_vars_combined :

(∀ *e v s e1*, ¬ In (inl **arr_var** *v*) (free_vars *e1 s*) → ¬ In (inl **arr_var** *v*) (free_vars (exp_graft *e v s e1*) *s*))

∧

(∀ *r v s e1*, ¬ In (inl **arr_var** *v*) (free_vars *e1 s*) → ¬ In (inl **arr_var** *v*) (ref_exp_free_vars (ref_exp_graft *r v s e1*) *s*)).

```
Proof.
  apply (exp_ref_exp_ind _ _); simpl; auto.
  destruct s; simpl; red_exprs; crush.
  destruct s; simpl; red_exprs; crush.
  destruct s; simpl; red_exprs; crush; red_exprs; crush; eauto.
  destruct s; simpl; red_exprs; crush; red_exprs; crush; eauto.
  destruct s; simpl; red_exprs; crush; red_exprs; crush; eauto.
  destruct s; simpl; red_exprs; crush; red_exprs; crush; eauto.
  crush ; red_exprs. crush. eauto. red_exprs. crush; eauto.
Qed.

Definition exp_graft_free_vars := (proj1 exp_graft_free_vars_combined).

intros. red; intros; red_exprs.
crush; eapply exp_graft_free_vars; eauto.
intros. red; intros; red_exprs; crush; eauto.
Qed.
```

Lemma exp_graft_free_vars2_combined :

(∀ *e e1 a s*, ¬ In *a* (free_vars *e1 s*) → ∀ *v*, In *a* (free_vars (exp_graft *e v s e1*) *s*) → In *a* (free_vars *e s*))

∧

```
(∀ r a s e, ¬ In a (free_vars e s) → ∀ v, In a (ref_exp_free_vars (ref_exp_graft r v s e) s) →
In a (ref_exp_free_vars r s)).
Proof.
  apply (exp_ref_exp_ind _ _); try solve [ simpl; auto ].

  destruct s; simpl; red_exprs; crush.

  destruct s; simpl; red_exprs; crush.

  intros. simpl in ×.

  red_exprs. apply in_or_app; crush; eauto.

  simpl; intros.

  red_exprs. apply in_or_app; crush; eauto.

  simpl; intros;

  red_exprs. apply in_or_app; crush; eauto.

  red_exprs ; right; apply in_or_app; crush; eauto.
Qed.

Definition exp_graft_free_vars2 := (proj1 exp_graft_free_vars2_combined).

Lemma pred_graft_free_vars2 :
  ∀ P e s a, ¬ In a (free_vars e s) → ∀ v, In a (pred_free_vars (pred_graft P v s e) s) →
    In a (pred_free_vars P s).
Proof.
  induction P; simpl; auto.

   Hint Resolve exp_graft_free_vars2.

   red_exprs; apply in_or_app; intuition; eauto.

   red_exprs. apply in_or_app; intuition; eauto.
Qed.

Lemma pred_mult_exists_unary :
  ∀ P s,
    ∀ fvs bs,
    length fvs = length bs →
```

```
    (∀ v, In v fvs → ¬ In (inl _ v) (pred_free_vars (pred_mult_exists P (combine fvs bs) s)
s)).
Proof.
  intros until 1.
  Require Import util.
  apply (two_list_ind var bvar (fun fvs bs ⇒
length fvs = length bs →
∀ v : var,
In v fvs →
¬
In (inl arr_var v)
  (pred_free_vars (pred_mult_exists P (combine fvs bs) s) s))).
 simpl. auto.
  crush.
  crush.
  simpl; intros.
  crush.
 contradict H3. destruct s; eapply pred_graft_free_vars; simpl; auto.
   eapply H2. eauto.
 destruct s.
eapply pred_graft_free_vars2. Focus 2. eapply H3; eauto; crush. auto.
eapply pred_graft_free_vars2. Focus 2. eapply H3; eauto; crush. auto.
auto.
Qed.
Lemma pred_mult_exists_unary3:
  ∀ P fvs bs,
    length fvs = length bs →
    ∀ a, In a (pred_free_vars (pred_mult_exists P (combine fvs bs) Right) Right) →
    In a (pred_free_vars P Right).
```

```
Proof.
  intro.
  apply (two_list_ind var bvar (fun fvs bs ⇒
    length fvs = length bs →
∀ a : var + arr_var,
In a (pred_free_vars (pred_mult_exists P (combine fvs bs) Right) Right) →
In a (pred_free_vars P Right)
  )).

  simpl. auto.
  crush.
  crush.
  simpl. intros. eapply H. crush.
  eapply pred_graft_free_vars2. Focus 2. eauto. crush.
Qed.

Fixpoint choose_free_num (vars : list bvar) : nat :=
  match vars with
  | nil ⇒ 0
  | (Bld n) :: vs ⇒
      let n_max := choose_free_num vs in
      if lt_dec n n_max then n_max else n + 1
  end.

Fixpoint gen_numbers (num start : nat) : list nat :=
   match num with
   | 0 ⇒ nil
   | (S n) ⇒ start :: (gen_numbers n (start + 1))
   end.

Definition choose_free_nums (vars : list bvar ) (num : nat) : list nat :=
  let top := choose_free_num vars in
  gen_numbers num top.
```

```
Definition choose_free_bvars (vars : list bvar ) (num : nat) : list bvar :=
  map (fun v ⇒ BId v) (choose_free_nums vars num).

Lemma gen_numbers_length :
  ∀ n s, length (gen_numbers n s) = n.
Proof.
 induction n; crush.
Qed.

Lemma choose_free_bvars_length :
  ∀ vars n, length (choose_free_bvars vars n) = n.
Proof.
   unfold choose_free_bvars. intros; rewrite map_length.
   unfold choose_free_nums.
  Hint Resolve gen_numbers_length. auto.
Qed.

Fixpoint exp_swap (e : exp) : exp :=
  match e with
  | Const n ⇒ Const n
  | LVar n ⇒ RVar n
  | RVar n ⇒ LVar n
  | BLVar n ⇒ BRVar n
  | BRVar n ⇒ BLVar n
  | Arr r e ⇒ Arr (ref_exp_swap r) (exp_swap e)
  | Iop op e1 e2 ⇒ Iop op (exp_swap e1) (exp_swap e2)
  end


with ref_exp_swap (r : ref_exp) : ref_exp :=


  match r with
  | LArrVar av ⇒ RArrVar av
```

```
  | RArrVar av ⇒ LArrVar av
  | ArrUpdate r e1 e2 ⇒ ArrUpdate (ref_exp_swap r) (exp_swap e1) (exp_swap e2)
  end.

Program Fixpoint pred_swap (P : pred) : pred :=
  match P with
  | PTrue ⇒ PTrue
  | PFalse ⇒ PFalse
  | PCmp op e1 e2 ⇒ PCmp op (exp_swap e1) (exp_swap e2)
  | PLop op p1 p2 ⇒ PLop op (pred_swap p1) (pred_swap p2)
  | PNeg p' ⇒ PNeg (pred_swap p')
  | PLExists bv p' ⇒ PRExists bv (pred_swap p')
  | PRExists bv p' ⇒ PLExists bv (pred_swap p')
  | PLForall bv p' ⇒ PRForall bv (pred_swap p')
  | PRForall bv p' ⇒ PLForall bv (pred_swap p')
  end.

Lemma exp_swap_denote_combined :
  (∀ e l_env r_env n, exp_denote e l_env r_env n → exp_denote (exp_swap e) r_env
l_env n)
  ∧
  (∀ r l_env r_env f, ref_exp_denote r l_env r_env f → ref_exp_denote (ref_exp_swap
r) r_env l_env f).
Proof.
  apply (exp_ref_exp_denote_ind _ _); crush.
  econstructor; auto.
Qed.

Definition exp_swap_denote := (proj1 exp_swap_denote_combined).

Lemma exp_swap_subst_combined :
  ∀ s,
  (∀ e x n, exp_subst (exp_swap e) x s n = exp_swap (exp_subst e x (select _ s Right Left)
```

```
n))
  ∧
  (∀ r x n, ref_exp_subst (ref_exp_swap r) x s n = ref_exp_swap (ref_exp_subst r x (select
_ s Right Left) n)).

Proof.
  destruct s; apply (exp_ref_exp_ind _ _); crush; destruct bvar_eq_dec; crush.
Qed.

Definition exp_swap_subst s := proj1 (exp_swap_subst_combined s).

Hint Rewrite exp_swap_subst : cpdt.

Lemma pred_swap_subst :
  ∀ s P x n, pred_subst_var (pred_swap P) x s n = pred_swap (pred_subst_var P x (select _
s Right Left) n).
Proof.
  destruct s; induction P; crush;
  destruct bvar_eq_dec; crush.
Qed.

Hint Resolve exp_swap_denote.
Hint Rewrite pred_swap_subst : cpdt.

Lemma pred_swap_denote :
  (∀ P env1 env2, pred_denote P env1 env2 →
    pred_denote (pred_swap P) env2 env1)
  ∧
 (∀ P env1 env2, npred_denote P env1 env2 →
    npred_denote (pred_swap P) env2 env1).
Proof.
  apply (pred_npred_denote_combined_ind
    (fun P env1 env2 ⇒ fun pr ⇒ _) (fun P env1 env2 ⇒ fun pr ⇒ _)); crush; eauto.

 econstructor; rewrite pred_swap_subst; eauto.
```

econstructor; rewrite pred_swap_subst; eauto.

econstructor. intro. rewrite pred_swap_subst; eauto.

econstructor. intro. rewrite pred_swap_subst; eauto.

Qed.

Lemma exp_swap_free_vars_combined :

(∀ *e*, free_vars (exp_swap *e*) Right = free_vars *e* Left) ∧

(∀ *r*, ref_exp_free_vars (ref_exp_swap *r*) Right = ref_exp_free_vars *r* Left).

Proof.

apply (exp_ref_exp_ind _ _); *crush*.

Qed.

Definition exp_swap_free_vars := (proj1 exp_swap_free_vars_combined).

Hint *Rewrite* exp_swap_free_vars : *cpdt*.

Lemma pred_swap_free_vars :

∀ *P*, pred_free_vars (pred_swap *P*) Right = pred_free_vars *P* Left.

Proof.

induction *P*; *crush*.

Qed.

Hint *Rewrite* pred_swap_free_vars : *cpdt*.

Definition selector_swap *s* := match *s* with Left ⇒ Right | Right ⇒ Left end.

Lemma exp_array_free_exp_swap_combined :

(∀ *e s*, **exp_array_free** *e s* → **exp_array_free** (exp_swap *e*) (selector_swap *s*))

∧

(∀ *r s*, **ref_exp_array_free** *r s* → **ref_exp_array_free** (ref_exp_swap *r*) (selector_swap *s*)).

Proof.

eapply (exp_ref_exp_array_free_ind _ _); *crush*.

Qed.

Definition exp_array_free_exp_swap :

```
  ∀ e, exp_array_free e Left → exp_array_free (exp_swap e) Right.
Proof.
intro.
pose proof (proj1 exp_array_free_exp_swap_combined) e Left. simpl in H.
auto.
Defined.

Hint Resolve exp_array_free_exp_swap.

Lemma pred_array_free_pred_swap :
  ∀ P, pred_array_free P Left → pred_array_free (pred_swap P) Right.
Proof.
  induction P; inversion 1; crush.
Qed.

Fixpoint exp_bound_vars (e : exp) (s : selector) : list bvar :=
  match e with
  | Const _ ⇒ nil
  | BLVar bv ⇒ (select _ s (bv : : nil) nil)
  | BRVar bv ⇒ (select _ s nil (bv : : nil))
  | LVar v ⇒ nil
  | RVar v ⇒ nil
  | Arr ref e ⇒ (ref_exp_bound_vars ref s) ++ (exp_bound_vars e s)
  | Iop op e1 e2 ⇒ (exp_bound_vars e1 s) ++ (exp_bound_vars e2 s)
  end

with ref_exp_bound_vars ref (s : selector) : list bvar :=
  match ref with
  | LArrVar av ⇒ nil
  | RArrVar av ⇒ nil
  | ArrUpdate ref' e1 e2 ⇒ (ref_exp_bound_vars ref' s) ++ (exp_bound_vars e1 s) ++
(exp_bound_vars e2 s)
```

```
  end.

Fixpoint pred_bound_vars (p : pred) (s : selector) : list bvar :=
  match p with
  | PTrue | PFalse ⇒ nil
  | PCmp op e1 e2 ⇒ (exp_bound_vars e1 s) ++ (exp_bound_vars e2 s)
  | PLop op p1 p2 ⇒ (pred_bound_vars p1 s) ++ (pred_bound_vars p2 s)
  | PNeg p ⇒ (pred_bound_vars p s)
  | PLExists x p' ⇒ (select _ s (x :: pred_bound_vars p' s) (pred_bound_vars p' s))
  | PRExists x p' ⇒ (select _ s (pred_bound_vars p' s) (x :: pred_bound_vars p' s))

  | PLForall x p' ⇒ (select _ s (x :: pred_bound_vars p' s) (pred_bound_vars p' s))
  | PRForall x p' ⇒ (select _ s (pred_bound_vars p' s) (x :: pred_bound_vars p' s))

end.

Lemma gen_numbers_values :
  ∀ n start v, In v (gen_numbers n start) → start ≤ v.
Proof.
  induction n; crush.
  eapply IHn in H0. crush.
Qed.

Hint Constructors NoDup.

Lemma gen_numbers_nodup :
  ∀ n start, NoDup (gen_numbers n start).
Proof.
   induction n; crush.
    constructor.
  red; intros; apply gen_numbers_values in H. crush.
  crush.
Qed.
```

```
Lemma nodup_map :
  ∀ (A B : Type) (f : A → B), (∀ x y, f x = f y → x = y) → ∀ l, NoDup l → NoDup (map
f l).
Proof.
  induction l. crush.
  inversion 1; subst. simpl. constructor.
  red; intro. eapply in_map_iff in H1. destruct_conjs.
  apply H in H2. crush.
  auto.
Qed.

Lemma choose_free_bvars_nodup :
  ∀ vars n, NoDup (choose_free_bvars vars n).
  unfold choose_free_bvars.
  unfold choose_free_nums.
  Hint Resolve gen_numbers_nodup.
  intro. SearchAbout NoDup.
intro. apply nodup_map; crush.
Qed.

Lemma choose_free_num_greater :
  ∀ bvars n, In (Bld n) bvars → n < choose_free_num bvars.
Proof.
  induction bvars.
    crush.
    simpl. intros.
 destruct H. destruct a. crush. destruct lt_dec; crush.
   eapply IHbvars in H. destruct a.
   destruct lt_dec. crush. crush.
Qed.
```

```
Lemma choose_free_num_greatest :
    ∀ bvars n, In (BId n) bvars → ∀ m l, In (BId m) (choose_free_bvars bvars l) → lt n m.
Proof.
  induction bvars.
    crush.
 intros. simpl in ×. unfold choose_free_bvars in ×.
 unfold choose_free_nums in ×.
    simpl in H0. crush.
    apply in_map_iff in H0. destruct_conjs; crush.
    destruct lt_dec. apply gen_numbers_values in H1. omega.
     apply gen_numbers_values in H1. omega.
    destruct a. destruct lt_dec. eauto.
    apply in_map_iff in H0. destruct_conjs; crush.
    apply gen_numbers_values in H2. apply choose_free_num_greater in H1. crush.
Qed.

Lemma choose_free_bvar_is_free :
    ∀ n bv bvars, In bv bvars → In bv (choose_free_bvars bvars n) → False.
Proof.
  crush. destruct bv. cut (lt n0 n0). crush.
  eapply choose_free_num_greatest.
  eapply H. eauto.
Qed.

End assertion_logic.

Notation "A '/\_p' B" := (PLop And A B) (at level 100) : accept_scope.
Notation "A '/ _p' B" := (PLop Or A B) (at level 101) : accept_scope.
Notation "E1 '=_p' E2" := (PCmp EQ E1 E2) (at level 90) : accept_scope.
Notation " E1 '<_p' E2" := (PCmp LT E1 E2) (at level 90) : accept_scope.
Notation " E1 '<=_p' E2" := (PCmp LEQ E1 E2) (at level 90) : accept_scope.
Notation "E1 +_p E2" := (Iop Plus E1 E2) (at level 80) : accept_scope.
```

Notation "~_p P" := (PNeg *P*) (at *level* 0) : *accept_scope*.

A.8 Substitution

```
Require Export AssertionLogic.

Require Import Coq.Logic.FunctionalExtensionality.

Require Import CpdtTactics.

Require Import util.

Require Export List.

Definition update_variable_select (s : selector) (env1 env2 : environment) v e :=
  match s with
  | Left ⇒ (update_variable env1 v (exp_eval e env1) , env2)
  | Right ⇒ (env1 , update_variable env2 v (exp_eval e env2))
  end.

Lemma exp_subst_bound_free :
  ∀ (e: Expressions.exp) (s : selector) b n, exp_subst (exp_inj e s) b s n = (exp_inj e s).
Proof.
  induction e; destruct s; crush.
Qed.

Hint Resolve exp_subst_bound_free.

Lemma exp_subst_selector :
  (∀ e s1 s2, s1 ≠ s2 → ∀ b n, exp_subst (exp_inj e s1) b s2 n = exp_inj e s1)
.
  induction e; destruct s1; destruct s2; crush.
Qed.

Hint Resolve exp_subst_selector.

Lemma exp_subst_graft_commute_combined :
  (∀ e,
    ∀ v e1 b (s1 s2 : selector) n,
```

```
    exp_subst (exp_graft e v s1 (exp_inj e1 s1)) b s2 n = exp_graft (exp_subst e b s2 n) v
s1 (exp_inj e1 s1))
  ∧
  (∀ r,
    ∀ v e1 b s1 s2 n,
      ref_exp_subst (ref_exp_graft r v s1 (exp_inj e1 s1)) b s2 n = ref_exp_graft (ref_exp_subst
r b s2 n) v s1 (exp_inj e1 s1)).
Proof.
  apply (exp_ref_exp_ind _ _); crush;
    destruct s1; destruct s2; red_exprs; crush;
      try destruct bvar_eq_dec; crush.
      apply exp_subst_selector. crush.
      apply exp_subst_selector. crush.
Qed.

Definition exp_subst_graft_commute := (proj1 exp_subst_graft_commute_combined).

Hint Rewrite exp_subst_graft_commute : cpdt.
Hint Rewrite exp_subst_bound_free : cpdt.

Lemma pred_subst_graft_commute :
  ∀ P v e b s1 s2 n,
    pred_subst_var (pred_graft P v s1 (exp_inj e s1)) b s2 n =
      pred_graft (pred_subst_var P b s2 n) v s1 (exp_inj e s1).
Proof.
  induction P; crush;
   destruct s2; simpl; auto; destruct bvar_eq_dec; crush.
Qed.

Lemma exp_subst_graft_arr_commute_combined :
    (∀ e av b s1 s2 n e1 e2,
    exp_subst (exp_graft_arr e av s1 (ArrUpdate (select _ s1 (LArrVar av) (RArrVar av))
(exp_inj e1 s1) (exp_inj e2 s1))) b s2 n =
```

```
    exp_graft_arr (exp_subst e b s2 n) av s1 (ArrUpdate (select _ s1 (LArrVar av) (RArrVar
av)) (exp_inj e1 s1) (exp_inj e2 s1)))
  ∧
    (∀ e av b s1 s2 n e1 e2,
    ref_exp_subst (ref_exp_graft_arr e av s1 (ArrUpdate (select _ s1 (LArrVar av) (RAr-
rVar av)) (exp_inj e1 s1) (exp_inj e2 s1))) b s2 n =
    ref_exp_graft_arr (ref_exp_subst e b s2 n) av s1 (ArrUpdate (select _ s1 (LArrVar av)
(RArrVar av)) (exp_inj e1 s1) (exp_inj e2 s1))).
Proof.
  apply (exp_ref_exp_ind _ _ );
    try solve [ crush; destruct s2; red_exprs; try destruct bvar_eq_dec; crush];

    crush; destruct s1; destruct s2; red_exprs; crush;
    rewrite ?exp_subst_selector; crush.
Qed.

Definition exp_subst_graft_arr_commute := (proj1 exp_subst_graft_arr_commute_combined).
Hint Rewrite exp_subst_graft_arr_commute : cpdt.

Lemma pred_subst_graft_arr_commute :
  ∀ P s1 s2 av b n e1 e2,
    pred_subst_var (pred_graft_arr P av s1 (ArrUpdate (select _ s1 (LArrVar av) (RArrVar
av)) (exp_inj e1 s1) (exp_inj e2 s1))) b s2 n =
      pred_graft_arr (pred_subst_var P b s2 n) av s1 (ArrUpdate (select _ s1 (LArrVar av)
(RArrVar av)) (exp_inj e1 s1) (exp_inj e2 s1)).
Proof.
  induction P; crush; destruct bvar_eq_dec; crush;
  destruct s2; crush.
Qed.

Hint Constructors exp_denote.
Hint Constructors ref_exp_denote.
```

```
Lemma exp_denote_graft :
  (∀ e s env1 env2 (n : nat),
    ∀ e' v,
    exp_denote (exp_graft e v s (exp_inj e' s)) env1 env2 n →
      match update_variable_select s env1 env2 v e' with
      | (env1', env2') ⇒ exp_denote e env1' env2' n
      end)
∧
 (∀ ref s env1 env2 f,
    ∀ e' v,
    ref_exp_denote (ref_exp_graft ref v s (exp_inj e' s)) env1 env2 f →
      match update_variable_select s env1 env2 v e' with
      | (env1', env2') ⇒ ref_exp_denote ref env1' env2' f
      end).
Proof.
 apply (exp_ref_exp_ind _ _).
   inversion 1; destruct s; crush.
   inversion 1. inversion 1.
   destruct s; crush; red_exprs; crush.
    econstructor. eapply exp_denote_sound in H;
      crush.
  Hint Resolve assign_state_same.
    inversion H; econstructor; crush.
    inversion H; econstructor; crush.
  destruct s; crush; red_exprs; crush.
    inversion H; econstructor; crush.
    eapply exp_denote_sound in H; econstructor; crush.
    inversion H; econstructor; crush.
  intros. inversion H1; crush.
```

```
  pose proof (H s).
  pose proof (H0 s).

   destruct s; crush.

  intros. pose proof (H s).
  pose proof (H0 s). destruct s;
 inversion H1; crush.

   destruct s; inversion 1; crush.

   destruct s; inversion 1; crush.

  intros;
  pose proof (H s); pose proof (H0 s); pose proof (H1 s);
  destruct s; inversion H2; crush; econstructor; crush...

Qed.

Lemma pred_denote_subst :
    (∀ (Q : pred) (env1 env2 : environment), pred_denote Q env1 env2 →
      ∀ s P (v : var) (e : Expressions.exp), Q = (pred_graft P v s (exp_inj e s)) →
        match update_variable_select s env1 env2 v e with
        | (env1', env2') ⇒ pred_denote P env1' env2'
        end)
    ∧
    (∀ (Q : pred) (env1 env2 : environment), npred_denote Q env1 env2 →
      ∀ s P (v : var) (e : Expressions.exp), Q = (pred_graft P v s (exp_inj e s)) →
        match update_variable_select s env1 env2 v e with
        | (env1', env2') ⇒ npred_denote P env1' env2'
        end).

Proof.
  Hint Constructors pred_denote.
  Hint Constructors npred_denote.

 apply (pred_npred_denote_combined_ind _ _);
```

```
    try solve[
      destruct P; try solve [crush]; inversion 1;
     try (pose proof (H s)); try (pose proof (H0 s)); crush;
     destruct s; crush
    ].
    destruct P; try solve [crush]; inversion 1; crush.
    apply exp_denote_graft in e.
    apply exp_denote_graft in e0.
  destruct s; crush; eauto.
    destruct P; crush. pose proof (H s); destruct s;
      econstructor; eapply H0;
      apply pred_subst_graft_commute.
    destruct P; crush. pose proof (H s); destruct s;
      econstructor; eapply H0;
      apply pred_subst_graft_commute.
  destruct P; crush. destruct s; crush.
     econstructor. intro. pose proof (H n Left); crush.
       eapply H0;
      apply pred_subst_graft_commute.
    econstructor. intro. pose proof (H n Right); crush.
       eapply H0;
      apply pred_subst_graft_commute.
destruct P; crush. destruct s; crush.
     econstructor. intro. pose proof (H n Left); crush.
       eapply H0;
      apply pred_subst_graft_commute.
    econstructor. intro. pose proof (H n Right); crush.
       eapply H0;
```

```
    apply pred_subst_graft_commute.

  destruct P; try solve [crush]; inversion 1; crush.
   apply exp_denote_graft in e.
   apply exp_denote_graft in e0.

 destruct s; crush; eauto.

Qed.

Hint Resolve exp_denote_sound_left exp_denote_sound_right.

Lemma exp_denote_arr_assign_combined :

  ∀ s,
    (∀ e env1 env2 av e1 e2 n1,
      exp_denote (exp_graft_arr e av s (ArrUpdate (select _ s (LArrVar av) (RArrVar
av)) (exp_inj e1 s) (exp_inj e2 s))) env1 env2 n1 →

      let update_env := (select _ s env1 env2) in
      let updated := update_array_contents update_env av (exp_eval e1 update_env)
(exp_eval e2 update_env) in
      let env1' := select _ s updated env1 in
      let env2' := select _ s env2 updated in

      exp_denote e env1' env2' n1)
  ∧
    (∀ ref env1 env2 av e1 e2 f,
      ref_exp_denote (ref_exp_graft_arr ref av s (ArrUpdate (select _ s (LArrVar av)
(RArrVar av)) (exp_inj e1 s) (exp_inj e2 s))) env1 env2 f →

      let update_env := (select _ s env1 env2) in
      let updated := update_array_contents update_env av (exp_eval e1 update_env)
(exp_eval e2 update_env) in
```

```
    let env1' := select _ s updated env1 in
    let env2' := select _ s env2 updated in

      ref_exp_denote ref env1' env2' f).
Proof.
  destruct s; simpl;

   apply (exp_ref_exp_ind _ _); crush ;
    try solve [red_exprs; crush; (inversion H || inversion H2) ; try (constructor);
crush | inversion H1; crush].

    red_exprs; crush; inversion H; crush.

      inversion H3; crush;
      constructor; unfold update_array_contents ; crush; red_exprs. crush.
      cut (exp_eval e1 env1 = n1) ;
        [ cut (exp_eval e2 env1 = n2) | idtac ] ; crush; eauto.

      inversion H; crush. constructor; crush; red_exprs; crush.
        rewrite eta_expansion. crush.

    red_exprs; crush; inversion H; crush.
      inversion H3; crush;
      constructor; unfold update_array_contents ; crush; red_exprs; crush.
      cut (exp_eval e1 env2 = n1) ;
        [ cut (exp_eval e2 env2 = n2) | idtac ] ; crush; eauto.

      inversion H; crush; constructor; crush; red_exprs; crush.
        rewrite eta_expansion; crush.
Qed.

Lemma exp_denote_arr_assign_right :
∀ (e : exp) (env1 env2 : environment) (av : arr_var)
          (e1 e2 : Expressions.exp) (n1 : nat),
```

```
        exp_denote
          (exp_graft_arr e av Left (ArrUpdate (LArrVar av) @ e1 <o> @ e2 <o>))
          env1 env2 n1 →
        exp_denote e
          (update_array_contents env1 av (exp_eval e1 env1) (exp_eval e2 env1))
          env2 n1.
Proof.
  apply exp_denote_arr_assign_combined.
Qed.

Lemma exp_denote_arr_assign_left :
∀ (e : exp) (env1 env2 : environment) (av : arr_var)
          (e1 e2 : Expressions.exp) (n1 : nat),
        exp_denote
          (exp_graft_arr e av Right
              (ArrUpdate (RArrVar av) (exp_inj e1 Right) (exp_inj e2 Right)))
          env1 env2 n1 →
        exp_denote e env1
          (update_array_contents env2 av (exp_eval e1 env2) (exp_eval e2 env2))
          n1.
Proof.
  apply exp_denote_arr_assign_combined.
Qed.

Hint Resolve exp_denote_arr_assign_left exp_denote_arr_assign_right.

Lemma pred_denote_arr_assign_combined :
  ∀ s,
    (∀ P (env1 env2 : environment),
     pred_denote P env1 env2 → ∀ Q (av : arr_var) (e1 e2 : Expressions.exp),
       P = pred_graft_arr Q av s (ArrUpdate (select _ s (LArrVar av) (RArrVar av))
(exp_inj e1 s) (exp_inj e2 s)) →
```

```
        let update_env := (select _ s env1 env2) in
        let upd := update_array_contents update_env av (exp_eval e1 update_env) (exp_eval
e2 update_env) in
        let env1' := select _ s upd env1 in
        let env2' := select _ s env2 upd in
          pred_denote Q env1' env2'
    )
  ∧
  (∀ P (env1 env2 : environment),
     npred_denote P env1 env2 → ∀ Q (av : arr_var) (e1 e2 : Expressions.exp),
       P = pred_graft_arr Q av s (ArrUpdate (select _ s (LArrVar av) (RArrVar av))
(exp_inj e1 s) (exp_inj e2 s)) →

        let update_env := (select _ s env1 env2) in
        let upd := update_array_contents update_env av (exp_eval e1 update_env) (exp_eval
e2 update_env) in
        let env1' := select _ s upd env1 in
        let env2' := select _ s env2 upd in
          npred_denote Q env1' env2').

Proof.

  Hint Rewrite pred_subst_graft_arr_commute : cpdt.

  destruct s; simpl ;

  apply (pred_npred_denote_combined_ind _ _);

    try solve [ destruct Q; try solve [crush]; inversion 1; crush ; eauto ];
      destruct Q; crush; econstructor; intro; eapply H; crush.
Qed.
```

```
Lemma exp_subst_var_graft_trans_combined :
  ∀ (s : selector),
    (∀ e a a0 n,
      ¬ In a0 (exp_bound_vars e s) → exp_subst (exp_graft e a s (select _ s (BLVar a0)
(BRVar a0))) a0 s n = exp_graft e a s (Const n))

    ∧

    (∀ ref a a0 n,
      ¬ In a0 (ref_exp_bound_vars ref s) → ref_exp_subst (ref_exp_graft ref a s (select _
s (BLVar a0) (BRVar a0))) a0 s n = ref_exp_graft ref a s (Const n)).
Proof.
 intros. destruct s;
 apply (exp_ref_exp_ind _ _ ); red_exprs; try (destruct bvar_eq_dec) ; crush.
Qed.

Definition exp_subst_var_graft_trans (s : selector) :=
proj1 (exp_subst_var_graft_trans_combined s).
Hint Rewrite exp_subst_var_graft_trans : cpdt.

Definition exp_subst_var_graft_trans_left := proj1 (exp_subst_var_graft_trans_combined
Left).
Definition ref_exp_subst_var_graft_trans_left := proj1 (exp_subst_var_graft_trans_combined
Left).

Lemma pred_subst_var_graft_trans :
  ∀ s P a a0 n,
    ¬ In a0 (pred_bound_vars P s) →
    pred_subst_var (pred_graft P a s (select _ s (BLVar a0) (BRVar a0))) a0 s n =
      pred_graft P a s (Const n).
Proof.
  induction P; crush; destruct s; try destruct bvar_eq_dec; crush.
```

```
Qed.

Lemma exp_graft_const_combined :
  (∀ e env1 env2 n,
    exp_denote e env1 env2 n →
      ∀ a s, exp_denote (exp_graft e a s (Const (st (select _ s env1 env2) a))) env1 env2
n)
  ∧
 (∀ ref env1 env2 n,
    ref_exp_denote ref env1 env2 n →
      ∀ a s, ref_exp_denote (ref_exp_graft ref a s (Const (st (select _ s env1 env2) a)))
env1 env2 n).
Proof.
  apply (exp_ref_exp_ind _ _); try solve [crush].
 inversion 1; crush. destruct s ; [ destruct var_eq_dec; crush | crush ].
 inversion 1; crush. destruct s ; try destruct var_eq_dec; crush.
 intros. inversion H1; crush.
  intros. inversion H1; crush.
  intros. inversion H2; crush. econstructor; crush.
Qed.

Definition exp_graft_const := (proj1 exp_graft_const_combined).
Hint Resolve exp_graft_const.

Lemma pred_graft_const :
  ∀ s,
    (∀ P env1 env2,
      pred_denote P env1 env2 →
      ∀ a , pred_denote (pred_graft P a s (Const (st (select _ s env1 env2) a))) env1
env2)
  ∧
  (∀ P env1 env2,
```

npred_denote *P env1 env2* →

∀ *a*, **npred_denote** (pred_graft *P a s* (Const (st (select _ *s env1 env2*) *a*))) *env1 env2*)

.

```
Proof.
  intro.
  apply (pred_npred_denote_combined_ind _ _); crush; eauto.

    econstructor; cut (Const (st (select _ s l_env r_env) a) = exp_inj (Expressions.Const
(st (select _ s l_env r_env) a)) s).
      intro. rewrite H0. rewrite pred_subst_graft_commute.
      simpl. apply H. crush.

    econstructor; cut (Const (st (select _ s l_env r_env) a) = exp_inj (Expressions.Const
(st (select _ s l_env r_env) a)) s).
      intro. rewrite H0. rewrite pred_subst_graft_commute.
      simpl. apply H. crush.

    econstructor; cut (Const (st (select _ s l_env r_env) a) = exp_inj (Expressions.Const
(st (select _ s l_env r_env) a)) s).
      intros. rewrite H0. rewrite pred_subst_graft_commute.
      simpl. apply H. crush.

    econstructor; cut (Const (st (select _ s l_env r_env) a) = exp_inj (Expressions.Const
(st (select _ s l_env r_env) a)) s).
      intros. rewrite H0. rewrite pred_subst_graft_commute.
      simpl. apply H. crush.

Qed.

Lemma exp_free_vars_graft_combined :
  ∀ s,
  (∀ e x n,
    free_vars (exp_graft e x s (Const n)) s = remove var_arr_var_eq_dec (inl _ x ) (free_vars
e s)) ∧
```

```
  (∀ ref x n,
    ref_exp_free_vars (ref_exp_graft ref x s (Const n)) s = remove var_arr_var_eq_dec (inl
_ x ) (ref_exp_free_vars ref s))
.
Proof.
  intro.
  Hint Rewrite remove_app : cpdt.
  apply (exp_ref_exp_ind _ _); destruct s; crush; red_exprs; crush.
Qed.
Definition exp_free_vars_graft (s : selector) := proj1 (exp_free_vars_graft_combined
s).
Definition ref_exp_free_vars_graft (s : selector) := proj2 (exp_free_vars_graft_combined
s).
Hint Rewrite exp_free_vars_graft : cpdt.
Lemma pred_free_vars_graft :
  ∀ (s : selector) (p : pred) (v : var + arr_var) x n,
    pred_free_vars (pred_graft p x s (Const n)) s = remove var_arr_var_eq_dec (inl _ x)
(pred_free_vars p s).
Proof.
  induction p; crush.
Qed.
Lemma exp_bound_vars_graft_combined :
  ∀ s,
  ∀ e' b,
  ¬ In b (exp_bound_vars e' s) →
   (
     (∀ e v, In b (exp_bound_vars (exp_graft e v s e') s) → In b (exp_bound_vars e s)) ∧
     ∀ ref v, In b (ref_exp_bound_vars (ref_exp_graft ref v s e') s) → In b (ref_exp_bound_vars
ref s)
```

```
  ).
Proof.
  intros. apply (exp_ref_exp_ind _ _); destruct s ; red_exprs; auto; try apply
in_or_app; (intuition; eauto);
  red_exprs; right; apply in_or_app; crush; eauto.
Qed.

Definition exp_bound_vars_graft s e' b pr := (proj1 (exp_bound_vars_graft_combined s
e' b pr)).

Hint Resolve exp_bound_vars_graft.
Hint Resolve in_or_app.

Lemma pred_bound_vars_graft :
  ∀ s b e,
    ¬ In b (exp_bound_vars e s) → ∀ P v, In b (pred_bound_vars (pred_graft P v s e) s) →
      In b (pred_bound_vars P s).
Proof.
  induction P; red_exprs; crush; eauto;
  destruct s; crush; eauto.
Qed.

Lemma pred_mult_exists_bound_vars :

 ∀ la lb ,
    length la = length lb →
     ∀ b, ¬ In b lb →
     ∀ P s, ¬ In b (pred_bound_vars P s) →
       ¬ In b (pred_bound_vars (pred_mult_exists P (combine la lb) s) s).
Proof.
  apply (two_list_ind var bvar (fun la lb ⇒
 length la = length lb →
   ∀ b : bvar,
```

```
    ¬ In b lb →
    ∀ (P : pred) (s : selector),
    ¬ In b (pred_bound_vars P s) →
    ¬ In b (pred_bound_vars (pred_mult_exists P (combine la lb) s) s))).
  crush.
  crush.
  crush.

  crush. destruct s; crush.
  eapply H5. eauto. eauto.
  eapply pred_bound_vars_graft. Focus 2. eauto.
   simpl. crush.
    eapply H5. eauto. eauto.
      eapply pred_bound_vars_graft. Focus 2. eauto.
   simpl. crush.
Qed.
```

A.9 UnaryAssertionLogic

```
Require Export AssertionLogic.
Require Import Substitution.
Require Import CpdtTactics.
Require Import util.
Require Import Tactics.

Section unary_assertion_logic.

Inductive unary_exp : exp → Prop :=
  | ue_const : ∀ n, unary_exp (Const n)

  | ue_lvar : ∀ v, unary_exp (LVar v)
```

```
  | ue_blvar : ∀ bv, unary_exp (BLVar bv)
  | ue_brvar : ∀ bv, unary_exp (BRVar bv)

  | ue_arr : ∀ ref, unary_ref_exp ref →
      ∀ e, unary_exp e →
        unary_exp (Arr ref e)
  | ue_op :
      ∀ e1, unary_exp e1 → ∀ e2, unary_exp e2 →
        ∀ op, unary_exp (lop op e1 e2)

with unary_ref_exp : ref_exp → Prop :=
  | ue_larrvar : ∀ av, unary_ref_exp (LArrVar av)
  | ue_larrupdate : ∀ ref, unary_ref_exp ref →
      ∀ e1, unary_exp e1 → ∀ e2, unary_exp e2 →
        unary_ref_exp (ArrUpdate ref e1 e2)
.

Hint Constructors unary_exp unary_ref_exp.

Scheme unary_exp_mut := Induction for unary_exp Sort Prop
with unary_ref_exp_mut := Induction for unary_ref_exp Sort Prop.

Combined Scheme unary_exp_ref_exp_ind from unary_exp_mut, unary_ref_exp_mut.

Lemma exp_inj_left_unary :
  ∀ e, unary_exp (exp_inj e Left).
Proof.
  induction e; crush.
Qed.

Lemma exp_denote_unary_combined :
  (∀ e, unary_exp e →
    ∀ l_env,
      ∀ r_env1 n1, exp_denote e l_env r_env1 n1 →
```

```
      ∀ r_env2 n2, exp_denote e l_env r_env2 n2 → n1 = n2)
  ∧
  (∀ ref, unary_ref_exp ref →
    ∀ l_env,
      ∀ r_env1 f1, ref_exp_denote ref l_env r_env1 f1 →
      ∀ r_env2 f2, ref_exp_denote ref l_env r_env2 f2 → f1 = f2).
Proof.
 apply (unary_exp_ref_exp_ind _ _);
   try solve [inversion 1; inversion 1; crush].
 intros. inversion H1; crush; inversion H2; crush.
  cut (f = f0); crush.
  cut (idx = idx0); crush.
  eauto.
  eauto.
  intros. inversion H1; crush. inversion H2; crush.
  cut (n0 = n1); crush.
  cut (n3 = n4); crush.
   eauto. eauto.
 intros. inversion H2; crush. inversion H3; crush.
  cut (n0 = n1); crush.
  cut (n2 = n3); crush.
  cut (f = f0); crush.
    eauto.
    eauto.
    eauto.
Qed.
Inductive unary_pred : pred → Prop :=
| up_true : unary_pred PTrue
| up_false : unary_pred PFalse
```

```
| up_cmp :
    ∀ e1, unary_exp e1 →
    ∀ e2, unary_exp e2 →
      ∀ op, unary_pred (PCmp op e1 e2)

| up_lop :
    ∀ p1, unary_pred p1 →
    ∀ p2, unary_pred p2 →
      ∀ op, unary_pred (PLop op p1 p2)

| up_neg : ∀ p, unary_pred p → unary_pred (PNeg p)

| up_lexists : ∀ p, unary_pred p → ∀ b, unary_pred (PLExists b p)
| up_lforall : ∀ p, unary_pred p → ∀ b, unary_pred (PLForall b p)

| up_rexists : ∀ p, unary_pred p → ∀ b, unary_pred (PRExists b p)
| up_rforall : ∀ p, unary_pred p → ∀ b, unary_pred (PRForall b p)

.

Definition upred := { x : pred | unary_pred x }.

Hint Constructors unary_pred.

Lemma exp_inj_unary :
 ∀ e, unary_exp @ e <o>.
Proof.
  induction e; crush.
Qed.
Hint Resolve exp_inj_unary.

Lemma exp_graft_unary_combined :
  (∀ x, unary_exp x → ∀ v (e : exp), unary_exp e → unary_exp (exp_graft x v Left
```

```
e))
  ∧
  (∀ x, unary_ref_exp x → ∀ v (e : exp), unary_exp e → unary_ref_exp (ref_exp_graft
x v Left e)).
Proof.
  apply (unary_exp_ref_exp_ind _ _); crush.
   destruct var_eq_dec; crush.
Qed.
Definition exp_graft_unary := proj1 exp_graft_unary_combined.
Lemma pred_graft_unary :
  ∀ x v (e : exp), unary_pred x → unary_exp e → unary_pred (pred_graft x v Left e).
Proof.
  Hint Resolve exp_graft_unary.
  induction 1; crush.
Qed.
Program Definition unary_pred_graft (P : upred) (v : var) (e : Expressions.exp ) :
upred :=
    pred_graft (proj1_sig P) v Left (exp_inj e Left).
Next Obligation.
destruct P. crush.
  eapply pred_graft_unary; eauto.
Defined.
Definition unary_pred_bound_vars (P : upred) := pred_bound_vars (proj1_sig P) Left.
Program Fixpoint upred_mult_exists (P : upred) (vars: list (var×bvar)) : upred :=
   match vars with
   | nil ⇒ P
   | (v, v') : : rest ⇒ (PLExists v' (pred_graft (proj1_sig (upred_mult_exists P rest)) v Left
(BLVar v')))
   end.
```

```
Next Obligation.
  econstructor. eapply pred_graft_unary; eauto.
Defined.

Lemma exp_graft_arr_unary_combined :
  (∀ x ,
    unary_exp x → ∀ av e1 e2, unary_exp (exp_graft_arr x av Left (ArrUpdate (LArrVar
av) @ e1 <o> @ e2 <o>)))
  ∧
 (∀ x ,
    unary_ref_exp x → ∀ av e1 e2, unary_ref_exp (ref_exp_graft_arr x av Left (Ar-
rUpdate (LArrVar av) @ e1 <o> @ e2 <o>))).
Proof.
  apply (unary_exp_ref_exp_ind _ _); crush.
  destruct arr_var_eq_dec; crush.
Qed.

Definition exp_graft_arr_unary_exp := proj1 exp_graft_arr_unary_combined.

Lemma pred_graft_arr_unary :
  ∀ x ,
    unary_pred x → ∀ av p e1 e2, p = pred_graft_arr x av Left (ArrUpdate (LArrVar av)
@ e1 <o> @ e2 <o>) → unary_pred p.
Proof.
  Hint Resolve exp_graft_arr_unary_exp.
  induction 1; crush; eauto.
Qed.

Program Definition unary_pred_graft_arr (P : upred) (av : arr_var) (e1 e2 : Expres-
sions.exp) : upred :=
  pred_graft_arr (proj1_sig P) av Left (ArrUpdate (LArrVar av) (exp_inj e1 Left) (exp_inj
e2 Left)).
Next Obligation.
```

```
  crush. destruct P. crush.

eapply pred_graft_arr_unary; eauto.
Defined.

Program Definition bexp_inj_unary (b : bexp) : upred := (bexp_inj b Left).
Next Obligation.
  induction b; crush. destruct b; crush.
Defined.

Definition upred_denote (P : upred) (env : environment) :=
  ∀ env2, pred_denote (proj1_sig P) env env2.

Lemma unary_exp_subst_combined :
  (∀ e, unary_exp e → ∀ s x n, unary_exp (exp_subst e x s n)) ∧
  (∀ (r : ref_exp), unary_ref_exp r → ∀ s x n, unary_ref_exp (ref_exp_subst r x s n)).
Proof.
  apply (unary_exp_ref_exp_ind _ _); crush.

  destruct bvar_eq_dec; destruct s; crush.

  destruct bvar_eq_dec; destruct s; crush.
Qed.

Definition unary_exp_subst := (proj1 unary_exp_subst_combined).
Hint Resolve unary_exp_subst.

Lemma unary_pred_subst :
  ∀ P, unary_pred P → ∀ s x n, unary_pred (pred_subst_var P x s n).
Proof.
  induction 1; simpl; auto.
  destruct s; simpl; intros; try destruct bvar_eq_dec; crush.

  destruct s; simpl; intros; try destruct bvar_eq_dec; crush.
  destruct s; simpl; intros; try destruct bvar_eq_dec; crush.
  destruct s; simpl; intros; try destruct bvar_eq_dec; crush.
Qed.
```

```
Lemma uexp_denote_right_irrel_combined :
  (∀ (e : exp) env1 env2 n, exp_denote e env1 env2 n → unary_exp e → ∀ env3,
exp_denote e env1 env3 n)
  ∧
  (∀ (r : ref_exp) env1 env2 n, ref_exp_denote r env1 env2 n → unary_ref_exp r →
∀ env3, ref_exp_denote r env1 env3 n) .
Proof.
  apply (exp_ref_exp_denote_ind _ _); crush.
   inversion H; crush.
   inversion H1; crush.
   inversion H1; crush.
   inversion H; crush.
   inversion H2; crush. econstructor; crush.
Qed.

Definition uexp_denote_right_irrel := proj1 uexp_denote_right_irrel_combined.

Hint Resolve uexp_denote_right_irrel.
Hint Resolve unary_pred_subst.

Lemma upred_denote_all :
  (∀ (P : pred) env1 env2, pred_denote P env1 env2 → unary_pred P → ∀ env3,
pred_denote P env1 env3)
  ∧
  (∀ (P : pred) env1 env2, npred_denote P env1 env2 → unary_pred P → ∀ env3,
npred_denote P env1 env3) .
Proof.
  apply (pred_npred_denote_combined_ind
    (fun P env1 env2 ⇒ fun pr ⇒ _) (fun P env1 env2 ⇒ fun pr ⇒ _)); simpl; intros;
auto;
 (inversion H || inversion H0 || inversion H1) ; subst; auto; eauto.

Qed.
```

```
Definition upred_satisfies (p1 p2 : upred) :=
  ∀ (env : environment), upred_denote p1 env → upred_denote p2 env.

Program Definition upred_neg (p1 : upred) : upred :=
  PNeg (proj1_sig p1).
Next Obligation.
  destruct p1; crush.
Defined.

Lemma exp_free_vars_nil_unary_combined :
  (∀ e, free_vars e Right = nil → unary_exp e)
  ∧
  (∀ r, ref_exp_free_vars r Right = nil → unary_ref_exp r).
Proof.
  apply (exp_ref_exp_ind _ _); crush.
   apply app_eq_nil in H1. crush.
   apply app_eq_nil in H1. crush.
   apply app_eq_nil in H2. crush.
   apply app_eq_nil in H4; crush.
Qed.

Definition exp_free_vars_nil_unary := (proj1 exp_free_vars_nil_unary_combined).

Hint Resolve exp_free_vars_nil_unary.

Lemma pred_free_vars_nil_unary :
  ∀ P, pred_free_vars P Right = nil →
     unary_pred P.
Proof.
  induction P; crush ;
  apply app_eq_nil in H; crush.
Qed.

Lemma pred_mult_exists_unary2 :
  ∀ P,
```

```
    pred_array_free P Right →
    ∀ fvs bs,
  (∀ v, In v fvs ↔ In (inl _ v) (pred_free_vars P Right)) →
    length fvs = length bs →
    unary_pred (pred_mult_exists P (combine fvs bs) Right).
Proof.
  intros. apply pred_free_vars_nil_unary.
eapply list_nil. intros.
destruct a.
eapply pred_mult_exists_unary. auto. eapply H0.
eapply pred_mult_exists_unary3. eauto. auto.
eapply pred_mult_exists_unary3 in H2. crush. eapply pred_array_free_not_in. eauto.
eauto. auto.
Qed.

Program Definition pred_projl (P : pred) (pr : pred_array_free P Right) : upred :=
  let fvs : list var := map (fun v ⇒ match v with | inl v ⇒ v | inr _ ⇒ (Id 0) end)
(pred_free_vars P Right) in
   pred_mult_exists P (combine fvs (choose_free_bvars (pred_bound_vars P Right) (length
fvs))) Right.
Next Obligation.
  eapply pred_mult_exists_unary2.
    auto. intros. crush. eapply in_map_iff in H.
    crush. destruct x. auto. exfalso; eapply pred_array_free_not_in; eauto.
    apply in_map_iff. ∃ (inl arr_var v). crush.
   rewrite choose_free_bvars_length. auto.
Defined.

Hint Rewrite pred_swap_free_vars : cpdt.

Program Definition pred_projr (P : pred) (pr : pred_array_free P Left) : upred :=
  let SP := pred_swap P in
```

```
  let fvs : list var := map (fun v => match v with | inl v => v | inr _ => (Id 0) end)
(pred_free_vars SP Right) in
    pred_mult_exists SP (combine fvs (choose_free_bvars (pred_bound_vars SP Right) (length
fvs))) Right.
Next Obligation.
  eapply pred_mult_exists_unary2.
  crush.
  Hint Resolve pred_array_free_pred_swap.
  auto.
 intros. crush. eapply in_map_iff in H.
    crush. destruct x. auto. exfalso; eapply pred_array_free_not_in; eauto.
    SearchAbout pred_swap.
    apply in_map_iff. ∃ (inl arr_var v). crush.
   rewrite choose_free_bvars_length. auto.
Defined.

Definition pred_left_satisfies_upred P Q :=
  ∀ env1 env2, pred_denote P env1 env2 → upred_denote Q env1.

Definition pred_right_satisfies_upred P Q :=
  ∀ env1 env2, pred_denote P env1 env2 → upred_denote Q env2.

Hint Resolve env_equiv_refl.

Lemma pred_proj1_satisfies :
   ∀ fvs bvs P env1 env2,
     length fvs = length bvs →
     NoDup bvs →
   ∀ (s : selector),
    (∀ v, In v bvs → In v (pred_bound_vars P s) → False) →
     pred_denote P env1 env2 →
     pred_denote (pred_mult_exists P (combine fvs bvs) s) env1 env2.
Proof.
```

```
  apply (two_list_ind var bvar (fun fvs bvs ⇒
    ∀ (P : pred)
  (env1 env2 : environment),
length fvs = length bvs →
NoDup bvs →
∀ (s : selector),
  (∀ v, In v bvs → In v (pred_bound_vars P s) → False) →
pred_denote P env1 env2 →
pred_denote (pred_mult_exists P (combine fvs bvs) s) env1 env2)).

  simpl. crush.

  crush.

  crush.

  crush. destruct s.

  econstructor. pose proof ( pred_subst_var_graft_trans Left). simpl in ×.

  rewrite H4.

eapply pred_denote_same_free_vars.

  apply pred_graft_const. eapply H.

 auto. solve_nodup. eauto.

      eauto. simpl. crush. crush.

eapply pred_mult_exists_bound_vars.

  auto.

  solve_nodup.

  crush. eauto.

  econstructor. pose proof ( pred_subst_var_graft_trans Right). simpl in ×.

  rewrite H4.

eapply pred_denote_same_free_vars.

  apply pred_graft_const. eapply H.

 auto. solve_nodup. eauto.

      eauto. simpl. crush. crush.
```

```
eapply pred_mult_exists_bound_vars.
  auto.
  solve_nodup.
  crush. eauto.
Qed.

Hint Resolve choose_free_bvar_is_free.

Lemma pred_projl_left_satisfies :
  ∀ P pr, pred_left_satisfies_upred P (pred_projl P pr).
Proof.
  unfold pred_left_satisfies_upred.
  unfold upred_denote.
  intros. eapply upred_denote_all.
 unfold pred_projl. simpl.
eapply pred_projl_satisfies.
rewrite choose_free_bvars_length. auto.
 eauto.
 unfold pred_projl. set (pred_projl_obligation_1 P pr).
 simpl. auto.
apply choose_free_bvars_nodup.
eauto.
eauto.
 unfold pred_projl. set (pred_projl_obligation_1 P pr).
 simpl. auto.
Qed.

Lemma pred_projr_right_satisfies :
  ∀ P pr, pred_right_satisfies_upred P (pred_projr P pr).
Proof.
  unfold pred_right_satisfies_upred.
  unfold upred_denote.
```

```
  intros. eapply upred_denote_all.
 unfold pred_projr. simpl.

eapply pred_projl_satisfies.
rewrite choose_free_bvars_length. auto.
apply choose_free_bvars_nodup.
eauto.

eapply pred_swap_denote.
 eauto.
 unfold pred_projr. set (pred_projr_obligation_1 P pr).
 simpl. auto.
Qed.

Lemma upred_satisfies_pred_satisfies :
  ∀ (P Q : upred), pred_satifies (proj1_sig P) (proj1_sig Q) → (upred_satisfies P Q).
Proof.
  unfold upred_satisfies. unfold upred_denote. destruct P. destruct Q.
unfold pred_satifies. simpl; auto.
Qed.

Program Definition UAnd (P1 P2 : upred) : upred :=
  PLop And P1 P2.
Next Obligation.
  destruct P1; destruct P2; crush.
Defined.

Lemma UAnd_inv :
∀ P Q s, upred_denote (UAnd P Q) s →
   upred_denote P s ∧ upred_denote Q s.
Proof.
  unfold upred_denote. simpl. intros.
  split; intros; pose proof (H env2); inversion H0; auto.
Qed.
```

```
Lemma upred_satifies_and :
  ∀ P1 P2, upred_satisfies P1 P2 → upred_satisfies P1 (UAnd P1 P2).
Proof.
  intros. unfold upred_satisfies in *; unfold UAnd; unfold upred_denote in *;
simpl. auto.
Qed.

Ltac apply_pred_projl :=
  match goal with
  | ⊢ pred_left_satisfies_upred ?RP ?P ⇒
      match (eval simpl in (pred_array_free_dec RP Right)) with
    | left ?t ?pr ⇒ apply (pred_projl_left_satisfies RP pr)
    | right _ ⇒ fail
    end
  end.

Ltac apply_pred_projr :=
  match goal with
  | ⊢ pred_right_satisfies_upred ?RP ?P ⇒
      match (eval simpl in (pred_array_free_dec RP Left)) with
    | left ?t ?pr ⇒ apply (pred_projr_right_satisfies RP pr)
    | right _ ⇒ fail
    end
  end.

End unary_assertion_logic.

Notation "^ b" := (bexp_inj_unary b) (at level 0).
```

A.10 OriginalAxiomatic

```
Require Import util.
Require Export AssertionLogic.
```

```
Require Export UnaryAssertionLogic.
Require Export Substitution.
Require Import List.
Require Import OriginalDynamic.
Require Import Coq.Arith.EqNat.
Require Import Coq.Program.Equality.
Require Import Coq.Bool.Bool.
Require Import Classical.
Require Import Coq.Logic.FunctionalExtensionality.
Require Import Coq.Relations.Relation_Operators.
Require Import Coq.Relations.Operators_Properties.
Require Import Coq.Program.Tactics.
Require Import CpdtTactics.
Require Import Tactics.

Open Scope accept_scope.

Hint Resolve bexp_inj_pred_denote_left bexp_inj_npred_denote_left.

Definition bexp_inj_true_pred_denote s be := (proj1 (bexp_inj_complete_combined s
be)).
Definition bexp_inj_false_npred_denote s be := (proj2 (bexp_inj_complete_combined s
be)).

Fixpoint pred_mult_forall (P : pred) (vars : list (var × bvar) ) {struct vars} : pred :=
  match vars with
  | nil ⇒ P
  | (v , v') : : rest ⇒ PLForall v' (pred_graft (pred_mult_forall P rest) v Left (BLVar v'))
  end.

Program Definition upred_mult_forall (P : upred) (vars : list (var × bvar) ) : upred :=
pred_mult_forall (proj1_sig P) vars.
Next Obligation.
  destruct P.
```

```
    induction vars.
      simpl. auto.
      simpl. destruct a. econstructor. eapply pred_graft_unary; eauto. constructor.
Defined.

Inductive original_axiomatic : upred → statement → upred → Prop :=

  | oa_skip : ∀ P Q, P = Q → original_axiomatic P Skip Q

  | oa_assign : ∀ Q v e,
      original_axiomatic (unary_pred_graft Q v e) (Statements.Assign v e) Q

  | oa_assign_arr :
    ∀ Q (av : arr_var) e1 e2,
      original_axiomatic (unary_pred_graft_arr Q av e1 e2) (AssignArr av e1 e2) Q

  | oa_havoc : ∀ P vars be (vars' : list bvar),
      length vars = length vars' →
      NoDup vars →
      NoDup vars' →
      (∀ v', In v' vars' → ¬ In v' (unary_pred_bound_vars P)) →
      (∀ s, upred_denote P s → (∃ s', havoc_sat vars be s s')) →
      original_axiomatic P (Havoc vars be) (UAnd (upred_mult_exists P (combine vars
vars')) (^be))

  | oa_relax : ∀ P vars be Q, original_axiomatic P (Assert be) Q →
      original_axiomatic P (Relax vars be) Q
```

| oa_assert : ∀ *P be*, **original_axiomatic** (UAnd *P* ^*be*) (Assert *be*) (UAnd *P* ^*be*)

| oa_assume : ∀ *P vars be*, **original_axiomatic** *P* (Assume *vars be*) (UAnd *P* ^*be*)

| oa_accept : ∀ *P l rbe*, **original_axiomatic** *P* (Accept *l rbe*) *P*

| oa_if : ∀ *P be st1 st2 Q*,
original_axiomatic (UAnd *P* ^*be*) *st1 Q* →
original_axiomatic (UAnd *P* (upred_neg ^*be*)) *st2 Q* →
original_axiomatic *P* (If *be st1 st2*) *Q*

| oa_while :
∀ *P be* (*st* : **statement**),
original_axiomatic (UAnd *P* ^*be*) *st P* →
original_axiomatic *P* (While *be st*) (UAnd *P* (upred_neg ^*be*))

| oa_seq : ∀ *P R Q st1 st2* , **original_axiomatic** *P st1 R* → **original_axiomatic** *R st2* *Q* → **original_axiomatic** *P* (Seq *st1 st2*) *Q*

| oa_conseq : ∀ *st P P' Q' Q*,
upred_satisfies *P P'* → **original_axiomatic** *P' st Q'* → upred_satisfies *Q' Q* →
original_axiomatic *P st Q*

.

Notation "|-o {{ P }} st {{ Q }}" := (**original_axiomatic** *P st Q*) (at *level* 70).

```
Definition pred_denote_arr_assign := proj1 (pred_denote_arr_assign_combined Left).

  Semantic judgment Notation "|=o {{ P }} st {{ Q }}" :=
 (∀ env,
    upred_denote P env → ∀ env' ol, original_eval_big <| st , env |> <# env', ol #>
→ upred_denote Q env') (at level 70).

Theorem original_axiomatic_soundness :
 ∀ st P Q, ⊢o {{ P }} st {{ Q }} → |=o {{ P }} st {{ Q }}.
Proof.
  Hint Constructors pred_denote npred_denote.
  Hint Resolve error_valid_config_false.
 induction 1 ;
  try solve [
    intros; invert_original_big; unfold upred_denote in × ; crush |
    intros; invert_original_big; eauto ].

  intros. invert_original_big. unfold upred_denote in H.
  unfold upred_denote. intro.
   eapply (proj1 pred_denote_subst) with (s:=Left) in H; crush; eauto.

  intros. invert_original_big. unfold upred_denote in ×.
  intro.
  eapply pred_denote_arr_assign. eauto. crush.

  intros. unfold upred_denote in *; crush. constructor. invert_original_big.
   unfold havoc_sat in H7. clear H3.
   destruct_conjs. clear H5.
  generalize dependent vars. intro. generalize dependent env.
  generalize dependent env'. generalize dependent vars'.
  generalize dependent vars.

  apply (two_list_ind var bvar (fun vars vars' ⇒
```

```
NoDup vars' →
(∀ v' : bvar, In v' vars' → In v' (unary_pred_bound_vars P) → False) →
∀ env' env : environment,
(∀ env3 : environment, pred_denote (proj1_sig P) env env3) →
length vars = length vars' →
NoDup vars →
(∀ vars'0 : list (var + arr_var),
 (∀ v : var, In v vars → ¬ In (inl arr_var v) vars'0) →
 env_equiv env env' vars'0) →
pred_denote (proj1_sig (upred_mult_exists P (combine vars vars'))) env' env2


)).
  crush.
    eapply pred_denote_same_free_vars. eauto. crush.
   crush. crush. crush.
   intros. crush.
     econstructor.
  pose proof (pred_subst_var_graft_trans Left). crush. rewrite H6.
  eapply pred_denote_same_free_vars.
  apply pred_graft_const. eapply H.
  solve_nodup.
  Hint Resolve in_cons.
  eauto.
  eauto.
  crush.
  solve_nodup.
```

```
  instantiate (1:=update_variable env' a (st env a)).
  intros.
  apply env_equiv_update.
  apply H5. crush. contradict H9. apply remove_In. eapply H7. eauto. eapply
In_remove_In. eauto.
  crush.
  rewrite pred_free_vars_graft.
  apply env_equiv_sym.
  apply update_variable_equiv. auto. apply remove_In. auto. crush.
  generalize H1. generalize H0. inversion H3; subst. generalize H8. clear.
  generalize dependent lb. generalize dependent la.
  apply (two_list_ind var bvar (fun la lb ⇒
    length la = length lb →
    NoDup (b :: lb) →
    (∀ v' : bvar,
      b = v' ∨ In v' lb → In v' (unary_pred_bound_vars P) → False) →
      ¬ In b (pred_bound_vars (proj1_sig (upred_mult_exists P (combine la lb))) Left)
  )).
  simpl. eauto.
  crush.
  crush.
  simpl; intros. inversion H1; subst. simpl in ×. crush.
  apply H5.
  Hint Constructors NoDup.
  solve_nodup.
  intro. intro. apply H2. crush.
  Hint Resolve pred_bound_vars_graft.
  eapply pred_bound_vars_graft with (e:=BLVar b0); crush; eauto.
```

```
  Hint Resolve bexp_inj_true_pred_denote bexp_inj_false_npred_denote.
  invert_original_big. unfold havoc_sat in ×.
    crush.
   intros; invert_original_big. eapply IHoriginal_axiomatic1; eauto.
   unfold upred_denote; crush.
   eapply IHoriginal_axiomatic2; eauto.
   unfold upred_denote; crush.
  intros. unfold upred_denote. crush. dependent induction H1; eauto.
    eapply IHoriginal_eval_big2. eauto. eauto.
    eapply IHoriginal_axiomatic.
      unfold upred_denote. crush. eauto. eauto. eauto.
      eauto.
    error_good_config.
    error_good_config.
   intros.
  invert_original_big; eauto.
    exfalso; eauto.
   eauto.
Qed.
Print Assumptions original_axiomatic_soundness.
Lemma valid_config_not_eq_wr :
   ∀ s ol, <# s , ol #> ≠ wr.
Proof.
  intros. discriminate.
Qed.
Lemma assertion_soundness' :
  ∀ (p : upred) (be : bexp), upred_satisfies p ^be →
   ∀ env, upred_denote p env → bexp_eval be env = false → False.
```

```
Proof.
  unfold upred_satisfies; unfold upred_denote. simpl; intros. contradict H1.
  apply not_false_iff_true. eapply (assertion_soundness Left).
  unfold pred_satifies.
 eauto. eapply H with (env2:= (mkEnv empty_state empty_heap)) in H0. apply H0.
Qed.

Lemma original_axiomatic_progress :
  ∀ st P Q , ⊢o {{ P }} st {{ Q }} →
    ∀ s, upred_denote P s → ∀ o, original_eval_big <| st, s |> o → o ≠ wr.
Proof.
  Hint Resolve assertion_soundness'.
  Hint Resolve valid_config_not_eq_wr.
  Hint Resolve original_axiomatic_soundness.
 intro.
   induction 1.
     intros; invert_original_big; eauto.
     intros; invert_original_big; eauto.
     intros; invert_original_big; eauto.
intros; invert_original_big; eauto.

   intros; invert_original_big; eauto.

   unfold upred_denote. simpl. intros. invert_original_big. eauto.
   cut (bexp_eval be s = true). crush. eapply (bexp_inj_soundness be s s Left).
   pose proof (H s). inversion H0; subst. auto.

    intros; invert_original_big; auto; crush.

  unfold upred_denote in *; crush; invert_original_big; eauto.

     intros; unfold UAnd in *; unfold upred_denote in ×.
     crush. invert_original_big.
    eapply IHoriginal_axiomatic1; eauto.
```

eapply *IHoriginal_axiomatic2*; eauto.

unfold *upred_denote* in × ; *crush*. intros; dependent induction *H1*; eauto.

eapply *IHoriginal_eval_big2*. eauto. *crush*.

eapply original_axiomatic_soundness. eauto. unfold *upred_denote*; *crush*. eauto.

eauto. *crush*.

intros; *invert_original_big*; eauto.

eauto.

Qed.

Print *Assumptions* original_axiomatic_progress.

Close Scope *accept_scope*.

A.11 IntermediateAxiomatic

Relaxed Axiomatic Semantics

Require Import AssertionLogic.

Require Import List.

Require Import RelaxedDynamic.

Require Import Coq.Arith.EqNat.

Require Import Coq.Program.Equality.

Require Import Coq.Bool.Bool.

Require Import Coq.Logic.FunctionalExtensionality.

Require Export OriginalAxiomatic.

Require Import CpdtTactics.

Inductive **intermediate_axiomatic** : upred → **statement** → upred → Prop :=

| ra_skip : ∀ (*P Q* : upred), **original_axiomatic** *P* Skip *Q* → **intermediate_axiomatic** *P* Skip *Q*

| ra_assign : $\forall\ P\ v\ e\ Q$, **original_axiomatic** P (Assign $v\ e$) $Q \rightarrow$ **intermediate_axiomatic** P (Assign $v\ e$) Q

| ra_assign_arr : $\forall\ P\ av\ e1\ e2\ Q$,
original_axiomatic P (AssignArr $av\ e1\ e2$) $Q \rightarrow$
intermediate_axiomatic P (AssignArr $av\ e1\ e2$) Q

| ra_havoc : $\forall\ P\ vars\ be\ Q$, **original_axiomatic** P (Havoc $vars\ be$) $Q \rightarrow$ **intermediate_axiomatic** P (Havoc $vars\ be$) Q

| ra_relax : $\forall\ P\ vars\ be\ Q$, **intermediate_axiomatic** P (Havoc $vars\ be$) $Q \rightarrow$ **intermediate_axiomatic** (UAnd P ^be) (Relax $vars\ be$) Q

| ra_assert : $\forall\ P\ be\ Q$, **original_axiomatic** P (Assert be) $Q \rightarrow$ **intermediate_axiomatic** P (Assert be) Q

| ra_assume : $\forall\ P\ vars\ be$, **intermediate_axiomatic** (UAnd P (^be)) (Assume $vars\ be$) (UAnd P (^be))

| ra_accept : $\forall\ P\ Q\ l\ rbe$, **original_axiomatic** P (Accept $l\ rbe$) $Q \rightarrow$ **intermediate_axiomatic** P (Accept $l\ rbe$) Q

| ra_if : $\forall\ P\ be\ st1\ st2\ Q$, **intermediate_axiomatic** (UAnd P (^be)) $st1\ Q \rightarrow$ **intermediate_axiomatic** (UAnd P (upred_neg (^be))) $st2\ Q \rightarrow$
intermediate_axiomatic P (If $be\ st1\ st2$) Q

| ra_while :
$\forall\ P\ be$ (st : **statement**),
intermediate_axiomatic (UAnd P (^be)) $st\ P \rightarrow$
intermediate_axiomatic P (While $be\ st$) (UAnd P (upred_neg (^be)))

| ra_seq : ∀ *P R Q st1 st2* , **intermediate_axiomatic** *P st1 R* → **intermediate_axiomatic** *R st2 Q* → **intermediate_axiomatic** *P* (Seq *st1 st2*) *Q*

| ra_conseq : ∀ *st P P' Q' Q*, upred_satisfies *P P'* → **intermediate_axiomatic** *P' st Q'* → upred_satisfies *Q' Q* → **intermediate_axiomatic** *P st Q*

.

Notation "|-r {{ P }} st {{ Q }}" := (**intermediate_axiomatic** *P st Q*) (at *level* 70).

Semantic preservation judgment *Notation* "|=r {{ P }} st {{ Q }}" :=

(∀ *s*, upred_denote *P s* → (∀ *ol s'*, **relaxed_eval_big** <| *st* , *s* |> <# *s'*, *ol* #> → upred_denote *Q s'*))

(at *level* 70).

Section intermediate_soundness.

Theorem intermediate_axiomatic_soundness :

∀ *st P Q*, ⊢r {{ *P* }} *st* {{ *Q* }} → |=r {{ *P* }} *st* {{ *Q* }}.

Proof.

Hint Resolve original_axiomatic_soundness.

Hint *Constructors relaxed_eval_big*.

Hint Resolve error_valid_config_false.

induction 1;

intros; try *solve*[*invert_relaxed*; eauto].

invert_relaxed. eapply *IHintermediate_axiomatic*; eauto. eapply UAnd_inv in *H0*. *crush*; eauto.

invert_relaxed. *invert_original_big*. unfold *upred_denote*.

simpl. eauto.

unfold *upred_denote* in *; simpl in ×. *invert_relaxed*; eauto.

Case While unfold *upred_denote* in *; simpl in ×. dependent induction *H1*; eauto.

```
    exfalso; eauto...

  unfold upred_denote in *; simpl in ×.
   invert_relaxed.
      eauto...
      exfalso; eauto...
      exfalso; eauto...
Qed.

End intermediate_soundness.

Section intermediate_axiomatic_progress.

Lemma not_error_valid_config :
  ∀ s ol, ¬ error <#s, ol #>.
Proof.
  intros. red; intro. inversion H.
Qed.

Lemma intermediate_axiomatic_progress :
  ∀ st P Q , ⊢r {{ P }} st {{ Q }} →
    ∀ s, upred_denote P s → ∀ o, relaxed_eval_big <| st, s |> o → ¬ error o.
Proof.
  Hint Resolve original_axiomatic_progress.
  Hint Resolve intermediate_axiomatic_soundness.
  Hint Resolve not_error_valid_config.
  induction 1; intros; try solve[invert_relaxed; try (invert_relaxed); try (invert_original_big);
eauto].
  invert_relaxed. inversion H6; crush.
    eauto.
  eapply original_axiomatic_progress; eauto.
  apply UAnd_inv in H0. inversion H0. invert_relaxed. eauto.
  invert_relaxed. inversion H5; crush.
```

```
    eauto.
  eapply original_axiomatic_progress; eauto.
   unfold upred_denote in *; crush. invert_relaxed; eauto.
   unfold upred_denote. invert_original_big. eauto.
   cut (bexp_eval be s = true). crush. eapply (bexp_inj_soundness be s s Left).
   pose proof (H s). inversion H2; subst. auto.
  invert_relaxed; unfold upred_denote in *; unfold UAnd in *; simpl in *; eauto.
  unfold upred_denote in *; crush. dependent induction H1; eauto.
  eapply IHrelaxed_eval_big2. eauto. crush. eapply intermediate_axiomatic_soundness;
eauto.
  unfold upred_denote in *; crush. eauto.
   crush.
Qed.
Print Assumptions intermediate_axiomatic_progress.
End intermediate_axiomatic_progress.
```

A.12 RelaxedAxiomatic

```
Require Import AssertionLogic.
Require Import List.
Require Import Tactics.
Require Import OriginalDynamic.
Require Import RelaxedDynamic.
Require Import Coq.Program.Tactics.
Require Import Coq.Program.Equality.
Require Import Coq.Arith.EqNat.
Require Import Coq.Bool.Bool.
Require Import Classical.
```

```
Require Export OriginalAxiomatic.

Require Export IntermediateAxiomatic.

Require Import Coq.Lists.SetoidList.

Require Import Coq.Arith.Peano_dec.

Require Import CpdtTactics.

Open Scope accept_scope.

Definition var_map_org_rvar (vars : list (var + arr_var)) : list (rel_var + rel_arr_var)
:=
    map (fun v =>
      match v with
      | inl v => inl _ (Org v)
      | inr av => inr _ (OrgArr av)
      end
     ) vars.

Definition var_map_rel_rvar (vars : list (var + arr_var)) : list (rel_var + rel_arr_var)
:=
    map (fun v =>
      match v with
      | inl v => inl _ (Rel v)
      | inr av => inr _ (RelArr av)
      end
     ) vars.

Definition accept_mods st :=
      (var_map_org_rvar (original_mods st)) ++ (var_map_rel_rvar (relaxed_mods st)).

Fixpoint rexp_inj (re : Expressions.rexp) : AssertionLogic.exp :=
  match re with
  | Expressions.RVar (Org v) => (LVar v)
  | Expressions.RVar (Rel v) => (RVar v)
  | RConst n => Const n
```

```
  | RArr (OrgArr av) e1 ⇒ Arr (LArrVar av) (rexp_inj e1)
  | RArr (RelArr av) e1 ⇒ Arr (RArrVar av) (rexp_inj e1)
  | RIop op e1 e2 ⇒ Iop op (rexp_inj e1) (rexp_inj e2)
end.

Fixpoint rbexp_inj (rbe : Expressions.rbexp) : pred :=
  match rbe with
  | RBConst true ⇒ PTrue
  | RBConst false ⇒ PFalse
  | RCmp op e1 e2 ⇒ PCmp op (rexp_inj e1) (rexp_inj e2)
  | RLop op be1 be2 ⇒ PLop op (rbexp_inj be1) (rbexp_inj be2)
  | RNeg be' ⇒ PNeg (rbexp_inj be')
 end.

Lemma rexp_inj_soundness :
  (∀ re env1 env2 n, exp_denote (rexp_inj re) env1 env2 n → rexp_eval re env1 env2 = n).
Proof.
  induction re; try destruct r; simpl; inversion 1; crush.
    inversion H6; crush.
    inversion H6; crush.
Qed.

Hint Resolve cmp_reflect_denote.
Hint Resolve not_cmp_reflect_denote.

Lemma rbexp_inj_soundness :
  ∀ be,
    (∀ env1 env2, pred_denote (rbexp_inj be) env1 env2 → rbexp_eval be env1 env2 =
true)
    ∧
    (∀ env1 env2, npred_denote (rbexp_inj be) env1 env2 → rbexp_eval be env1 env2 =
false).
Proof.
```

induction *be* ; split; simpl; try destruct *b*; auto; inversion 1; *crush*.

apply rexp_inj_soundness in *H3*.

apply rexp_inj_soundness in *H6*. *crush*.

apply rexp_inj_soundness in *H3*.

apply rexp_inj_soundness in *H6*. *crush*.

apply andb_false_iff. auto.

Qed.

Definition uexp := {*x* : **exp** | **unary_exp** *x*}.

Definition uref_exp := { *x* : **ref_exp** | **unary_ref_exp** *x*}.

Program Definition join (*P_o P_r* : upred) : **pred** :=

(PLop Expressions.And (proj1_sig *P_o*) (pred_swap *P_r*)).

Definition join_bexp (*be* : **bexp**) :=

(PLop Expressions.And (bexp_inj *be* Left) (bexp_inj *be* Right)).

Definition join_neg_bexp (*be*: **bexp**) :=

(PLop Expressions.And (PNeg (bexp_inj *be* Left)) (PNeg (bexp_inj *be* Right))).

Fixpoint pred_mult_exists_right (*P* : **pred**) (*vars*: list (**var**×**bvar**)) : **pred** :=

match *vars* with

| nil ⇒ *P*

| (*v*, *v'*) : : *rest* ⇒ (PRExists *v'* (pred_graft (pred_mult_exists_right *P rest*) *v* Right (BRVar *v'*)))

end.

Inductive **accept_axiomatic** : **pred** → **statement** → **pred** → Prop :=

| aa_skip : ∀ *P*, **accept_axiomatic** *P* Skip *P*

| aa_accept : ∀ *RP aid rbe*,

pred_satifies *RP* (rbexp_inj *rbe*) →

accept_axiomatic *RP* (Accept *aid rbe*) (PLop Expressions.And *RP* (rbexp_inj

rbe))

| aa_assert : ∀ *RP be*,

pred_satifies (PLop Expressions.And *RP* (bexp_inj *be* Left)) (bexp_inj *be* Right) →

accept_axiomatic *RP* (Assert *be*) (PLop Expressions.And *RP* (join_bexp *be*))

| aa_assume : ∀ *RP be* (*vars* : list **var**),

pred_satifies (PLop Expressions.And *RP* (bexp_inj *be* Left)) (bexp_inj *be* Right) →

accept_axiomatic *RP* (Assume *vars be*) (PLop Expressions.And *RP* (join_bexp *be*))

| aa_assign : ∀ *Q v e*,

accept_axiomatic (pred_graft (pred_graft *Q v* Left (exp_inj *e* Left)) *v* Right (exp_inj *e* Right)) (Statements.Assign *v e*) *Q*

| aa_relax : ∀ *be RP* (*vars* : list **var**) (*vars'* : list **bvar**),

length *vars* = length *vars'* →

NoDup *vars* →

NoDup *vars'* →

(∀ *v'*, In *v' vars'* → ¬ In *v'* (pred_bound_vars *RP* Right)) →

(∀ *s_o s_r*, **pred_denote** *RP s_o s_r* → (∃ *s'*, havoc_sat *vars be s_r s'*)) →

accept_axiomatic

(PLop Expressions.And *RP* (bexp_inj *be* Right))

(Relax *vars be*)

(PLop Expressions.And (pred_mult_exists *RP* (combine *vars vars'*) Right) (join_bexp *be*))

| aa_if_converge : ∀ *RP be st1 st2 RQ*,

pred_satifies *RP* (PLop Or (join_bexp *be*) (join_neg_bexp *be*)) →

accept_axiomatic (PLop Expressions.And *RP* (join_bexp *be*)) *st1 RQ* →

accept_axiomatic (PLop Expressions.And *RP* (join_neg_bexp *be*)) *st2 RQ* →

accept_axiomatic *RP* (If *be st1 st2*) *RQ*

| aa_while_converge : ∀ *RP be st*,

pred_satifies *RP* (PLop Or (join_bexp *be*) (join_neg_bexp *be*)) →

accept_axiomatic (PLop Expressions.And *RP* (join_bexp *be*)) *st RP* →

accept_axiomatic *RP* (While *be st*) (PLop Expressions.And *RP* (join_neg_bexp *be*))

| aa_seq : ∀ *RP st1 RR st2 RQ*,

accept_axiomatic *RP st1 RR* →

accept_axiomatic *RR st2 RQ* →

accept_axiomatic *RP* (Seq *st1 st2*) *RQ*

| aa_conseq : ∀ *st RP RP' RQ' RQ*,

pred_satifies *RP RP'* →

accept_axiomatic *RP' st RQ'* →

pred_satifies *RQ' RQ* →

accept_axiomatic *RP st RQ*

| aa_diverge : ∀ *RP P_o P_r st Q_o Q_r*,

accept_free *st* →

pred_left_satisfies_upred *RP P_o* →

original_axiomatic *P_o st Q_o* →

pred_right_satisfies_upred *RP P_r* →

intermediate_axiomatic *P_r st Q_r* →

```
      accept_axiomatic RP st (join Q_o Q_r)

  | aa_constancy : ∀ st RP RR RQ,
      (∀ rv, In rv (pred_free_vars RR Left) → ¬ In rv (original_mods st)) →
      (∀ rv, In rv (pred_free_vars RR Right) → ¬ In rv (relaxed_mods st)) →
      accept_axiomatic RP st RQ →
      accept_axiomatic (RP ∧_p RR) st (RQ ∧_p RR)

.

Section aa_derived.

Lemma pred_satisfies_refl :
  ∀ RP, pred_satifies RP RP.
Proof.
  intro. unfold pred_satifies. crush.
Qed.

Lemma aa_conseq_left :
  ∀ st RP RP' RQ,
      pred_satifies RP RP' →
      accept_axiomatic RP' st RQ →
        accept_axiomatic RP st RQ.
Proof.
  intros. apply aa_conseq with (RP':=RP') (RQ':=RQ); auto. apply pred_satisfies_refl.
Qed.

Lemma aa_dup :
 ∀ st RP RR RQ,
    accept_axiomatic (RP ∧_p RR ∧_p RR) st RQ →
    accept_axiomatic (RP ∧_p RR) st RQ.
Proof.
  intros. eapply aa_conseq; eauto.
```

```
  unfold pred_satifies. intros. econstructor. auto. inversion H0; subst; auto.
  apply pred_satisfies_refl.
Qed.

End aa_derived.

Definition accept_map := list (accept_id × rbexp).

Definition eq_aid (m1 m2 : accept_id × environment) := (fst m1) = (fst m2).

Definition beq_aid (a1 a2 : accept_id) := if eq_nat_dec a1 a2 then true else false.

Definition acceptable_obs (amap : accept_map) (obs1 obs2 : (accept_id × environment)) :=
    eq_aid obs1 obs2 ∧
    ∃ x, InA (fun a b ⇒ (fst a) = (fst b)) (fst obs1, x) amap ∧ rbexp_eval x (snd obs1)
(snd obs2) = true.

Inductive acceptable (amap : accept_map) : obs_list → obs_list → Prop :=
    | acceptable_nil : acceptable amap nil nil
    | acceptable_cons:
        ∀ obs_o obs_r ol_o ol_r,
          acceptable_obs amap obs_o obs_r →
          acceptable amap ol_o ol_r →
          acceptable amap (obs_o::ol_o) (obs_r::ol_r)
.

Fixpoint compute_map (st : statement) :=
  match st with
  | Accept aid rbe ⇒ (aid, rbe)::nil
  | If be st1 st2 ⇒
        (compute_map st1 ++ compute_map st2)
  | While be st ⇒ compute_map st
  | Seq st1 st2 ⇒
      (compute_map st1 ++ compute_map st2)
  | _ ⇒ nil
```

```
  end.

Section FindApp.

Variable A B : Type.

Variable eqA : A → A → Prop.

Hypothesis eqA_equiv : Equivalence eqA.

Hypothesis eqA_dec : ∀ x y : A, {eqA x y}+{~ (eqA x y)}.

Variable eqAB : (A×B)->(A×B)->Prop.

End FindApp.

Lemma acceptable_app :
      ∀ amap ol1_o ol1_r ol2_o ol2_r,
        acceptable amap ol1_o ol1_r →
        acceptable amap ol2_o ol2_r →
          acceptable amap (ol1_o ++ ol2_o) (ol1_r ++ ol2_r).
Proof.

 intro.

Require Import util.

apply (two_list_ind (accept_id × environment) (accept_id × environment)
  (fun ol1_o ol1_r ⇒
    ∀ ol2_o ol2_r ,
      acceptable amap ol1_o ol1_r → acceptable amap ol2_o ol2_r →
      acceptable amap (ol1_o ++ ol2_o) (ol1_r ++ ol2_r))).

   simpl. auto.

   inversion 1.

   inversion 1.

   inversion 2; subst.

      rewrite ← ?app_comm_cons.

      econstructor; auto.

Qed.
```

```
Lemma acceptable_app_left :
  ∀ st1 st2 ol_o ol_r,
    acceptable (compute_map st1) ol_o ol_r → acceptable (compute_map st1 ++
compute_map st2) ol_o ol_r.
Proof.
  Hint Constructors acceptable.
  induction ol_o; induction ol_r.
    eauto.
    inversion 1.
    inversion 1.
    inversion 1; subst.
      inversion H3; subst. destruct_conjs.
      econstructor. econstructor. auto.
      ∃ H1. simpl; split; auto.
   simpl in ×. apply InA_app_iff.

Lemma Equivalance_fst_eq :
   Equivalence (fun a1 b : nat × rbexp ⇒ (fst a1) = (fst b)).
Proof.
  econstructor.
    unfold Reflexive.
     intros. auto.
    unfold Symmetric.
      intros. eauto.
    unfold Transitive.
     intros. rewrite H. auto.
Qed.
  apply Equivalance_fst_eq.
  eauto.
  auto.
```

```
Qed.

Lemma acceptable_app_right :
  ∀ st1 st2 ol_o ol_r,
    acceptable (compute_map st2) ol_o ol_r → acceptable (compute_map st1 ++ com-
pute_map st2) ol_o ol_r.
Proof.
Hint Constructors acceptable.
  induction ol_o; induction ol_r.
    eauto.
    inversion 1.
    inversion 1.
    inversion 1; subst.
      inversion H3; subst. destruct_conjs.
      econstructor. econstructor. auto.
      ∃ H1. simpl; split; auto.
      simpl in ×.
      apply InA_app_iff.
        apply Equivalance_fst_eq.
      right; auto.
      auto.
Qed.

Lemma accept_free_nil_org :
  ∀ st, accept_free st → ∀ s s' ol, original_eval_big <| st, s |> <# s', ol #> → ol = nil.
Proof.
  Require Import Coq.Program.Tactics.
  induction st; simpl; intros;

   try solve [ tauto | invert_original_big ; auto].
   do 2 invert_original_big. auto.
```

```
    destruct_conjs ; invert_original_big; eauto.

  dependent induction H0; eauto.

  assert (ol_1 = nil ∧ ol_2 = nil).
    eauto.

    destruct_conjs ; subst; auto.

  destruct_conjs ; invert_original_big.
  assert (ol_1 = nil ∧ ol_2 = nil).
    eauto.

  destruct_conjs ; subst; auto.

    eauto.
    eauto.
Qed.

Lemma accept_free_nil_rel :
  ∀ st, accept_free st → ∀ s s' ol, relaxed_eval_big <| st, s |> <# s', ol #> → ol = nil.
Proof.
  Require Import Coq.Program.Tactics.

    Hint Resolve accept_free_nil_org.
  induction st; intros;
    try solve [ invert_relaxed ; eauto].

    do 2 invert_relaxed; invert_original_big; auto.

    simpl in *; destruct_conjs ; invert_relaxed; eauto.

  dependent induction H0; eauto.

  assert (ol_1 = nil ∧ ol_2 = nil).
    eauto.

    destruct_conjs ; subst; auto.

  simpl in *; destruct_conjs ; invert_relaxed.
```

```
  assert (ol_1 = nil ∧ ol_2 = nil).
    eauto.
  destruct_conjs ; subst; auto.
   eauto.
   eauto.
Qed.
Local Open Scope accept_scope.
Ltac invert_relaxed :=
  match goal with
  | [ H : relaxed_eval_big _ _ ⊢ _] ⇒ inversion H; subst; clear H
  end.
Lemma error_valid_config_false :
      ∀ s ol , error <#s , ol#> → ∀ RP s_o s_r, pred_denote RP s_o s_r.
   Proof.
     inversion 1.
   Qed.
Section helpers.
End helpers.
Ltac magic :=
  repeat match goal with
        | [H : pred_denote (PLop And _ _ ) _ _ ⊢ _ ] ⇒ inversion H; subst; clear H
        | [H : pred_denote (PNeg _) _ _ ⊢ _ ] ⇒ inversion H; subst; clear H
      end.
Lemma diverge_contradict :
   ∀ s_o s_r be b1 b2,
     pred_denote (PLop Or (join_bexp be) (join_neg_bexp be)) s_o s_r →
      bexp_eval be s_o = b1 →
```

```
    bexp_eval be s_r = b2 →
    b1 ≠ b2 →
      False.
Proof.
  intros. destruct b1, b2; try congruence.
 inversion H; crush. inversion H7; crush.
 apply bexp_inj_soundness in H5.
 apply bexp_inj_soundness in H9. crush.
  inversion H7; crush.
  inversion H5; crush. inversion H9; crush.
  apply bexp_inj_soundness in H4.
 apply bexp_inj_soundness in H6. crush.
 inversion H; crush. inversion H7; crush.
 apply bexp_inj_soundness in H5;
 apply bexp_inj_soundness in H9; crush.
  inversion H7; crush.
  inversion H5; crush. inversion H9; crush.
  apply bexp_inj_soundness in H4.
 apply bexp_inj_soundness in H6. crush.
Qed.

Hint Resolve intermediate_axiomatic_soundness.

Theorem relaxed_axiomatic_soundness :
 ∀ st RP RQ, accept_axiomatic RP st RQ →
   ∀ s_o s_r, pred_denote RP s_o s_r →
   ∀ s_o' ol_o', original_eval_big <| st, s_o |> <# s_o', ol_o' #> →
   ∀ s_r' ol_r', relaxed_eval_big <|st, s_r|> <# s_r', ol_r' #> →
     pred_denote RQ s_o' s_r'.
```

```
Proof.
  Hint Resolve error_valid_config_false.
  induction 1; simpl; intros.
  Hint Constructors pred_denote.
  invert_relaxed; repeat invert_original_big; auto.

Lemma upred_join_denote :
  ∀ Q1 env1,
    upred_denote Q1 env1 → ∀ Q2 env2, upred_denote Q2 env2 →
      pred_denote (join Q1 Q2) env1 env2.
Proof.
  intros. unfold upred_denote in ×.
    constructor. destruct Q1. crush.
    eapply pred_swap_denote. eauto.
Qed.

  Hint Constructors pred_denote.
  invert_relaxed.
  repeat invert_original_big.
  pose proof (H _ _ H0). eauto.
  intros; invert_relaxed; do 2 invert_original_big; repeat constructor ; auto.
  intros. invert_relaxed.
    repeat invert_original_big.
    repeat constructor ; auto.
   repeat invert_relaxed. repeat invert_original_big.
   eapply (proj1 pred_denote_subst) with (s:=Right) in H; crush; eauto.
   eapply (proj1 pred_denote_subst) with (s:=Left) in H; crush; eauto.
  inversion H4; subst; clear H4.
  invert_original_big. invert_original_big.
  invert_relaxed. invert_relaxed. invert_original_big.
```

```
  unfold havoc_sat in H6. destruct H6.

  constructor.

    clear H6. clear H12. clear H5. clear H3.
generalize dependent vars. intro. generalize dependent s_r'.
  generalize dependent s_r. generalize dependent s_o'.
  generalize dependent vars'. generalize dependent vars.
 apply (two_list_ind var bvar (fun vars vars' ⇒
   NoDup vars' →
(∀ v' : bvar, In v' vars' → ¬ In v' (pred_bound_vars RP Right)) →
∀ s_o' s_r : environment,
pred_denote RP s_o' s_r →
∀ s_r' : environment,
length vars = length vars' →
NoDup vars →
(∀ vars'0 : list (var + arr_var),
 (∀ v : var, In v vars → ¬ In (inl arr_var v) vars'0) →
 env_equiv s_r s_r' vars'0) →
pred_denote (pred_mult_exists RP (combine vars vars') Right) s_o' s_r'
)).

    crush. eapply pred_denote_same_free_vars. eauto.

    eauto. eauto.

    crush.

    crush.

    intros. simpl. econstructor.

  pose proof (pred_subst_var_graft_trans Right). simpl in H6. rewrite H6.

  eapply pred_denote_same_free_vars. eapply pred_graft_const.

  eapply H. solve_nodup. crush. eauto. eauto. auto. solve_nodup.

  instantiate(1:= update_variable s_r' a (st s_r a)).

  intros. apply env_equiv_update.
```

```
  apply H5. crush. contradict H9. apply remove_In. eapply H7. eauto. eapply
In_remove_In. eauto.
  eauto. simpl. rewrite pred_free_vars_graft.
  apply env_equiv_sym.
  apply update_variable_equiv. auto. apply remove_In. auto.

   generalize H1. generalize H0. inversion H3; subst.
   generalize H8. clear.
  generalize dependent lb. generalize dependent la.

  apply (two_list_ind var bvar (fun la lb ⇒
length la = length lb →
NoDup (b :: lb) →
(∀ v' : bvar, In v' (b :: lb) → ¬ In v' (pred_bound_vars RP Right)) →
¬ In b (pred_bound_vars (pred_mult_exists RP (combine la lb) Right) Right)
)).

  crush. eauto.
  crush.
  crush.

  simpl; intros. inversion H1; subst. simpl in ×. crush.

 eapply H5. Hint Constructors NoDup. solve_nodup. eauto. crush.
 eauto. eauto.

  Hint Resolve pred_bound_vars_graft.
  eapply pred_bound_vars_graft with (e:=BRVar b0); crush; eauto.

  Hint Resolve bexp_inj_true_pred_denote bexp_inj_false_npred_denote.
  constructor; eauto.

    pose proof (H _ _ H2).
      invert_original_big; invert_relaxed.
      eapply IHaccept_axiomatic1; eauto.
      repeat constructor; eauto.
```

exfalso; eapply diverge_contradict; eauto; discriminate.

exfalso; eapply diverge_contradict; eauto; discriminate.

eapply *IHaccept_axiomatic2*; eauto.

constructor. eauto. *constructor*; simpl; repeat *constructor*; eauto.

intros.

generalize dependent *s_r'*.

generalize dependent *s_r*.

dependent induction *H2*.

intros.

dependent induction *H3*.

repeat (*constructor* ; simpl) ; auto.

pose proof (*H _ _ H2*).

exfalso; eapply diverge_contradict; eauto; discriminate.

eauto...

eauto...

pose proof (*IHoriginal_eval_big2 _ _ H H0 IHaccept_axiomatic*).

clear *H0 IHoriginal_eval_big1 IHoriginal_eval_big2*.

intros; dependent induction *H3*.

pose proof (*H _ _ H0*).

exfalso; eapply diverge_contradict; eauto; discriminate.

assert (**pred_denote** (PLop Expressions.And *RP* (join_bexp *be*)) *s_o s_r*).

repeat *constructor*; auto.

eauto.

eauto...

eauto...

eauto...

eauto...

```
    intros; invert_original_big; invert_relaxed; eauto.

  eauto.

    eapply upred_join_denote; eauto.

     intros. inversion H2; subst. econstructor.

        eauto...

        pose proof (original_eval_not_in_mods_constant _ _ _ _ H3).
        pose proof (relaxed_eval_not_in_mods_constant _ _ _ _ H4).

       eapply pred_denote_same_free_vars; eauto.
Qed.

Section accept_axiomatic_acceptable.

Lemma error_valid_config_acceptable :
  ∀ s ol m ol1 ol2, error <#s, ol #> → acceptable m ol1 ol2.
Proof.
  intros; exfalso; eauto.
Qed.

Theorem relational_assertion_soundness :
 ∀ st RP RQ, accept_axiomatic RP st RQ →
    ∀ s_o s_r, pred_denote RP s_o s_r →
    ∀ s_o' ol_o , original_eval_big <| st, s_o |> <# s_o', ol_o #> →
    ∀ s_r' ol_r , relaxed_eval_big <| st, s_r |> <# s_r', ol_r #> →
        acceptable (compute_map st) ol_o ol_r
.
Proof.
  Hint Constructors acceptable.
  Hint Resolve error_valid_config_acceptable.
  Hint Resolve relaxed_axiomatic_soundness.
  Hint Resolve acceptable_app_right acceptable_app_left.
  Hint Resolve acceptable_app.
```

```
induction 1; simpl; intros.
 repeat invert_relaxed; repeat invert_original_big; auto.
  invert_relaxed. repeat invert_original_big.
  repeat constructor; auto. eexists. simpl.
  split. econstructor. simpl. auto.
  eauto.
  apply rbexp_inj_soundness; auto.
  repeat invert_relaxed; repeat invert_original_big; auto.
  repeat invert_relaxed; repeat invert_original_big; auto.
  repeat invert_relaxed; repeat invert_original_big; auto.
  repeat invert_relaxed; repeat invert_original_big; auto.
   pose proof (H _ _ H2).
   invert_original_big; invert_relaxed.
       apply acceptable_app_left. eapply IHaccept_axiomatic1; eauto.
         repeat constructor; auto.
      exfalso; eapply diverge_contradict; eauto; discriminate.
     exfalso; eapply diverge_contradict; eauto; discriminate.
      apply acceptable_app_right; eapply IHaccept_axiomatic2; eauto.
        repeat (constructor; simpl); auto.
   generalize dependent s_r'.
   generalize dependent s_r.
   dependent induction H2.
     intros. dependent induction H3.
    eauto.
     exfalso; eapply diverge_contradict; eauto; discriminate.
  eauto.
  eauto.
```

```
    intros. dependent induction H3.
        exfalso; eapply diverge_contradict; eauto; discriminate.
    assert (pred_denote (PLop Expressions.And RP (join_bexp be)) s_o s_r).
      repeat constructor; auto.
    assert (pred_denote RP s' s'0).
      eauto.
     apply acceptable_app; eauto.
    eauto.
    eauto.
    eauto.
    eauto.
      invert_original_big; invert_relaxed; eauto.
        cut (pred_denote RR s' s'0); eauto 6.
  eauto.
    Hint Resolve accept_free_nil_org accept_free_nil_rel.
    assert (ol_o = nil ∧ ol_r = nil).
      eauto.
    destruct_conjs; subst; auto.
  inversion H2; subst. eauto.
Qed.
Print Assumptions relational_assertion_soundness.

Theorem relaxed_axiomatic_relative_progress :
  ∀ st RP RQ, accept_axiomatic RP st RQ →
    ∀ s_o s_r,
      pred_denote RP s_o s_r →
      ∀ o_r, relaxed_eval_big <| st, s_r |> o_r →
```

```
    ∀ o_o, original_eval_big <| st, s_o |> o_o → ¬ error o_o →
      ¬ error o_r.
Proof.
  Hint Resolve relaxed_axiomatic_soundness.
  Hint Resolve original_axiomatic_progress.
 induction 1.
  intros. invert_relaxed. invert_original_big; auto. eauto.
  intros. invert_relaxed; invert_original_big. eauto.
  intros. invert_relaxed. do 2 invert_original_big.
    crush. eauto.
    crush. eauto.
    cut (bexp_eval be s_r = true).
      congruence...
      eapply (proj1 (bexp_inj_soundness be _ _ Right)). eauto.
      auto.
  intros.
  invert_relaxed. do 2 invert_original_big.
    eauto.
    eauto.
    cut (bexp_eval be s_r = true).
      congruence...
   eapply (proj1 (bexp_inj_soundness be _ _ Right)). eauto. auto.
  intros. invert_relaxed; invert_original_big. eauto.
 intros. inversion H4; subst; clear H4.
  invert_original_big. invert_original_big.
    invert_relaxed. invert_relaxed. invert_original_big.
      eauto.
      contradict H12. eapply H3. eauto.
```

```
  invert_relaxed. invert_relaxed. invert_original_big.
    eauto.
     contradict H12. eapply H3. eauto.

Hint Constructors pred_denote npred_denote.

intros; pose proof (H _ _ H2).
  invert_original_big; invert_relaxed.
  cut (pred_denote (PLop Expressions.And RP (join_bexp be)) s_o s_r); eauto.
    intros; repeat constructor ; eauto.
  exfalso; eapply diverge_contradict; eauto; discriminate...
  exfalso; eapply diverge_contradict; eauto; discriminate...
  cut (pred_denote (PLop Expressions.And RP (join_neg_bexp be)) s_o s_r); eauto.
     repeat (constructor; simpl) ; auto.

 intros.
 generalize dependent s_r.
 dependent induction H3.
   intros.
    dependent induction H3.
 eauto.
eauto.
 exfalso; eapply diverge_contradict; eauto; discriminate...
 exfalso; eapply diverge_contradict; eauto; discriminate...

 intros.
     pose proof (IHoriginal_eval_big2 _ _ H H0 IHaccept_axiomatic).
     clear IHoriginal_eval_big1 IHoriginal_eval_big2.
     dependent induction H3.
     eauto.
```

```
        eauto.

          eapply IHaccept_axiomatic; eauto. repeat econstructor; eauto.

        eapply H5. eauto. eauto.
        eapply relaxed_axiomatic_soundness; eauto. repeat econstructor; eauto.
        auto.

      intros. auto.
      auto.

    intros. invert_relaxed.
       eauto.
    invert_original_big.
      eauto.
      eauto.
      eauto.
  invert_original_big.
     eauto.
      eauto.
     eauto.
    eauto.
    intros. eapply intermediate_axiomatic_progress; eauto.
    intros. inversion H2; eauto.
Qed.

Theorem relaxed_axiomatic_progress :
 ∀ st P_o Q_o, original_axiomatic P_o st Q_o →
 (∀ s_o o, upred_denote P_o s_o → original_eval_big <| st, s_o |> o → o ≠ ba) →
 (∀ s_o, upred_denote P_o s_o → ∃ o_o, original_eval_big <| st, s_o |> o_o) →
 ∀ RP RQ, accept_axiomatic RP st RQ →
```

```
    pred_left_satisfies_upred RP P_o →
  ∀ s_o s_r, pred_denote RP s_o s_r →
    ∀ o_r, relaxed_eval_big <| st, s_r |> o_r → ¬ error o_r.
Proof.
  intros. destruct_conjs.
  cut (¬ error o_r ∨ o_r = ba ∨ o_r = wr).
    intros. destruct H6. auto.
    edestruct H1. eauto. destruct x.
    eapply relaxed_axiomatic_relative_progress; eauto.
    exfalso. eapply original_axiomatic_progress; eauto.
    exfalso. eapply H0; eauto.
    destruct o_r. left. eauto. eauto. eauto.
Qed.

Corollary relaxed_axiomatic_progress_modulo :
  ∀ P_o st Q_o, original_axiomatic P_o st Q_o →
  ∀ RP RQ, accept_axiomatic RP st RQ →
    pred_left_satisfies_upred RP P_o →
  ∀ s_o s_r, pred_denote RP s_o s_r →
    ∀ o_r, relaxed_eval_big <| st, s_r |> o_r → (o_r = ba ∨ o_r = wr) →
    ∀ o_o, original_eval_big <| st, s_o |> o_o → o_o = ba.
Proof.
  intros. destruct o_o.
   exfalso. eapply relaxed_axiomatic_relative_progress; eauto. Hint Constructors error. crush.
   exfalso. eapply original_axiomatic_progress; eauto.
   auto.
Qed.

End accept_axiomatic_acceptable.

Close Scope accept_scope.
```

Bibliography

[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] J. Ansel, Y. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. CGO, 2011.

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

[6] T. Bao, Y. Zheng, and X. Zhang. White box sampling in uncertain data processing enabled by program analysis. OOPSLA, 2012.

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

[8] G. Barthe, P. D'Argenio, and T. Rezk. Secure information flow by self-composition. CSFW, 2004.

[9] G. Barthe, D. Demange, and D. Pichardie. A formally verified ssa-based middle-end: Static single assignment meets compcert. ESOP, 2012.

[10] G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. POPL, 2009.

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

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

[13] C. Bienia, S. Kumar, J. Pal Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. PACT, 2008.

[14] M. Blum and S. Kanna. Designing programs that check their work. STOC, 1989.

[15] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of computer and system sciences*, 1993.

[16] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *Transactions on Parallel and Distributed Systems*, 3(6), 1992.

[17] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *International Journal of High Performance Computing Applications*, 2009.

[18] M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. PLDI, 2012.

[19] M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Verified integrity properties for safe approximate program transformations. PEPM, 2013.

[20] M. Carbin, S. Misailovic, M. Kling, and M. Rinard. Detecting and escaping infinite loops with jolt. In *ECOOP*, 2011.

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

[22] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in olden. PPOPP, 1995.

[23] F. Chaitin-Chatelin and V. Fraysse. *Lectures on finite precision computations*. 1996.

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

[25] P. Cousot and M. Monerau. Probabilistic abstract interpretation. ESOP, 2012.

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

[27] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. ISCA '10.

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

[29] A. Di Pierro and H. Wiklicky. Concurrent constraint programming: Towards probabilistic abstract interpretation. PPDP, 2000.

[30] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, August 1975.

[31] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. MICRO, 2003.

[32] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. ASPLOS, 2012.

[33] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. MICRO, 2012.

[34] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: probabilistic soft error reliability on the cheap. ASPLOS, 2010.

[35] A. Filieri, C. Păsăreanu, and W. Visser. Reliability analysis in symbolic pathfinder. ICSE, 2013.

[36] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. PLDI, 2002.

[37] M. Hiller, A. Jhumka, and N. Suri. On the placement of software mechanisms for detection of data errors. DSN, 2002.

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

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

[40] 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.

[41] M. Kling, S. Misailovic, M. Carbin, and M. Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. OOPSLA, 2012.

[42] K. Knobe and V. Sarkar. Array ssa form and its use in parallelization. POPL, 1998.

[43] D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 1981.

[44] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian. Mitigating soft error failures for multimedia applications by selective data protection. CASES, 2006.

[45] L. Leem, H. Cho, J. Bau, Q. Jacobson, and S. Mitra. Ersa: error resilient system architecture for probabilistic applications. DATE, 2010.

[46] N. Leveson, S. Cha, J. C. Knight, and T. Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE TSE*, 1990.

[47] N. Leveson and P. Harvey. Software fault tree analysis. *Journal of Systems and Software*, 3(2), 1983.

[48] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. HPCA, 2007.

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

[50] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and Martin Rinard. Automatic input rectification. ICSE, 2012.

[51] Xiph.org Video Test Media. http://media.xiph.org/video/derf.

[52] J. Meng, A. Raghunathan, S. Chakradhar, and S. Byna. Exploiting the forgiving nature of applications for scalable parallel execution. IPDPS, 2010.

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

[54] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. OOPSLA, 2014.

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

[56] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. *ACM TECS Special Issue on Probabilistic Embedded Computing*, 2013.

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

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

[59] D. Monniaux. Abstract interpretation of probabilistic semantics. SAS, 2000.

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

[61] C. Morgan, A. McIver, and K. Seidel. Probabilistic predicate transformers. *TOPLAS*, 1996.

[62] D. Murta and J. N. Oliveira. Calculating fault propagation in functional programs. Technical report, Univ. Minho, 2013.

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

[64] S. Narayanan, J. Sartori, R. Kumar, and D. Jones. Scalable stochastic processors. DATE, 2010.

[65] J. Nelson, A. Sampson, and L. Ceze. Dense approximate storage in phase-change memory. ASPLOS Ideas & Perspectives, 2011.

[66] K. Palem. Energy aware computing through probabilistic switching: A study of limits. *IEEE Transactions on Computers*, 2005.

[67] K. Pattabiraman, V. Grover, and B. Zorn. Samurai: protecting critical data in unsafe languages. EuroSys, 2008.

[68] J. H. Perkins, S. Kim, S. Larsen, S. P. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. C. Rinard. Automatically patching errors in deployed software. SOSP, 2009.

[69] F. Perry, L. Mackey, G.A. Reis, J. Ligatti, D.I. August, and D. Walker. Fault-tolerant typed assembly language. PLDI, 2007.

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

[71] P. Prata and J. Silva. Algorithm based fault tolerance versus result-checking for matrix computations. FTCS, 1999.

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

[73] J. Reed and B. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. ICFP, 2010.

[74] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August. Swift: Software implemented fault tolerance. CGO, 2005.

[75] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

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

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

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

[79] M. 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.

[80] M. Rinard, C. Cadar, D. Dumitran, D.M. Roy, T. Leu, and W.S. Beebee Jr. Enhancing server availability and security through failure-oblivious computing. OSDI, 2004.

[81] M. Rinard, H. Hoffmann, S. Misailovic, and S. Sidiroglou. Patterns and statistical analysis for understanding reduced resource computing. OOPSLA Onwards!, 2010.

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

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

[84] 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.

[85] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. PLDI, 2013.

[86] C. Schlesinger, K. Pattabiraman, N. Swamy, D. Walker, and B. Zorn. Yarra: An extension to c for data integrity and partial safety. CSF '11.

[87] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. DSN, 2002.

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

[89] M. Smith. Probabilistic abstract interpretation of imperative programs using truncated normal distributions. *Electronic Notes in Theoretical Computer Science*, 2008.

[90] A. Thomas and K. Pattabiraman. Error detector placement for soft computation. DSN, 2013.

[91] x264. http://www.videolan.org/x264.html.

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

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

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

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