Operational Refinement for Compiler Correctness

Robert W. Dockins

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Andrew W. Appel

September 2012

Abstract

Compilers are an essential part of the software development process. Programmers all over the world rely on compilers every day to correctly translate their intentions, expressed as high-level source code, into executable low-level machine code. But what does it mean for a compiler to be correct?

This question is surprisingly difficult to answer. Despite the fact that various groups have made concerted efforts to prove the correctness of compilers since at least the early 1980's, no clear consensus has arisen about what it means for a compiler to be correct. As a result, it seems that no two compiler verification efforts have stated their correctness theorems in the same way.

In this dissertation, I will advance a new approach to compiler correctness based on refinements of the operational semantics of programs. The cornerstones of this approach are *behavioral refinement*, which allows programs to improve by going wrong less often, and *choice refinement*, which allows compilers to reduce the amount of internal nondeterminism present in a program. I take particular care to explain *why* these notions of refinement are the correct formal interpretations of the informal ideas above.

In addition, I will show how these notions of refinement can be realistically applied to compiler verification efforts. First, I will present a toy language, WHILE-C, and show how choice and behavioral refinement can be used to verify the correctness of several interesting program transformations. The WHILE-C language and the transformations themselves are simple enough to be presented here in full detail. I will also show how the ideas of behavioral and choice refinement may be applied to the CompCert formally verified compiler [Ler09a], a realistic compiler for a significant subset of C.

Acknowledgements

To everyone who has shared this journey with me during my time at Princeton: thank you. I hope you remember our time together as fondly as I do.

I would like especially to thank my adviser, Andrew Appel, who has helped to guide my journey to a pleasant and timely conclusion. Your fruitful policy of benign neglect has always been tempered by an open door and a genuine interest in my progress. I have enjoyed working with you greatly.

I would also like to thank the members of my dissertation committee for their support: David Walker, Xavier Leroy, David August and Mark Braverman. In particular, I would like to thank David Walker and Xavier Leroy who made time during a very busy part of the year to read and comment on early drafts of this manuscript.

For a multitude of interesting research discussions and delightful lunchtime chats I would like to thank the usual suspects: C.J. Bell, Lennart Beringer, Joey Dodds, Cole Schlesinger and Gordon Stewart. To my ofttime collaborator and coauthor Aquinas Hobor: my thanks. We've done some good work together.

To my parents, Richard and Donna Dockins, who have always supported my academic endeavors, I owe you my eternal gratitude. I am immensely fortunate to have been granted the opportunities you made available. I think your son is finally ready to finish school and grow up now.

To my other parents, Linda and Norman Vance, thank you for your continued love and support, and thank you especially for allowing me to drag your daughter all over the country in pursuit of my dreams.

Most importantly, I would like to thank my lovely wife Rachel, more than words can express. You have stood by me for these many long years, through the good times and the bad, and you have provided me with the moral, emotional, and financial support I needed to go on. I could not have done this without you. Thank you.

This work has been funded by: the National Science Foundation, grant CNS-0910448; the Air Force Office of Scientific Research, grant FA9550-09-1-0138; and by a generous scholarship from the Siebel Scholars Foundation.

In memory of Anna, our faithful friend.

Contents

Chapter 1

Introduction

Compilers are some of the most basic and ubiquitous tools in all of computer science. As traditionally imagined, compilers translate programs in some high-level language into machine-executable form, allowing the program to be run directly on some hardware device of interest.

A compiler, like any other software artifact, might not behave in the way that its designer intended; that is, the compiler might have *bugs*. Bugs in a compiler might cause the compiler to fail in some obvious way, like crashing when compiling some programs. However, the compiler might also fail in some subtle way that causes misbehaviors in compiled programs. This class of bugs is exceptionally difficult to find, understand, and remove for users of the compiler. The bug is not apparent in the text of the program being compiled — instead it is somewhere in the compiler itself. The programmer may well not have access to the source code of the compiler, or even if he does, he may lack the time or expertise to understand it.

The possibility of compiler bugs is a major concern for safety- and security-critical software. In such settings, programmers may have spent considerable effort verifying the correctness of their programs through simulation, code review, or various formal methods. Bugs in a compiler that cause miscompilation can render all that effort meaningless. Extensive testing of the compiled code is sometimes a viable alternative. However, the old adage, attributed to Edsger W. Dijkstra, states that testing can only conclusively demonstrate the *presence* of bugs, not their absence. When lives or a sufficient amount of money are at stake, we might wish to rely on other methods, in addition to testing, to verify the correctness of programs. If we cannot trust a compiler, then we are left only with the option of reviewing, simulating, and formally verifying the low-level code directly — a difficult, unpleasant and time-consuming task.

Instead, we might wish to use a trustworthy compiler, one we believe has no bugs. Then, we could review, verify, etc. the high-level code instead. Unfortunately, compilers are large, complicated programs in their own right, even for fairly simple languages. Worse, the input space for compilers is the programs of the source language — such a space is highly discontinuous (i.e., small changes in the input may lead to large, unpredictable changes in the output), which makes testing difficult.

To gain a high level of confidence in a compiler, we may wish to formally prove that the compiler has no bugs. Provided we can believe such a proof, we are now in a position to trust the compiler; more accurately, we no longer *have* to trust the compiler, because we trust the proof *about* the compiler. Although the effort required to prove the correctness of a compiler is quite large, this effort can be amortized over all the programs subsequently compiled. The availability of a trustworthy compiler may allow significant savings by shifting work on low-level languages, which are difficult to understand, to high-level languages, which are designed with human cognitive abilities in mind. Thus we can economically justify the difficult work of verifying a compiler.

Now we have reduced the question of believing the correctness of a compiler to believing the correctness of a *proof about the compiler*. It is not immediately obvious that this is a better situation. Suppose the proof is mistaken — then the compiler may still have bugs! For a large and complicated program (like a compiler), the proof of its correctness may be even larger and more complicated still. Nonetheless, this problem, too, can be solved.

If we carry out the correctness proof with sufficient rigor, it is possible to write a program which checks the proof for validity. Now we are in a good position, because it is much easier to gain confidence in a small proof checking procedure than it is to gain confidence in a large proof. As we shall see below, using computer assistance to check proofs of compiler correctness has become a relatively popular activity in the academic community.

Once we have performed such a machine-checked proof, we have shifted our burden of trust. We no longer trust the compiler, nor do we trust the compiler correctness proof, but rather we trust the general purpose proof checker. In other words, our *trusted computing base*, that in which we must trust to believe in the correctness of our compiler, includes neither the compiler nor its correctness proof.

However, the proof checker is not the only thing in our trusted computing base, according to the analysis of Pollack [Pol98]. We must also include the *statement* of the correctness theorem (although not its proof). In other words, we must believe that the statement of the correctness theorem (together with all the definitions leading up to it) has adequately captured the correctness property we had in mind.

It is this latter concern that motivates the work appearing in this thesis. What do we (or what *ought* we) mean by the phrase "compiler correctness?" This question is informal, even philosophical, in nature, and thus cannot yield to a purely formal approach. In other words, no answer to this question can be definitively shown to be the right answer by proof. Instead, the best we can do is to advance a hypothesis and accumulate evidence that our hypothesis is a good one. Over time, one hopes that the community will converge to a formal definition that is widely held to be "right."

The famous Church-Turing thesis is an example of this sort of phenomenon, where the informal notion of "effective computability" was associated with the formal notion of "computable by a Turing-machine," an association that has so far withstood the test of time and provided fertile ground for the entire field of Computer Science.

While I do not suppose that any of the hypotheses I advance in this thesis will rival the status of Church and Turing's, I at least hope to convince the reader that this question regarding the meaning of "compiler correctness" is a question worthy of study, and to make some reasonable attempt to answer the question.

1.1 Why Compiler Correctness?

A skeptic may object to this line of work by pointing out that compiler correctness efforts are far from a new phenomenon — surely some coherent theory of what compiler correctness means has already emerged. Surprisingly, however, little consensus has arisen in the compiler verification community about the form a compiler correctness theorem should take. Instead, each major compiler verification effort has developed its own notion of compiler correctness, which must be understood and defended individually. Some correctness statements are very "gritty," and involve explicit correspondences between the operational states of the source and target languages. Others are quite abstract and require readers to absorb a significant amount of mathematical machinery to understand. Others still formulate simple definitions that are easy to understand but suffer from being too weak. Although significant similarities can be found among many of the various efforts, they are not stated within a common framework, so results from different verification efforts cannot be easily understood and compared.

Let us examine a selection of the published results in compiler correctness and see what conclusions we can draw. These results are presented, rather arbitrarily, in order of publication date.

CLI. The first result we will examine was produced by the research group at Computational Logic Inc. in the late '80s. This work is justifiably well-known — it represents the first (to my knowledge) machine-verified correctness proof for a realistic compiler. In fact, this work was just one piece of an entire stack (the so-called CLI short stack [BHMY89]) of machine-verified correctness proofs going all the way down to a gate-level specification of the FM8502 microprocessor. The CLI short stack was mechanically verified using Kaufmann's interactive enhancements to Nqthm, the then-current incarnation of the Boyer-Moore theorem prover [Kau88].

Each of the stages in this stack is specified in basically the same way, using an *interpreter equivalence* theorem [You89].

Definition 1.1.1 (Interpreter equivalence (paraphrased))**.** *Let p be a high-level program and o be a well-formed initial state for p. Let* run *be a function computing a final state in the high-level semantics, and let* execute *be a function computing a final state in the low-level semantics.* compile *translates a high-level program and initial state to a low-level program and initial state;* lift *turns a low-level state into a corresponding high-level state.*

The high-level semantics is interpreter equivalent to the low-level semantics if:

$$\mathsf{run}(p, o) = \mathsf{lift}(\mathsf{execute}(\mathsf{compile}(p, o)))$$

This definition has a number of advantages. First, it is rather simple, which makes it easy to understand and explain. Second, it is *syntax agnostic* — in other words, the form of the theorem constrains only the operational behavior of the program, without any reference to syntactic properties.[1] Third, the form of the theorem composes in sequence. In other words, if I have two transformation passes each satisfying interpreter equivalence, I can combine to two passes in sequence to get a new interpreter equivalence diagram over the whole. This property means we can verify a compiler transformation pass at a time, combining the verifications of each individual pass into a verification of the whole compiler.

However, this definition also has a number of significant problems. Most importantly, it has a rather explicit and clumsy handling of computational resources. An upper bound on the number of clock cycles and the amount of memory to be used by the program must be included in the initial state o. In part, this situation is due to limitations imposed by the theorem prover. It is significantly easier to define a total function in Nqthm than it is to define some formula representing a relation.

[1]Contrast with methods based on contextual equivalence, which require the compared programs to be expressed in some common syntax.

By fixing in advance a specific number of clock cycles, it was possible to write total functions for the program semantics. A special "out of resources" state is reached if the allotted time or space bounds are exceeded. At any rate, it means the interpreter equivalence theorem does not directly address unbounded program behaviors. Because the semantics of each language is defined as a function, this theorem style also cannot properly deal with nondeterminism.

TPL. The second compiler verification effort we will examine proves the correctness of a compiler for a small, hard-real-time programming language (TPL) to the instruction set of the Transputer architecture [MO97]. This work differs from all the others we will examine in three significant aspects: it is not machine verified; it makes hard real-time assumptions; and it avoids the use of operational semantics, using instead a combination of denotational and axiomatic methods.

The main lemma Müller-Olm proves about his compiler is the following:

Definition 1.1.2 (Correct program translation)**.** *Suppose given prog* $\in \mathsf{Prog}$ *and* $m \in \mathsf{IS}$*. We call* m correct code *for prog iff*

$$I_5^{\emptyset}(\epsilon, m) \geq \mathsf{MP}(\mathit{prog})$$

Roughly, it says that the denotation of the compiled program $I_5^{\emptyset}(\epsilon, m)$ is included in the denotation of the source program $\mathsf{MP}(\mathit{prog})$; in other words, the output program *refines* or *is better than* the input program. Although not this particular instantiation, the general idea of program refinement will be critical in later chapters of this thesis.

The function MP is defined by recursion on the syntax of *prog*. For Müller-Olm, the denotation of a program is its weakest-precondition predicate transformer, and $\geq$ means that the "larger" program secures every postcondition for at least as many initial conditions. In other words, for each postcondition Q and for each initial state σ such that running *prog* from state σ halts in a state satisfying Q, running σ with code m will also halt in a state satisfying Q.

Unfortunately, this extremely compact definition conceals quite a lot of fairly complicated mathematics. In particular, the notation $I_5^{\emptyset}$ refers to the top-level of a 5-deep stack of abstractions that are defined earlier in a rather dense chapter of the book spanning more than 30 pages.

Recognizing this problem, Müller-Olm proves a weaker, but more user-friendly result.

Theorem 1.1.3 (Direct relationship to run phase)**.** *Suppose given prog* $\in \mathsf{Prog}$ *and* $m \in \mathsf{IS}$*. If* m *is correct code for* Prog *then:*

$$\mathsf{MP}(\mathit{prog}) \ \leq \ Tc^{+} \ ; \ [\mathit{Loaded}(m) \wedge \mathtt{Ip} = s_{\mathrm{P}} \wedge \mathtt{Oreg} = 0 \wedge \mathit{Wptr} = s_{\mathrm{W}}] \ ; \ \mathit{Run} \ ; \ Tc^{-}$$

Roughly, this theorem says that, whenever *prog* is correct code for m (as in definition 1.1.2), the denotation of $\mathsf{MP}(\mathit{prog})$ is included in the denotation of a program that "boots" the Transputer into an initial state where program m is loaded and then begins executing.

This theorem statement is now pretty acceptable from the standpoint of a trusted computing base. One must understand only the basic setup (i.e., that the denotations of programs are weakest-precondition predicate transformers) and the semantics of the high and low-level languages to evaluate the meaning of this theorem.

Although this setup is quite elegant in many ways, it falls short, I think, by taking the weakest-precondition notion as basic. This essentially means that the compiler preserves only precondition/-postcondition properties, and only for terminating programs. These restrictions are unsatisfying because the main application for hard real-time compilers is embedded control software — such software is typically long-running reactive software rather than terminating batch processes. Reasoning only about terminating programs rules out programs of the form "run forever, reacting to inputs by performing operations X, Y and Z."

CompCert. The CompCert verified C is a compiler for a large subset of the C language, currently targeting several target architectures (PowerPC, x86 and ARM) with a machine-verified correctness proof in Coq [Com]. Among the compilers examined here, CompCert (and its fork CompCertTSO) is the most "realistic" in terms of the input and output languages it considers.

It is a long-term goal of the Verified Software Toolchain project [App11] to connect CompCert to concurrent separation logic [HAZ08], and the work in this thesis was originally prompted by a desire to better understand the proper software engineering interface between CompCert and a high-level program verification logic. Chapter 10 is devoted to discussing how the work in this thesis could be applied to the CompCert compiler.

In an overview paper discussing the CompCert proof, Leroy spends a fair amount of space discussing the form of his correctness result [Ler06]. The form of the correctness result for CompCert at the time Leroy published his paper was roughly:

Definition 1.1.4 (Certified compilers)**.** *Suppose S is an input program, and suppose the compiler succeeds in producing a compiled program C. Provided S has well-defined semantics (does not go wrong) and terminates producing value v, then C also does not go wrong, terminates, and produces value v.*

This correctness statement has the pleasing property that is is quite simple. One must understand little more than the operational semantics of the source and target languages. Unfortunately, this early paper on CompCert actually proves a rather weak theorem. It only holds for safe, terminiating programs, and only tells us about the final value produced by the program (its exit code). It does not say anything about side effects that might occur during the program execution.

However, as Leroy notes, the actual proofs of the individual compiler passes prove facts about the compiler that are significantly stronger than the final result. They show that memory accesses, function calls, etc. line up in an essentially lockstep fashion. In other words, the main theorem for the compiler is *weaker than what is true.* This weakness limits our ability to believe the correctness of the compiler because it allows the compiler to do things we would probably deem inappropriate, even though the compiler does not, in fact, do such things. It also limits our ability to later use the compiler correctness theorem to establish other results of interest.

As stated, the definition above makes three essential assumptions: 1) the source program is safe, 2) the source program terminates and 3) the target program is deterministic. Assumptions 1 and 2 appear directly in the statement, but assumption 3 is latent. If assumption 3 is violated then the correctness statement above is still true, but it fails to be very useful. In a nondeterministic setting, we would also like to require that every value produced by C could have been produced by S — otherwise the compiler would be allowed to introduce new behaviors, which is not what we wish at all.

In the years since publication of the overview paper mentioned above, the CompCert correctness proof has been gradually improved in a variety of ways. In a later overview paper [Ler09a] and a related journal article [Ler09b], Leroy discusses an improved correctness statement that drops the requirement for programs to terminate and also discusses traces of observable events that the program may generate as it runs. The latest version of CompCert[2] has an even stronger correctness statement based on simulation relations.

I defer the discussion of these improvements until chapter 10, but here I simply note that the history of the CompCert project has seen a march toward stronger correctness statements. One question that arises is: will this process ever converge? That is, can we ever reach the point where it is possible to definitively say that a compiler correctness theorem is as strong as it ever needs to be? I believe the approach I advocate in this thesis should help us to get a handle on this question.

Verisoft. One portion of the Verisoft project involved showing the correctness of a compiler from C0 (a C-like source language) to VAMP assembly [LP08]. The correctness proof is developed in the

[2]The latest release of CompCert is version 1.11 as of the time of writing.

Isabelle/HOL proof assistant. Like the CompCert and CLI proofs, the Verisoft effort defines the semantics of the source and target languages operationally.

Their correctness statement is:

Definition 1.1.5 (Simulation theorem)**.** *Let p be a C0 program, c^0 the corresponding initial configuration of the C0 machine, and d some well-formed initial assembler configuration which contains the compiled code $code_s(p)$ at address 0. Then, it holds for all steps i of the C0 machine executing program p that there exists an assembler step number $s(i)$ and an allocation function $alloc^i$ such that the C0 machine after i steps is consistent with the assembler machine after $s(i)$ steps.*

However, this is only true if the following requirements are fulfilled.

- *The program p has to be translatable for our compiler...*
- *We must not reach an error state up to step i of the C0 computation: $c^i \neq \bot$*
- *There must not be a stack overflow up to step i of the C0 computation: $\forall j \leq i : \neg \mathit{ovfl}_{stack}(c_j)$.*

Unlike the CompCert statement from above, this correctness statement does not seem to suffer from being weaker than what is true. It suffers the opposite problem, which is that it fails to abstract from the setting sufficiently. The reader must carefully parse and understand quite a bit to know what has been proved. Indeed, the definition of "consistent with the assembler machine" is a relation not given above and the (rather complicated) definition must be found elsewhere in the paper.

For some purposes, *preciseness* of the correctness statement is the most important factor. In particular, if the correctness theorem will be later used to establish some other facts of independent interest, then the believability of the statement of correctness is less critical than that it is not too weak. For such purposes, the statement above is perfectly adequate.

However, if we, the readers, are to independently understand the statement of correctness and believe that it is the right thing to prove, then this statement falls quite short of what we would like. This statement provides too much detail, and is too specific to the particular setting and languages involved.

As with most of the previous statements we have seen, this statement makes the implicit assumption that all programs are deterministic. Otherwise, this forward-direction simulation argument is not sufficient, for the same reason as before (the compiler could insert additional behaviors).

CompCertTSO. Ševčík et al. [ŠVZN+11] have developed a fork of the CompCert compiler that deals explicitly with issues arising from the weak memory models of modern hardware — in particular, they reason about the total store ordering (TSO) model found in x86 hardware. Like CompCert, CompCertTSO is verified in Coq.

In this setting, the determinism assumption relied on by other statements fails; both source and target programs are nondeterministic. Furthermore, as the source program is compiled down to the target hardware, certain implementation choices get made that reduce the amount of nondeterminism present. To deal with these issues, CompCertTSO defines its correctness statement in terms of measured backward simulations. In the following, S is a source language program and T is a compiled target language program.

Definition 1.1.6 (Measured backward simulations)**.** *A family of relations* $R^i : \mathsf{States}(S) \times \mathsf{States}(T)$, *indexed by elements* i *of a well-founded order* $>$, *is a* measured backward simulation *if, whenever* $s\ R^i\ t$ *and* $t \xrightarrow{ev} t'$ *for* $ev \neq \mathtt{oom}$, *then either*

1. $\exists s'. s \xrightarrow{\tau}{}^{*}\ s' \xrightarrow{\mathtt{fail}}$ *(s can reach a semantic error), or*

2. $\exists s'\ j.\ s \xrightarrow{\tau}{}^{*} \xrightarrow{ev} s' \wedge s'\ R^j\ t'$ *(s can do a matching step), or*

3. $\exists j. ev = \tau \wedge i > j \wedge s\ R^j\ t'$ *(t stuttered, with a decreasing measure).*

The overall correctness statement is then that the initial state of the source program is related to the initial state of the target program by a measured backward simulation. Note the clause $ev \neq \mathtt{oom}$; the special event `oom` represents running out of memory. Therefore, these backward simulations say nothing about what happens if the target program reaches a state where allocation fails. In contrast, standard CompCert assumes an unbounded memory model where allocation always succeeds. See §11.1 for a discussion of this issue.

This particular definition is now quite close to the style of definition I will advocate later in this thesis. It is slightly intimidating to look at, but the definition fits easily onto one page, and is pretty generic with respect to the setting and languages.

However, it, too, suffers from stating a condition that is weaker than what is true. By requiring only backward simulation, this definition fails to preserve any progress properties. In particular, it allows any program whatever to be compiled into a program that immediately deadlocks. Of course, the compiler doesn't actually do this, and we would like the correctness result to tell us so. Later in this thesis, I will examine a definition that precisely captures those progress properties that are preserved even under nondeterministic refinement (cf. chapter 6).

Vellvm. The Vellvm project [ZNMZ12] is concerned with formalizing the semantics of the Low Level Virtual Machine (LLVM) in Coq with the aim of proving the correctness of a number of interesting LLVM-to-LLVM optimization passes. In a recent paper, Zhou et al. illustrated this basic idea by implementing a simple program transformation that improves the safety of an LLVM program. Essentially, they insert enough dynamic checks to transform a program that goes wrong into one that instead aborts predictably with an error.

Theorem 1.1.7 (SoftBound is correct)**.** *Let* SBtrans$(P) = \lfloor P' \rfloor$ *denote that the SoftBound pass instruments a well-formed program P to be P'. A SoftBound instrumented program P' either aborts on detecting spatial memory violations or preserves the LLVM semantics of the original program P. P' is not stuck by any spatial memory violation.*

This statement, like the CompCert statement above, only claims the preservation of program semantics for safe programs. For input programs that go wrong, all the theorem says is that the output program will abort — it makes no claims about the program behavior before then.

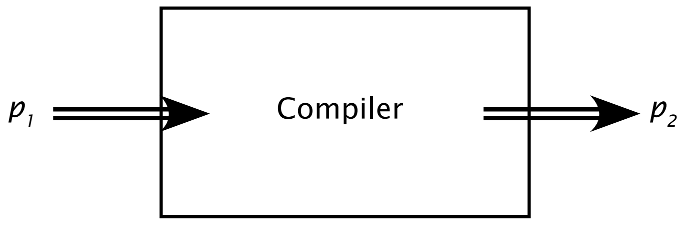
This claim, like several previously, is weaker than what is true. In fact, the statement in the paper does not seem to appear anywhere in the formal proof development; it is a paraphrase of the formal statement, which is much closer in style to the Verisoft statement from above (it claims the preservation of an explicit forward simulation relation).

The fact that the authors paraphrased their own result (but did not bother to formally prove it as stated) speaks, I think, to their desire to present a more “digestible” version of the result that can be easily understood by their readers.

1.1.1 A Strawman Proposal

The main thing to note about these results is that no two of them are stated the same way. There are some similarities, naturally, but the results above are not obviously positioned in some overarching framework that would allow them to be easily compared and understood. More generally, it seems to me that none of the results above is *obviously the correct statement.*

Perhaps obvious correctness is too much to hope for. If we look at the analogous case of Turing’s thesis, one is hard pressed to argue that Turing machines are *obviously* the right mathematical abstraction of computability. Instead, then, perhaps our task should be to pose a thesis about the meaning of compiler correctness and accumulate evidence for its appropriateness with the hope that we might eventually convince the community that we have arrived at the correct notion. To produce such evidence, we must show that our notion is not too strong, so that all “reasonable” program transformations are allowed, but also that it is not too weak, so that all interesting program properties are preserved.



1) Operational: same behavior
contextual equivalence, bisimulation, trace congruence $p_1 \cong p_2$

2) Denotational: same denotation
domain theory, trace/failures, games $[\![p_1]\!] = [\![p_2]\!]$

3) Deductive: same specifications
Hoare logic, temporal logics $p_1 \models f \text{ iff } p_2 \models f$

4) Algebraic: provably equal
$\beta\eta$-equivalence, CCS equivalence $\vdash p_1 = p_2$

Figure 1.1: Compiler correctness: a strawman proposal

In posing such a thesis, we can draw from the rich vocabulary developed in the last fifty or so years of programming language research. Our first attempt at such a thesis is given below:

Thesis (Same meaning correctness)**.** *Suppose we have two programs: p_1, in some input language, and p_2 in some output language. p_2 is an* acceptable translation *of p_1 if both programs have the* same meaning.

Given this, our notion of compiler correctness is clear: a correct compiler, when given an input program p_1, produces a target program p_2 that is an acceptable translation of p_1. That is, correct compilers produce compiled programs with the same meaning as their input programs.

The crux of this thesis is how we define "same meaning" — that is, what program equivalence should we use? Depending on one's philosophical bent, one may chose any of the four major approaches (operational, denotational, deductive or algebraic) as ground truth for what constitutes program meaning. It is currently popular to take the operational approach as ground truth (especially for low-level languages) but this is certainly not universal. Figure 1.1 lists some of the most well-known notions of program equivalence.

This proposal based on "same meaning" has a certain basic appeal. It seems to fit very closely with most people's intuition about how compilers should behave, and it allows us to leverage the considerable research effort that has been spent over the years developing and understanding various kinds of program equivalences.

Unfortunately, this strawman proposal breaks down in the face of the ugly realities of compiling realistic languages. The examples below should go a long way toward explaining why none of the compiler correctness statements we have examined are stated using the strawman “same meaning” framework.

1.1.2 Dismantling the Strawman

In this section, I will demonstrate some example programs, written in C, that demonstrate the limitations of the strawman proposal given above. Here I apply the semantics of C as defined by the 2011 ISO C standard [ISO11]. Each example below exercises one of the ugly corners of the C standard. However, the problems I will illustrate are not specific to C; similar issues crop up frequently in languages designed at approximately the same level of abstraction.

First, consider the following two programs.

```
 1 int foo() {
 2     int i;
 3 
 4     for(i=0; i<2; i++) {
 5         int x;
 6 
 7         if(i==0) {
 8             x = 0;
 9         } else {
10             x++;
11             return x;
12         }
13     }
14 }
```

Listing 1.1(a)

```
 1 int bar() {
 2     int i;
 3 
 4     int x;
 5     for(i=0; i<2; i++) {
 6 
 7         if(i==0) {
 8             x = 0;
 9         } else {
10             x++;
11             return x;
12         }
13     }
14 }
```

Listing 1.2(b)

The only difference between foo and bar occurs in lines 4 and 5; it relates to the placement of the declaration **int** x. For foo, it appears inside the scope of the for loop, whereas for bar, it appears outside. Should a C compiler be allowed to transform the function foo into the function bar?

The process of pulling variable declarations inside block scopes into outer scopes (renaming as necessary to avoid clashes) is called *scope extrusion*. C compilers usually will perform scope extrusion, or some other transformation that has the same effect, early in compilation process. This is necessary to perform standard compiler passes, like register allocation and stack frame layout. For example, `gcc` (version 4.4.3 with i486-linux-gnu target) compiles the functions foo and bar to identical assembly language, up to naming of labels. CompCert performs a scope extrusion pass very early in its compilation pipeline.

However, it is *not* the case that these two programs have the same meaning. According to the ISO C standard, foo will exhibit *undefined behavior* whereas bar will return the value 1. This is because foo, in the second iteration of the for loop (at line 10), will access the value of an uninitialized variable. Although x was written in the first iteration (at line 8), the program left the scope of its declaration at the end of the loop body (line 13). The x accessed at line 10 is a fresh, uninitialized variable and accessing it causes undefined behavior.

For bar, the value of x is maintained across iterations of the loop because its declaration appears outside the scope of the loop body — thus x is initialized in the first iteration and used in the second.

C compilers are justified in performing scope extrusion because: scope extrusion produces a semantically identical program if every variable is written before it is read, and because the C standard defines reading from an uninitialized variable to produce "undefined behavior." Clause 3.4.3 of the C standard defines how the term "undefined behavior" should be interpreted:

> **undefined behavior** [is] behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements
>
> NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a manner characteristic of the environment..., to terminating a translation or execution...

Essentially, because executing foo will cause the program to go wrong, the compiler is justified in translating it into a function with arbitrary behavior; bar is just one of the infinite possibilities.

Note that the distinction between foo and bar is not just a curiosity. foo contains no loop-carried dependencies (because the lifetime of x does not extend beyond the loop body), whereas bar does. In a realistic program, the absence of such loop-carried dependencies may enable certain aggressive loop optimizations, so it seems we should not just decide that foo means the same thing as bar.

Thus, we must acknowledge that foo and bar do not have the same meaning, but that nonetheless the compiler is justified in translating foo into bar.

The following listing presents a second example in the same vein. This example relates to coalescing locally declared arrays.

```
int foo() {
  int x[10];
  int y[10];

  y[2] = 42;
  return x[12];
}
```

Listing 1.3(a)

```
int bar() {
  int xy[20];
  int* x = xy + 0;
  int* y = xy + 10;

  y[2] = 42;
  return x[12];
}
```

Listing 1.4(b)

In this example, we have merged the arrays x and y into a single array, with the original arrays being placed in disjoint areas of the new one. This merging is quite similar to what happens when the compiler lays out local data in the stack frame.

Function foo contains a classic out-of-bounds memory access error, resulting in undefined behavior, but bar returns a well-defined result (42). Once again bar is just one of the possible behaviors allowed by the C standard.

A third example illustrates a different issue, having to do with the separate notion of *unspecified behavior*.

```
int foo() {
  int x;

  x = f1() + f2();

  return x;
}
```

Listing 1.5(a)

```
int bar() {
  int x;

  x = f1();
  x = x + f2();

  return x;
}
```

Listing 1.6(b)

```
int baz() {
  int x;

  x = f2();
  x = x + f1();

  return x;
}
```

Listing 1.7(c)

During compilation of C programs, complex expressions are generally reduced to a series of simpler statements, assigning intermediate values to new temporary variables as necessary. In the end, the evaluation of an expression is completely serialized so that each step only performs some basic operation. In C, expression evaluation can have side-effects, including assignment to local or global variables, and calling functions. In what order should these side-effects occur? Should we compile foo into bar or baz?

According to the C standard, both are acceptable translations! This is because the order in which subexpressions get evaluated is an example of unspecified behavior, as defined by clause 3.4.4 of the C standard:

> **unspecified behavior** [is] use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance
>
> EXAMPLE An example of unspecified behavior is the order in which arguments to a function are evaluated.

The C standard call attention to the fact that the order of evaluation for function arguments is unspecified, but it is also the case that the order of evaluation with respect to arithemetic operators (like addition) is unspecified. Thus, both bar and baz are acceptable translations — clearly they are not equal. Once again, we must acknowledge that the notion of "acceptable translation" cannot be reduced to the idea of "same meaning."

1.1.3 Refinements for Compiler Correctness

Fortunately, all is not lost. We can salvage the situation by allowing "acceptable translation" to be a preorder on programs rather than an equivalence. A preorder is simply a reflexive and transitive binary relation — compared to an equivalence, we drop the requirement that the relation be symmetric. Such preorders on program meaning are usually called *refinements* and, intuitively, they capture the notion that programs are allowed to improve in some way.

Thus, we present a modified thesis:

Thesis (Improvement correctness)**.** *Suppose we have two programs: p_1, in some input language, and p_2 in a target language. p_2 is an* acceptable translation *of p_1 if p_2 is an improvement over p_1.*

Just as before, the crux of the definition is the meaning of "improvement." Improved in what sense? For Müller-Olm, it basically meant "satisfies more total correctness properties."

In this thesis, I will examine two different notions in improvement that are related to the two problems illustrated above. First, I will examine improvement with respect to undefined behavior. For this type of refinement, an improved program will exhibit undefined behavior (go wrong) in

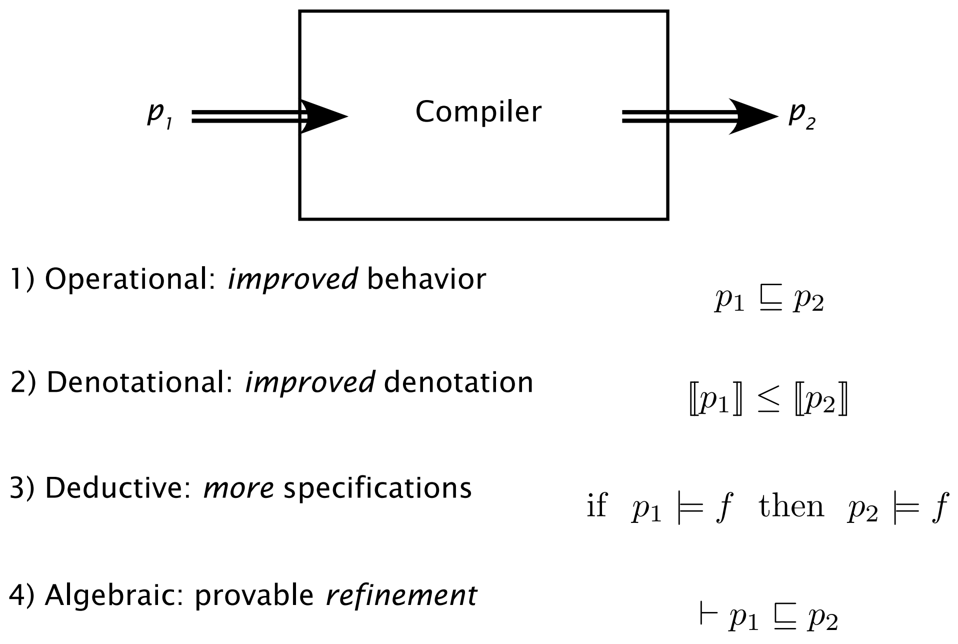

p_1
Compiler
p_2
1) Operational: *improved* behavior
$p_1 \sqsubseteq p_2$
2) Denotational: *improved* denotation
$[\![p_1]\!] \leq [\![p_2]\!]$
3) Deductive: *more* specifications
if $p_1 \models f$ then $p_2 \models f$
4) Algebraic: provable *refinement*
$\vdash p_1 \sqsubseteq p_2$

Figure 1.2: Compiler correctness via refinements

fewer situations. This corresponds to the fact that compilers are allowed to turn undefined behavior into arbitrary well-defined behavior. Second, I will examine refinement with respect to specification nondeterminism, in which improved programs have less nondeterminism. This corresponds to the fact that compilers are allowed to resolve (certain kinds of) nondeterministic choices when compiling programs. For example, a compiler may choose to evaluate expressions using a standard left-to-right evaluation order.

In fact, I shall make a considerably more concrete proposal, delineated below.

1. Take the operational point of view as primary; in particular, consider small-step operational semantics. That is, for each language of interest, take the definition of program meaning to be an assignment that sends each program to some transition system that describes the way program execution proceeds step-by-step.

2. Define refinements between program meanings using bisimulation methods.

3. Choose the most discriminating relation between program meanings that is available, subject to having a "nice" theory and allowing the desired class of program transformations.

4. Relate the chosen operational notion of refinement to an associated temporal logic of program specifications. This temporal logic will help us to understand exactly what program properties are guaranteed to be preserved by refinement.

$$\begin{tikzcd}
& \sqsubseteq_{\mho\Delta B} \arrow[rr, dash] \arrow[dl, dash] \arrow[dd, dash] & & \sqsubseteq_{\mho CB} \arrow[dl, dash] \arrow[dd, dash] \\
\sqsubseteq_{\mho} \arrow[rr, dash] \arrow[dd, dash] & & \sqsubseteq_{\mho C} \arrow[dd, dash] & \\
& \cong_{\Delta B} \arrow[rr, dash] \arrow[dl, dash] & & \sqsubseteq_{CB} \arrow[dl, dash] \\
\cong \arrow[rr, dash] & & \sqsubseteq_{C} &
\end{tikzcd}$$

B	Branching bisimulation/refinement
Δ	Divergence sensitive
$\mho$	Behavioral refinement
C	Choice refinement

Figure 1.3: The refinement cube

I shall attempt a detailed defense of these points in chapter 2. For now, let me simply say that the operational point of view is by far the most popular among serious compiler verification efforts.

1.2 Thesis Outline

Following the program outlined above, I start with the theory of strong bisimulation on labeled transition systems. This theory is generally acknowledged to be the strongest relation between the operational meaning of programs that "makes sense." To obtain a theory that is appropriate for reasoning about realistic compilers, we will have to weaken the theory in three separate ways: by abstracting away from the number of internal computation steps taken by a process, by allowing refinement with respect to undefined behavior, and by allowing refinement with respect to unspecified behavior. Each of these three enhancements can be applied orthogonally — starting from the theory of strong bisimulation, this results in 8 separate theories. These theories are organized into a cube, as in figure 1.3.

The main technical content of this thesis involves examining the theory of these 8 systems. Before diving into the main results, however, chapter 2 lays out some preliminary definitions and concepts that will be used throughout the remainder of the thesis. In chapter 3, I shall examine the bottom-left corners of the cube. These corners correspond to the theory of strong bisimulation and the theory of (divergence-sensitive) branching bisimulation. Both theories represent previous work and are well-known. I review these theories because the basic sequence of definitions and results forms a skeleton that will be repeated in later chapters. The differences that arise from refinement can thus be seen in sharp relief against the foundational symmetric theories.

Chapter 4 contains the development of the theory of behavioral refinement, which allows programs to be improved by going wrong less often. This corresponds to the top-left vertices in the cube. In chapter 5, I take a detour to examine the proof theory of the characteristic logic for strong behavioral refinement. This examination will help us to understand what principles of reasoning are appropriate when considering programs that may have undefined behavior.

Next, in chapter 6, I examine a theory of choice refinement, which captures improvement with respect to nondeterministic choices. These systems are in the bottom-right of the cube. Chapter 7 examines the combination of choice and behavioral refinement, corresponding to the top-right corners of the refinement cube.

Once I have completed an examination of the basic theories, I will attempt to address the question of whether these theories can actually be used to handle the sorts of problems we examined above. In chapter 9, I will present a small case study language, just sophisticated enough to exhibit the problems with local variable scopes and expression evaluation order. I will show that the theory of behavioral choice refinement is sufficient to justify both scope extrusion and the specialization from an unspecified evaluation order to a deterministic one (left-to-right evaluation).

In chapter 10 I will discuss how these ideas relate to the current state of the CompCert C compiler and show how, with only minor changes, the correctness statement for CompCert could be modified to directly use the refinements presented here. In particular, I will show that the vast majority of the CompCert compiler requires *no changes at all* to be proved correct with respect to the behavioral refinements of chapter 4. I give a generic construction proving that forward simulations are sufficent when the target language is deterministic. This generic construction is then straightforward to apply to the main CompCert correctness lemma for the deterministic portion of the compiler. Furthermore, I believe only minor changes would be required to prove the remainder correct with respect to the behavioral choice refinement of chapter 7.

Finally, in chapter 11, I will examine some possible directions for future work in this area and make some concluding remarks.

Chapter 2

Preliminaries

In this chapter, I have collected together some topics that apply to the whole proof development and do not easily fit into the discussion in later chapters.

2.1 Metalogic of the Formalization

All the definitions and proofs appearing in this thesis (except where specifically indicated) have been formalized and machine checked using the proof assistant Coq [Coq]. The metalogic of the proof assistant itself is the Calculus of Inductive Constructions (CiC) [CH88, CP90], a dependently typed lambda-calculus closely related to Martin-Löf's type theory [Mar73]. Interpreted as a logic under the types-as-propositions view, CiC is a higher-order logic with inductive data types and (terminating) recursive functions.

As with most variants of type theory, CiC is a *constructive* logic, in that it eschews all forms of choice principles and the axiom of the excluded middle. The logic also fails to validate most forms of extensionality principles; most notably, the principle of functional extensionality is not available by default.

One may extend the logic by asserting additional axioms, which the proof system will accept without proof. Using this mechanism, one can augment the base metalogic with the excluded middle, various choice principles, extensionality, etc. as desired. Obviously, an injudicious choice of axioms can lead to an inconsistent logic, so one is advised to use extreme care when asserting axioms.

Certain sets of axioms are known to be consistent additions to the metalogic, relative to the consistency of other well-known logical systems (e.g., ZFC). In particular, the addition of functional extensionality, propositional extensionality, the excluded middle, and relational choice are known to be consistent (up to ZFC with inaccessable cardinals) [Wer97], as is any subset of these axioms.

To the extent possible, I wish to retain the constructive character of the logic, but I consider extensionality constructively acceptable. Thus I will freely make use of extensionality principles, strictly avoid any choice principles, and use the excluded middle only where it seems unavoidable. My goal is to develop all the basic theory of refinements and the soundness directions of proofs without using EM, and to allow the use of EM only for the completeness directions of proofs. This goal has largely been achieved. Whenever a proof relies on the axiom of the excluded middle (either directly, or via some applied lemma), the statement of the proof will be marked with the symbol $^{\mathsf{EM}}$ to indicate such use.

Occasionally I will present some concepts and definitions in this thesis in a way that differs slightly from the formalized proof. This is usually done to achieve a more readable notation. Whenever this occurs, or whenever there is some point regarding the formalization that is not immediately clear from the informal presentation, I will include a note, as follows:

Note on formalization. *This indicates a note regarding details of the formalization.*

Readers uninterested in the details of the formalization may safely skip these notes.

The main part of the proof development, covering all the concepts and proofs from this thesis, apart from the case study of chapter 9, totals about 38K lines of code, including comments and whitespace. The proofs from chapter 9 total another 15K lines of code. I have not been especially careful to produce short, beautiful proof scripts, so those totals could perhaps be significantly reduced. For comparison, the proof scripts from CompCert version 1.11 total about 112K lines of code. All proofs have been developed and tested using Coq version 8.3pl1. The formal proof development may be downloaded from the author's personal website. Those readers interested in additional detail beyond what can reasonably be covered here are encouraged to consult the formal proof development.

2.2 Labeled Transition Systems

The basic substrate I will use to study operational semantics is the labeled transition system (LTS). In its most basic form, the labeled transition system is simply a set of states equipped with a relation indicating which states can transition to which other states, and what observable event is produced by the transition. Mathematically (and pictorially) a labeled transition system is just a directed graph where the arcs carry labels.

Definition 2.2.1 (Labeled Transition System)**.** *Suppose we have fixed in advance a set $\mathcal{O}$ of observable transitions. Then a* Labeled Transition System *(or LTS) is a tuple $\langle \mathcal{S}, \longrightarrow \rangle$ where $\mathcal{S}$ is a set of states and $\longrightarrow \subseteq \wp(\mathcal{S} \times \mathcal{O} \times \mathcal{S})$ is a transition relation. We write $x \xrightarrow{o} x'$ for $(x, o, x') \in \longrightarrow$.*

We can model a computational process by specifying an LTS and and a current state. Then we imagine the computational process proceeding step-by-step, as the current state transitions to a new current state along one of the arcs. The behavior of the process is determined by the labels on the arcs; in other words, observers can only distinguish computations by the actions they engage in, not by the identity of the states.

LTSs are an appealing substrate for reasoning about operational semantics because they are both *simple* and *expressive*. The LTSs themselves have almost no structure, which makes them easy to define. Furthermore, the idea of states with transitions between them is a simple and intuitive way to think about computational systems. Almost any notion of computation can be embedded into an LTS — the states can represent the states of some abstract machine (e.g., a Turing machine), a term in a reduction calculus (e.g., λ-calculus terms), the status of a proof search procedure (e.g., a Prolog state), etc. Structural Operational Semantics (SOS) is a popular technique for defining operational semantics in terms of labeled transition systems [Plo81, AFV01].

The main drawback with LTSs as substrates for defining operational semantics is that they are quite *intensional*; that is, they specify too much detail about how computation proceeds. The states of an LTS and the shape of the graph generated by a particular program are details we wish to ignore when considering the semantics of programs. Contrast this with the denotational approach to programming language semantics, where the goal is to build some appropriate abstraction so that irrelevant details about the way the program is constructed are invisible.

The usual solution to this problem is to construct some equivalence or preorder over LTSs which abstracts away from this irrelevant detail. The issue with this approach is that there are a multitude of different ways one can do this. The academic literature contains literally hundreds of different variations. Unfortunately, this amazing variety of choices leaves us with a puzzling question: which relation should we use?

A major goal of this thesis is to help guide the program and compiler verification community to make the "right" choices for their problem domains. As such, much of this thesis is devoted to explaining *why* I believe a particular notion is the correct one for the domain of program transformations. For now, I will simply say that the design space for constructing these relations is quite large, so one must spend some effort to defend any particular definition.

Processes. A labeled transition system describes the potential ways a program may execute. However, we usually think of a running process as being in some particular state. Thus, to fully specify a process, one needs to specify not only the LTS, but also some particular "current" state within that LTS.

Definition 2.2.2 (Process)**.** *Suppose $\mathcal{L}$ is some LTS, and $x \in \mathcal{S}$ is some state from $\mathcal{L}$. Then the pair $\langle \mathcal{L}, x \rangle$ is a* process*. Frequently, I will abuse notation and simply write the state x to stand for the process $\langle \mathcal{L}, x \rangle$ when $\mathcal{L}$ can be inferred from context. Likewise, I will freely write a process in positions where a state is expected.*

The basic model I have in mind is that a programming language is specified by giving some syntax together a mapping that sends each program to a process; the state of this process can be thought of as the "initial" state of the program. Compilers manipulate program syntax, but the refinement notions I describe here discuss processes. The mapping from syntax to processes (i.e., the operational semantics) is what connects these notions.

Process histories. Sometimes I will be interested in a more sophisticated notion of process *histories* that record not just the current state of a process, but also the sequence of previous states visited by a program execution and the observable actions produced. Process histories behave a bit like the back button of a web browser. Navigating through links causes the browser to keep track of the sequence of web pages you have previously visited; pressing the "back" button pops the most recent page off the stack and displays it again. From there you can now follow new links along a different path than the one you first explored.

Process histories are a technical device that will allow us to define observations on processes that "go back in time" to previously-visited states. This additional observational power, while not completely intuitive, neatly characterizes branching bisimulation, the notion of bisimulation I have chosen to handle internal actions. See chapter 3 for a discussion of why branching bisimulation is the most appropriate choice.

Definition 2.2.3 (Process history)**.** *Suppose $\mathcal{L}$ is an LTS and let n be a nonnegative integer. Let $x_0, \cdots, x_n$ be a (nonempty) sequence of states, and let $a_1, \cdots, a_n$ be a (possibly empty) sequence of actions. Then the sequence $[x_0, a_1, x_1, a_2, \cdots, a_n, x_n]$ is a* process history *if $x_i \xrightarrow{a_{i+1}} x_{i+1}$ for each $i < n$.*

As a diagram, we may write this condition as:

$$x_0 \xrightarrow{a_1} x_1 \xrightarrow{a_2} x_2 \cdots x_{n-1} \xrightarrow{a_n} x_n$$

Let h be some history as defined above. I write $\mathsf{init}(h)$ *to mean the initial state x_0, and I write* $\mathsf{curr}(h)$ *to mean the current state x_n.*

Note on formalization. *In the formal proof, histories are represented as a dependent record containing 1) a list of pairs of state and actions 2) the current state, and 3) a proof that each state/action pair steps to the next in the sequence.*

2.3 Refinements

As discussed in the introduction, I will be primarily concerned with finding *refinement* relations that describe when it is allowable to transform an input program to an output program. For now, I wish to simply lay down some ground rules that will help to navigate the design space and allow us to rule out some "obviously" inappropriate choices.

First, a bit of notation: I will generically use the symbol $\sqsubseteq$ to stand for a refinement relation satisfying the conditions below. When the identity of a particular refinement cannot be easily inferred from the context, subscripts will be used to identify which refinement is meant. In the special case where the refinement is also symmetric (and thus an equivalence relation), I use the symbol $\cong$ instead.

Condition 1: reflexivity. This basic condition simply lets us know that the null transformation (which returns the program unchanged) is legal. From a practical standpoint, this condition is perhaps not so compelling, although it seems rather strange to contemplate a notion of program transformation that fails to be reflexive. When it comes to the semantics of characteristic logics, this condition will be important to certain technical arguments, and so it seems appropriate to require it as a basic sanity condition.

Condition 2: transitivity. This condition lets us know that we can take two valid transformations and connect them together in sequence. This property is absolutely critical if we wish to verify realistic compilers. Compilers are almost always broken down into a series of passes or phases, each of which does a specific part of the overall compilation process. In order for our verification to scale, we must be able to verify each phase independently and then combine the results for each phase to get an overall correctness theorem. Transitivity is precisely the property that allows us to do this.

Condition 3: greatest fixpoint of a monotone operator. Each of the refinements/equivalences we will examine will be specified in basically the same way. First, we will specify a set of binary relations on processes satisfying some conditions; call this set X. Then we will say that program x is refined by program y if there exists some relation $R \in X$ such that $(x, y) \in R$.

Although I will usually not explicitly point this out, the class X will always be generated by some monotone operator F, so that $X = \{R \mid R \subseteq F(R)\}$. Then the refinement relation we wish to define is $\bigcup X$, the union of all the relations in X. By the Knaster-Tarski fixpoint theorem [Tar55], $\bigcup X = F(\bigcup X)$ is the greatest fixpoint of F. Furthermore, X is closed under unions, so $\bigcup X \in X$ and $\bigcup X$ is the largest relation in X.

This approach using greatest fixpoints gives us notions based on the "equivalence of indistinguishables;" by taking the largest relation closed under F, we relate as many programs as possible, unless they can somehow be distinguished by the conditions specified in the definition of F. In other words, the operator F can be seen as defining the powers of an external observer. If the observer cannot "observe" some difference between programs using the powers granted to him, the programs must be equivalent.

Condition 4: the finer, the better. This condition is not formal in the same way as the others; it is perhaps more of a "guiding principle." What I mean is that where there is a choice between several alternatives, each of which have satisfactory theories, I will prefer to choose the one which distinguishes the most programs. This choice "breaks ties" in a way that preserves as many program properties as possible. This is appropriate for our compiler correctness setting where we do not know in advance what properties our users will care about. Choosing the finest available relation that still allows our correctness proof to go through will satisfy the needs of as many use cases as possible.

Note that there is some tension between conditions 3 and 4. Condition 3 drives us to choose *large* relations, and demands that we relate programs unless there is some reason we must distinguish them. Condition 4 drives us to choose *small* relations that preserve as many program properties as possible. This tension is perfectly natural, because it reflects the tension that arises from the problem — how to choose a relation that relates enough programs so that our compiler correctness theorems will go through, but not too many, so that the resulting theorem says something meaningful.

2.4 Logical Characterizations

For each of the refinements I consider, I will also present a characteristic logic. The formulae of the characteristic logics may be thought of as program specifications. Following the notion of the equivalence of indistinguishables, a characteristic logic delineates observations that may be made by observers in a manner we shall make precise briefly. The main idea is that two programs are indistinguishable if they satisfy all the same specifications.

We use characteristic logics to gain understanding of the refinements we define, and to gain confidence that we have defined the right notion. If all the specifications we care to prove about programs can be expressed in the characteristic logic, then we know that the refinement we have defined is sufficiently tight. On the other hand, if all the program transformations we desire can be proved sound with respect to the refinement, we know that it is sufficiently loose.

The characteristic logic is also of practical use, in the sense that it gives us a vocabulary for discussing program verification. We may define the *program verification* task as the task of proving that a program satisfies some particular specification of interest. The *compiler verification* task involves showing that a compiler preserves the truth of all specifications. If we compile a verified program with a verified compiler, we are in the desirable situation where we know that the compiled program satisfies the specification of interest.

The most basic requirement we will make of a specification logic is that it must be sound with respect to the refinement.

Definition 2.4.1 (Soundness)**.** *Suppose $\sqsubseteq$ is a refinement relation, and suppose $\mathcal{F}$ is a set of formulae. We write $x \models f$ to mean that the process x satisfies specification f.*

The specification logic $\langle \mathcal{F}, \models \rangle$ is sound *for $\sqsubseteq$ if the following holds for all x and y:*

$$x \sqsubseteq y \qquad \text{implies} \qquad \forall f \in \mathcal{F},\ x \models f \ \Rightarrow\ y \models f$$

Soundness means that every specification in $\mathcal{F}$ is preserved by refinement. In practical terms, this is usually the property we are interested in when it comes to compiler correctness concerns; we want to know that the specifications we prove are preserved by the compilation process.

However, we also might wish to know that our logic is a good one, in the sense that it allows us to write down as many specifications as possible. There are several ways we can say that a logic characterizes a refinement relation. The weakest of these is called *adequacy* by Pnueli [Pnu85], and is simply the converse of soundness.

Definition 2.4.2 (Adequacy)**.** *Suppose $\sqsubseteq$ is a refinement relation, and suppose $\mathcal{F}$ is a set of formulae. We write $x \models f$ to mean that the process x satisfies specification f.*

The specification logic $\langle \mathcal{F}, \models \rangle$ is adequate *for $\sqsubseteq$ if the following holds:*

$$\forall f \in \mathcal{F},\ x \models f \ \Rightarrow\ y \models f \qquad \text{implies} \qquad x \sqsubseteq y$$

This definition means any two programs distinguishable under refinement are distinguishable by some formula of the characteristic logic. In other words, the logic and the refinement have the same distinguishing power.

Despite the fact that adequacy says that there are somehow "enough" formulae, a logic that is merely adequate is frequently too weak for practical use. This is because it might require an infinite set of formulae to completely specify the meaning of a program, or indeed, to capture some specification of interest.

Thus, we might wish to require a strong notion called *expressivity* [Pnu85].

Definition 2.4.3 (Expressivity)**.** *Suppose $\sqsubseteq$ is a refinement relation. Then a specification logic $\langle \mathcal{F}, \models \rangle$ is* expressive *for $\sqsubseteq$ if, for each process x, there exists a characteristic formula $\mathsf{CF}(x)$ such that, for all y,*

$$x \sqsubseteq y \qquad \text{iff} \qquad y \models \mathsf{CF}(x)$$

Expressivity hinges on the notion that there is a *single* formula that completely specifies the behavior of a program. In essence, the characteristic formula gives a logical account of the program's semantics. It is not hard to see that expressivity implies adequacy.

There is yet a third, even stronger notion we might require, that I call *denotability.*

Definition 2.4.4 (Refinement closed)**.** *Suppose P is a set of programs. We say P is* refinement closed *if, for all $x \sqsubseteq y$, $x \in P$ implies $y \in P$.*

Definition 2.4.5 (Denotability)**.** *Suppose $\sqsubseteq$ is a refinement relation. Then a specification logic $\langle \mathcal{F}, \models \rangle$* denotes the specifications for $\sqsubseteq$ *if, for each refinement-closed set P, there exists some formula $\mathsf{interp}(P) \in \mathcal{F}$ such that:*

$$z \in P \qquad \text{iff} \qquad z \models \mathsf{interp}(P)$$

Denotability means that every refinement closed set of programs can be denoted by some formula in the specification logic. This is the strongest notion of a connection between refinement and a characteristic logic that it seems reasonable to posit. It implies that *any* property of programs, no matter how complicated or difficult to describe, can be written down in the language of the characteristic logic.

Denotability implies expressivity because we can take $\mathsf{CF}(x)$ to be $\mathsf{interp}(\{y \mid x \sqsubseteq y\})$. In general, denotability may be strictly stronger than expressivity. For the logics we will examine in this thesis, however, they will go hand-in-hand. See theorem 2.5.7 below.

2.5 Logical Skeletons

Throughout this thesis (and in the formal proof development), I will need to perform a series of very similar constructions to build the specification logics for the various refinements I consider. Here I have factored out the common elements of these constructions so they can be discussed just once. Then, in subsequent chapters, I will invoke these constructions, varying the parameters as appropriate for each setting.

The logics used in this thesis are all basically variations of branching-time temporal logics. The differences between them boil down to what modalities and atomic propositions exist and their meanings. The meaning of each formula is given by a Kripke-style possible worlds semantics [Kri63]. In these settings, the "worlds" will represent either program states or sequences of program states, and the Kripke semantics can be interpreted as defining when a program satisfies a specification.

To use either of the constructions I describe below, a user must provide the following data.

Definition 2.5.1 (Modal logic specification)**.** *To specify a logic, the user must provide:*

- *A set $\mathcal{W}$ of* worlds*,*
- *$\sqsubseteq$, an* accessibility preorder *on $\mathcal{W}$,*
- *a set $\mathcal{A}$ of* atoms*,*
- *a set of $\mathcal{M}$ of* modes*,*
- *and an interpretation function for modes and atoms.*

When $m \in \mathcal{M}$, we write $[\![m]\!]_{\mathcal{M}}$ to mean the interpretation of m, which must be a function $\wp(\mathcal{W}) \to \wp(\mathcal{W})$. Likewise, for $a \in \mathcal{A}$, $[\![a]\!]_{\mathcal{A}}$ is an interpretation of atoms, in $\wp(\mathcal{W})$.

The interpretation of each mode must be monotone (with respect to set inclusion) and preserve refinement-closure, and the interpretation of atoms must be refinement-closed. We use the symbol $\mathcal{X}$ to range over such specifications, which are tuples of the form: $\langle \mathcal{W}, \sqsubseteq, \mathcal{A}, \mathcal{M}, [\![-]\!]_{\mathcal{A}}, [\![-]\!]_{\mathcal{M}} \rangle$.

Note on formalization. *In the informal presentation, I will typically use the vocabulary of set theory, as is consistent with usual practice. However, the formal proof is done in type theory, where sets,* per se, *are not available. In the formal proof the word "set" is usually translated to* `Type`, *the sort of types in CiC. However, it will sometimes instead mean a predicate on types,* `A -> Prop`, *for some type* `A`. *In particular power sets, e.g.,* $\wp(A)$, *are essentially always understood as* `A -> Prop`. *Relations, likewise, are usually specified in curried form. For example, a binary relation on worlds will have type* `world -> world -> Prop`.

The specification of a modal logic is defined as a dependent record in the formal proof, and the logic constructions described below are implemented taking the input data as a parameter.

Once the user has supplied these data, we can construct two different flavors of logic. The first is a simple propositional logic containing the modes and atoms specified by the user. The propositional logics will be suitable for adequacy proofs, and are similar to most presentations found in the literature. The second logic we can construct is based on the modal μ-calculus [Koz83]; in addition to the basic propositional connectives, it also has operators for fixpoints, infinite conjunction and disjunction, and for building relations in addition to propositions. This additional power will allow us to prove expressivity and denotability.

2.5.1 A Propositional Skeleton

Suppose the user has supplied a logic specification $\mathcal{X} = \langle \mathcal{W}, \sqsubseteq, \mathcal{A}, \mathcal{M}, [\![-]\!]_{\mathcal{A}}, [\![-]\!]_{\mathcal{M}} \rangle$. Let a range over atoms in $\mathcal{A}$, and let m range over modes in $\mathcal{M}$. Then the formulae of the propositional logic induced by $\mathcal{X}$ are defined inductively as follows:

$$f ::= \mathsf{atom}\ a \ \mid\ \mathsf{mode}\ m\ f \ \mid\ \top \ \mid\ \bot \ \mid\ f \wedge f \ \mid\ f \vee f \ \mid\ f \Rightarrow f$$

Although we define the syntax of modalities here using the mode literal, this literal will be elided when I define specific logics with concrete modes. That is, I will write $[o]P$ instead of $\mathsf{mode}\ [o]\ P$. This should always be understood as an abbreviation for the more verbose mode form. Likewise for atom.

To each formula, we assign a denotation: the set of worlds that satisfy that formula. We write $[\![f]\!]$ for the denotation of a formula, and we introduce the notation $w \models f$ as an abbreviation for $w \in [\![f]\!]$.

The denotations of formulae are defined by recursion on the structure of f via the following:

$$[\![\top]\!] \equiv \mathcal{W} \tag{2.1}$$

$$[\![\bot]\!] \equiv \emptyset \tag{2.2}$$

$$[\![f_1 \wedge f_2]\!] \equiv [\![f_1]\!] \cap [\![f_2]\!] \tag{2.3}$$

$$[\![f_1 \vee f_2]\!] \equiv [\![f_1]\!] \cup [\![f_2]\!] \tag{2.4}$$

$$[\![f_1 \Rightarrow f_2]\!] \equiv \{w \mid \forall w' \in \mathcal{W}.\ w \sqsubseteq w' \;\Rightarrow\; w' \in [\![f_1]\!] \;\Rightarrow\; w' \in [\![f_2]\!]\} \tag{2.5}$$

$$[\![\mathsf{mode}\ m\ f]\!] \equiv [\![m]\!]_{\mathcal{M}}([\![f]\!]) \tag{2.6}$$

$$[\![\mathsf{atom}\ a]\!] \equiv [\![a]\!]_{\mathcal{A}} \tag{2.7}$$

For a reader familiar with Kripke semantics of logics, all these clauses should be unsurprising, except perhaps the clause for $\Rightarrow$. To obtain a refinement-closed interpretation of implication, we must explicitly quantify over all $\sqsubseteq$-accessible worlds. This quantification over accessible worlds gives the logic an *intuitionistic* interpretation.

In the special case where $\sqsubseteq$ is symmetric, it is not hard to show that $\Rightarrow$ has the classical semantics one may have expected: $\overline{[\![f_1]\!]} \cup [\![f_2]\!]$.

We can now prove, once and for all, the soundness of propositional logic with respect to the accessibility preorder $\sqsubseteq$.

Proposition 2.5.2 (Soundness for propositional logic)**.** *For each $f \in \mathcal{F}$, the denotation of f, $[\![f]\!]$, is refinement-closed.*

Proof. By induction on f. The cases for $\top$, $\bot$, and $\Rightarrow$ are easy. The cases for $\wedge$ and $\vee$ follow from the induction hypothesis. The case for atoms relies on the client-supplied fact that the interpretation of atoms is refinement-closed. The case for modalities relies on the fact that the interpretation of modes preserves refinement-closure and the induction hypothesis. $\square$

In addition, we can prove soundness with respect to part of the proof theory of intuitionistic modal logic, including modus ponens, the rule of regularity for all modes, and the standard axioms of intuitionistic propositional logic.

2.5.2 A Modal μ-calculus Skeleton

As before, assume $\mathcal{X} = \langle \mathcal{W}, \sqsubseteq, \mathcal{A}, \mathcal{M}, [\![-]\!]_{\mathcal{A}}, [\![-]\!]_{\mathcal{M}} \rangle$ is a logic specification supplied by the user.

This presentation of the modal μ-calculus requires some more sophistocated ideas than the propositional logic from the previons section. In particular, we need a type system to rule out the possibility of writing down badly-formed formulae. Unfortunately, a presentation of this calculus that is faithful to the formal proof development is rather notationally heavy. Thus, before I delve into the details of the type system, here I give a high-level explanation of the connectives of the logic.

As with the propositional case, we have the basic propositions built from atom, mode and $\Rightarrow$, which behave pretty much like before. However, instead of binary conjunction and disjunction, we instead have infinite conjunction $\bigwedge$, and infinite disjunction $\bigvee$. The infinite connectives behave rather like universal and existential quantifiers. I include them so that it is possible to prove expressivity results without needing finiteness assumptions about processes.

In addition, I include least and greatest fixpoint operators, the distinguishing operators of the μ-calculus. The formula $\mu X.F(X)$ represents the smallest solution to the recursive equation $X = F(X)$, where F is a formula in which the variable X may occur free. In other words μ is the least fixpoint operator. Dually $\nu X.F(X)$ is the largest solution to the equation $X = F(X)$ — that is, ν is the greatest fixpoint operator. Using the type system, we will arrange it so that fixpoint formulae always satisfy sufficent conditions for the desired recursive solutions to exist and be unique.

The fixpoint operators are useful for describing looping or recursive behavior of programs. In general, μ describes finite recursive behavior — that is, loops that are guaranteed to eventually halt. On the other hand, ν describes recursive behavior that may or may not halt.

The extensions of modal logic with infinite connectives and fixpoint operators are both fairly standard. However, I will also extend the logic with λ abstractions and applications, which is rather unusual. This enhancement lets me directly write predicates and relations in addition to propositions. Suppose I write the formula $\lambda x : I.\ F(x)$ — the meaning of this formula is not just a proposition, but a predicate on values of type I. Likewise, $\lambda x : I.\ \lambda y : J.\ R(x, y)$ represents a 2-place relation on values of type I and J.

Once we have predicates and relations, it makes sense to apply them. Suppose f is some formula representing a predicate over values of type X, and suppose x is such a value. Then the formula $f@x$ applies x to f, and denotes the truth of the statement "x is in the class of values denoted by f." The formula $f@x$ is analogus to the set-theoretic notion of set membership, $x \in f$. I avoid the set-theoretic notation here because I do not wish to imply that I am employing the toolkit of set theory. In particular, there is no sense in which it is possible to quantify over predicates f, as one could do with sets.

Using λ abstractions and application makes it possible to express recursive specifications that carry "ghost" state — that is, extra data that can be referenced as part of the program specification, but that does not appear anywhere in the state of the program itself. In this thesis, I primarily use this capability to make it possible to give very consise presentations of characteristic formulae. Compared with the framework for presenting characteristic formulae of Aceto et al. [AILS12], I do not require the auxiliary notion of declarations and fixpoints invoving (infinite) series of declarations. Instead, λ abstractions serve this purpose. An added advantage of my approach is that my presentation generalizes smoothly to arbitrarily-nested fixpoints, whereas a presentation using declarations must do some additional work to support nesting.

Types. Once we add λ abstractions to our formulae, we need a type system to keep things straight. This will keep us from applying predicates to the wrong types of data. The type system will also serve a second purpose — it requires that recursion variables occur only in positive positions. This ensures that the interpretation of terms is monotone and that the fixpoint operators behave as expected.

Thus, we present a simple type system for terms.

$$\sigma \quad ::= \quad \mathbb{T} \;\mid\; I \to \sigma$$

The primitive type $\mathbb{T}$ stands for the type of propositions. An arrow type $I \to \mathbb{T}$ represents the type of predicates on I, and $I \to J \to \mathbb{T}$ represents a 2-place relation on I and J , etc. Note that nesting of types is not allowed on the left-hand side of arrows, so higher-order types are not allowed. Although we use the same symbol here $\to$ as we used before to indicate functions in the metalogic, here we simply mean a syntactic construct for building types. In practice, confusion should not arise, as one can tell from context which is meant.

In what follows, I will use the metavariable e to represent terms of the μ-calculus. Formally, terms are defined in a way that they explicitly carry their type, so the following definition actually defines a syntactic form e_σ, which represents a term e of type σ. Usually, the type of e can be inferred from context and will be elided for readability. However, it is occasionally necessary to explicitly reference the type of a term. Thus, the reader should be aware that a term appearing without a type ascription is simply an abbreviation for the full term.

We inductively define the set of modal μ-calculus terms over $\mathcal{X}$ as follows:

$$\begin{array}{rll} e_\sigma ::= & X_\sigma & \\ & \mid \mathsf{atom}\ a & \text{where } \sigma = \mathbb{T} \\ & \mid \mathsf{mode}\ m\ e_\sigma & \\ & \mid \bigwedge_I\ e_{(I\to\sigma)} & \\ & \mid \bigvee_I\ e_{(I\to\sigma)} & \\ & \mid e_\sigma \Rightarrow e_\sigma & \\ & \mid \mu X.\ e_\sigma & \\ & \mid \nu X.\ e_\sigma & \\ & \mid \lambda x : I.\ e_{\sigma'} & \text{where } \sigma = I \to \sigma' \\ & \mid e_{(I\to\sigma)}@i & i \text{ of type } I \end{array}$$

Here a is drawn from the set of atoms $\mathcal{A}$ and m is from the set $\mathcal{M}$ of modes as specified by the input signature. The metavariable X is drawn from a set of recursion variables and I represents an arbitrary index set.

Note on formalization. *To be really careful, one needs the following additional global technical requirement. Fix in advance an infinite cardinal κ: the set of terms will be an element of this infinite cardinal. Restrict all the index sets appearing in any term so that they are members of some $\kappa' < \kappa$. This ensures that the term construction above is well-founded and can thus acutally be constructed. In particular, this means that the set of terms cannot itself appear in any index set. Thus, this logic has a first-order feel.*

In the formal development, the index sets I are just Coq types. Furthermore the λ forms are actually functions in Coq from I to terms, following the methodology of higher-order abstract syntax [PE88]. The restrictions involving infinite cardinals sketched above are in fact implemented via universe checking in Coq's `Type` hierarchy. Thus, despite using the approach of higher-order abstract syntax, we actually only achieve binders for first-order variables.

In terms of the type system (see below), this means that there is no explicit variable type context for the "first order" variables bound by λ. These variables are actually managed by the native typechecker of Coq. The recursion variables, however, are managed manually by the separate type system. This explains why there is an explicit typing context for the recursion variables but not for the first-order variables.

In essence, what we have done is relegate the role of binding first-order variables to the metalogic.

This is convenient, in terms of the formalism, because we do not have to give a syntactic account of binding and substitution, etc., for first order variables. However, it means that the logic we have defined here is in some sense too large. The infinitary conjunction and disjunction operators $\bigwedge$ and $\bigvee$ go beyond the expressive power of the comparable first-order quantifiers $\forall$ and $\exists$. This is because the metalogic functions $f : I \to e$ that appear inside the formal constructor for λ terms are allowed to do arbitrary computation in the metalogic to compute some term given an $i \in I$. This is the "exotic terms" problem that occurs when one attempts to use the higher-order abstract syntax in Coq [DFH95]. In contrast, terms built from $\forall$ and $\exists$ cannot examine the first-order data — the term is uniformly given *with respect to the first-order data.*

One result of this decision is that the abstract syntax of terms defined above has no corresponding concrete syntax, in the following technical sense. Fix a finite alphabet Σ; let Σ^* represent finite sequences of symbols from Σ. Suppose parse *is some mapping from Σ^* to terms. Then* parse *cannot be surjective because the set Σ^* is countable, but the set of terms is uncountable — this can be shown by a standard diagonalization argument. Thus, any attempt to assign a concrete syntax to the abstract syntax of terms must necessarily leave some of the abstract terms inexpressible.*

I only rarely make interesting use of this additional expressive power. For example, all the results regarding characteristic formulae we will examine seem as though they would also be definable in a comparable setting with true first-order variables. The only result that I believe will fail when moving to a true first-order setup is the proof that expressivity of a characteristic logic implies the denotability of all specifications (see theorem 2.5.7).

Let V^+ be a set of *positive* recursion variables and let V^- be a set of *negative* variables — we everywhere require that the set of positive and negative variables are disjoint. Define $V = V^+ \cup V^-$; we call V the set of variables. Let $\Gamma \in V \to \sigma$ be a function mapping each variable to its type. We write $V^-, V^+, \Gamma \vdash e \ : \ \sigma$ to mean that e is a formula of type σ with free variables from V having types assigned by Γ. The typing rules are given in figure 2.1.

For the purposes of presentation, we adopt the convention that bound variables are chosen to be distinct from all free variables; in particular the X appearing in the rules for μ and ν is assumed to be fresh. In both rules, $\Gamma[X \mapsto \sigma]$ denotes the function with domain $V \cup \{X\}$ which behaves like Γ on V and assigns σ to X. By our variable convention, $X \notin V$ so this extension is well-behaved.

Several points concerning the type system are worthy of note. In the rules for $\bigwedge$ and $\bigvee$, a relation of type $I \to \sigma$ is consumed to produce a term of type σ; essentially, the type I is the domain of quantification for the operator. $\bigwedge$ corresponds to universal quantification and $\bigvee$ corresponds to

$\boxed{V^-, V^+, \Gamma \vdash e : \sigma}$

$$\frac{X \in V^+}{V^-, V^+, \Gamma \vdash X : \Gamma(X)}\ \text{Var}$$

$$\frac{a \in A}{V^-, V^+, \Gamma \vdash a : \mathbb{T}}\ \mathsf{atom} \qquad \frac{V^-, V^+, \Gamma \vdash e : \sigma \qquad m \in \mathcal{M}}{V^-, V^+, \Gamma \vdash \mathsf{mode}\ m\ e : \sigma}\ \mathsf{mode}$$

$$\frac{V^-, V^+ \cup \{X\}, \Gamma[X \mapsto \sigma] \vdash e : \sigma}{V^-, V^+, \Gamma \vdash \mu X.\ e : \sigma}\ \mu \qquad \frac{V^-, V^+ \cup \{X\}, \Gamma[X \mapsto \sigma] \vdash e : \sigma}{V^-, V^+, \Gamma \vdash \nu X.\ e : \sigma}\ \nu$$

$$\frac{V^-, V^+, \Gamma \vdash e : I \to \sigma}{V^-, V^+, \Gamma \vdash \bigvee_I e : \sigma}\ \bigvee \qquad \frac{V^-, V^+, \Gamma \vdash e : I \to \sigma}{V^-, V^+, \Gamma \vdash \bigwedge_I e : \sigma}\ \bigwedge \qquad \frac{\begin{array}{c} V^+, V^-, \Gamma \vdash e_1 : \sigma \\ V^-, V^+, \Gamma \vdash e_2 : \sigma \end{array}}{V^-, V^+, \Gamma \vdash e_1 \Rightarrow e_2 : \sigma}\ \Rightarrow$$

$$\frac{(\forall i \in I.\ \ V^-, V^+, \Gamma \vdash e[i/x] : \sigma)}{V^-, V^+, \Gamma \vdash (\lambda x : I.\ e) : I \to \sigma}\ \lambda \qquad \frac{V^-, V^+, \Gamma \vdash e : I \to \sigma \qquad i \in I}{V^-, V^+, \Gamma \vdash e@i : \sigma}\ @$$

Figure 2.1: Typing of formulae

existential quantification. In the cases for μ and ν, note that the bound variable X becomes free and is adjoined to V^+, and that its type is recorded in Γ. In the case for variables, only positive variables are allowed to appear.

Note that in the case for $\Rightarrow$, the left-hand side formula is typed in a context where its positive and negative variables have been swapped. This corresponds to the usual syntactic restriction that recursion variables must occur only in positive positions; that is, under an even (possibly 0) number of left-hand-sides of implications. There is no rule for typing negative variables because they are not allowed to occur. Instead, negative variables *become* positive variables when they get swapped in the rule for $\Rightarrow$.

In the rule for λ, the syntax $e[i/x]$ means the term e where value i is substituted for x. This substitution is accomplished via function application in the metalogic, according to the methodology of higher-order abstract syntax.

Note on formalization. *In the formal proof, several issues mentioned above do not arise, as they are excluded by construction. In particular, the disjointness of positive and negative variables and the free variable convention are all statically enforced* by construction *because we use a nameless representation of variables. In fact, the formal proof defines the terms and type system simultaneously, using a single inductive definition that "bakes in" the type system into the definition of terms.*

This means that the Coq type of μ-calculus terms rules out ill-typed terms by construction — we cannot even write down untypeable terms as they will be rejected by the metalogic.

Derived operators. We can derive constants for truth and falsehood by taking the conjunction (resp. disjunction) of the empty set. We can also get operators for binary conjunction and disjunction by using $\bigwedge$ and $\bigvee$ over a 2-point set. We use the usual intuitionistic abbreviation for negation. Here ι_σ is used as notation for the empty function; that is, ι_σ is the unique function with empty domain and codomain terms with type σ.

$$\begin{aligned}
\top_\sigma &\equiv \textstyle\bigwedge_\emptyset \lambda\iota_\sigma \\
\bot_\sigma &\equiv \textstyle\bigvee_\emptyset \lambda\iota_\sigma \\
f_1 \wedge f_2 &\equiv \textstyle\bigwedge_2 \Lambda_2(\lambda i.f_i) \\
f_1 \vee f_2 &\equiv \textstyle\bigvee_2 \Lambda_2(\lambda i.f_i) \\
\neg p &\equiv p \Rightarrow \bot
\end{aligned}$$

Note on formalization. *I have resorted to an abuse of notation here. The presentation I have given for lambda terms makes it difficult to express the desired construction for $\top$ and $\bot$. In the formal proof, ι is implementated via case distinction on an empty inductive type, so there is no interesting function body.*

Semantics. In a way that is now standard, we give semantics to the terms of our logic via a Kripke interpretation [Kri63]. Roughly, this means each formula is assigned a denotation, the set of *worlds* on which it holds. In our case, worlds correspond to programs and thus the denotation of a formula defines what programs satisfy it.

In an ordinary, classical interpretation, the denotations of formulae are just sets; no special structure is required. In our case, however, we want to restrict our attention to the sets which are closed with respect to refinement.

Before giving the semantics of formula, we briefly review some fixpoint theory. The standard interpretation of a μ-calculus involves choosing some complete lattice and assigning an element of the lattice for each formula, given some interpretation of the variables (a valuation). This assignment is required to be monotone with respect to the valuation. Because the lattice is complete and the assignment is monotone, we can apply the Knaster-Tarski fixpoint theorem [Tar55] to calculate least and greatest fixpoints.

As we have said, the standard classical semantics uses the powerset lattice (sets with no particular structure), a well-known complete lattice. In our case we will instead use the lattice of *refinement-closed* sets over an index set I, which we write $\mathcal{K}(I)$ and call interpretations. The refinement-closed sets over I form a complete sublattice of $\wp(I \times \mathcal{W})$. The lattice order is the inclusion relation on sets, with least upper bounds given by union and greatest lower bounds given by intersection.

$$\mathcal{K}(I) \equiv \{K \in \wp(I \times \mathcal{W}) \ \mid \ \forall i \in I.\ \forall w\ w' \in \mathcal{W}.\ \ w \sqsubseteq w' \Rightarrow (i, w) \in K \Rightarrow (i, w') \in K\}$$

We write $\mathcal{K}$ for $\mathcal{K}(1)$, where by 1 we mean a 1-point set. Whenever we have a set $K \in \mathcal{K}$, we simply write $w \in K$ rather than the more verbose $(*, w) \in K$ ($*$ is the unique element in 1).

Recall that the Knaster-Tarski theorem applies to a monotone function $g : \mathcal{K}(I) \to \mathcal{K}(I)$, so that:

$$\mathsf{lfp}(g) \equiv \bigcap\{K \in \mathcal{K}(I) \mid g(K) \subseteq K\} \qquad\qquad \mathsf{gfp}(g) \equiv \bigcup\{K \in \mathcal{K}(I) \mid K \subseteq g(K)\}$$

Where $\mathsf{lfp}(g)$ is the least fixed point of g and $\mathsf{gfp}(g)$ is the greatest fixed point of g. These operators will give us the interpretation of the (syntactic) fixpoint operators μ and ν.

For each type σ, we define $\mathsf{el}(\sigma)$, the set of elements of type σ.

$$\begin{array}{lcl} \mathsf{el}(\mathbb{T}) & \equiv & 1 \\ \mathsf{el}(I \to \sigma) & \equiv & I \times \mathsf{el}(\sigma) \end{array}$$

Thus $\mathsf{el}(\sigma)$ is just a tuple of all the types appearing in σ, and the $\mathcal{K}(\mathsf{el}(\sigma))$ are interpretations of type σ.

Let $V^-, V^+, \Gamma \vdash e : \sigma$, so that e is a term of type σ with free variables in $V = V^- \cup V^+$. Let $\Xi \in \Pi v : V.\ \ \mathcal{K}(\mathsf{el}(\Gamma(v)))$ be a *valuation*, i.e., a function mapping free variables $v \in V$ to interpretations of the appropriate type. Then the denotation of a formula $[\![e_\sigma]\!]_\Xi$ is an interpretation in $\mathcal{K}(\mathsf{el}(\sigma))$. The clauses in figure 2.2 inductively define the interpretation for each syntax former of the logic. In each clause we assume the terms are well-typed; however, for readability we omit type annotations. For closed e, we write $[\![e]\!]$ for $[\![e]\!]_\iota$ (recall ι is the empty function). When e is closed, we will use the notation $z, w \models e$ as an abbreviation for $(z, w) \in [\![e]\!]$. We lift the ordering on interpretations to an ordering on the valuations Ξ by writing $\Xi \subseteq \Xi'$ iff $\Xi(v^+) \subseteq \Xi'(v^+)$ for each $v^+ \in V^+$ and $\Xi'(v^-) \subseteq \Xi(v^-)$ for each $v^- \in V^-$.

Proposition 2.5.3 (μ-calculus monotonicity)**.** *Let e be a μ-calculus term where $V^-, V^+, \Gamma \vdash e : \sigma$. Let Ξ, Ξ' be valuations for Γ with $\Xi \subseteq \Xi'$. Then $[\![e]\!]_\Xi \subseteq [\![e]\!]_{\Xi'}$.*

Proof. By induction on e. $\square$

$$\begin{array}{lcl}
[\![X]\!]_\Xi & \equiv & \Xi(X) \\
[\![\mathsf{atom}\ a]\!]_\Xi & \equiv & \{(z,w) \ \mid\ w \in [\![a]\!]_\mathcal{A}\} \\
[\![\mathsf{mode}\ m\ e]\!]_\Xi & \equiv & \{(z,w) \ \mid\ [\![m]\!]_\mathcal{M}(\{w' \mid (z,w') \in [\![e]\!]_\Xi\})\ \} \\
[\![\bigvee_I e]\!]_\Xi & \equiv & \{(z,w) \ \mid\ \exists i \in I.\ \ ((i,z),w) \in [\![e]\!]_\Xi\} \\
[\![\bigwedge_I e]\!]_\Xi & \equiv & \{(z,w) \ \mid\ \forall i \in I.\ \ ((i,z),w) \in [\![e]\!]_\Xi\} \\
[\![e_1 \Rightarrow e_2]\!]_\Xi & \equiv & \{(z,w) \ \mid\ \forall w'.\ \ w \sqsubseteq w' \Rightarrow (z,w') \in [\![e_1]\!]_\Xi \Rightarrow (z,w') \in [\![e_2]\!]_\Xi\} \\
[\![\mu X.e]\!]_\Xi & \equiv & \mathsf{lfp}\ (\lambda m.\ [\![e]\!]_{\Xi[X \mapsto m]}) \\
[\![\nu X.e]\!]_\Xi & \equiv & \mathsf{gfp}\ (\lambda m.\ [\![e]\!]_{\Xi[X \mapsto m]}) \\
[\![\lambda x : I.\ e]\!]_\Xi & \equiv & \{((i,z),w) \ \mid\ (z,w) \in\ [\![e[i/x]]\!]_\Xi\} \\
[\![e@i]\!]_\Xi & \equiv & \{(z,w) \ \mid\ ((i,z),w) \in\ [\![e]\!]_\Xi\}
\end{array}$$

Figure 2.2: Semantics of formulae

Because $[\![e]\!]_\Xi$ is monotone and the valuation extension operator $\Xi[X \mapsto m]$ is monotone, we are justified in using the lfp and gfp operators to calculate least and greatest fixpoints.

Proposition 2.5.4 (μ-calculus soundness)**.** *Let e be a μ-calculus term where $V^-, V^+, \Gamma \vdash e : \sigma$. Let Ξ be a valuation for Γ, and let $z \in \mathsf{el}(\sigma)$. Then $[\![e]\!]_\Xi$ is refinement closed. That is:*

$$w \sqsubseteq w' \quad \textit{and} \quad (z,w) \in [\![e]\!]_\Xi \qquad \textit{implies} \qquad (z,w') \in [\![e]\!]_\Xi$$

Proof. By induction on e. □

Derived operators. From the syntactic definition of the derived operators, we can prove that they have the following denotations.

$$\begin{array}{lcl}
[\![\top_\sigma]\!]_\Xi & = & \mathsf{el}(\sigma) \times \mathcal{W} \\
[\![\bot]\!]_\Xi & = & \emptyset \\
[\![f_1 \wedge f_2]\!]_\Xi & = & [\![f_1]\!]_\Xi \cap [\![f_2]\!]_\Xi \\
[\![f_1 \vee f_2]\!]_\Xi & = & [\![f_1]\!]_\Xi \cup [\![f_2]\!]_\Xi \\
[\![\neg f]\!]_\Xi & = & \{(z,w) \mid \forall w' \in \mathcal{W}.\ w \sqsubseteq w' \rightarrow (z,w') \notin\ [\![f]\!]_\Xi\}
\end{array}$$

Note the quantification over accessible worlds in the denotation of $\neg$; this comes from the denotation of $\Rightarrow$. Semantically, this is what causes the failure of the excluded middle. Negation is much stronger than simply saying that a formula fails to hold in the current world; it must also continue to fail in all accessible worlds. As with the propositional logic, the axiom of the excluded middle is valid if the accessibility preorder is symmetric.

Denotability. Recall from §2.4 I claimed that, for the logics in this thesis, expressivity (definition 2.4.3) implies denotability (definition 2.4.5). This is essentially because we can use infinite disjunction together with a characteristic formula to inject any refinement-closed property into a formula.

Suppose $\langle \mathcal{F}, \models \rangle$ is a specification logic over a set of worlds $\mathcal{W}$ with accessability relation $\sqsubseteq$. Let CF be a function from worlds to formulae such that $\mathsf{CF}(x)$ is a characteristic formula for x.

Definition 2.5.5 (Interpretation formula)**.** *Let $P \subseteq \mathcal{W}$ be some refinement-closed set. Then the interpretation of P is a formula as follows:*

$$\mathsf{interp}(P) \qquad \equiv \qquad \bigvee_{\{w' \ \mid \ w' \in P\}} \mathsf{CF}(w') \tag{2.8}$$

Proposition 2.5.6.

$$w \in P \qquad \textit{iff} \qquad w \models \mathsf{interp}(P)$$

Proof. Unwinding the definitions of the symbols, we see that $w \models \mathsf{interp}(P)$ holds whenever there exists some world $w' \in P$ where $w \models \mathsf{CF}(w')$. Because CF is a characteristic formula, this is the same as saying that there exists $w' \in P$ with $w' \sqsubseteq w$. Finally, because P is refinement closed and $\sqsubseteq$ is reflexive, this is equivalent to claiming that $w \in P$. □

Theorem 2.5.7 (Expressivity implies denotability)**.** *In any instantiation of the μ-calculus that admits characteristic formulae, we can build representation formulae for any refinement closed set of worlds.*

In other words, for the μ-calculus logical skeleton, expressivity implies denotability of all specifications.

Note that this theorem relies on the ability to quantify over an arbitrary refinement-closed set of worlds using $\bigvee$. I suspect that restricting the syntax of the μ-calculus to true first-order formulae would make these representation formulae inexpressible, causing this result to fail.

Chapter 3

Bisimulations

In this chapter, I will review the theory of strong bisimulation and the theory of branching bisimulation. Most of the material in this chapter is due to previous work by others; however, I will be reviewing aspects of these theories in some detail because the sequence of definitions and properties will recur, with modifications, in later chapters. It is useful to review these properties in their most basic form so that the differences due to the refinements I will introduce are easily highlighted.

In addition, I have made several minor improvements in the theory of branching bisimulation over what has been previously published in the literature. The related work section at the end of this chapter contains a detailed discussion of which results are due to whom and which results are improvements.

3.1 Strong Bisimulation

Strong bisimulation is generally regarded as the finest equivalence on LTS processes that makes sense [San12]. It abstracts entirely away from the identity of the individual states in an LTS as well as most graph "shape" properties. Instead, we are only interested in being able to distinguish LTS states by their connectivity properties; that is, what sort of transitions are available from a given state, what transitions are available after performing certain actions, etc.

Recall that an LTS $\mathcal{L}$ is a set of states $\mathcal{S}$ together with a transition relation $\longrightarrow$ (see definition 2.2.1). We write $x \xrightarrow{o} x'$ to mean that state x steps to state x' producing observable o.

Definition 3.1.1 (Strong bisimulation relation)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are LTSs. Suppose $R \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ is a binary relation on the states of $\mathcal{L}_1$ and $\mathcal{L}_2$. We say that R is a* strong bisimulation relation *if,* *whenever $(x, y) \in R$, the following two conditions hold:*

$$\forall o\ x'.\ \ x \xrightarrow{o} x' \ \text{ implies }\ \exists y'.\ y \xrightarrow{o} y' \text{ and } (x', y') \in R \tag{3.1}$$

$$\forall o\ y'.\ \ y \xrightarrow{o} y' \ \text{ implies }\ \exists x'.\ x \xrightarrow{o} x' \text{ and } (x', y') \in R \tag{3.2}$$

Definition 3.1.2 (Strong bisimlarity)**.** *Suppose a and b are LTS processes. Then we say a is* strongly bisimilar *to b, and write $a \cong b$ if there exists a bisimulation relation R such that $(a, b) \in R$.*[1]

Terminology note. Note that there are two clauses in definition 3.1.1; clause (3.1) says that every action taken by the left process can be matched by the process on the right, and clause (3.2) says that actions on the right can be matched on the left. I call clause (3.1), going from left-to-right, the *forward* direction of the bisimulation and (3.2), going from right-to-left, the *backward* direction of the bisimulation. There is disagreement in the literature, with some authors adopting this convention and others adopting the opposite convention, which is why I take this moment to explicitly pin down the terminology.

Although it makes no difference for a symmetric relation, I always orient the refinement symbol so that the input (less refined) process is on the left and the output (more refined) process is on the right. Thus, the forward direction of bisimulation means that every action of the input program is matched by the output program and the backward direction means that every action of the output program is matched by the input program.

Basic properties. Recall from §2.3 that I require several sanity properties of the refinements and equivalences I examine. These proofs are routine, but I will show the details here to give the flavor. In later sections, I will not be so thorough.

First, I need to show bisimulation is reflexive. This is done in two steps: first, show that the relation of equality on states is a strong bisimulation relation, then use that to demonstrate bisimilarity.

Proposition 3.1.3. *Let $\mathcal{L}$ be an LTS. The identity relation on $\mathcal{S}$ is a bisimulation relation.*

Proof. Assume $x, y \in \mathcal{S}$ such that $x = y$. We need to show the two properties from definition 3.1.1. Both are trivial. □

Proposition 3.1.4. *Let a be an LTS process. Then $a \cong a$.*

[1]Note the slight abuse of notation: we intend that the current state of a is related to the current state of b.

Proof. By proposition 3.1.3, $=$ is a bisimulation relation, and clearly $a = a$. $\square$

The next property we need is transitivity. This, again, is done in two parts. First we need to show that the composition of two bisimulation relations is again a bisimulation. Then we exploit this property to show transitivity.

Definition 3.1.5 (Relation composition). *Suppose $S \subseteq \wp(A \times B)$ and $T \subseteq \wp(B \times C)$ are binary relations for some sets A, B, and C. Then the composition $S \fatsemi T \subseteq \wp(A \times C)$ is defined as follows:*

$$S \fatsemi T \qquad \equiv \qquad \{(a, c) \mid \exists b.\ \ (a, b) \in S \text{ and } (b, c) \in T\} \tag{3.3}$$

Note that the "direction" of the composition is reversed from the usual order for function composition. I find this order more intuitive when applied to relations.

Proposition 3.1.6. *Suppose $\mathcal{L}_1$, $\mathcal{L}_2$, and $\mathcal{L}_3$ are LTSs with $R_1 \subseteq \wp(\mathcal{S}_1 \times \mathcal{S}_2)$ and $R_2 \subseteq \wp(\mathcal{S}_2 \times \mathcal{S}_3)$ both strong bisimulation relations. Then $R_1 \fatsemi R_2 \subseteq \wp(\mathcal{S}_1 \times \mathcal{S}_3)$ is a strong bisimulation relation.*

Proof. We prove only the first of the two clauses from definition 3.1.1; the other follows by symmetry. Suppose $(x, z) \in R_1 \fatsemi R_2$. Then there exists some y such that $(x, y) \in R_1$ and $(y, z) \in R_2$. Further suppose that $x \xrightarrow{o} x'$ for some o and x'. Then, because R_1 is a bisimulation, there exists some y' such that $y \xrightarrow{o} y'$ and $(x', y') \in R_1$. Now, because R_2 is a bisimulation, there exists a z' where $z \xrightarrow{o} z'$ and $(y', z') \in R_2$. Now we have the desired z', and must simply show $(x', z') \in R_1 \fatsemi R_2$. This follows from the definition of relation composition using y'. $\square$

Proposition 3.1.7. *Suppose a, b and c are LTS processes with $a \cong b$ and $b \cong c$. Then $a \cong c$.*

Proof. By definition 3.1.2, there exist bisimulation relations R_1 and R_2 such that $(a, b) \in R_1$ and $(b, c) \in R_2$. By proposition 3.1.6, $R_1 \fatsemi R_2$ is a bisimulation relation. Clearly $(a, c) \in R_1 \fatsemi R_2$, so $a \cong c$ as desired. $\square$

Since I am using the symbol $\cong$, I have implicitly asserted that strong bisimilarity is symmetric. This follows from the fact that the inverse relation R^{-1} is a strong bisimulation whenever R is. The reader may readily verify that this is the case.

The third condition I required (of the 4 from chapter 2) is that $\cong$ must be generated as the greatest fixpoint of a monotone operator on relations. To make this evident, consider F, an operator on relations:

$$\begin{aligned} F(R) \quad \equiv \quad \{\ (x,y)\ \mid\ & \forall o\ x'.\ \ (x \xrightarrow{o} x')\ \rightarrow\ \exists y'.\ (y \xrightarrow{o} y') \wedge (x',y') \in R\ \text{ and} \\ & \forall o\ y'.\ \ (y \xrightarrow{o} y')\ \rightarrow\ \exists x'.\ (x \xrightarrow{o} x') \wedge (x',y') \in R\ \} \end{aligned} \tag{3.4}$$

From a quick examination of equation 3.4, it should be clear that F is monotone (R occurs only in "positive" positions). Then it is evident that definition 3.1.2 is simply the construction of the greatest fixpoint of F via the Knaster-Tarski fixpoint theorem [Tar55]. To reiterate, $F(\cong) = \cong$, and $\cong$ is the largest relation satisfying this equation.

For future refinements, I will not explicitly give the operator F of which we are taking the fixpoint, but the reader should be able to reconstruct it from the associated definition easily.

The fourth condition requires that I choose the finest relation available. As I have already mentioned, strong bisimulation is generally regarded as the finest equivalence relation on LTSs that makes sense. In particular, it is strictly finer than equivalence of trace sets (i.e., the automata theory view of LTSs). This strength is one of the major reasons I chose strong bisimulation as the bedrock notion for the rest of these investigations. Each of the enhancements we will examine in the following pages can be viewed as loosening the restrictions of strong bisimulation as little as possible to allow the class of transformations required by realistic compilers.

3.1.1 Characteristic Logic

It is well-known that strong bisimulation is characterized by a simple modal logic, called Hennessy-Milner logic (HML) [HM80]. The main feature of this logic is a set of modalities that allow the observation of program behavior. For each observation $o \in \mathcal{O}$, there is a modality $[o]$ and a dual modality $\langle o \rangle$. Informally, the meaning of these modalities are:

$[o]P$	After every o-transition from the current state, P holds.
$\langle o \rangle P$	There exists an o-transition such that afterward P holds.

$[o]$ constrains the behavior of programs by asserting facts that must be true no matter what o-transition is selected. Roughly, properties that can be stated using only $[o]$ correspond to safety properties; they constrain what the program is *allowed* to do. Dually, properties stated using only $\langle o \rangle$ are progress properties; they specify what the program is *obligated* to (be able to) do. Mixing both modalities allows the specification of rich properties with both safety and progress aspects.

Recall from §2.4 that to build both the propositional and the fixpoint characteristic logics, we need to specify a set of worlds, an accessibility relation, a set of modes and atoms, and interpretations for the same. For HML, these data are:

- The worlds $\mathcal{W}$ are the LTS processes;
- the accessibility relation is $\cong$, strong bisimilarity;
- and there are no atoms, so $\mathcal{A} = \emptyset$ and
- the modes and interpretations are given as below.

$$\mathcal{M} := [o] \mid \langle o \rangle$$

$$[\![[o]]\!]_{\mathcal{M}}(P) \equiv \{w \mid \forall w'.\ w \xrightarrow{o} w' \Rightarrow w' \in P\} \tag{3.5}$$

$$[\![\langle o \rangle]\!]_{\mathcal{M}}(P) \equiv \{w \mid \exists w'.\ w \xrightarrow{o} w' \wedge w' \in P\} \tag{3.6}$$

It should be clear that the interpretation of modes is monotone in P (P occurs only in positive positions). We must simply show that $[\![-]\!]_{\mathcal{M}}$ preserves $\cong$-closure.

Proposition 3.1.8. *Suppose P is a $\cong$-closed set of LTS processes. Then $[\![m]\!]_{\mathcal{M}}(P)$ is $\cong$-closed for each $m \in \mathcal{M}$.*

Proof. First suppose $m = \langle o \rangle$ for some $o \in \mathcal{O}$. We are given that $a \in [\![\langle o \rangle]\!](P)$ and $a \cong b$ for some processes a and b; we must show that $b \in [\![\langle o \rangle]\!](P)$. By the interpretation of $\langle o \rangle$, there exists an a' such that $a \xrightarrow{o} a'$ with $a' \in P$. By the forward direction of $a \cong b$, there exists b' such that $b \xrightarrow{o} b'$ and $a' \cong b'$. By the refinement closure of P, $b' \in P$. Thus, $b \in [\![\langle o \rangle]\!](P)$, as required.

Next suppose $m = [o]$ for some $o \in \mathcal{O}$ and $a \in [\![[o]]\!](P)$ and $a \cong b$ for some processes a and b; we must show that $b \in [\![[o]]\!](P)$. Thus we also suppose that $b \xrightarrow{o} b'$ for some b', and now we must show $b' \in P$. By the backward direction of $a \cong b$, there exists an a' where $a \xrightarrow{o} a'$ and $a' \cong b'$. Because $a \in [\![[o]]\!](P)$, we have that $a' \in P$. Finally, we have $b' \in P$ by $\cong$-closure of P. □

This proof is particularly illuminating because it highlights the critical reason why HML characterizes strong bisimulation: $\langle o \rangle$ describes the forward direction of bisimulation whereas $[o]$ describes the backward direction.

3.1.2 Propositional Adequacy

Now that we have specified the modalities of HML, we can instantiate the construction from §2.5.1 to get the propositional version of Hennessy-Milner logic. We use the symbol $\mathcal{F}_{\text{HM}}$ to refer to the formulae of propositional Hennessy-Milner logic.

The proofs above suffice for soundness of the logic (see §2.5.1); recall that soundness means that whenever $a \cong b$, if $a \models f$ for some $f \in \mathcal{F}_{\text{HM}}$, then $b \models f$.

The rest of this section is devoted to proving adequacy of propositional HM logic. As is typically the case, we will need a finiteness assumption to complete this proof.

Definition 3.1.9 (Image-finite LTS). *Suppose $\mathcal{L}$ is an LTS. We say $\mathcal{L}$ is* image-finite *if, for each state $x \in \mathcal{S}$ and $o \in \mathcal{O}$, the set $\{x' \mid x \xrightarrow{o} x'\}$ is finite. An LTS process is image-finite if its underlying LTS is image-finite.*

In an image-finite LTS, each state has at most a finite number of possible successor states for each observation. For propositional characteristic logics, one must usually restrict the result to the image-finite LTSs because one has only finite conjunctions and disjunctions available in the logic. This restriction may be lifted if one adds infinite conjunction/disjunction [HS85].

Theorem 3.1.10 (Adequacy for HM). $^{\mathsf{EM}}$ [2] *Suppose a and b are image-finite LTS processes where $a \models f$ implies $b \models f$ for all $f \in \mathcal{F}_{\text{HM}}$. Then $a \cong b$.*

Proof. It suffices to prove $R \equiv \{(a, b) \mid \forall f \in \mathcal{F}_{\text{HM}}.\ a \models f \Rightarrow b \models f\}$ is a bisimulation relation. Suppose the opposite for contradiction. Then there exists $(a, b) \in R$ where either the forward or the backward direction of bisimulation fails.

Suppose it is the forward direction. Then there exists a' and o where $a \xrightarrow{o} a'$ but for every b' with $b \xrightarrow{o} b'$, $(a', b') \notin R$. Thus, for every such b', there exists some $f \in \mathcal{F}_{\text{HM}}$ such that $a' \models f$ but $b' \not\models f$. Because b is image-finite, we can finitely enumerate all such b', and thus also finitely enumerate the corresponding formulae. Let $f_0, \cdots, f_n$ be this sequence of formulae and let $\bigwedge_i f_i$ represent the (finite) conjunction of all these formulae. Clearly $a' \models \bigwedge_i f_i$; in turn this implies $a \models \langle o \rangle (\bigwedge_i f_i)$. Because $(a, b) \in R$, $b \models \langle o \rangle (\bigwedge_i f_i)$. Thus there must be some b_i such that $b \xrightarrow{o} b_i$ and $b_i \models \bigwedge_i f_i$. This cannot be, as $b_i \not\models f_i$.

Now suppose the backward direction fails. There exists b' and o where $b \xrightarrow{o} b'$ but for every a' with $a \xrightarrow{o} a'$, $(a', b') \notin R$. Thus, for every such a', there exists some $f \in \mathcal{F}_{\text{HM}}$ such that $a' \models f$ but $b' \not\models f$. Because a is image-finite we can finitely enumerate all such a' and

[2]Recall that the symbol $^{\mathsf{EM}}$ indicates that the proof of this statment uses the axiom of the excluded middle.

thus the corresponding f_i. As before let $f_0, \cdots, f_n$ be this sequence of formulae and let $\bigvee_i f_i$ represent the (finite) disjunction of all these formulae. Then we have that $a \models [o](\bigvee_i f_i)$. Because $(a, b) \in R$, $b \models [o](\bigvee_i f_i)$. Therefore, $b' \models \bigvee_i f_i$, which implies $b' \models f_i$ for some i. This is impossible. □

Once again, we see a tight connection between $\langle o \rangle$ and the forward direction of bisimulation and a corresponding connection between $[o]$ and the backward direction.

3.1.3 Characteristic Formula for HM

In this section, I will show how we can use the additional expressive power of the modal μ-calculus to prove expressivity in addition to adequacy. Instantiating the logic of §2.5.2 using the same data as before, we obtain the fixpoint logic $\mathcal{T}_{\text{HM}}$. Compared to the propositional logic above, we now have infinite conjunction and disjunction, least and greatest fixpoint operators, and lambdas.

Definition 3.1.11 (Characteristic formula for HM)**.** *Suppose $a = \langle \mathcal{L}, z \rangle$ is an LTS process. The characteristic formula for the LTS $\mathcal{L}$ is:*

$$\begin{aligned}&\mathsf{CF}(\mathcal{L}) \equiv \nu M.\ \lambda x : \mathcal{S}.\\ &\qquad\left(\bigwedge_{o \in \mathcal{O}} \quad \bigwedge_{\{x' \ \mid\ x \xrightarrow{o} x'\}} \langle o \rangle M @ x'\right) \ \wedge\ \left(\bigwedge_{o \in \mathcal{O}} [o] \bigvee_{\{x' \ \mid\ x \xrightarrow{o} x'\}} M @ x'\right)\end{aligned}$$

Then the characteristic formula for the process a is $\mathsf{CF}(a) \equiv \mathsf{CF}(\mathcal{L}) @ z$.

This definition is written in the μ-calculus logical skeleton of §2.5.2. Note that the definition of $\mathsf{CF}(\mathcal{L})$ has a λ appearing inside the fixpoint operator ν. Because the λ is nested inside the ν, $\mathsf{CF}(\mathcal{L})$ is a formula of type $\mathcal{S} \to \mathbb{T}$; that is, a predicate on states. Thus M is a variable of the same type. The notation $M @ x'$ *applies* the predicate represented by M to the argument x'. This is well-typed because $x' \in \mathcal{S}$, as required. To get a characteristic formula on processes, we must apply $\mathsf{CF}(\mathcal{L})$ to the initial state of the process. Note that this outermost application cannot be β-reduced through the λ because the fixpoint operator ν interposes. If we attempted to do the β-reduction, we would get an entirely different formula that does not mean what we want.

The basic idea behind the characteristic formula is to restate the definition of bisimilarity using the vocabulary of the characteristic logic. The various parts of CF can be lined up one-to-one with the definitions 3.1.1 and 3.1.2. The outermost symbol is ν and, by construction, every well-typed formula is monotone with respect to the recursion variables. $\mathsf{CF}(\mathcal{L})$ is therefore evidently the greatest fixpoint of a monotone operator. The two main conjuncts of $\mathsf{CF}(\mathcal{L})$ correspond to the two directions of bisimulation; the clause with $\langle o \rangle$ is the forward direction and the clause with $[o]$ is the backward direction.

To prove that $\mathsf{CF}(a)$ is indeed the characteristic formula, we need two basic lemmas. The first tells us that any process satisfies its own characteristic formula. The second says that any other process satisfying the characteristic formula is bisimilar. Both lemmas mostly involve unwinding the denotation of CF and observing its equivalence with strong bisimulation. Note that for this proof, we do not need any finiteness assumptions, nor do we need to use the excluded middle.

Proposition 3.1.12. *Suppose a is an LTS process. Then $a \models \mathsf{CF}(a)$.*

Proof. By the denotation of ν, we need to provide a set satisfying the fixpoint condition. We choose $R = \{(x, y) \mid x \cong y\}$ Because $\cong$ is reflexive, $(a, a) \in R$, so all that remains is to show that R satisfies the body of CF. This is a completely straightforward manipulation of the symbols, and I omit the details. □

Proposition 3.1.13. *Suppose $b \models \mathsf{CF}(a)$. Then $a \cong b$.*

Proof. It suffices to show that $R = \{(x, y) \mid y \models \mathsf{CF}(x)\}$ is a bisimulation relation. As before, the proof follows by straightforward manipulation of the symbols. □

Theorem 3.1.14 (Expressiveness for HM)**.** *Suppose a and b are LTS processes. Then:*

$$a \cong b \qquad \text{iff} \qquad b \models \mathsf{CF}(a)$$

Proof. First, suppose $a \cong b$. By proposition 3.1.12, $a \models \mathsf{CF}(a)$. By soundness, $b \models \mathsf{CF}(a)$, as desired. Next, suppose $b \models \mathsf{CF}(a)$. Then, by proposition 3.1.13, $a \cong b$. □

Theorem 3.1.15 (Infinite-branching Adequacy for HM)**.** *Suppose a and b are arbitrary LTS processes such that $a \models f$ implies $b \models f$ for all $f \in \mathcal{T}_{\mathrm{HM}}$. Then $a \cong b$.*

Proof. Instantiate the assumption using $\mathsf{CF}(a)$. Finish by applying propositions 3.1.12 and 3.1.13. □

3.2 Branching Bisimulation

The theory of strong bisimulation is the simplest and most distinguishing theory in the family of bisimulation-based methods. This makes it a good place to start our investigations. However, it is *too* strong for the purpose we intend: to reason about computations. In strong bisimulation, each and every transition taken by a process is observable to the environment. However, in this setting, we prefer to view computation as occurring in a manner independent of time; in particular, we often want to abstract away from *how many* steps a computation takes. This allows the compiler to perform optimization passes and to reduce complicated high-level operations into more basic low-level operations. These passes typically shrink or grow the number of steps required to complete a computation. In the absence of information about how much real time an operation takes, the best we can do is abstract away from time altogether. This is our main reason for considering weaker forms of bisimulation.

There are a wide variety of different ways this abstraction can be done. Most methods share the same basic idea: we designate some transitions as "hidden" and mark them with a special symbol (usually τ). Hidden transitions cannot be observed by the environment, except by the way they affect the later observable transitions offered by the process. It turns out that this informal idea is a little bit tricky to render formally; hence the wide variety of different notions of bisimulation with hidden transitions.

According to our guiding principle, we should choose the notion (if it exists) which subsumes all the others. It is not immediately obvious that such a "tightest" notion should exist. Fortunately, a good candidate definition has been identified by van Glabbeek and Weijland [vGW96], called *branching bisimulation*. In a comprehensive analysis, val Glabbeek constructed a lattice consisting of some 150 different variants of weak bisimualations (including all the known variants in the literature) [vG93]. The apex of this lattice is branching bisimulation. Thus, one could argue that branching bisimulation is the strongest equivalence on LTSs with hidden transitions that makes sense.

One can also show that branching bisimulation coincides with the *weakest* congruence relation on process calculi (i.e., for particular fragments of CCS) with (interleaving) parallel composition [vGW96]. Thus branching bisimulation is both the strongest extensional (considering only behaviors) equivalence as well as the weakest intentional (considering syntax) equivalence. This gives me sufficient confidence to declare branching bisimulation the "right" starting point for our investigation of bisimulations with hidden transitions.

Definition 3.2.1 (τ-Labeled Transition Systems)**.** *Suppose we have fixed in advance a nonempty set $\mathcal{O}$ of observable transitions. Let τ be a new distinguished symbol not in $\mathcal{O}$. Then a τ-*Labeled Transition System *(or τLTS) is a tuple $\langle \mathcal{S}, \longrightarrow \rangle$ where $\mathcal{S}$ is a set of states and $\longrightarrow \subseteq \wp(\mathcal{S} \times \mathcal{O} \cup \{\tau\} \times \mathcal{S})$ is a transition relation.*

In short, a τLTS is an LTS where we have added a new label for arcs, called τ. I will use the symbols α and β to represent labels in $\mathcal{O} \cup \{\tau\}$, and I will reserve the symbol o for observable labels in $\mathcal{O}$.

Note on formalization. *In the formal development, I represent $\mathcal{O} \cup \{\tau\}$ as an* `option` *type. τLTSs are represented as a dependent record as follows:*

```
Record LTS :=
  { state : Type; steps : state -> option O -> state -> Prop }.
```

Definition 3.2.2 (Weak transitions)**.** *Suppose $\mathcal{L}$ is a τLTS with state space $\mathcal{S}$. For each $\alpha \in \mathcal{O} \cup \{\tau\}$ and $x, x' \in \mathcal{S}$ we define the* weak α-transition *as follows:*

$$x \overset{\alpha}{\Longrightarrow} x' \equiv \begin{cases} x \xrightarrow{\tau}{}^{*}\, x' & \text{when } \alpha = \tau \\ x \xrightarrow{\tau}{}^{*} \xrightarrow{o} \xrightarrow{\tau}{}^{*}\, x' & \text{when } \alpha = o \in \mathcal{O} \end{cases}$$

Here $\xrightarrow{\tau}{}^{*}$ refers to the reflexive, transitive closure of τ-transitions.

Weak transitions capture the basic notion of time-abstraction. A weak τ-transition indicates the passage of some unknown, finite, and possibly zero amount of time. A weak o-transition indicates the occurrence of some observable, which is preceded and followed by some amount of time.

The earliest notion of bisimulation with hidden transitions, called observation equivalence by Milner [Mil80] (but typically called weak bisimulation by others) is simply the definition of strong bisimulation with ordinary LTS transitions replaced by weak transitions. Although this generalization is seems quite natural, its theory is not entirely satisfactory. For example, weak bisimulation is not a congruence with respect to nondeterministic choice.

Although technically a little more complicated, branching bisimulation addresses this deficiency and has a satisfactory theory. To define it, we need one additional technical definition.

Definition 3.2.3 (τ-path)**.** *Suppose $\mathcal{L}$ is a τLTS with state space $\mathcal{S}$, and suppose $P \subseteq \mathcal{S}$ is some property of states. We say there is a τ-path from x to y along P whenever there is a finite sequence starting at x and ending in y such that each element of the sequence has a τ-transition to the next, and each state in the sequence is in P. When this is the case we write $x \overset{\tau}{\Longrightarrow}_P y$. We allow the empty path where $x = y$, indicating no actual steps are taken.*

Branching bisimulation comes in two flavors: divergence-sensitive and divergence-insensitive. First I will present the insensitive version because the definition is a bit simpler.

$$\begin{tikzcd}
x \arrow[r, dash, "R"] \arrow[d, "\tau"'] & y \arrow[dl, dash, dashed, "R"] \\
x' &
\end{tikzcd}$$

$\alpha = \tau$

Figure 3.1a: Forward stuttering

$$\begin{tikzcd}
 & y \arrow[d, Rightarrow, "\tau"] \arrow[dl, dash, "R"'] \\
x \arrow[r, dash, dashed, "R"] \arrow[d, "\alpha"'] & \exists y' \arrow[d, "\alpha"] \\
x' \arrow[r, dash, dashed, "R"] & \exists y''
\end{tikzcd}$$

Figure 3.1b: Forward matching step

$$\begin{tikzcd}
x \arrow[r, dash, "R"] \arrow[dr, dash, dashed, "R"'] & y \arrow[d, "\tau"] \\
 & y'
\end{tikzcd}$$

$\alpha = \tau$

Figure 3.1c: Backward stuttering

$$\begin{tikzcd}
x \arrow[d, Rightarrow, "\tau"'] \arrow[dr, dash, "R"] & \\
\exists x' \arrow[r, dash, dashed, "R"] \arrow[d, "\alpha"'] & y \arrow[d, "\alpha"] \\
\exists x'' \arrow[r, dash, dashed, "R"] & y'
\end{tikzcd}$$

Figure 3.1d: Backward matching step

Figure 3.2: Branching bisimulation

Definition 3.2.4 (Branching bisimulations)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are LTSs. Suppose $R \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ is a binary relation on the states of $\mathcal{L}_1$ and $\mathcal{L}_2$. Let $R(x, -)$ refer to the set $\{y \mid (x, y) \in R\}$. Likewise, $R(-, y) \equiv \{x \mid (x, y) \in R\}$. We say that R is a* branching bisimulation relation *if, whenever $(x, y) \in R$, the following two conditions hold:*

$$\begin{aligned} &x \xrightarrow{\alpha} x' \text{ implies either } \alpha = \tau \wedge (x', y) \in R, \text{ or} \\ &\quad \exists y'\ y''.\ y \overset{\tau}{\Longrightarrow}_{R(x,-)} y' \wedge y' \xrightarrow{\alpha} y'' \wedge (x, y') \in R \wedge (x', y'') \in R \end{aligned} \tag{3.7}$$

$$\begin{aligned} &y \xrightarrow{\alpha} y' \text{ implies either } \alpha = \tau \wedge (x, y') \in R, \text{ or} \\ &\quad \exists x'\ x''.\ x \overset{\tau}{\Longrightarrow}_{R(-,y)} x' \wedge x' \xrightarrow{\alpha} x'' \wedge (x', y) \in R \wedge (x'', y') \in R \end{aligned} \tag{3.8}$$

Definition 3.2.5 (Branching bisimlarity)**.** *Suppose a and b are LTS processes. Then we say a is* branching bisimilar *to b, and write $a \cong_B b$ if there exists a branching bisimulation relation R such that $(a, b) \in R$.*

These definitions are a little intimidating to read. The definition is easier understood by examining the diagrams of figure 3.2. The a and b diagrams show the two possible options of the forward direction, and c and d show the two options for the backward direction. States with a leading $\exists$ are states that must be shown to exist; dotted lines are pairs that must be shown to be in R.

As before, the reflexivity and transitivity of branching bisimilarity follows by showing that the identity relation is a bisimulation and that bisimulations are closed under composition; I omit the details (cf. [vGW96]). It should be evident from the definition that branching bisimilarity is the greatest fixpoint of a monotone operator. I have already discussed how branching bisimulation is the strongest among its family.

Stuttering closure. The stuttering closure is an operation on relations that relates some additional intermediate states. Stuttering closure helps us state and prove some intermediate lemmas, it also gives us a way to make it easier to prove branching bisimulations.

Definition 3.2.6 (Stuttering closure)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are two τLTSs. Then the* stuttering closure *is an operation on relations between $\mathcal{L}_1$ and $\mathcal{L}_2$, defined as follows:*

$$\begin{aligned}\mathsf{ST}(R) \equiv \{(x, y) \ \mid \ & \exists x_0\ x_1\ y_0\ y_1. \\ & x_0 \overset{\tau}{\Longrightarrow} x \wedge x \overset{\tau}{\Longrightarrow} x_1 \wedge y_0 \overset{\tau}{\Longrightarrow} y \wedge y \overset{\tau}{\Longrightarrow} y_1 \wedge (x_0, y_1) \in R \wedge (x_1, y_0) \in R\}\end{aligned}$$

Essentially $(x, y) \in \mathsf{ST}(R)$ whenever the pair (x, y) is sandwiched between two other pairs in R.

Proposition 3.2.7. *Stuttering closure is a closure operation on relations. That is:*

- $R \subseteq \mathsf{ST}(R)$
- $R \subseteq S$ *implies* $\mathsf{ST}(R) \subseteq \mathsf{ST}(S)$
- $\mathsf{ST}(R) = \mathsf{ST}(\mathsf{ST}(R))$

Proof. Immediate from the definition. □

Definition 3.2.8 (Pre-branching bisimulation)**.** *A relation R is a pre-branching bisimulation if it satisfies the definition of branching bisimulation relations (definition 3.2.4), except the τ-path symbol $\overset{\tau}{\Longrightarrow}_P$ is replaced everywhere by the weak transition symbol $\overset{\tau}{\Longrightarrow}$.*

In a pre-branching bisimulation, we simply weaken the diagrams (b) and (d) from figure 3.2 so that each intermediate state is not required to be in the relation R, but only the beginning and ending states. Thus, finding a pre-branching bisimulation is strictly easier than demonstrating a branching bisimulation.

Proposition 3.2.9. *Suppose R is a branching bisimulation. Then R is a pre-branching bisimulation.*

However, for the purposes of showing that two processes are branching bisimilar, a pre-branching bisimulation suffices.

Proposition 3.2.10. *Suppose R is a pre-branching bisimulation. Then $\mathsf{ST}(R)$ is a branching bisimulation.*

Corollary. *Suppose R is a pre-branching bisimulation and $(x, y) \in R$. Then $x \cong_B y$.*

Corollary. *Suppose R is a branching bisimulation. Then $\mathsf{ST}(R)$ is a branching bisimulation.*

Corollary. *Branching bisimilarity is stuttering-closed.*

$$x \ \cong_B \ y \qquad \text{iff} \qquad (x, y) \in \mathsf{ST}(\cong_B)$$

Divergence. When we introduce hidden transitions into the model of processes, we allow for a new type of behavior: divergence. In a τLTS, a *divergence* is (essentially) an infinite path of τ-transitions. This can arise either from a cycle in the LTS graph, or from a path passing through an infinite sequence of distinct states. A divergence is basically an "infinite loop," a computation that consumes resources without interacting with the environment.

The basic version of branching bisimulation defined above is divergence insensitive. This means that it does not distinguish between a process which has divergences from one that does not. There are some situations where such a view is appropriate; however, regarding divergent behavior as equivalent to terminating behavior is not at all appropriate for our compiler correctness concerns. Furthermore, the guiding principles I have adopted demand the finest possible relations, so the divergence-sensitive version is the main one studied here.

Definition 3.2.11 (Divergences)**.** *Suppose $\mathcal{L}$ is some τLTS, and $D \subseteq \mathcal{S}$ is a set of states. Then D is a* divergence *if, for each $x \in D$ there exists $x' \in D$ such that $x \xrightarrow{\tau} x'$. We say a state x* diverges *if there exists a divergence D with $x \in D$.*

In the literature, divergence is usually defined using infinite paths (that is, functions $\mathbb{N} \to \mathcal{S}$) rather than divergence sets, as I have here. The two notions are basically equivalent for our present aims. It is easy to show that each infinite path induces a divergence set; however, some version of the axiom of choice is required to generate a path from a divergence set. Frequently it occurs in some proof that that a divergence set is found easily to hand, whereas a path is not. To avoid invoking choice principles, I have taken the divergence set notion as primitive.

Definition 3.2.12 (Divergence-sensitive branching bisimulations)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are LTSs. Suppose $R \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ is a binary relation on the states of $\mathcal{L}_1$ and $\mathcal{L}_2$. We say that R is a* divergence-sensitive branching bisimulation relation *if R is a branching bisimulation and, in addition, satisfies the following for each $(x, y) \in R$:*

- *For each divergence D in $\mathcal{L}_1$ where $x \in D$, there exists $x' \in D$ and y' such that $y \xrightarrow{\tau} y'$ and $(x', y') \in R$.*
- *For each divergence D in $\mathcal{L}_2$ where $y \in D$, there exists $y' \in D$ and x' such that $x \xrightarrow{\tau} x'$ and $(x', y') \in R$.*

Definition 3.2.13 (Divergence-sensitive branching bisimlarity)**.** *Suppose a and b are LTS processes. Then we say a is* divergence-sensitive branching bisimilar *to b, and write $a \cong_{\Delta B} b$ if there exists a divergence-sensitive branching bisimulation relation R such that $(a, b) \in R$.*

As with (insensitive) branching bisimulation, $\cong_{\Delta B}$ is reflexive and transitive, and arises as the greatest fixpoint of a monotone operator.

The main idea behind divergence-sensitivity is to make sure that divergences in one process get "carried over" to the other. Consider the forward direction. There is some divergence D containing the state x. Then we know that y must be able to take at least one τ-step to y', and furthermore the new y' is still related to a divergent state x'. The fact that definition 3.2.13 transfers divergences is mare more readily apparent by the following alternate characterization.

Proposition 3.2.14. *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are LTSs. Suppose $R \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ is a binary relation on the states of $\mathcal{L}_1$ and $\mathcal{L}_2$. Then R is a divergence-sensitive branching bisimulation relation iff R is a branching bisimulation and R satisfies the following for each $(x, y) \in R$:*

- *For each divergence D_x in $\mathcal{L}_1$ where $x \in D_x$, there exists a divergence D_y in $\mathcal{L}_2$ where $y \in D_y$ satisfying: $\forall y' \in D_y.\ \exists x' \in D_x.\ (x', y') \in R$.*
- *For each divergence D_y in $\mathcal{L}_2$ where $y \in D_y$, there exists a divergence D_x in $\mathcal{L}_1$ where $x \in D_x$ satisfying: $\forall x' \in D_x.\ \exists y' \in D_y.\ (x', y') \in R$.*

This characterization makes it clear that divergent behavior corresponds to divergent behavior. Moreover, the divergent behaviors must be tightly linked to each other. It is not completely clear, just from examining the definition, that this property is the right thing to require. In particular, it is stronger than the obvious requirement (that x diverges iff y diverges). For a defense of this definition, proof of transitivity and a variety of equivalent alternatives, the reader should reference the account of van Glabbeek et al. [vGLT09].

As with branching bisimulation, divergence sensitivity is preserved by stuttering closure.

Proposition 3.2.15. *Suppose R is a pre-branching bisimulation satisfying the clauses from definition 3.2.12. Then $\mathsf{ST}(R)$ is a divergence-sensitive branching bisimulation.*

Corollary. *Suppose R is a divergence-sensitive branching bisimulation; then so is $\mathsf{ST}(R)$.*

Corollary. *Divergence-sensitive branching bisimulation is stuttering-closed.*

$$x \ \cong_{\Delta B} \ y \qquad \text{iff} \qquad (x, y) \in \mathsf{ST}(\cong_{\Delta B})$$

Inductive branching bisimulation. I have also developed an alternate definition, separate from those of van Glabbeek et al., that I think most clearly illustrates what is happening when we shift to the divergence-sensitive definition. The intuition is as follows. The reason branching bisimulation cannot distinguish a divergent state from a non-divergent state is because one can apply the "stuttering" diagrams (diagrams (a) and (c) from figure 3.2) an infinite number of times in sequence. Basically, we want to remove this possibility by only allowing the stuttering diagrams to be used a finite number of times in sequence. Formally, we do this by using an *inductive* (least-fixpoint) definition nested inside the coinductive (greatest-fixpoint) definition.

Definition 3.2.16 (Inductive branching bisimulation)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are LTSs. Suppose $R \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ is a binary relation on the states of $\mathcal{L}_1$ and $\mathcal{L}_2$. Let $F(X)$ and $B(X)$ be operators on relations defined as follows:*

$$\begin{aligned} F(X) \equiv \{\ (x, y) \mid \forall \alpha\ x'.\ (x \xrightarrow{\alpha} x') \Rightarrow \\ &(\alpha = \tau \wedge (x', y) \in X \wedge (x', y) \in R) \vee \\ &(\exists y',\ y''.\ \ y \overset{\tau}{\Longrightarrow}_{R(x,-)} y' \wedge y' \xrightarrow{\alpha} y'' \wedge (x, y') \in R \wedge (x', y'') \in R)\ \} \end{aligned}$$

$$\begin{aligned} B(X) \equiv \{\ (x, y) \mid \forall \alpha\ y'.\ (y \xrightarrow{\alpha} y') \Rightarrow \\ &(\alpha = \tau \wedge (x, y') \in X \wedge (x, y') \in R) \vee \\ &(\exists x',\ x''.\ \ x \overset{\tau}{\Longrightarrow}_{R(-,y)} x' \wedge x' \xrightarrow{\alpha} x'' \wedge (x', y) \in R \wedge (x'', y') \in R)\ \} \end{aligned}$$

Then let T_F be the least fixpoint of $F(X)$ and T_B be the the least fixpoint of $B(X)$. In other words, T_F is the smallest relation such that $T_F = F(T_F)$; likewise for T_B. Such relations exist by the Knaster-Tarski fixpoint theorem [Tar55].

We say that R is an inductive branching bisimulation *if for all $(x, y) \in R$, $(x, y) \in T_F$ and $(x, y) \in T_B$.*

Definition 3.2.17 (Inductive branching bisimilarity)**.** *Suppose a and b are LTS processes. Then we say a is* inductively branching bisimilar *to b, and write $a \cong_{IB} b$ if there exists an inductive branching* *bisimulation relation R such that $(a, b) \in R$.*

Note on formalization. *In the formal proof, I use the native Coq facility for inductive definitions to build the inner least-fixpoints rather than going through Knaster-Tarski. For our present purposes, this difference is inconsequential.*

This definition is a bit more technical than the previous ones because of the need to make precise the nesting of fixpoints. The main idea is that by making T_F and T_B inductive definitions, we eliminate the possibility of using the stuttering diagrams infinitely many times in sequence. It turns out that this is precisely what divergence-sensitive branching bisimulation is doing.

Proposition 3.2.18. *Suppose R is an inductive branching bisimulation; then R is a divergence-sensitive branching bisimulation.*

Proof. First we must show that R is a plain branching bisimulation; this should be clear from the definitions. Next we need to prove the conditions from definition 3.2.12 Here I prove only the forward direction; the other is symmetric.

Assume $(x, y) \in R$, and thus $(x, y) \in T_F$. We proceed by induction on T_F. We are given a divergence D such that $x \in D$ and must find x' and y' so that $x' \in D$, $y \xrightarrow{\tau} y'$, and $(x', y') \in R$.

Because $x \in D$, there exists an $x_1 \in D$ such that $x \xrightarrow{\tau} x_1$. Applying the induction hypothesis to x_1, we get two cases, corresponding to the disjunction in the definition of $F(X)$. In the first case, we get that $(x_1, y) \in R$ and that the condition from definition 3.2.12 holds on (x_1, y) (this is the inductive step); this allows us to complete the proof.

In the second case, we obtain $y \overset{\tau}{\Longrightarrow}_{R(x,-)} y'$, $y' \xrightarrow{\tau} y''$, and $(x', y'') \in R$ for some y', y''. If the path from y to y' is empty (i.e., $y = y'$), then we complete the proof with x_1 and y''. If the path is nonempty, then there is some y_1 where $y \xrightarrow{\tau} y_1$ with $(x, y_1) \in R$. Now we can complete the proof with x and y_1. □

Proposition 3.2.19. EM *Suppose R is a divergence-sensitive branching bisimulation; then R is an inductive branching bisimulation.*

Proof. Here I again show only the forward direction. The other follows by symmetry. Suppose $(x, y) \in R$; we want to show that $(x, y) \in T_F$. Suppose the opposite for contradiction: then $(x, y) \notin T_F$.

Let $D = \{x' \mid (x', y) \in R \wedge (x', y) \notin T_F\}$. Note $x \in D$. Further, D is a divergence set. If not, there must exist some $x' \in D$, but for all $x' \xrightarrow{\tau} x''$, $x'' \notin D$. But this means $(x', y) \in T_F$

because (roughly) x' cannot use stuttering diagram (a). This gives us a contradiction and shows that D is a divergence.

Because D is a divergence containing x, there must exist some $x' \in D$ and y' where $y \xrightarrow{\tau} y'$ and $(x', y') \in R$. Because $x' \in D$, we know that $(x', y) \in R$ but $(x', y) \notin T_F$. However, $(x', y) \in T_F$, a contradiction.

To see why $(x', y) \in T_F$, consider all x'' where $x' \xrightarrow{\alpha} x''$. Because $(x', y) \in R$, we either get an instance of diagram (a) or diagram (b) from figure 3.2. If we get diagram (b), we can show $(x', y) \in T_F$ directly. However, if we get diagram (a), we know that $\alpha = \tau$. Now we can use the fact that $(x', y') \in R$; we get a diagram of type (a) or (b). If we have diagram (a), we can fill in a diagram of type (b) because we know that $y \xrightarrow{\tau} y'$. Likewise, if we get a diagram of type (b), we can fill out a new diagram of type (b), with the step $y \xrightarrow{\tau} y'$ appended to the front. □

Corollary. $^{\mathsf{EM}}$

$$a \cong_{\Delta B} b \qquad \text{iff} \qquad a \cong_{IB} c$$

Note that only one direction of this proof uses the excluded middle. It turns out that this use of the excluded middle is essential. More precisely, *some* nonconstructive proof principle is required to prove proposition 3.2.19. The result implies the Limited Principle of Omniscience (LPO), a well-known nonconstructive consequence of the excluded middle [BB85].[3] If interpreted constructively, LPO implies the existence of a solution to the halting problem; thus it can have no constructive proof. This means that $a \cong_{\Delta B} b$ is a weaker statement than $a \cong_{IB} b$ in constructive logic, although they are equivalent classically.

Based on the equivalence above, it should be unsurprising that inductive branching bisimulation is reflexive and transitive. However, proving transitivity directly (as opposed to passing through $\cong_{\Delta B}$ and invoking the excluded middle) is rather delicate. As usual, we prove transitivity by showing that inductive branching bisimulation relations are closed under composition. The proof is similar in many ways to proving that the lexicographic ordering of two well-founded orders is itself well-founded — the proof goes by nested induction. The trick is to find the right induction hypothesis. In the following, I sketch the proof for the forward direction. The backward direction is basically the same with the roles of R and S swapped. Suppose X, Y and Z are LTSs and R is an inductive branching bisimulation between X and Y, and that S is an inductive branching bisimulation between Y and Z.

[3] I would like to thank Wim Veldman for a very helpful discussion on this matter.

Lemma 3.2.20. *Suppose* $(x_0, y) \in R$ *and* $(y, z_0) \in S$. *Further suppose the following:*

$$
\begin{aligned}
&\forall x\ x'\ y'\ y''. \\
&\quad (x \xrightarrow{\tau} x') \Rightarrow \\
&\quad (y \overset{\tau}{\Longrightarrow}_{R(x,-)} y') \Rightarrow \\
&\quad (y' \xrightarrow{\tau} y'') \Rightarrow \\
&\quad (x', y'') \in R \Rightarrow \\
&\qquad ((x', z) \in (R \mathbin{⨾} S) \wedge (x', z) \in T_F) \vee \\
&\qquad (\exists z'\ z''. \\
&\qquad\quad z \overset{\tau}{\Longrightarrow}_{(R ⨾ S)(x,-)} z' \wedge \\
&\qquad\quad z' \xrightarrow{\tau} z'' \wedge \\
&\qquad\quad (x, z') \in (R \mathbin{⨾} S) \wedge (x', z'') \in (R \mathbin{⨾} S)).
\end{aligned}
$$

Then $(x_0, z_0) \in T_F(R \mathbin{⨾} S)$.

Proof. Because $(x_0, y) \in R$, we have that $(x_0, y) \in T_F(R)$. The proof goes by induction on $T_F(R)$. The main assumption is used to complete the proof when we reach the case of a non-stuttering τ-transition. □

Lemma 3.2.21. *Suppose* $(x_0, y_0) \in R$ *and* $(y_0, z_0) \in S$. *Then* $(x_0, z_0) \in T_F(R \mathbin{⨾} S)$.

Proof. Because $(y_0, z_0) \in S$, $(y_0, z_0) \in T_F(S)$. The proof goes by induction on $T_F(S)$, using lemma 3.2.20. □

Proposition 3.2.22. $R \mathbin{⨾} S$ *is an inductive branching bisimulation.*

Proof. Use lemma 3.2.21 for the forward direction and show the backward direction by symmetry. □

Corollary. *Inductive branching bisimulation is transitive.*

3.3 Back-and-forth Bisimulation

Branching bisimulation can be adequately characterized by a modification of Hennessy-Milner logic [vGW96]. Instead of the HM modality $\langle o \rangle$, one instead has a new *binary* modality $P\langle o \rangle Q$ which means a program can perform a (possibly empty) series of τ transitions along a path always satisfying P, and then take an o-transition to a state satisfying Q.

However, it is unclear if this logic is also *expressive* for branching bisimulation. Despite considerable effort, I have been unable to devise a way to build characteristic formulae for this logic.[4]

Nonetheless, it is possible to shift perspective slightly and obtain a logic which is both adequate (in its propositional form) and also expressive (in its μ-calculus form). Instead of single process states, we now consider entire process histories. Recall from §2.2 that a process history is essentially a record of some particular (finite) path through an LTS. By using histories when we build the characteristic logic, rather than states, we can build specifications that look "back in time" to previous states in the run.

It is not immediately obvious, but this ability to look back in time is equivalent to the additional observational power we obtain by moving from weak bisimulation to branching bisimulation.

In this section I develop *back-and-forth* bisimulation [DNV95] which is the generalization of branching bisimulation from process states to process histories, and I show how these two notions are related. In the next section, I will give characteristic logics for back-and-forth bisimulation.

Definition 3.3.1 (History transitions)**.** *Suppose $\mathcal{L}$ is some LTS, and h is a process history in $\mathcal{L}$.*

We say h' extends *h with action α (and write $h \xrightarrow{\alpha} h'$) provided h' is the same sequence as h with a new α and x appended to the end such that*

$$\mathsf{curr}(h) \xrightarrow{\alpha} x$$

In other words, $h' = h \bullet [\alpha, x]$.

We say h' weakly extends *h with action α by analogy to weak transitions of states:*

$$h \overset{\alpha}{\Longrightarrow} h' \equiv \begin{cases} h \xrightarrow{\tau}{}^{*}\, h' & \text{when } \alpha = \tau \\ h \xrightarrow{\tau}{}^{*} \xrightarrow{o} \xrightarrow{\tau}{}^{*}\, h' & \text{when } \alpha = o \in \mathcal{O} \end{cases} \tag{3.9}$$

Thus, a weak τ-extension is a series (possibly 0) of τ-extensions, and a weak o-extension is a series of τ-extensions, followed by an o-extension, followed by a series of τ-extensions.

Definition 3.3.2 (Back-and-forth bisimulation)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are τLTSs. Let R be a binary relation on process histories from $\mathcal{L}_1$ and $\mathcal{L}_2$. Then R is a back-and-forth bisimulation if, for all*

[4]Nor have I found any way to prove that such characteristic formulae cannot exist!

$(g, h) \in R$, the following hold:

$$\forall \alpha\ g'.\ g \overset{\alpha}{\Longrightarrow} g' \rightarrow \exists h'.h \overset{\alpha}{\Longrightarrow} h' \wedge (g', h') \in R \tag{3.10}$$

$$\forall \alpha\ h'.\ h \overset{\alpha}{\Longrightarrow} h' \rightarrow \exists g'.g \overset{\alpha}{\Longrightarrow} g' \wedge (g', h') \in R \tag{3.11}$$

$$\forall \alpha\ g'.\ g' \overset{\alpha}{\Longrightarrow} g \rightarrow \exists h'.h' \overset{\alpha}{\Longrightarrow} h \wedge (g', h') \in R \tag{3.12}$$

$$\forall \alpha\ h'.\ h' \overset{\alpha}{\Longrightarrow} h \rightarrow \exists g'.g' \overset{\alpha}{\Longrightarrow} g \wedge (g', h') \in R \tag{3.13}$$

The first two clauses involve going "forward" in time and the last two clauses involve going "backward" in time — hence the name back-and-forth bisimulation. Unfortunately, this puts us in an awkward situation with regard to terminology. There are now two dimensions along which we have terminology for going forward and backward: across the relation (from inputs to outputs) and with respect to time. Rather than adopting nonstandard terminology, I will simply be careful to apply enough modifiers to distinguish which sense is meant. Note that the four clauses above correspond to the four possible ways of moving across these two dimensions.

Like vanilla branching bisimulation, back-and-forth bisimulation is divergence insensitive. We add clauses to the definition in a way totally analogous to the case for branching bisimulation.

Definition 3.3.3 (History divergences)**.** *Suppose D is a set of process histories. Then D is a* history divergence *if, for every $h \in D$, there exists an $h' \in D$ such that $h \xrightarrow{\tau} h'$.*

Definition 3.3.4 (Divergence-sensitive back-and-forth bisimulation)**.** *Suppose R is a binary relation on the process histories of $\mathcal{L}_1$ and $\mathcal{L}_2$. Then R is a divergence-sensitive back-and-forth bisimulation if R satisfies definition 3.3.2 as well as the following, for each $(g, h) \in R$:*

- *For each history divergence D where $g \in D$, there exists $g' \in D$ and h' where $h \xrightarrow{\tau} h'$ and $(g', h') \in R$.*
- *For each history divergence D where $h \in D$, there exists $h' \in D$ and g' where $g \xrightarrow{\tau} g'$ and $(g', h') \in R$.*

Definition 3.3.5 (Divergence-sensitive back-and-forth bisimilarity)**.** *Suppose g and h are process histories. Then g is divergence-sensitive back-and-forth bisimilar to h (written $g \cong_{\Delta BF} h$) if there exists some divergence-sensitive back-and-forth bisimulation R where $(g, h) \in R$.*

Now I can define what I mean when I say that back-and-forth bisimulations generalize branching bisimulations to process histories.

Definition 3.3.6 (Related histories)**.** *Suppose R is some binary relation on τLTS process states. Then we can lift R to a relation on process histories (written $\mathsf{rel}(R)$). $\mathsf{rel}(R)$ is the smallest relation defined by the following rules:*

$$\frac{(x,y) \in R}{([x],\ [y]) \in \mathsf{rel}(R)} \qquad \frac{(g,h) \in \mathsf{rel}(R) \qquad (x,y) \in R}{(g \bullet [\alpha, x],\ h \bullet [\alpha, y]) \in \mathsf{rel}(R)}$$

$$\frac{(g,\ h) \in \mathsf{rel}(R) \qquad (\mathsf{curr}(g), y) \in R}{(g,\ h \bullet [\tau, y]) \in \mathsf{rel}(R)} \qquad \frac{(g,\ h) \in \mathsf{rel}(R) \qquad (x, \mathsf{curr}(h)) \in R}{(g \bullet [\tau, x],\ h) \in \mathsf{rel}(R)}$$

Basically, related histories are histories that progress *almost* in lockstep. The initial states of both histories must be related by R, and each observable event they produce must be matched by a corresponding event in the other. Either history may "run ahead" of the other by making a τ transition, so long as the resulting states are still related by R.

The main theorem of this section is that back-and-forth bisimilarity is just branching bisimilarity lifted in this way. To prove this theorem, we will need several additional technical definitions.

Definition 3.3.7 (X-closure)**.** *Suppose R is a relation on process histories. The X-closure of R (written $\mathcal{X}(R)$) is defined as so that $(g,h) \in \mathcal{X}(R)$ whenever:*

$$\begin{aligned} &\exists g_0.\ \exists h_0. \\ &\quad g_0 \overset{\tau}{\Longrightarrow} g \ \wedge\ h_0 \overset{\tau}{\Longrightarrow} h \ \wedge \\ &\quad (g_0, h) \in R \ \wedge\ (g, h_0) \in R \end{aligned} \tag{3.14}$$

Definition 3.3.8 (Current relation)**.** *Suppose R is a relation on process histories. The* current relation *of R (written $\mathsf{curr}(R)$) is defined as*

$$\mathsf{curr}(R) \equiv \{(x,y) \mid \exists (g,h) \in R.\ x = \mathsf{curr}(g) \wedge y = \mathsf{curr}(h)\} \tag{3.15}$$

Taking the X-closure of a history relation relates some additional "internal" states, much the same way that stuttering closures does for branching bisimulations. The main point is that taking the X-closure of a back-and-forth bisimulation is again a back-and-forth bisimulation. The following propositions are stated without proof. They are mostly straightforward once properly stated; see the formal development for details.

Proposition 3.3.9. *Suppose R is a divergence-sensitive back-and-forth bisimulation. Then $\mathcal{X}(R)$ is a divergence-sensitive back-and-forth bisimulation.*

Proposition 3.3.10. *Suppose R is a back-and-forth bisimulation. Then $R \subseteq \mathsf{rel}(\mathsf{curr}(\mathcal{X}(R)))$.*

Proposition 3.3.11. *Suppose R is a divergence-sensitive back-and-forth bisimulation. Let $S \equiv \mathsf{curr}(\mathcal{X}(R))$. Then S is a divergence-sensitive branching bisimulation.*

Proposition 3.3.12. *Suppose R is a divergence-sensitive branching bisimulation. Then $\mathsf{rel}(R)$ is a divergence-sensitive back-and-forth bisimulation.*

Theorem 3.3.13.

$$x \cong_{\Delta B} y \qquad \text{iff} \qquad (x, y) \in \mathsf{curr}(\cong_{\Delta BF}) \tag{3.16}$$

$$g \cong_{\Delta BF} h \qquad \text{iff} \qquad (g, h) \in \mathsf{rel}(\cong_{\Delta B}) \tag{3.17}$$

This series of results explains the way that branching bisimulation is related to back-and-forth bisimulation. Theorem 3.3.13, in particular, shows that they are interdefinable by the relation transformers curr and rel.

3.4 Characteristic Logic for Back-and-forth Bisimulation

Divergence-sensitive back-and-forth bisimulation is characterized by a modal logic, similar to the logic of Hennessy and Milner. The main differences are that:

1. the worlds are process histories rather than process states
2. modalities use *weak* extensions in their definition
3. there are new *backward* modalities, and
4. there are new modalities for handling divergence.

More formally, the data needed to build the characteristic logic for back-and-forth bisimulation are:

- The worlds $\mathcal{W}$ are the LTS histories;
- the accessibility relation is $\cong_{\Delta BF}$,
- the modes are given as below,
- and there are no atoms, so $\mathcal{A} = \emptyset$.

$$\mathcal{M} := [\alpha] \mid \langle\alpha\rangle \mid \overset{\leftarrow}{[\alpha]} \mid \overset{\leftarrow}{\langle\alpha\rangle} \mid \Delta \mid \nabla$$

$$[\![[\alpha]]\!](P) \equiv \{h \mid \forall h'.\ (h \overset{\alpha}{\Longrightarrow} h') \Rightarrow h' \in P\} \tag{3.18}$$

$$[\![\langle\alpha\rangle]\!](P) \equiv \{h \mid \exists h'.\ (h \overset{\alpha}{\Longrightarrow} h') \wedge h' \in P\} \tag{3.19}$$

$$[\![\overleftarrow{[\alpha]}]\!](P) \equiv \{h \mid \forall h'.\ (h' \overset{\alpha}{\Longrightarrow} h) \Rightarrow h' \in P\} \tag{3.20}$$

$$[\![\overleftarrow{\langle\alpha\rangle}]\!](P) \equiv \{h \mid \exists h'.\ (h' \overset{\alpha}{\Longrightarrow} h) \wedge h' \in P\} \tag{3.21}$$

$$[\![\Delta]\!](P) \equiv \{h \mid \exists D.D \text{ is a divergence } \wedge h \in D \wedge (\forall h'.\ h' \in D \Rightarrow h' \in P)\} \tag{3.22}$$

$$[\![\nabla]\!](P) \equiv \{h \mid \forall D.D \text{ is a divergence } \Rightarrow h \in D \Rightarrow (\exists h'.\ h' \in D \wedge h' \in P)\} \tag{3.23}$$

Note that these six modalities correspond directly to the six clauses in the definition of divergence-sensitive back-and-forth bisimulation. It should be straightforward to see that the above modalities are monotone operators in P. Furthermore, each one preserves the property of being closed with respect to $\cong_{\Delta BF}$ — each modality exercises a different clause of the definition of divergence-sensitive back-and-forth bisimulation.

The new modalities Δ and ∇ deserve some mention; these modalities allow us to observe the divergence and convergence properties of programs. Basically, ΔP holds when a process can diverge along a path where every state on the path satisfies P. In other words, ΔP means "can diverge while satisfying P." In particular $\Delta\top$ simply means that there is some divergence, and $\Delta\bot$ is equal to $\bot$. Another interesting formula is $[\tau](\Delta\top)$, which says that every τ-reachable state diverges — that is, the program cannot converge.

Dually, ∇ discusses convergence properties. The formula ∇P means that every divergence emerging from the current state eventually contains a state satisfying P. In other words, ∇P means "the process converges unless P is eventually satisfied." In particular, $\nabla\bot$ means that the program certainly converges, and $\nabla\top$ equals $\top$. Dual to the formula above, $\langle\tau\rangle(\nabla\bot)$ means that it is possible for the program to converge.

The propositional form of this logic is denoted $\mathcal{F}_{\Delta BF}$. As before, the main result regarding this logic is adequacy. With the addition of τ-actions, however, we need a stronger finiteness assumption.

Definition 3.4.1 (Weakly image-finite)**.** *Suppose $\mathcal{L}$ is a τLTS. We say $\mathcal{L}$ is* weakly image-finite *if, for each state x and α, the set $\{x' \mid x \overset{\alpha}{\Longrightarrow} x'\}$ is finite. A process (or history) is weakly image-finite if its underlying LTS is. In other words, for each state, the set of states reachable by a weak transition from it is finite.*

Despite the name, the assumption that an LTS is weakly image-finite is a stronger assumption than that it is image-finite. Unfortunately, this finiteness fact is not quite in the form we need to do the adequacy proof below, because it discusses states and state transitions rather than histories and history transitions. Thus, before moving on to the main adequacy proof, we need to prove some technical facts about finiteness.

Proposition 3.4.2. *Suppose h is a τELTS process history. Then the set $\{h' \mid h' \overset{\alpha}{\Longrightarrow} h\}$ is finite.*

Proof. The sequence of states in a history is finite, therefore one can step backward only a finite number of times. □

It turns out that the set $\{h' \mid h \overset{\alpha}{\Longrightarrow} h'\}$ of weakly-reachable histories is not finite in general, even when the underlying LTS is weakly image-finite. This is because the runs in an LTS may contain cycles — that is, sequences of τ steps that return to a previously-visited state. When a cycle exists, there are an infinite number of possible histories corresponding to the number of times the cycle is traversed. However, we are saved by the fact that there is no observable difference between these histories. In particular, a history with cycles is always bisimilar to a history without cycles. Furthermore the acyclic histories *do* form a finite set. This is sufficent for the adequacy proof below.

Proposition 3.4.3. EM *Suppose $\mathcal{L}$ is a weakly image-finite τELTS. Then for each history h in $\mathcal{L}$ and for each $\alpha \in \mathcal{O} \cup \{\tau\}$, there exists a finite set of process histories H such that:*

$$h' \in H \qquad \textit{implies} \qquad h \overset{\alpha}{\Longrightarrow} h' \tag{3.24}$$

$$h \overset{\alpha}{\Longrightarrow} h' \qquad \textit{implies} \qquad \exists h'' \in H \wedge h' \cong_{\Delta BF} h'' \tag{3.25}$$

Theorem 3.4.4 (Adequacy for $\mathcal{F}_{\Delta BF}$)**.** EM *Suppose g and h are weakly image-finite τLTS process histories where for all $f \in \mathcal{F}_{\Delta BF}$, $g \models f$ implies $h \models f$. Then $g \cong_{\Delta BF} h$.*

Proof. As with strong bisimulation, suppose the opposite. Then one of the six clauses in the definition of $\cong_{\Delta BF}$ fails. In each case we can build a contradiction using the modality associated with the clause. Some care is needed to properly exploit finiteness, but the subcases are otherwise relatively straightforward. □

The μ-calculus form of this logic is denoted $\mathcal{T}_{\Delta BF}$.

Definition 3.4.5 (Characteristic formula for $\mathcal{T}_{\Delta BF}$)**.** *For a given τLTS $\mathcal{L}$, the characteristic formula* *for $\mathcal{L}$ is:*

$$
\begin{aligned}
&\mathsf{CF}(\mathcal{L}) \equiv \nu X.\ \lambda h. \\
&\qquad \left(\bigwedge_{\alpha} \bigwedge_{\{h' \ \mid \ h \overset{\alpha}{\Longrightarrow} h'\}} \langle\alpha\rangle X@h' \right) \wedge \left(\bigwedge_{\alpha} [\alpha] \bigvee_{\{h' \ \mid \ h \overset{\alpha}{\Longrightarrow} h'\}} X@h' \right) \wedge \\
&\qquad \left(\bigwedge_{\alpha} \bigwedge_{\{h' \ \mid \ h' \overset{\alpha}{\Longrightarrow} h\}} \overleftarrow{\langle\alpha\rangle}\ X@h' \right) \wedge \left(\bigwedge_{\alpha} \overleftarrow{[\alpha]} \bigvee_{\{h' \ \mid \ h' \overset{\alpha}{\Longrightarrow} h\}} X@h' \right) \wedge \\
&\qquad \left(\bigwedge_{\{D \ \mid \ D \text{ is a divergence} \ \wedge \ h \in D\}} \Delta \left(\bigvee_{\{h' \ \mid \ h' \in D\}} X@h' \right) \right) \wedge \left(\nabla \left(\bigvee_{\{h' \ \mid \ h \xrightarrow{\tau} h'\}} X@h' \right) \right)
\end{aligned}
$$

When h is a process history with LTS $\mathcal{L}$, we define $\mathsf{CF}(h) \equiv \mathsf{CF}(\mathcal{L})@h$.

As before, we observe that each of the six clauses above corresponds to one of the six clauses in the definition of $\cong_{\Delta BF}$. Only the clause for Δ requires any real work. When unfolded, this clause states the condition from proposition 3.2.14 (appropriately lifted to histories) rather than the one from definition 3.3.4.

These observations culminate in the main theorem.

Theorem 3.4.6 (Expressivity for $\mathcal{T}_{\Delta BF}$)**.** *For all process histories g and h,*

$$g \cong_{\Delta BF} h \qquad \text{iff} \qquad h \models \mathsf{CF}(g)$$

Discussion. The logic for back-and-forth bisimulation is technically quite elegant. Each of the six modalities is orthogonal, and it is quite easy to relate them to the clauses of the relational bisimulation definition. Furthermore, this form allows us to precisely identify which observations correspond to which features of the relation: $[\alpha]$ and $\langle\alpha\rangle$ express the observations due to weak bisimulation, $\overleftarrow{[\alpha]}$ and $\overleftarrow{\langle\alpha\rangle}$ express observations due to the branching structure, and Δ and ∇ express observations of divergence and convergence.

Unfortunately, the backward modalities $\overleftarrow{[\alpha]}$ and $\overleftarrow{\langle\alpha\rangle}$ are rather strange. It is a little difficult to imagine how to use them for specifying realistic program specifications. Nontrivial nestings of forward and backward modalities quickly become mind-bending, and it is difficult to understand what one has written down. In contrast, the "until" modality ($P\langle\alpha\rangle Q$) found in van Glabbeek and Weijland's characteristic logic for branching bisimulation [vGW96] seems quite a bit more natural for expressing specifications of interest.

Of course, we could simply add the until modality to the characteristic logic — back-and-forth bisimulation preserves the truth of such specifications, so soundness is unharmed; and adding new expressive power cannot harm adequacy/expressiveness. Even better would be if we could show how to define the until modality in terms of the backward modalities. Sadly, this does not seem possible (although I have no proof to that effect); nor does it seem possible to define the backward modalities in terms of until modalities.

This leads to an odd situation where two logics both have a similar global expressivity property (adequacy), but neither is known to be definable in terms of the other. Although strange, this situation is not contradictory — there is no particular reason *global* expressivity properties (like adequacy) should imply *local* expressivity properties (interdefinability).

This situation is similar to the case where two programming languages are both Turing-complete, but neither may express the constructs of the other without a global program rearrangement. Felleisen has an excellent treatment of this topic for programming languages [Fel91], which is derived from earlier work by Kleene [Kle52, §74] (which treated logics, bringing us full circle).

Despite the oddness of the backward modalities, we will continue to examine systems based on back-and-forth bisimulation. This is primarily because I do not know if it is possible to find characteristic formulae for logics with "until" modes rather than backward modes. The backward modes also have pleasing proof-theoretic connections to the forward modalities. For example, the following bidirectional rule of inference is valid for back-and-forth logic:

$$\frac{\overset{\leftarrow}{\langle\alpha\rangle}\; P \vdash Q}{\overline{P \vdash [\alpha]Q}}$$

This symmetrical structure of modalities leads me to believe that an examination of the proof theory of back-and-forth logic would perhaps be easier than for a logic with until modalities.

3.5 Related Work

The basic idea of strong bisimulation as a notion of program equivalence goes back to Hennessy and Milner [HM80, HM85], and the characteristic logic for it bears their names. Bisimulation was reformulated by Park [Par81] into the modern greatest-fixpoint style. Davide Sangiorgi has written an excellent introductory text on bisimulation methods with an emphasis on their uses in Computer Science [San12].

Characteristic formulae for strong (and weak) bisimulation were studied by Müller-Olm [MO98]. Aceto et al. have presented a general framework for characteristic formulae that differs somewhat from my own [AILS12]. My presentation of characteristic formulae for strong bisimulation is only novel, perhaps, because of its concise statement using the powerful operators of my variant of the modal μ-calculus. However, characteristic formulae for back-and-forth bisimulation have not previously been presented, to my knowledge.

The idea of weak bisimulation first appeared in Milner's seminal book on The Calculus of Communicating Systems [Mil80], where it was called observation equivalence. The stronger notion of branching bisimulation is developed in a line of research undertaken by R. van Glabbeek and his collaborators [vG93, vGW96, vGLT09]; they have argued (successfully, I think) that branching bisimulation is the strongest equivalence on LTS processes that makes sense from the perspective of program equivalence. The definition of inductive branching bisimulation (definition 3.2.16) and the proof that it is (classically) equivalent to divergence-sensitive branching bisimulation is new.

Back-and-forth bisimulation and its characteristic logic were examined by De Nicola and Vaandrager [DNV95] — one of three logical systems they investigated. The presentation here improves on theirs in several aspects. De Nicola and Vaandrager proved only equation 3.16 of theorem 3.3.13 above; the result of equation 3.17 is new. Also, they showed only adequacy for a propositional logic, whereas I have also shown expressivity for the modal μ-calculus variant. Finally, De Nicola and Vaandrager did not examine the divergence-sensitive variant.

Some have argued that methods based on bisimulation are too strong — that is, that they distinguish too many programs [Uli92, BIM95, NV07]. These authors have advanced alternate notions of program equivalance based on weaker notions like copy+refusal testing, ready simulation, and trace equivalence. For my present purposes, the arguments advanced by these authors are largely irrelevant.

I am not attempting to claim here that bisimulation or bisimulation-based refinements are the One True Way to think about program equivalence/refinement. Instead, I make the more modest claim that *for the purposes of reasoning about compiler correctness* we want the finest possible relations that still admit the desired classes of program transformations. This allows the compiler to be completely agnostic about the notion of program equivalence/refinement prefered by its users while still satisfying all their needs.

Chapter 4

Behavioral Refinement

As I argued in chapter 1, a theory of compiler correctness for programs that may go wrong cannot be symmetric. Otherwise, we are unable to specialize undefined behavior into arbitrary behavior, which defeats the entire purpose of having the concept of undefined behavior in the first place.

In this chapter, I shall examine a theory of refinement that allows exactly this kind of specialization — I call this theory *behavioral* refinement. Both a strong theory and a weak (i.e., branching) theory will be developed. Among the programs that may go wrong, the safe programs (i.e., those that will never go wrong) are an important subset. For both the strong and the weak theory, I shall examine several ways we can characterize the safe programs, and we shall see how this sheds light on the theory of behavioral refinement.

The technical material in this section will closely follow the template laid out in chapter 3. By and large, the proofs follow quite similar patterns, and I will be omitting most of them. Only where some interesting new facet is uncovered will I go into detail. Readers interested in further details are encouraged to consult the formal proof development.

4.1 Strong Behavioral Refinement

One way of characterizing the problem with bisimulation is that it does not allow specialization for programs that go wrong. This, however, doesn't go far enough. In fact, the labeled transition systems on which bisimulations are based cannot even *express* programs that might go wrong.

Compare with the popular approach to operational semantics proposed by Wright and Felleisen [WF94]. In their approach, there are essentially three sorts of program states: those which are still running, those which are *values*, and those which are neither running nor values. This last sort is often called *stuck* states, and they represent going wrong.

With labeled transition systems, one instead has only two sorts of states: running states and halted (or deadlocked) states. There is no representation for going wrong. My solution to this expressiveness problem is the simplest change I can imagine that addresses the issue. I simply add a new, distinguished error state that represents going wrong. This new state is written $\mho$, and is pronounced "wrong." I call the resulting structures extended labeled transition systems.

Definition 4.1.1 (Lifted states)**.** *Let $\mathcal{S}$ be some set of states. We define the lifted states $\mathcal{S}^{\mho} = \{\mho\} \cup \mathcal{S}$, as the set $\mathcal{S}$ extended with a new distinguished element $\mho \notin \mathcal{S}$.*

Definition 4.1.2 (Extended Labeled Transistion System)**.** *Fix $\mathcal{O}$, a nonempty set of observations. An extended labeled transition system over $\mathcal{O}$ is a tuple $\langle \mathcal{S}, \longrightarrow \rangle$, where $\longrightarrow$ is a 3-place relation on $\mathcal{S} \times \mathcal{O} \times \mathcal{S}^{\mho}$. When $(s, o, s') \in \longrightarrow$, we write $s \xrightarrow{o} s'$.*

Note that while $\mho$ may be the destination of a transition, it can never be the source. This comports well with the Wright/Felleisen notion of stuck states, which, by definition, cannot take any more steps. It is also worth noting at this point that the basic idea of designating a particular state to represent dynamic errors is not new — this idea was already present in Plotkin's notes on structural operational semantics from 1981 [Plo81]. What is new is our treatement of this designated error state via refinement, as discussed below.

An alternative design would be to designate a subset of the states as wrong states, and impose the restriction that wrong states may not step. For our purposes, the two choices seem essentially the same. We may always take a system with a set of wrong states and collapse them down into a single state without changing any observable properties of programs.

Using extended labeled transition systems (ELTSs) we can represent programs that may exhibit undefined behavior; any time the program performs an action, the result of which is undefined, we can represent that with an arc with $\mho$ as the destination state.

It is a bit inconvenient that $\longrightarrow$ is defined over different source and destination types. We can lift the step relation to take $\mathcal{S}^{\mho}$ as the source space in a straightforward way. For reasons that should become clear later, we call this lifting *must* stepping.

Definition 4.1.3 (Must stepping)**.** *Suppose $\mathcal{L}$ is an ELTS with state space $\mathcal{S}$. Then we define the must-step relation $\underset{must}{\longrightarrow} \subseteq \wp(\mathcal{S}^{\mho} \times \mathcal{O} \times \mathcal{S}^{\mho})$ as follows:*

$$x \underset{must}{\xrightarrow{o}} x' \qquad \equiv \qquad x \neq \mho \ \wedge \ x \xrightarrow{o} x' \tag{4.1}$$

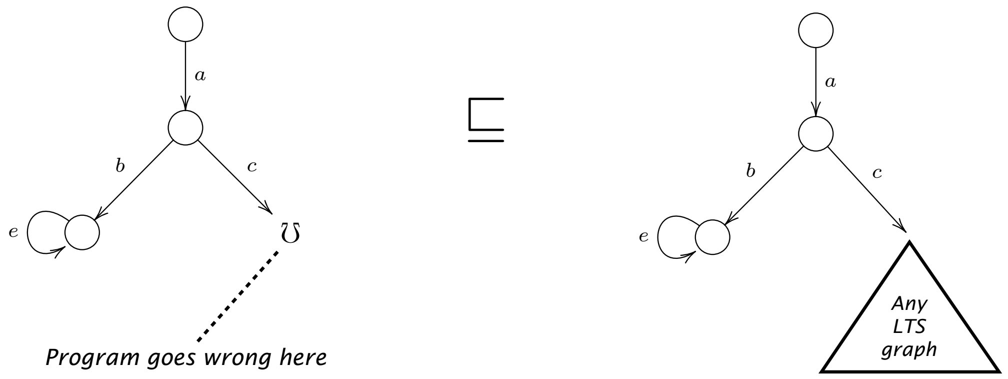


Figure 4.1: ℧-expansion

The next question that presents itself is: how should we modify the notion of bisimulation to account for undefined behavior? Perhaps the most obvious route would be to take the same basic structure as bisimulation, and require that the two programs go wrong at the same time.

Definition 4.1.4 (Strong ℧-bisimulation)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are ELTSs. Let $R \subseteq \wp(\mathcal{S}_1^{℧} \times \mathcal{S}_2^{℧})$ be a binary relation on the lifted state spaces of $\mathcal{L}_1$ and $\mathcal{L}_2$. We say R is a* strong ℧-bisimulation[1] *if, whenever $(x, y) \in R$ the following hold:*

$$x = ℧ \qquad \textit{iff} \qquad y = ℧ \tag{4.2}$$

$$\forall o\ x'.\ \ x \xrightarrow[\textit{must}]{o} x' \ \text{ implies } \ \exists y'.\ y \xrightarrow[\textit{must}]{o} y' \text{ and } (x', y') \in R \tag{4.3}$$

$$\forall o\ y'.\ \ y \xrightarrow[\textit{must}]{o} y' \ \text{ implies } \ \exists x'.\ x \xrightarrow[\textit{must}]{o} x' \text{ and } (x', y') \in R \tag{4.4}$$

We write $x \cong_{℧} y$ if there exists a ℧-bisimulation R relation with $(x, y) \in R$.

Note that equation 4.3 is true vacuously if $x = ℧$ because the clause $x \xrightarrow[\text{must}]{o} x'$ can only hold when $x \neq ℧$; likewise for 4.4. Thus, a strong ℧-bisimulation is essentially just a strong bisimulation over the lifted state space, except that ℧ is only ever related to ℧.

Note that $\cong_{℧}$ is a reflexive, transitive relation generated as the greatest fixpoint of a monotone operator. It is a very natural extension of bisimulation that simply treats ℧ as a new state distinct from every other state. However, this relation is symmetric, and thus is inappropriate for handling the examples from chapter 1.

Instead, we need to find an *asymmetric* relation that lets us refine undefined behavior into arbitrary behavior. Figure 4.1 gives a graphical example; basically anywhere a ℧ appears, we may replace it by an arbitrary subgraph. There are several equivalent ways to define the relation we

[1]The term ℧-bisimulation should be pronounced "behavioral bisimulation." I use the terms interchangably.

want, but the most elegant requires an additional notion of stepping, called "may" stepping that complements the must-step relation defined above.

Definition 4.1.5 (May stepping)**.** *Suppose $\mathcal{L}$ is an ELTS with state space $\mathcal{S}$. Then we define the may-step relation $\underset{may}{\longrightarrow} \subseteq \wp(\mathcal{S}^{\mho} \times \mathcal{O} \times \mathcal{S}^{\mho})$ as follows:*

$$x \underset{may}{\xrightarrow{o}} x' \qquad \equiv \qquad (x \neq \mho \ \wedge \ x \xrightarrow{o} x') \vee (x = x' = \mho) \tag{4.5}$$

The may-step relation includes the must-step relation, but also strictly more. Once a process enters the $\mho$ state, arbitrary transitions are allowed by the may-step relation, remaining in the $\mho$ state all the while. In particular, any observation at all may be generated via may-steps from $\mho$. The must-step relation captures the transitions that a program must be able to make; that is, well-defined transitions that are guaranteed to be preserved from input to output programs. The may-step relation captures transitions that are *allowed* to be present in output programs.

We can think of the transition system defined by must-stepping as a sort of lower bound on program behavior, and the system defined by may-stepping as an upper bound — actual program behavior falls somewhere between. Then refinement can be imagined as the process of shrinking the interval by increasing the lower bound and decreasing the upper bound. The definition of behavioral refinement below captures this intuition.

Definition 4.1.6 (Strong $\mho$-refinement)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are ELTSs. Let $R \subseteq \wp(\mathcal{S}_1^{\mho} \times \mathcal{S}_2^{\mho})$ be a binary relation on the lifted state spaces of $\mathcal{L}_1$ and $\mathcal{L}_2$. We say R is a* strong $\mho$-refinement[2] *if, whenever $(x, y) \in R$ the following hold:*

$$\forall o\ x'.\ \ x \underset{must}{\xrightarrow{o}} x' \ \text{ implies } \ \exists y'.\ y \underset{must}{\xrightarrow{o}} y' \text{ and } (x', y') \in R \tag{4.6}$$

$$\forall o\ y'.\ \ y \underset{may}{\xrightarrow{o}} y' \ \text{ implies } \ \exists x'.\ x \underset{may}{\xrightarrow{o}} x' \text{ and } (x', y') \in R \tag{4.7}$$

Suppose x and y are ELTS processes. We say y $\mho$-refines x, and write $x \sqsubseteq_{\mho} y$ if there exists a strong $\mho$-refinement R with $(x, y) \in R$.

Clause 4.6 is essentially the "forward" direction of the refinement, and refers only to the must-step relation; likewise 4.7 is the "backward" direction and uses the may-step relation. In short, obligations are carried forward, and permissions are carried backward. As we shall see, this basic idea will translate smoothly to the branching case, as well as the corresponding enhancement of

[2] As with $\mho$-bisimulation, this term should be pronounced "behavioral refinement."

chapter 7. As usual, $\sqsubseteq_{\mho}$ is reflexive, transitive and the greatest fixpoint of a monotone operator.

Unlike some authors, I orient refinement so that $x \sqsubseteq y$ means that y is "better" than x. That is, y goes wrong less often. Think: bigger is better.

Although this definition is elegant, it is not completely obvious that it is the generalization we want. Some readers may find the following alternate definition more understandable.

Definition 4.1.7 (Strong $\mho$-refinement (alternate))**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are two ELTSs over $\mathcal{O}$, and let R be a two-place relation on the states of $\mathcal{L}_1$ and $\mathcal{L}_2$ (note:* not *the lifted states). We say R is a refinement relation if, whenever $(x, y) \in R$, both of the following hold.*

1. *For each $x \xrightarrow{o} x'$ there exists y' such that*
   - *$y \xrightarrow{o} y'$ and*
   - *if $x' \neq \mho$ then $y' \neq \mho$ and $(x', y') \in R$.*
2. *For each $y \xrightarrow{o} y'$ there exists x' such that*
   - *$x \xrightarrow{o} x'$ and*
   - *if $x' \neq \mho$ then $y' \neq \mho$ and $(x', y') \in R$.*

We say that y refines x (or that x is refined by y) and write $x \sqsubseteq_{\mho'} y$ if $x = \mho$ or if $x \neq \mho$ and $y \neq \mho$ and there exists an R which is a refinement relation and $(x, y) \in R$.

Note that this definition of refinement relations is asymmetric; in both directions we include the identical phrase: if $x' \neq \mho$ then $y' \neq \mho$ and $(x', y') \in R$. In both directions, if the program being refined (state x) steps to $\mho$, no requirements are made about the future behavior of the refined program (state y'). Instead we only require that *if* x steps to some $x' \neq \mho$, then y' is also not $\mho$ and the new states are again in the refinement relation. This rather minor difference is what allows us to consider $\mho$ as a placeholder for arbitrary behavior.

Proposition 4.1.8. *Both definitions of strong $\mho$-refinement are equivalent.*

$$x \sqsubseteq_{\mho} y \qquad \textit{iff} \qquad x \sqsubseteq_{\mho'} y$$

This alternate style of definition does not seem to generalize well to the branching case. In contrast, the may/must formulation smoothly generalizes, so it will be the definition mainly studied here.

In addition, I will present the following evidence that $\mho$-refinement is the appropriate generalization of bisimulation to handle undefined behavior: first, $\mho$-refinements are asymmetric versions of $\mho$-bisimulations in a way we can make precise; second, $\mho$-refinement can be characterized in a precise way by the intensional process of replacing $\mho$ states with arbitrary subgraphs; and third, $\mho$-refinement is characterized by a nice generalization of Hennessy-Milner logic.

I begin by showing that $\mho$-refinements really are asymmetric versions of $\mho$-bisimulations. The essence of the following proposition is that a relation is a $\mho$-bisimulation iff it is a refinement in both directions. Stated another way, $\mho$-bisimulation is the symmetric notion of $\mho$-refinement.

Proposition 4.1.9. *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are ELTSs. Let $R \subseteq \wp(\mathcal{S}_1^{\mho} \times \mathcal{S}_2^{\mho})$ Then R is a $\mho$-bisimulation iff R is a $\mho$-refinement and R^{-1} is a $\mho$-refinement.*

Proof. The forward direction is easy; I will not recite the details. The backward direction is also mostly easy. The only interesting bit is showing that clause 4.2 holds when we assume that both R and R^{-1} are $\mho$-refinements. Assume $(x, y) \in R$ and $x = \mho$, we must show $y = \mho$ (I elide the symmetric case). By definition $(y, x) \in R^{-1}$. Recall that we require the set of observations, $\mathcal{O}$, to be nonempty; let $o \in \mathcal{O}$ be some arbitrary observation. Note that $x \xrightarrow[\text{may}]{o} \mho$. Because R^{-1} is a refinement, there exists some y' where $y \xrightarrow[\text{may}]{o} y'$. There are two cases — either $y \xrightarrow[\text{must}]{o} y'$ or $y = y' = \mho$. In the second case, we have that $y = \mho$ and we are done. In the remaining case, we can use the other direction of refinement to show that there exists x' where $x \xrightarrow[\text{must}]{o} x'$; however, this is a contradiction because $x = \mho$. □

The next evidence I give for $\mho$-refinement shows that it corresponds quite tightly to the graphical intuition about editing graphs to replace $\mho$ states. I make this intuition precise below by defining $\mho$-expansions. Basically, the idea is that an ELTS $\mathcal{L}_1$ can be $\mho$-expanded into $\mathcal{L}_2$ if we can map the state space of $\mathcal{L}_1$ into $\mathcal{L}_2$ in a one-to-one manner such that a transition exists in $\mathcal{L}_2$ exactly when a corresponding transition exists in $\mathcal{L}_1$ — except that transitions to a $\mho$ may instead go to an arbitrary state in $\mathcal{L}_2$.

Definition 4.1.10 ($\mho$-expansion)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are ELTSs. Let $f : \mathcal{S}_1 \to \mathcal{S}_2$ be a function from the state space of $\mathcal{L}_1$ to the state space of $\mathcal{L}_2$. We say that f is a $\mho$-expansion if the following conditions hold:*

$$f \text{ is injective} \tag{4.8}$$

$$x \xrightarrow{o} x' \text{ and } x' \neq \mho \text{ implies } f(x) \xrightarrow{o} f(x') \tag{4.9}$$

$$x \xrightarrow{o} \mho \text{ implies } \exists y'.\ \ f(x) \xrightarrow{o} y' \tag{4.10}$$

$$f(x) \xrightarrow{o} y' \text{ implies } \exists x'.\ \ x \xrightarrow{o} x' \text{ and } (x' = \mho \vee f(x') = y') \tag{4.11}$$

When x and y are ELTS processes, we say that x $\mho$-expands into y, and write $x \preceq_\mho y$ when there exists a $\mho$-expansion f where either $x = \mho$ or $f(x) = y$.

Note that, unlike the bisimulation definitions we have seen so far, $\mho$-expansion is quite intensional; it only holds between two processes if they have very similar graph structure. We can *make* it extensional by considering it up to bisimulation; that is, work with the relation $\cong_\mho \fatsemi \preceq_\mho \fatsemi \cong_\mho$. The main theorem of interest here is that this relation is precisely the same as $\sqsubseteq_\mho$. The forward direction of this proof is easy.

Lemma 4.1.11. *Suppose x $(\cong_\mho \fatsemi \preceq_\mho \fatsemi \cong_\mho)$ y; then $x \sqsubseteq_\mho y$.*

Proof. $\sqsubseteq_\mho$ is transitive and $\cong_\mho$ is included in $\sqsubseteq_\mho$ by proposition 4.1.9. Thus, it suffices to show that $x \preceq_\mho y$ implies $x \sqsubseteq_\mho y$. By the definition of $\preceq_\mho$, there exists a $\mho$-expansion function f with $f(x) = y$ or $x = \mho$. It suffices to show that the relation $R = \{(x, y) \mid x = \mho \vee f(x) = y\}$ is a $\mho$-refinement.

Thus, suppose $(x, y) \in R$. If $x = \mho$ clauses 4.6 and 4.7 are both direct. Then suppose instead that $f(x) = y$ and $x \neq \mho$.

To prove 4.6, suppose $x \xrightarrow[\text{must}]{o} x'$ for some x'; we must find y' such that $y \xrightarrow[\text{must}]{o} y'$ and $(x', y') \in R$. Suppose $x' \neq \mho$. Then we set $y' = f(x')$ and finish using clause 4.9. If $x' = \mho$, we instead invoke 4.10 to find y'; then $(x', y') \in R$ because $x' = \mho$.

To prove 4.7, suppose $y \xrightarrow[\text{may}]{o} y'$ for some y'; we must find x' such that $x \xrightarrow[\text{may}]{o} x'$ and $(x', y') \in R$. We know that $y = f(x)$ (and thus that $y \neq \mho$), so we may use 4.11 to find x' and finish. □

The other direction of this proof, however, requires some additional ingenuity. Given a process x that is refined by a process y, we must find a pair of processes q_1 and q_2 where $x \cong_\mho q_1$, $q_1 \preceq_\mho q_2$ and $q_2 \cong_\mho y$.

The details of this proof are a bit tedious, so here I will simply give a high-level sketch. The main intellectual work in this proof is finding the correct constructions for q_1 and q_2 so that the above proof obligations can be discharged.

Suppose we are given two ELTSs X and Y and a $\mho$-refinement relation R between X and Y. Then we can build a new ELTS, Q_1 with state space defined by the pairs of real states in R; that is, the state space of Q_1 is the set $\{(x, y) \mid (x, y) \in R \wedge x \neq \mho \wedge y \neq \mho\}$. The step relation of Q_1 is defined by the following two rules:

$$\frac{x \xrightarrow{o} x' \qquad y \xrightarrow{o} y' \qquad x' \neq \mho \qquad y' \neq \mho}{(x, y) \xrightarrow{o} (x', y')} \qquad \frac{x \xrightarrow{o} \mho}{(x, y) \xrightarrow{o} \mho}$$

Now we construct a second ELTS, Q_2, to be a $\mho$-expansion of Q_1. Let the state space of Q_2 be the disjoint union of pairs in R and states in Y. The $\mho$-expansion mapping injects the pairs in R into the state space of Q_2; this function is obviously injective. The stepping relation of Q_2 is defined by the following rules:

$$\frac{x \xrightarrow{o} x' \qquad y \xrightarrow{o} y' \qquad x' \neq \mho \qquad y' \neq \mho}{\mathsf{inl}\ (x,y) \xrightarrow{o} \mathsf{inl}\ (x',y')} \qquad \frac{x \xrightarrow{o} \mho \qquad y \xrightarrow{o} y' \qquad y' \neq \mho}{\mathsf{inl}\ (x,y) \xrightarrow{o} \mathsf{inr}\ y'}$$

$$\frac{x \xrightarrow{o} \mho \qquad y \xrightarrow{o} \mho}{\mathsf{inl}\ (x,y) \xrightarrow{o} \mho} \qquad \frac{y \xrightarrow{o} y' \qquad y \neq \mho}{\mathsf{inr}\ y \xrightarrow{o} \mathsf{inr}\ y'} \qquad \frac{y \xrightarrow{o} \mho}{\mathsf{inr}\ y \xrightarrow{o} \mho}$$

With these two intermediate ELTSs defined, we can prove the main lemma.

Lemma 4.1.12. *Suppose $x \sqsubseteq_\mho y$; then $x\ (\cong_\mho \fatsemi \preceq_\mho \fatsemi \cong_\mho)\ y$.*

Proof. Because $x \sqsubseteq_\mho y$, there exists a $\mho$-refinement R with $(x,y) \in R$. Construct Q_1 and Q_2 as above. Set $q_1 = (x,y)$ and set $q_2 = \mathsf{inl}\ (x,y)$. It is fairly straightforward to show that $x \cong_\mho q_1$, $q_1 \preceq_\mho q_2$ and $q_2 \cong_\mho y$ (proofs elided). □

Theorem 4.1.13. *The relation $\sqsubseteq_\mho$ is exactly $(\cong_\mho \fatsemi \preceq_\mho \fatsemi \cong_\mho)$.*

Corollary. *$\sqsubseteq_\mho$ is the smallest transitive relation containing $\cong_\mho$ and $\preceq_\mho$.*

This final corollary shows how $\sqsubseteq_\mho$ satisfes the fourth condition from §2.3. Any preorder finer than $\sqsubseteq_\mho$ must either distinguish some bisimilar states, or it must not allow some expansion of undefined behavior. Therefore, we cannot hope to find any smaller relation.

Throughout the rest of this thesis I will attempt, and mostly succeed, at showing corresponding results for the other notions of refinement we encounter.

4.1.1 Characteristic Logic for Strong Behavioral Refinement

As with strong bisimulation, we can characterize strong $\mho$-refinement using a simple modal logic containing two families of modalities $[o]$ and $\langle o \rangle$. Unlike standard Hennessy-Milner logic, the characteristic logic for $\mho$-refinement is an intuitionistic logic; I call this variant Intuitionistic Hennessy-Milner logic (IHM).

The data needed to specify this logic are:

- The worlds are the set of ELTS processes,
- the accessibility relation is $\sqsubseteq_\mho$, $\mho$-refinement,

- there are no atoms, and
- modes are as below.

$$\mathcal{M} := [o] \mid \langle o \rangle$$

$$[\![[o]]\!]_{\mathcal{M}}(P) \equiv \{w \mid \forall w'.\ w \xrightarrow[\text{may}]{o} w' \Rightarrow w' \in P\} \tag{4.12}$$

$$[\![\langle o \rangle]\!]_{\mathcal{M}}(P) \equiv \{w \mid \exists w'.\ w \xrightarrow[\text{must}]{o} w' \wedge w' \in P\} \tag{4.13}$$

Compared to standard HM logic, the main difference is the fact that $\sqsubseteq_{\mho}$ is asymmetric and that the clauses for $[o]$ and $\langle o \rangle$ use the two different notions of stepping in their definitions. In these definitions, the meaning of the may/must step relations is most clear; $[o]$ relates to safety properties and constrains what a program may do and $\langle o \rangle$ relates to progress properties that constrain what a program must (be able to) do.

It happens that $[o]$ could equivalently be defined with the following clause:

$$[\![[o]]\!]_{\mathcal{M}}(P) \equiv \{w \mid \forall w'\ w''.\ \ w \sqsubseteq_{\mho} w' \Rightarrow w' \xrightarrow[\text{must}]{o} w'' \Rightarrow w'' \in P\}$$

This definition provides an alternate explanation; $[o]P$ means after every o-transition P will hold, even when we consider all possible refinements of the current program. In chapter 5 I present a slightly different account of strong $\mho$-refinement that takes this definition. It is closer to the standard way to define intuitionistic modal logics.

Here, I present the may-stepping definition because it is the form that generalizes properly to the case with hidden transitions. In that setting, the two definitions are not equivalent and the may-stepping form is the one that has the nicer theory.

It is evident from the definitions that the interpretation of modes is monotone and preserves $\sqsubseteq_{\mho}$-closure. Soundness of the logic follows immediately.

Adequacy (for image-finite ELTSs) follows in a way substantially similar to the proof for classical HM. However, some additional cases arise because of the presence of $\mho$. To address them, we need the following results.

Proposition 4.1.14. *$\mho$ is the bottom of the refinement order. That is, $\mho \sqsubseteq_{\mho} y$ for arbitrary y.*

Proof. It suffices to show that the relation $R = \{(\mho, y) \mid y \in \mathcal{S}_Y\}$ is a $\mho$-refinement relation.

In the forward direction, we assume $\mho \xrightarrow[\text{must}]{o} x'$, an immediate contradiction. In the backward direction, it suffices to show there exists x' such that $\mho \xrightarrow[\text{may}]{o} x'$ and $(x', y') \in R$. Setting $x' = \mho$ satisfies both obligations. □

In other words $\mho$ is the worst program; any other program is better.

Proposition 4.1.15. *For every formula $f \in \mathcal{F}_{IHM}$:*

$$\mho \models f \qquad \textit{iff} \qquad f \textit{ is valid.}$$

Proof. If f is valid then it holds in every world, so the backward direction is easy. In the forward direction, assume $\mho \models f$. We must show $w \models f$ for arbitrary w. This follows from soundness of IHM and proposition 4.1.14. □

When interpreted from the logic side, we see that $\mho$ satisfies no interesting specifications, but only the tautologies. Thus, there is a very real sense in which $\mho$ gives us no information about what may happen when we run the program — absolutely any behavior at all is allowed.

Theorem 4.1.16 (Adequacy for IHM)**.** EM *Suppose a and b are finite-branching LTS processes where $a \models f$ implies $b \models f$ for all $f \in \mathcal{F}_{\text{IHM}}$. Then $a \sqsubseteq_{\mho} b$.*

Proof. This proof follows in a manner substantially similar to the proof for HM (cf. proposition 3.1.10). However, there are two additional cases. The first is when we have that $x \xrightarrow[\text{must}]{o} x'$ but $y = \mho$. However, this is a contradiction. Note that $x \models \langle o \rangle \top$ (because $x \xrightarrow[\text{must}]{o} x'$) which implies $y \models \langle o \rangle \top$; however $y = \mho$ so this cannot be. The second case is when we have that $y \xrightarrow[\text{may}]{o} y'$ and $x = \mho$, and we must show that $x \xrightarrow[\text{may}]{o} x'$ for some x' related to y'. We set $x' = \mho$ and the case is completed by appeal to proposition 4.1.14. □

Turning to the μ-calculus version of the logic, we can find a characteristic formula of IHM via a straightforward modification of the CF for HM.

Definition 4.1.17 (Characteristic formula for IHM)**.** *Suppose $a = \langle \mathcal{L}, z \rangle$ is an ELTS process. The characteristic formula for the ELTS $\mathcal{L}$ is:*

$$\begin{aligned}&\mathsf{CF}(\mathcal{L}) \equiv \nu M.\ \lambda x : \mathcal{S}.\\ &\qquad \left(\bigwedge_{o \in \mathcal{O}} \ \bigwedge_{\{x' \ \mid \ x \xrightarrow[must]{o} x'\}} \langle o \rangle M@x' \right) \ \wedge \ \left(\bigwedge_{o \in \mathcal{O}} [o] \bigvee_{\{x' \ \mid \ x \xrightarrow[may]{o} x'\}} M@x' \right)\end{aligned}$$

Then the characteristic formula for the process a is $\mathsf{CF}(a) \equiv \mathsf{CF}(\mathcal{L})@z$.

Again, the main proof obligation is to show that this definition reflects the relational definition of ℧-refinement. I omit the details.

Theorem 4.1.18 (Expressiveness for IHM)**.** *Suppose a and b are ELTS processes. Then:*

$$a \sqsubseteq_{\mho} b \qquad \text{iff} \qquad b \models \mathsf{CF}(a)$$

4.1.2 Program Safety

Now that we have enhanced the notion of programs so that it is possible to express programs that go wrong, it is possible to select out the class of safe programs, i.e., those programs that never go wrong.

It may seem perverse to make the effort to explicitly represent programs that might exhibit undefined behavior only to turn around and study the properties of those that are always defined. However, this is actually an entirely sensible thing to do. For example, one popular view of the function of type systems in programming languages is to conservatively approximate the set of safe programs. Under this view, we are frequently interested in the *soundness* of a type system, a property which means that every program accepted by the type system is safe. In order for an investigation of type systems under this view to be interesting, it must be at least conceivable for a program to go wrong — otherwise every type system would be trivially sound.

Here, I am not interested in questions about type systems per se; nonetheless, I think it is appropriate and interesting to investigate the properties of safe programs. This investigation will, I think, further serve to convince the reader that the abstraction of ℧-refinement properly accounts for the notion of undefined behavior.

The main result in this section is that the set of safe processes can be characterized in several equivalent ways: as the processes that never go wrong (℧ is unreachable); as the maximal elements of the $\sqsubseteq_{\mho}$ preorder; and as the processes that satisfy the axiom of the excluded middle in IHM.

First, I will define safety by a standard “cannot go wrong” definition that I will take as the ground truth.

Definition 4.1.19 (Reachability)**.** *Let $\mathcal{L}$ be an ELTS. Then lifted state x can reach x' in one step if $x \xrightarrow[\textit{must}]{o} x'$ for some o; in this case I write $x \longrightarrow x'$. Then $x \longrightarrow^* x'$ is the reflexive, transitive closure of one-step reachability. When $x \longrightarrow^* x'$, we say x' is reachable from x.*

Definition 4.1.20 (State safety)**.** *Let $\mathcal{L}$ be an ELTS. Then lifted state x is* safe *if every state* *reachable from x is non-$\mho$. In symbols:*

$$\mathsf{safe}\ x \qquad \equiv \qquad \forall x'.\ \ x \longrightarrow^* x' \Rightarrow x' \neq \mho$$

The notion of safety is lifted to ELTS processes in the obvious way.

The first theorem I will prove about safe processes is that they coincide with the maximal elements of the preorder $\sqsubseteq_\mho$. This should make perfect sense; $\sqsubseteq_\mho$ is supposed to capture the notion of improvement with respect to undefined behavior, and the safe processes are precisely those which cannot be improved any further.

Definition 4.1.21 ($\mho$-maximal processes)**.** *Suppose x is an ELTS process. We say x is maximal if, for all processes y, $x \sqsubseteq_\mho y$ implies $x \cong_\mho y$.*

Basically, x is maximal if every improvement of x is actually equal to x.

Proposition 4.1.22. *Suppose x is a safe ELTS process. Then x is $\mho$-maximal.*

Proof. We are given that x is safe, and that $x \sqsubseteq_\mho$ for some y; we must show that $x \cong_\mho y$. Because $x \sqsubseteq_\mho$, there exists a $\mho$-refinement R with $(x, y) \in R$. Set $R' = \{(x, y) \mid (x, y) \in R \wedge x \text{ is safe}\}$. Clearly $(x, y) \in R'$. Thus it suffices to show R' is a $\mho$-bisimulation. This follows from a straightforward manipulation of the definitions. □

We also have the converse; every maximal process is safe. This is basically because every process can be refined to some safe process. Thus every maximal process is bisimilar to a safe process, and this transports safety back to the original process. The following propositions formalize this argument.

Proposition 4.1.23. *Every ELTS process is refined by some safe ELTS process.*

Proof. We are given an ELTS $\mathcal{L}$ and a state x. We can "edit" $\mathcal{L}$ by modifying the transition relation to so that every arc going to $\mho$ instead turns into a self-loop. Call the new ELTS $\mathcal{L}'$. More explicitly, every transition in $\mathcal{L}$ where $x \xrightarrow{o} \mho$ becomes a transition $x \xrightarrow{o} x$ in $\mathcal{L}'$. It should be evident that every state in $\mathcal{L}'$ is safe. It is also not hard to see that the relation $R = \{(x, y) \mid x = \mho \vee x = y\}$ is a $\mho$-refinement between $\mathcal{L}$ and $\mathcal{L}'$. This is sufficient to show that the process $\langle \mathcal{L}, x\rangle$ is refined by $\langle \mathcal{L}', x\rangle$. □

Proposition 4.1.24. *Suppose x is a safe process and $x \sqsubseteq_\mho y$. Then y is a safe process.*

Proof. We are given that x is safe and there exists a $\mho$-refinement R with $(x, y) \in R$. We are also given that $y \longrightarrow^* y'$; we must show $y' \neq \mho$. The proof goes by induction on the number of steps from y to y'. The base case requires that $(x, y) \in R$ and x safe implies $y \neq \mho$; this is straightforward from the definition of $\mho$-refinement. The inductive case requires that when x is safe and $x \longrightarrow x'$, then x' is safe. This follows directly from the definition of safety. □

Proposition 4.1.25. *Suppose x is $\mho$-maximal. Then x is a safe ELTS process.*

Proof. By proposition 4.1.23, there exists a safe process $\hat{x}$ such that $x \sqsubseteq_\mho \hat{x}$. Because x is maximal, $x \cong_\mho \hat{x}$. $\cong_\mho$ is symmetric and implies $\sqsubseteq_\mho$, so $\hat{x} \sqsubseteq_\mho x$. $\hat{x}$ is safe, so by proposition 4.1.24, x is safe. □

Corollary. *An ELTS process x is safe iff it is $\mho$-maximal.*

The final, and perhaps most surprising, result is that the safe programs are precisely those that behave classically in the characteristic temporal logic IHM. There are two versions of this result — one for the propositional system and one for the μ-calculus system.

In both cases, we can rely on the following definition for what it means to behave classically.

Definition 4.1.26 (Classical process)**.** *A process x is* classical *if it satisfies every instance of the axiom of the excluded middle.*

$$\mathsf{classical}\ x \qquad \equiv \qquad \forall f.\ \ x \models (f \vee \neg f)$$

First, I show that safe processes are classical. This is essentially because the refinement order is symmetric on the safe processes.

Proposition 4.1.27. $^{\mathsf{EM}}$ *Suppose x is a safe process. Then x is classical.*

Proof. Consider an arbitrary formula f; either $x \models f$ or $x \not\models f$. In the first case, we are done. In the second case, we must show that $x \models \neg f$; that is, we must show that for all x' where $x \sqsubseteq_\mho x'$, $x' \not\models f$. Because x is safe, we know that $x \cong_\mho x'$ by proposition 4.1.22, which implies $x' \sqsubseteq_\mho x$. Now, suppose $x' \models f$. By soundness, it must be that $x \models f$, a contradiction. □

Going in the other direction, the first thing to notice is that classical processes cannot be $\mho$.

Proposition 4.1.28. *Suppose x is a classical process. Then $x \neq \mho$.*

Proof. Because x is classical, $x \models \langle o \rangle \top \vee \neg \langle o \rangle \top$ for arbitrary $o \in \mathcal{O}$. Recall that $\mathcal{O}$ must be nonempty, so we can find some such o. This gives two cases. If $x \models \langle o \rangle \top$, then $x \neq \mho$ because of the interpretation of $\langle o \rangle$. On the other hand suppose $x \models \neg \langle o \rangle \top$ and $x = \mho$ (otherwise we are done). Unwinding the interpretation of this formula, it says that every refinement of x must not have any o-transition (recall that $\neg f$ is an abbreviation for $f \Rightarrow \bot$, and that $\Rightarrow$ is defined by quantification over more refined processes). However because $x = \mho$, we can refine x into any process at all, including one with an o-transition, which yields a contradiction. □

Next we need to show that classical processes step to classical processes; that is, the property of being classical is preserved by stepping.

Proposition 4.1.29. $^{\mathsf{EM}}$ *Suppose x is a classical process and $x \xrightarrow[must]{o} x'$. Then x' is a classical process.*

Proof. Suppose x' is not classical for contradiction. Then there exists some f such that $x' \not\models f$ and $x' \not\models \neg f$. Now, because x is classical, either $x \models [o](f \vee \neg f) \vee \neg[o](f \vee \neg f)$. In the first case we get $x \models [o](f \vee \neg f)$, which gives a contradiction. Thus, assume $x \models \neg[o](f \vee \neg f)$. By proposition 4.1.23, there exists a safe process that refines x; call this process $\hat{x}$. Now $\hat{x}$ is safe, and all the processes reachable from $\hat{x}$ are safe. Thus, using proposition 4.1.27, we can show that $\hat{x} \models [o](f \vee \neg f)$. However, this contradicts $x \models \neg[o](f \vee \neg f)$ because $x \sqsubseteq_{\mho} \hat{x}$. □

Proposition 4.1.30. $^{\mathsf{EM}}$ *Suppose x is classical. Then x is a safe process.*

Proof. Assume $x \longrightarrow^{*} x'$; we must show $x' \neq \mho$. The proof goes by induction on the number of steps from x to x'. The base case uses proposition 4.1.28, and the inductive case uses proposition 4.1.29. □

Corollary. $^{\mathsf{EM}}$ *A process x is safe iff it is classical.*

For the μ-calculus version of the logic, we can even go one step further and show that safety is internalizable; that is, there exists a formula that a world satisfies iff it is safe.

Definition 4.1.31 (Safety formula)**.**

$$\mathsf{safety} \quad \equiv \quad \nu X.\ \bigwedge_{o \in \mathcal{O}} ([o]X) \wedge ([o]\bot \vee \langle o \rangle \top)$$

The key to this formula is the phrase $[o]\bot \vee \langle o \rangle \top$, which claims that transition o is either definintely impossible, or definintely possible. This formula holds on a process iff that process is non-$\mho$ — the $\mho$ process refuses to commit to offering action o or not, but every real state will do so. The greatest fixpoint going through $[o]$ ensures that the formula continues to hold on every reachable state.

Proposition 4.1.32.

$$x \text{ is safe} \qquad \textit{iff} \qquad x \models \mathsf{safety}$$

Proof. In the forward direction, we basically need to show that the property of being safe is preserved by stepping and that being safe implies $[o]\bot \vee \langle o \rangle \top$. Both are straightforward. In the other direction, we need to show that $x \models \mathsf{safety}$ is preserved by stepping and that $w \models [o]\bot \vee \langle o \rangle \top$ implies $x \neq \mho$. Again, both are straightforward. □

The safety formula gives us another way to prove proposition 4.1.30. If x is classical then $x \models \mathsf{safety}$ or $x \models \neg\mathsf{safety}$. In the first case, we are done. The second case gives a contradiction because every process can be refined into a safe process. In other words, $\neg\mathsf{safety}$ is unsatisfiable; thus a classical process must satisfy safety.

4.2 Branching Behavioral Refinement

Hopefully by now the reader is convinced that strong $\mho$-refinement properly captures the notion of improving programs with respect to undefined behavior. Our task now is to drag this notion into the setting with hidden transitions by analogy to the developments of §3.2.

The main insight that makes the generalization to hidden actions work is the idea that we transform the definition of branching bisimulation by using *must* stepping for the forward direction and *may* stepping for the backward direction. Once the may/must stepping setup is used, the results are almost anticlimactic. Indeed, this is the point — the definitions and proofs have been carefully arranged to highlight their similarity.

Essentially, we just take all the definitions relating to branching bisimulation and edit them to use must stepping for the forward directions and may stepping for the backward directions. The results from §3.2 carry over with very few changes. Likewise, the properties relating to undefined behavior we saw from earlier in this chapter carry over to this setting with only minor modifications.

In this section, I will elide the majority of the proofs; only where interesting new issues due to undefined behavior arise will I delve into details. Of course, full details are available in the mechanized proof development.

Definition 4.2.1 (Extended τ-Labeled Transition Systems)**.** *Suppose we have fixed in advance a nonempty set $\mathcal{O}$ of observable transitions. Let τ be a new distinguished symbol not in $\mathcal{O}$. Then an* Extended τ-Labeled Transition System *(or τELTS) is a tuple $\langle \mathcal{S}, \longrightarrow \rangle$ where $\mathcal{S}$ is a set of states and $\longrightarrow \subseteq \wp(\mathcal{S} \times \mathcal{O} \cup \{\tau\} \times \mathcal{S}^{\mho})$ is a transition relation.*

Definition 4.2.2 (May and must τ-paths)**.** *Suppose $\mathcal{L}$ is a τLTS with state space $\mathcal{S}$, and suppose $P \subseteq \mathcal{S}^{\mho}$ is some property of lifted states. We say there is a must τ-path from x to y along P*

whenever there is a finite sequence starting at x and ending in y such that each element of the sequence has a must τ-transition to the next, and each state in the sequence is in P. When this is the case we write $x \overset{\tau}{\underset{\textit{must}}{\Longrightarrow}}_P y$. We allow the empty path where $x = y$, indicating no actual steps are taken. Likewise $x \overset{\tau}{\underset{\textit{may}}{\Longrightarrow}}_P y$ refers to a path where each state is connected by a may step.

Definition 4.2.3 (Branching $\mho$-refinement)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are τELTSs. Suppose $R \subseteq \mathcal{S}_1^{\mho} \times \mathcal{S}_2^{\mho}$ is a binary relation on the states of $\mathcal{L}_1$ and $\mathcal{L}_2$. We say that R is a* branching $\mho$ refinement relation *if, whenever $(x, y) \in R$, the following two conditions hold:*

$$\begin{aligned} & x \xrightarrow[\textit{must}]{\alpha} x' \text{ implies either } \alpha = \tau \wedge (x', y) \in R, \text{ or} \\ & \qquad \exists y'\ y''.\ \ y \overset{\tau}{\underset{\textit{must}}{\Longrightarrow}}_{R(x,-)} y' \wedge y' \xrightarrow[\textit{must}]{\alpha} y'' \wedge (x, y') \in R \wedge (x', y'') \in R \end{aligned} \tag{4.14}$$

$$\begin{aligned} & y \xrightarrow[\textit{may}]{\alpha} y' \text{ implies either } \alpha = \tau \wedge (x, y') \in R, \text{ or} \\ & \qquad \exists x'\ x''.\ \ x \overset{\tau}{\underset{\textit{may}}{\Longrightarrow}}_{R(-,y)} x' \wedge x' \xrightarrow[\textit{may}]{\alpha} x'' \wedge (x', y) \in R \wedge (x'', y') \in R \end{aligned} \tag{4.15}$$

Here $R(x, -)$ refers to the set $\{y \mid (x, y) \in R\}$. Likewise, $R(-, y) \equiv \{x \mid (x, y) \in R\}$.

Just as with stepping, we need now to distinguish two sorts of divergences: a must-divergence is a divergence using only must-steps, and a may-divergence is a divergence using may-steps.

Definition 4.2.4 (Must and may divergences)**.** *Suppose $\mathcal{L}$ is some τELTS, and $D \subseteq \mathcal{S}^{\mho}$ is a set of states. Then D is a* must divergence *if, for each $x \in D$ there exists $x' \in D$ such that $x \xrightarrow[\textit{must}]{\tau} x'$. Likewise D is a* may divergence *if, for each $x \in D$ there exists $x' \in D$ such that $x \xrightarrow[\textit{may}]{\tau} x'$.*

Definition 4.2.5 (Divergence-sensitive branching $\mho$-refinements)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are τELTSs. Suppose $R \subseteq \mathcal{S}_1^{\mho} \times \mathcal{S}_2^{\mho}$ is a binary relation on the states of $\mathcal{L}_1$ and $\mathcal{L}_2$. We say that R is a* divergence-sensitive branching $\mho$-refinement relation *if R is a branching $\mho$-refinement and, in addition, satisfies the following for each $(x, y) \in R$:*

- *For each must divergence D in $\mathcal{L}_1$ where $x \in D$, there exists $x' \in D$ and y' such that $y \xrightarrow[\textit{must}]{\tau} y'$ and $(x', y') \in R$.*
- *For each may divergence D in $\mathcal{L}_2$ where $y \in D$, there exists $y' \in D$ and x' such that $x \xrightarrow[\textit{may}]{\tau} x'$ and $(x', y') \in R$.*

Definition 4.2.6 (Divergence-sensitive branching $\mho$-refinement)**.** *Suppose a and b are τELTS processes. Then we say a is* divergence-sensitive $\mho$-refined *by b, and write $a \sqsubseteq_{\mho\Delta B} b$ if there exists a divergence-sensitive branching $\mho$-refinement relation R such that $(a, b) \in R$.*

In the usual way, we can show that branching $\mho$-refinement is reflexive and transitive (for both the vanilla and the divergence-sensitive versions).

The one basic definition that does not generalize directly from the strong $\mho$-refinement case to the branching case is the definition of $\mho$-bisimulation. Recall from definition 4.1.4 that $\mho$-bisimulation requires, for each $(x, y) \in R$, $x = \mho \leftrightarrow y = \mho$. This requirement is too strong for the branching bisimulation case, where we wish to abstract away from time. If we require $x = \mho \leftrightarrow y = \mho$, then we distinguish a process which has already gone wrong from a program that will certainly go wrong after one more step, violating the time abstraction we wish to achieve.

Instead, we can take the characterization from proposition 4.1.9 as a definition — this notion is the one that generalizes properly. Thus, we say that a relation is a $\mho$-bisimulation iff it is a $\mho$-refinement in both directions.

Definition 4.2.7 (Branching $\mho$-bisimulation)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are τELTSs. Suppose $R \subseteq \mathcal{S}_1^{\mho} \times \mathcal{S}_2^{\mho}$ is a binary relation on the states of $\mathcal{L}_1$ and $\mathcal{L}_2$. We say that R is a* (divergence-sensitive) branching $\mho$-bisimulation relation *if R is a (divergence-sensitive) branching $\mho$-refinement and R^{-1} is a (divergence-sensitive) branching $\mho$-refinement.*

We write $x \cong_{\mho B} y$ when there exists a branching $\mho$-bisimulation relating x and y. We write $x \cong_{\mho \Delta B} y$ when there exist a divergence-sensitive branching $\mho$-bisimulation relating x and y.

In the divergence-sensitive case, branching $\mho$-bisimulation may be characterized in a way similar to strong $\mho$-bisimulation; that is, divergence-sensitive branching $\mho$-bisimulation is equivalent to the must-stepping diagrams in both directions together with a clause about undefined behavior.

This extra clause says that x goes wrong iff y goes wrong; here "goes wrong" means that every path from a state is finite, contains only τ-transitions and terminates in a $\mho$. In other words, x goes wrong if it is the root of a finite directed acyclic graph (of only τ arcs) where every sink state is a $\mho$. In the non-divergence-sensitive case, drop the condition that every path be finite — equivalently allow a general directed graph rather than a finite DAG. In either case, going wrong means the program will engage in no more "must" observable events and cannot safely halt.

The concept of inductive branching bisimulation generalized to this new setting simply by applying the recipe: forward directions use must-stepping and backward directions use may-stepping. The basic theory of inductive branching $\mho$-refinement proceed by proofs mostly analogous to the ones from §3.2, as does the proof of equivalence with divergence-sensitive branching $\mho$-refinement. The minor differences that do occur have to do with the handling of cases related to $\mho$; these cases are generally straightforward.

The stuttering closure operation also generalizes well, although it might not be entirely obvious how to write down the correct definition. Should we use may or must stepping in the definition? Oddly enough, it turns out not to matter! For states, $\underset{\text{must}}{\xrightarrow{\tau}}^{*}$ is the same relation as $\underset{\text{may}}{\xrightarrow{\tau}}^{*}$, even though $\underset{\text{must}}{\xrightarrow{\tau}}$ is not the same as $\underset{\text{may}}{\xrightarrow{\tau}}$. It turns out to be most convenient to state the definition using may stepping. Let $\underset{\text{may}}{\overset{\tau}{\Longrightarrow}}$ be the reflexive, transitive closure of $\underset{\text{may}}{\xrightarrow{\tau}}$.

Definition 4.2.8 (ELTS stuttering closure)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are to τELTSs. Then the* stuttering closure *is an operation on relations between $\mathcal{L}_1$ and $\mathcal{L}_2$, defined as follows:*

$$\begin{aligned}\mathsf{ST}(R) \equiv \big\{(x,y) \;\mid\; &\exists x_0\; x_1\; y_0\; y_1. \\ &x_0 \underset{\text{may}}{\overset{\tau}{\Longrightarrow}} x \wedge x \underset{\text{may}}{\overset{\tau}{\Longrightarrow}} x_1 \wedge y_0 \underset{\text{may}}{\overset{\tau}{\Longrightarrow}} y \wedge y \underset{\text{may}}{\overset{\tau}{\Longrightarrow}} y_1 \wedge (x_0,y_1) \in R \wedge (x_1,y_0) \in R\big\} \qquad (4.16)\end{aligned}$$

With this definition, we can show that $\sqsubseteq_{\mho B}$, $\sqsubseteq_{\mho\Delta B}$, $\cong_{\mho B}$ and $\cong_{\mho\Delta B}$ are all stuttering-closed.

As with strong $\mho$-refinement, we can show that branching $\mho$-refinement can be characterized by the intensional process of replacing $\mho$ states with arbitrary subgraphs. As before, I will give only a high-level overview of this proof — it goes in a similar manner to the proof for strong $\mho$-refinement.

Suppose we are given two ELTSs X and Y and a branching $\mho$-refinement relation R between X and Y. Then we can build a new ELTS, Q_1 with state space defined by the pairs of states in R; that is, the state space of Q_1 is the set $\{(x,y) \mid (x,y) \in R\}$ The step relation of Q_1 is defined by the following rules:

$$\frac{x \underset{\text{must}}{\xrightarrow{\alpha}} x' \qquad y \underset{\text{must}}{\xrightarrow{\alpha}} y'}{(x,y) \xrightarrow{\alpha} (x',y')} \qquad \frac{}{(\mho,y) \xrightarrow{\tau} \mho} \qquad \frac{x \underset{\text{must}}{\xrightarrow{\tau}} x'}{(x,y) \xrightarrow{\tau} (x',y)} \qquad \frac{y \underset{\text{must}}{\xrightarrow{\tau}} y'}{(x,y) \xrightarrow{\tau} (x,y')}$$

Now we construct a second ELTS, Q_2, to be a $\mho$-expansion of Q_1. Let the state space of Q_2 be the disjoint union of pairs in R and states in Y. The $\mho$-expansion mapping injects the pairs in R into the state space of Q_2; this function is obviously injective. The stepping relation of Q_2 is defined by the following rules:

$$\frac{x \underset{\text{must}}{\xrightarrow{\alpha}} x' \qquad y \underset{\text{must}}{\xrightarrow{\alpha}} y'}{\mathsf{inl}\ (x,y) \xrightarrow{\alpha} \mathsf{inl}\ (x',y')} \qquad \frac{x \underset{\text{must}}{\xrightarrow{\tau}} x'}{\mathsf{inl}\ (x,y) \xrightarrow{\tau} \mathsf{inl}\ (x',y)} \qquad \frac{y \underset{\text{must}}{\xrightarrow{\tau}} y'}{\mathsf{inl}\ (x,y) \xrightarrow{\tau} \mathsf{inl}\ (x,y')}$$

$$\frac{}{\mathsf{inl}\ (\mho,y) \xrightarrow{\tau} \mathsf{inr}\ y} \qquad \frac{y \underset{\text{must}}{\xrightarrow{\alpha}} y'}{\mathsf{inr}\ y \xrightarrow{\alpha} \mathsf{inr}\ y'} \qquad \frac{}{\mathsf{inr}\ \mho \xrightarrow{\tau} \mho}$$

Using these intermediate τELTSs we can prove the following.

Proposition 4.2.9.

$$x \sqsubseteq_{\mho B} y \quad \textit{iff} \quad x \; (\cong_{\mho B} \mathbin{⨾} \preceq_{\mho} \mathbin{⨾} \cong_{\mho B}) \; y$$

In this proposition $\preceq_{\mho}$ is exactly the same definition of $\mho$-expansion we used in the strong $\mho$-refinement case above (cf. definition 4.1.10).

Corollary. *$\sqsubseteq_{\mho B}$ is the smallest transitive relation containing $\cong_{\mho B}$ and $\preceq_{\mho}$.*

Although I suspect the corresponding fact is also true for the divergence-sensitive version, I have been unable to find a proof. The construction of Q_1 and Q_2 above may introduce additional divergences. This means we cannot use them directly in a proof for the divergence-sensitive versions, and it is not obvious how (or indeed if) these constructions can be modified to properly preserve divergence behavior. Thus I present the following conjecture as an item of future work.

Conjecture.

$$x \sqsubseteq_{\mho \Delta B} y \quad \textit{iff} \quad x \; (\cong_{\mho \Delta B} \mathbin{⨾} \preceq_{\mho} \mathbin{⨾} \cong_{\mho \Delta B}) \; y$$

4.3 Back-and-forth Behavioral Refinement

As with branching bisimulation, we need to take a detour through "back-and-forth" refinement to get an expressive characteristic logic. Our first task is to update the notion of process histories. There are a couple of possible design decisions about how to do this; however, all the reasonable variations essentially boil down to whether we want the states of a history to be connected by the must-step relation or the may-step relation. Both choices seem reasonable at first, but it turns out only if we choose the may-step setup does the theory work out.

Definition 4.3.1 (ELTS Process history)**.** *Suppose $\mathcal{L}$ is an ELTS and let n be a nonnegative integer. Let $x_0, \cdots, x_n$ be a (nonempty) sequence of lifted states (in $\mathcal{S}^{\mho}$, and let $\alpha_1, \cdots, \alpha_n$ be a (possibly empty) sequence of actions (in $\mathcal{O} \cup \{\tau\}$). Then the sequence $[x_0, \alpha_1, x_1, \alpha_2, \cdots, \alpha_n, x_n]$ is a* process history *if $x_i \xrightarrow[\mathsf{may}]{a_{i+1}} x_{i+1}$ for each $i < n$.*

As a diagram, we may write this condition as:

$$x_0 \xrightarrow[\mathsf{may}]{\alpha_1} x_1 \xrightarrow[\mathsf{may}]{\alpha_2} x_2 \cdots x_{n-1} \xrightarrow[\mathsf{may}]{\alpha_n} x_n$$

Let h be some history as defined above. I write $\mathsf{init}(h)$ to mean the initial state x_0, and I write $\mathsf{curr}(h)$ to mean the current state x_n.

We define the notions of may and must history extension, and weak may and must history extension by analogy to the corresponding definitions over LTS histories (cf. definition 3.3.1).

Now we are ready to define the back-and-forth version of $\mho$-refinement.

Definition 4.3.2 (Back-and-forth $\mho$ refinement)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are τELTSs. Let R be a binary relation on ELTS process histories from $\mathcal{L}_1$ and $\mathcal{L}_2$. Then R is a back-and-forth $\mho$-refinement if, for all $(g,h) \in R$, the following hold:*

$$\forall \alpha\ g'.\ \ g \overset{\alpha}{\underset{must}{\Longrightarrow}} g' \ \Rightarrow\ \exists h'.\ \ h \overset{\alpha}{\underset{must}{\Longrightarrow}} h' \wedge (g', h') \in R \tag{4.17}$$

$$\forall \alpha\ h'.\ \ h \overset{\alpha}{\underset{may}{\Longrightarrow}} h' \ \Rightarrow\ \exists g'.\ \ g \overset{\alpha}{\underset{may}{\Longrightarrow}} g' \wedge (g', h') \in R \tag{4.18}$$

$$\forall \alpha\ g'.\ \ g' \overset{\alpha}{\underset{may}{\Longrightarrow}} g \ \Rightarrow\ \exists h'.\ \ h' \overset{\alpha}{\underset{may}{\Longrightarrow}} h \wedge (g', h') \in R \tag{4.19}$$

$$\forall \alpha\ h'.\ \ h' \overset{\alpha}{\underset{may}{\Longrightarrow}} h \ \Rightarrow\ \exists g'.\ \ g' \overset{\alpha}{\underset{may}{\Longrightarrow}} g \wedge (g', h') \in R \tag{4.20}$$

We further say that R is a divergence-sensitive back-and-forth $\mho$-refinement relation if it also satisfies the following:

$$\begin{aligned}&\textit{For each history must-divergence } D \textit{ in } \mathcal{L}_1 \textit{ where } g \in D,\\ &\qquad \textit{there exists } g' \in D \textit{ and } h' \textit{ such that } h \xrightarrow[must]{\tau} h' \textit{ and } (g', h') \in R.\end{aligned} \tag{4.21}$$

$$\begin{aligned}&\textit{For each history may-divergence } D \textit{ in } \mathcal{L}_2 \textit{ where } h \in D,\\ &\qquad \textit{there exists } h' \in D \textit{ and } g' \textit{ such that } g \xrightarrow[may]{\tau} g' \textit{ and } (g', h') \in R.\end{aligned} \tag{4.22}$$

We write $g \sqsubseteq_{\mho BF} h$ when there exists a back-and-forth $\mho$-refinement where $(g,h) \in R$. Likewise, we write $g \sqsubseteq_{\mho \Delta BF} h$ when there exists a divergence-sensitive back-and-forth $\mho$-refinement where $(g,h) \in R$.

Possibly contrary to intuition, both of the backward-in-time clauses use the may-step relation. This is related to the fact that process histories have the may-step relation built-in. The divergence-sensitive version is the unsurprising generalization of divergence-sensitive back-and-forth bisimulation (cf. definition 3.3.4). The definition of related histories (definition 3.3.6) is essentially unchanged.

The main result relating back-and-forth $\mho$-refinement to branching $\mho$-refinement is the following, which is completely analogous to the result for branching bisimulation.

Theorem 4.3.3.

$$x \sqsubseteq_{\mho\Delta B} y \qquad \text{iff} \qquad (x, y) \in \mathsf{curr}(\sqsubseteq_{\mho\Delta BF}) \tag{4.23}$$

$$g \sqsubseteq_{\mho\Delta BF} h \qquad \text{iff} \qquad (g, h) \in \mathsf{rel}(\sqsubseteq_{\mho\Delta B}) \tag{4.24}$$

Next, I present the characteristic logic for divergence-sensitive back-and-forth $\mho$-refinement. The data needed to specify the logic are:

- The worlds are the ELTS process histories,
- the accessibility relation is $\sqsubseteq_{\mho\Delta BF}$,
- there are no atoms, and
- the modes are given as below.

$$\mathcal{M} := [\alpha] \mid \langle\alpha\rangle \mid \overset{\leftarrow}{[\alpha]} \mid \overset{\leftarrow}{\langle\alpha\rangle} \mid \Delta \mid \nabla$$

$$[\![[\alpha]]\!](P) \equiv \{h \mid \forall h'.\ (h \underset{\text{may}}{\overset{\alpha}{\Longrightarrow}} h') \Rightarrow h' \in P\} \tag{4.25}$$

$$[\![\langle\alpha\rangle]\!](P) \equiv \{h \mid \exists h'.\ (h \underset{\text{must}}{\overset{\alpha}{\Longrightarrow}} h') \wedge h' \in P\} \tag{4.26}$$

$$[\![\overset{\leftarrow}{[\alpha]}]\!](P) \equiv \{h \mid \forall h'.\ (h' \underset{\text{may}}{\overset{\alpha}{\Longrightarrow}} h) \Rightarrow h' \in P\} \tag{4.27}$$

$$[\![\overset{\leftarrow}{\langle\alpha\rangle}]\!](P) \equiv \{h \mid \exists h'.\ (h' \underset{\text{may}}{\overset{\alpha}{\Longrightarrow}} h) \wedge h' \in P\} \tag{4.28}$$

$$[\![\Delta]\!](P) \equiv \{h \mid \exists D. D \text{ is a must-divergence} \wedge h \in D \wedge (\forall h'.\ h' \in D \Rightarrow h' \in P)\} \tag{4.29}$$

$$[\![\nabla]\!](P) \equiv \{h \mid \forall D. D \text{ is a may-divergence} \Rightarrow h \in D \Rightarrow (\exists h'.\ h' \in D \wedge h' \in P)\} \tag{4.30}$$

As before, these modalities are directly related to the clauses in the definition of back-and-forth $\mho$-refinement. Each modality is monotone and preserves closure with respect to $\sqsubseteq_{\mho\Delta BF}$, ensuring soundness.

As usual, we achieve an adequacy result for the weakly image-finite ELTSs in the propositional system and an expressiveness result for the μ-calculus system.

Theorem 4.3.4 (Adequacy for $\mathcal{F}_{\mho\Delta BF}$). EM *Suppose g and h are weakly image-finite τELTS process histories where for all $f \in \mathcal{F}_{\mho\Delta BF}$, $g \models f$ implies $h \models f$. Then $g \sqsubseteq_{\mho\Delta BF} h$.*

Definition 4.3.5 (Characteristic formula for $\mathcal{T}_{\mho\Delta BF}$)**.** *For a given τELTS $\mathcal{L}$, the characteristic formula for $\mathcal{L}$ is:*

$$\begin{aligned}
\mathsf{CF}(\mathcal{L}) \equiv \nu X.\ \lambda h.& \\
&\left(\bigwedge_{\alpha} \bigwedge_{\{h' \mid h \overset{\alpha}{\underset{must}{\Longrightarrow}} h'\}} \langle\alpha\rangle X@h'\right) \wedge \left(\bigwedge_{\alpha} [\alpha] \bigvee_{\{h' \mid h \overset{\alpha}{\underset{may}{\Longrightarrow}} h'\}} X@h'\right) \wedge \\
&\left(\bigwedge_{\alpha} \bigwedge_{\{h' \mid h' \overset{\alpha}{\underset{may}{\Longrightarrow}} h\}} \overleftarrow{\langle\alpha\rangle}\, X@h'\right) \wedge \left(\bigwedge_{\alpha} \overleftarrow{[\alpha]} \bigvee_{\{h' \mid h' \overset{\alpha}{\underset{may}{\Longrightarrow}} h\}} X@h'\right) \wedge \\
&\left(\bigwedge_{\{D \mid D \text{ is a must-divergence} \wedge h \in D\}} \Delta\left(\bigvee_{\{h' \mid h' \in D\}} X@h'\right)\right) \wedge \\
&\left(\nabla\left(\bigvee_{\{h' \mid h \overset{\tau}{\underset{may}{\longrightarrow}} h'\}} X@h'\right)\right)
\end{aligned}$$

When h is a process history with τELTS $\mathcal{L}$, we define $\mathsf{CF}(h) \equiv \mathsf{CF}(\mathcal{L})@h$.

Theorem 4.3.6 (Expressivity for $\mathcal{T}_{\mho\Delta BF}$)**.** *For all process histories g and h,*

$$g \sqsubseteq_{\mho\Delta BF} h \qquad \text{iff} \qquad h \models \mathsf{CF}(g)$$

Both of these proofs follow the same pattern as the corresponding proofs for back-and-forth bisimulation. Note that if we drop the modalities Δ and ∇, corresponding characterization results follow for the divergence-insensitive version of back-and-forth refinement.

Safe histories. The theory of safe programs we developed for strong $\mho$-refinement also lifts in a nice way to the branching refinement case. We say an ELTS history is safe if every state appearing in the history is safe — this is equivalent to requiring that the initial state be safe. Safety of a state in a τELTS is exactly the same as for an ELTS; it means the $\mho$ state is unreachable.

With this definition made, we get entirely corresponding results. The safe processes (resp. histories) are precisely those maximal with respect to the $\sqsubseteq_{\mho\Delta B}$ refinement order (resp. $\sqsubseteq_{\mho\Delta BF}$ refinement order). Furthermore, the safe histories are precisely those that satisfy every instance of the excluded middle; this result holds for both the propositional and μ-calculus versions. All of these results hold for essentially the same reasons as before.

For the μ-calculus logic, we can also internalize the safety predicate as before. We need only a minor change to handle going backward in the history. This formula works for both the divergence-sensitive and the divergence-insensitive versions of the logic.

Definition 4.3.7 (Safety formula)**.**

$$\mathsf{safety} \quad \equiv \quad \nu X. \bigwedge_{\alpha \in \mathcal{O} \cup \{\tau\}} ([\alpha] X) \wedge (\overleftarrow{[\alpha]}\, X) \wedge ([\alpha]\bot \vee \langle\alpha\rangle\top)$$

Proposition 4.3.8.

$$h \textit{ is a safe history} \qquad \textit{iff} \qquad h \models \mathsf{safety}$$

4.4 Related Work

Partial bisimulations. There is a line of closely related work on "bisimulation preorders," or "partial bisimulations" pursued by Milner [Mil81], Stirling [Sti87], Walker [Wal90], and Abramsky [Abr91]. This work deals with *underspecified* processes, and the bisimulation preorders they consider allow programs to become more specified. Both technically and in spirit, this line work is similar in many ways to the present work, so here I shall take some space to clearly delineate where and how I believe this new work adds to the picture.

The common theme through all the above cited papers is the study of labeled transition systems that have been augmented with a distinguished collection of states that are identified as "divergent." The notation $x\!\uparrow$ is used to indicate that x "may diverge." The corresponding notation $x\!\downarrow$ is used to mean that it is *not* the case that $x\!\uparrow$, and is read, x "certainly converges." The following definition, due to Abramsky [Abr91, defn 2.2], is lifted directly, changing only the notation slightly.

Definition 4.4.1 (Partial bisimulation)**.** *Suppose $\mathcal{L}$ is a transition system. A binary relation R is* a partial bisimulation *if, for all $(x, y) \in R$:*

$$\forall o\ x'.\ \ x \xrightarrow{o} x' \Rightarrow \exists y'.\ y \xrightarrow{o} y' \wedge (x', y') \in R \tag{4.31}$$

$$x\!\downarrow \ \ \Rightarrow \ \ (y\!\downarrow \wedge\, (\forall o\ y'.\ \ y \xrightarrow{o} y' \Rightarrow \exists x'.\ x \xrightarrow{o} x' \wedge (x', y') \in R)) \tag{4.32}$$

It is not hard to see that ELTSs and strong $\mho$-refinement then arise as a special case of partial bisimulation. Given an ELTS, we can define an LTS with divergence by using the must-step relation and defining $x\!\uparrow$ to hold exactly when $x = \mho$. There is additionally a variation of Hennessy-Milner logic that characterizes these partial bisimulations. The key clauses are [Abr91]:

$$x \models \langle o \rangle P \equiv \exists x'.\ x \xrightarrow{o} x' \wedge x' \models P \tag{4.33}$$

$$x \models [o]P \equiv x{\downarrow} \wedge \forall x'.\ x \xrightarrow{o} x' \Rightarrow x' \models P \tag{4.34}$$

From these definitions, along with a standard logical skeleton, an adequacy results follows.

A reader disinclined to be generous might declare that the existence of this previous work critically undermines the novelty of the present work. Nonetheless, I feel the present work is worthwhile, and even novel, for the reasons I put forth below.

The major difference between my work and the previous work above is philosophical in nature. For Abramsky, Milner, Stirling and Walker, underspecification arises from a phenomenon they call "divergence." This, taken together with the surrounding contextual work, betrays a mindset that equates divergence (nontermination) with catastrophic failure (undefined behavior/going wrong).

In the concurrency calculi which are the main motivations for the above-mentioned authors' work, undefined behavior typically arises from certain kinds of unguarded recursion. For example, in Abramsky's treatment of partial bisimulations, he uses them to induce an ordered space in which he can do domain theory; the meaning of recursive equations are, as usual, given by the least fixpoint construction in the domain. That is, recursion is defined directly with respect to the "becoming more specified" order induced by partial bisimulation. The bottom element of these domains are interpreted as totally unspecified behavior, similar to $\mho$. Thus, a recursive "infinite loop" in these systems results in the phenomenon I call "going wrong."

One can see the same mechanism at work in Stirling's account. When he generalizes to the case with hidden τ-actions, those states which can engage in an infinite sequence of τ steps are defined to be "globally divergent," processes which are extremely underspecified. Walker's account [Wal90] is even more explicit. This is in sharp contrast to my own account of branching $\mho$-refinement, where a divergent process is a well-defined process that happens to have nonterminating behavior.

It is technically possible to generalize partial bisimulations to include τ-actions in a way that distinguishes between nontermination and going wrong. However, such a generalizaion is essentially precluded by the philosophical underpinnings of the work, underpinnings that are deeply embedded even into the choice of terminology. Certainly, I do not believe that such a generalization was in the mind of the authors of the above papers.

For our present investigations into compiler correctness, I am not of the opinion that identifying nontermination with going wrong is the most useful frame of mind. Instead, I prefer to regard nontermination (i.e., divergence) as a distinct concept from undefined behavior (i.e., going wrong). From a practical point of view, it seems clear that the designers of the C specification regard these concepts as distinct. I think most mainstream programming language researchers would also hold this view. From a theoretical point of view, as well, I think it is better to distinguish these notions. Equating nontermination with going wrong essentially means one is only interested in total correctness properties, whereas treating them as distinct allows one to address partial correctness as well. Naturally we expect compilers to preserve both partial and total correctness, so we should adopt a framework that allows us to do so.

There is also an important difference between my extension to HM logic and those of Abramsky, Stirling, etc. In their extensions, the modality $[o]$ implies the absence of divergence (undefined behavior), whereas in my system $[o]P$ may hold even on the $\mho$-state, provided P is tautological. Both of these extensions collapse into HM logic when there is no divergence (undefined behavior), but I think that the extension I define is more "natural" in that it breaks down very nicely along the may/must step organizational principle and comports more directly to the intuitive meaning of the modal operators. As I have shown, the may/must structure lifts in a completely general way to the setting with τ-actions, whereas the generalization done by Stirling [Sti87] feels more ad-hoc.

In summary, one can show that strong $\mho$-refinement arises as a special case of partial bisimulation. However, this is mostly an accident; in particular, when we examine generalizations of the two systems to include τ-actions, the resulting systems are quite different. There is, in particular, no interesting notion of a divergence-sensitive partial bisimulation preorder. Also, although the characteristic logics for both systems are intuitionistic generalizations of Hennessy-Milner logic, they are incompatible generalizations with rather different interpretations. In chapter 5, I shall examine the proof theory of the logic for strong $\mho$-refinement. This examination should help to highlight the logical differences between these systems.

Modal transition systems. Another closely related body of work has to do with a concept called *modal transition systems* (MTSs). Originally introduced by Larsen and Thomsen [LT88], modal transition systems are essentially a set of states equipped with two transition systems, a *must* transition and a *may* transition, where the must relation is included in the may relation.

The presentation I have adopted for this chapter makes the connection to modal transition systems clear — the ELTSs arise as a particular special case. Once this special case is identified, strong behavioral refinement arises as an instance of Larsen's specification refinements [Lar90].

Once again, the fact that ELTSs and $\mho$-refinement arise as special cases of a more general theory does not negate the value of studying the special case. As with partial bisimulations, the philosophical mindset underpinning my work is quite distinct from the underpinnings underlying MTSs. The work on MTSs has largely focused on the stepwise-refinement use case. Stepwise refinement is a development methodology whereby a system is developed by first building a high-level specification; this specification is then made more explicit (refined) by making some implementation choices. Multiple levels of internal specification layers may be passed through until a concrete implementation is reached.

The main idea here is that there is a *stack* of specifications of varying levels of detail, each written by developers. Automatic tools are used to verify that each step is a refinement of the previous one. These refinements are then put together transitively to show that the implementation is a refinement of the original specification. As such, much of the work on MTSs has revolved around computational apsects. One asks questions like: “does one MTS refine another?” and “given to MTSs, is there any implementation that satisfies both?” and one is interested in the computational complexity of these questions.

In contrast, I am interested in showing the correctness of program transformations. In such a setting, the computational questions above are largely irrelevant. One typically does not perform proof search to show refinements of particular programs, but instead the transformation is proved to produce refinements, once and for all.[3] In any case, the idea of using modal transition systems to model undefined behavior in the sense that I use it (i.e., going wrong) seems to be novel, as is the theory of safe programs. In addition, my extension of MTS refinement to the branching and back-and-forth versions is novel, as is my logical characterization of the same. Although Fischbein, Braberman and Uchitel study a generalization of MTS refinement to the branching case [FBU09], the definition they give is quite distinct from the one I give and it is not immediately clear if it defines the same relation. In any case Fischbein et al. do not consider diveregence-sensitivity, as I do, nor do they consider characteristic logics.

[3]However, one might consider translation validation a form of proof search, depending on the precise nature of the untrusted translation oracle and the certificate checking procedure.

Chapter 5

Proof Theory of Behavioral Refinement

Note to the reader. This chapter is essentially an extended aside. It is largely self-contained and may be read independently of the remainder of this thesis. There is some duplication of definitions that appear in chapter 2, and some definitions are given in a different form than from chapter 4. Unlike the rest of this thesis, I will freely use the axiom of the excluded middle in the proofs appearing in this chapter.

5.1 Introduction

In this chapter, I will examine the proof theory of the logic for strong behavioral refinement. This examination should give us a better feeling for how the modalities behave as logical operators. Hopefully it will also convince the reader that the characteristic logic for strong behavioral refinement is a reasonable logic in its own right.

I begin by defining the strong behavioral refinement.[1]

Definition 5.1.1 (Behavioral refinement)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are two ELTSs over $\mathcal{O}$, and let R be a two-place relation on the states of $\mathcal{L}_1$ and $\mathcal{L}_2$. We say R is a refinement relation if, whenever $(x, y) \in R$, both of the following hold.*

1. *For each $x \xrightarrow{o} x'$ there exists y' such that*
   - *$y \xrightarrow{o} y'$ and*

[1] This definition is the same as the "alternate" definition from chapter 4. (cf. definition 4.1.7)

- *if $x' \neq \mho$ then $y' \neq \mho$ and $(x', y') \in R$.*

2. *For each $y \xrightarrow{o} y'$ there exists x' such that*
   - *$x \xrightarrow{o} x'$ and*
   - *if $x' \neq \mho$ then $y' \neq \mho$ and $(x', y') \in R$.*

We say that y refines x (or that x is refined by y) and write $x \sqsubseteq y$ if there exists an R which is a refinement relation and $(x, y) \in R$.

Note that the definition of refinement relations is asymmetric; in both directions we include the identical phrase: if $x' \neq \mho$ then $y' \neq \mho$ and $(x', y') \in R$. In both directions, if the program being refined (state x) steps to $\mho$, no requirements are made about the future behavior of the refined program (state y'). Instead we only require that *if* x steps to some $x' \neq \mho$, then y' is also not $\mho$ and the new states are again in the refinement relation. This rather minor difference is what allows us to consider $\mho$ as a placeholder for arbitrary behavior.

5.2 A Characteristic Logic

Here again, I will present a characteristic logic for behavioral refinement. It too differs slightly from chapter 4, but only in presentation.

I call this logic intuitionistic Hennessy-Milner logic (IHM). The formulae of IHM are built from the basic propositional connectives and two families of modalities:

$$f := \bot \mid \top \mid f \wedge f \mid f \vee f \mid f \Rightarrow f \mid [o]f \mid \langle o \rangle f$$

We wish to view these formulae as program specifications; this leads to a Kripke possible-worlds semantics where the worlds are program states.

Definition 5.2.1 (Worlds, refinement and stepping)**.** *Let $\mathcal{L}$ be an ELTS, and let x be a lifted state from $\mathcal{L}$. Then the pair $\langle \mathcal{L}, x \rangle$ is called a* world*.*

Let $w_1 = \langle \mathcal{L}_1, x \rangle$ and $w_2 = \langle \mathcal{L}_2, y \rangle$. We say w_1 is refined by w_2 (and write $w_1 \sqsubseteq w_2$) if either $x = \mho$ or if $x \neq \mho$ and $y \neq \mho$ and $x \sqsubseteq y$. We sometimes abuse notation to write $x \sqsubseteq y$ instead of $\langle \mathcal{L}_1, x \rangle \sqsubseteq \langle \mathcal{L}_2, y \rangle$ when $\mathcal{L}_1$ and $\mathcal{L}_2$ can be inferred from context.

We write $w \overset{o}{\Longrightarrow} w'$, and say w o-steps to w', when $w = \langle \mathcal{L}, x \rangle$ and $w' = \langle \mathcal{L}, x' \rangle$ for some $\mathcal{L}$, x and x', where $x \neq \mho$ and $x \xrightarrow{o} x'$. Note in particular that $w \overset{o}{\Longrightarrow} w'$ means that w does not contain $\mho$.

We desire that, whenever a world w_1 satisfies some specification, every w_2 which refines it also satisfies that specification. Thus we assign semantics to each formula that is closed with respect to the refinement preorder $\sqsubseteq$. We follow the basic recipe for intuitionistic modal logics laid out by Alex Simpson in his dissertation [Sim94].

The semantics of formulae are defined inductively by the following clauses. We write $w \models f$ to mean that the world w satisfies a formula (specification) f.

$w \models \top$	always
$w \models \bot$	never
$w \models f_1 \wedge f_2$	if $w \models f_1$ and $w \models f_2$
$w \models f_1 \vee f_2$	if $w \models f_1$ or $w \models f_2$
$w \models f_1 \Rightarrow f_2$	if, for all w' where $w \sqsubseteq w', w' \models f_1$ implies $w' \models f_2$
$w \models \langle o \rangle f$	if there exists w' where $w \overset{o}{\Longrightarrow} w'$ and $w' \models f$
$w \models [o] f$	if, for all w' w'' where $w \sqsubseteq w'$ and $w' \overset{o}{\Longrightarrow} w'', w'' \models f$

Note that the clauses for $\Rightarrow$ and $[o]$ explicitly quantify over more refined worlds. These quantifications give the semantics its intuitionistic character.

Proposition 5.2.2 (Refinement closure)**.** *If $w \sqsubseteq w'$ and $w \models f$, then $w' \models f$.*

Proof. The proof goes by induction on f. The only interesting case is for $\langle o \rangle$, where we must make use the forward direction of refinement. □

Our goal is to demonstrate that the formulae of IHM have the same distinguishing power as strong refinement, a property called *adequacy* by Pnueli [Pnu85]. As is typically the case, we must limit ourselves to the image-finite transition systems because we have only finite conjunctions. This restriction can be lifted by moving to a logic with infinite conjunction/disjunction [HS85].

In fact, we prove something slightly stronger than adequacy, because it turns out we do not require implication to distinguish distinct ELTS states.

Definition 5.2.3 (Image-finite)**.** *Let $\mathcal{L}$ be an ELTS. We say $\mathcal{L}$ is* image-finite *if, for each state x and observation o, the set $\{x' \mid x \xrightarrow{o} x'\}$ is finite.*

Definition 5.2.4 (Specification refinement)**.** *Let w_1 and w_2 be worlds. We say that w_2 is a specification refinement of w_1, (and write $w_1 \leq w_2$) if, for all implication-free formulae f, $w_1 \models f$ implies $w_2 \models f$.*

Proposition 5.2.5 (Adequacy)**.** *Suppose w_1 and w_2 are worlds containing image-finite transition systems and that $w_1 \leq w_2$. Then, $w_1 \sqsubseteq w_2$.*

It suffices to show that $\leq$ is a behavioral refinement. The details of this proof closely follow the proof for classical Hennessy-Milner logic [HM80] and we will not repeat them here.[2] The cases dealing with $\mho$, however deserve some mention. These cases rely on the following proposition which shows that the property of being defined is internalized by a formula.

Proposition 5.2.6 (Definiteness)**.** *$x \neq \mho$ iff $\langle \mathcal{L}, x \rangle \models [o]\bot \vee \langle o \rangle \top$.*

Proof. If x is non-$\mho$, then it satisfies either $[o]\bot$ or $\langle o \rangle \top$, depending on whether it has an o-transition or not. In the other direction, if x satisfies $\langle o \rangle \top$, it must be non-$\mho$ by the definition of $\langle o \rangle$. If x satisfies $[o]\bot$ instead, then it must also be non-$\mho$. Suppose otherwise; then x refines any world, including one containing an o-transition. From this we can get a contradiction via the semantics for $\bot$. □

5.3 A Hilbert System

In this section, we will develop an axiomatic presentation of IHM that is sound and *nearly* complete for the Kripke semantics presented above. This serves to help us understand the rules of reasoning appropraite for IHM; but it also serves to show that IHM, and the behavioral refinement is characterizes, are well-behaved systems with a nice theory. In other words, behavioral refinement is not just a kludge, and the proof theory of IHM helps us to undersdand what species of logic IHM is.

The axiomization we present is a modification of (multimodal) IK, the minimal normal intuitionistic modal logic examined by Simpson [Sim94] and Servi [Ser84]. I call this system $\mathcal{H}_{\text{IHM}}$. The rules of inference and axioms for $\mathcal{H}_{\text{IHM}}$ are found in figure 5.1. Note that IHM does not contain propositional variables, so the variables appearing in the axioms must be understood as propositional metavariables, and the axioms themselves must be properly understood as axiom schemata. Furthermore, we intend that "axiom" 0 include all substitution instances of propositional tautologies; that is, we explicitly intend that axiom 0 to allow formulae containing modalities. One could equivalently include any standard listing of axioms for propositional intuitionistic logic.

The inference rules and axioms IK0-IK5 of $\mathcal{H}_{\text{IHM}}$ are exactly the same as Simpson's IK, whereas axiom IK6 and axioms D1 and D2 are new. We have included a star by IK5 to indicate that this axiom has bit of a special status. It is sound for the models of IHM, but is not used in the

[2] Interested readers may consult the mechanized proof scripts.

$\boxed{\vdash_{\mathcal{H}} P}$

Inference rules:

$$\frac{\vdash_{\mathcal{H}} (P \Rightarrow Q) \quad \vdash_{\mathcal{H}} P}{\vdash_{\mathcal{H}} Q}$$

$$\frac{\vdash_{\mathcal{H}} P}{\vdash_{\mathcal{H}} [o]P}$$

$$\frac{P \text{ is an axiom}}{\vdash_{\mathcal{H}} P}$$

Axioms:

IK0	Any intuitionistic tautology
IK1	$[o](P \Rightarrow Q) \Rightarrow ([o]P \Rightarrow [o]Q)$
IK2	$[o](P \Rightarrow Q) \Rightarrow (\langle o\rangle P \Rightarrow \langle o\rangle Q)$
IK3	$\langle o\rangle\bot \Rightarrow \bot$
IK4	$\langle o\rangle(P \vee Q) \Rightarrow \langle o\rangle P \vee \langle o\rangle Q$
IK5*	$(\langle o\rangle P \Rightarrow [o]Q) \Rightarrow [o](P \Rightarrow Q)$
IK6	$[o](P \vee Q) \Rightarrow [o]P \vee [o]Q \vee (\langle o\rangle P \wedge \langle o\rangle Q)$
D1	$([o_1]\bot \vee \langle o_1\rangle\top) \Rightarrow ([o_2]\bot \vee \langle o_2\rangle\top)$
D2	$[o]P \Rightarrow P \vee [o]\bot \vee \langle o\rangle\top$

Figure 5.1: The Hilbert system $\mathcal{H}_{\text{IHM}}$

completeness proof we present in this section, so its inclusion in the list of axioms is somewhat provisional. We shall return to this issue in more detail later.

Without much difficulty, one can show that all the axioms of $\mathcal{H}_{\text{IHM}}$ are theorems of classical K; thus it makes sense to call it an intuitionistic variant of K. It also includes the axioms of IK, so $\mathcal{H}_{\text{IHM}}$ lies between IK and K in proving power. In fact, it is not hard to show that it lies strictly between. $\mathcal{H}_{\text{IHM}}$ shares with IK the property that adding the axiom of the excluded middle causes the logic to have the same proving power as K.[3]

It is routine to verify that all the axioms and inference rules of $\mathcal{H}_{\text{IHM}}$ are sound with respect to IHM models, which gives soundness of the overall system.

Definition 5.3.1 (Validity)**.** *We say a formula f is* valid *if it holds in all worlds. That is, if $w \models f$ holds for all w.*

Proposition 5.3.2 ($\mathcal{H}_{\text{IHM}}$ soundness)**.** *If $\vdash_{\mathcal{H}} f$ then f is valid.*

Discussion of the axioms. Before diving into the completeness proof, it is worth taking a moment to examine the axioms and to understand what role they play in the overall system. IK1 together with the rule of necessity show that $\mathcal{H}_{\text{IHM}}$ is a normal modal logic. Among other things, this implies

[3]Note that this property relies on the troublesome axiom IK5.

that $[o]$ is a box-type modality which commutes with conjunction:

$$\begin{array}{rcl}[o]\top & \Leftrightarrow & \top \\ [o](P \wedge Q) & \Leftrightarrow & [o]P \wedge [o]Q\end{array}$$

Axiom IK2 defines the one half of the connection between $[o]$ and $\langle o \rangle$; it allows one to transport facts from under a $[o]$ to under a $\langle o \rangle$. Axioms IK1 and IK2 with the necessitation rule allows one to prove the following admissible rules:

$$\frac{\vdash_{\mathcal{H}} P \Rightarrow Q}{\vdash_{\mathcal{H}} [o]P \Rightarrow [o]Q} \qquad \frac{\vdash_{\mathcal{H}} P \Rightarrow Q}{\vdash_{\mathcal{H}} \langle o \rangle P \Rightarrow \langle o \rangle Q}$$

Axioms IK3 and IK4 simply establish that $\langle o \rangle$ commutes with disjunction in the way expected of a diamond-style modality. They prove facts dual to the ones above for $[o]$:

$$\begin{array}{rcl}\langle o \rangle\bot & \Leftrightarrow & \bot \\ \langle o \rangle(P \vee Q) & \Leftrightarrow & \langle o \rangle P \vee \langle o \rangle Q\end{array}$$

Axiom IK5 provides another connection between $[o]$ and $\langle o \rangle$. However, in the opinion of the author, it is the least intuitive of the axioms of IK and the most difficult to motivate philosophically. It does, however, have the following important consequence:

$$\langle o \rangle P \Rightarrow \bot \qquad \Leftrightarrow \qquad [o](P \Rightarrow \bot)$$

This is critical for showing that IK and $\mathcal{H}_{\mathrm{IHM}}$ collapse into K in the presence of the axiom of the excluded middle.

Axiom IK6 lets one decompose a disjunction appearing under $[o]$. Roughly, one can read IK6 as requiring that there be "enough" worlds to serve as counterexamples when $[o]$ statements fail. If $[o](P \vee Q)$ holds but neither $[o]P$ nor $[o]Q$, there must be some *reason* each one fails; that is, some worlds reachable via o-transitions that fail to satisfy P and Q. The world falsifying P must satisfy Q and vice-versa, hence $\langle o \rangle P \wedge \langle o \rangle Q$.

The axioms D1 and D2 are called the axioms of *definitiveness* because they logically characterize the properties of ELTSs having to do with the $\mho$ pseudostate; that is, they discuss the properties of being defined. Recall from the previous section that the formula $[o]\bot \vee \langle o \rangle\top$ holds on a world iff the world is non-$\mho$. The axiom D1 essentially says that being defined is a property independent of observable o. It makes no sense to be defined with respect to one observation but undefined with respect to another. This is because being defined is a property of states, not of transitions. Axiom D2 also tells us something about the nature of definitiveness, but the story it tells is a bit more subtle. If the formula $[o]P$ holds on some world, it must either be because the world is well-defined and all the reachable states from here satisfy P *or*, if the world contains $\mho$, it must be that P is a tautology. Axiom D2 is essentially an axiomatic way of capturing this property.

5.4 Completeness for $\mathcal{H}_{\text{IHM}}$

The completeness proof for IHM involves building a canonical model, in the style of Henkin. Our completeness proof is similar in many respects to the completeness of IK [Ser84], and involves a very similar construction using prime sets. The main novelty in our construction is the additional requirement that the canonical sets be *definite*.

Definition 5.4.1 (Prime/definite sets)**.** *Let G be a set of formulae. We say that G is* deductively closed *if, for every x and y such that $\vdash_{\mathcal{H}} x \Rightarrow y$ if $x \in G$, then $y \in G$. We say G is* disjunctive *if, whenever $x \vee y \in G$, either $x \in G$ or $y \in G$. G is* consistent *if $\perp \notin G$.*

Then, we say G is a prime set *if G is deductively closed, disjunctive and consistent. In addition, we require all the formulae in G to be implication-free.*

Finally, we say G is definite *if it includes the formula $[o]\perp \vee \langle o\rangle\top$ for each $o \in \mathcal{O}$.*

Definition 5.4.2 (Canonical model for IHM)**.** *The states of the canonical ELTS for IHM are the definite prime sets.*

Suppose G is a lifted *definite prime set. Then $G^\bullet$ maps G into a set of formulae as follows*

$$G^\bullet = \begin{cases} \{P \mid \vdash_{\mathcal{H}} P \text{ and } P \text{ implication-free } \} & \text{when } G = \mho \\ G & \text{when } G \neq \mho \end{cases}$$

Thus $G^\bullet$ contains just G if $G \neq \mho$, but it contains the implication-free theorems of $\mathcal{H}_{\text{IHM}}$ otherwise. Then we define $G_1 \xrightarrow{o} G_2$ to hold exactly when the following are true:

$$\text{for all } P, [o]P \in G_1 \text{ implies } P \in G_2^\bullet$$
$$\text{for all } P, P \in G_2^\bullet \text{ implies } \langle o\rangle P \in G_1$$

Before we proceed to the heart of the proof, we will need two technical lemmas. The first allows us to construct prime sets.

Definition 5.4.3 (Avoidance)**.** *Suppose X and Y are sets of formulae. We say that X* avoids *Y if, for every $X' \subseteq X$ and $Y' \subseteq Y$ with X' and Y' finite, $\bigwedge X' \Rightarrow \bigvee Y'$ is* not *a theorem of $\mathcal{H}_{\text{IHM}}$.*

Sometimes we say that X avoids a formula F. This simply means that $\bigwedge X' \Rightarrow F$ fails to be a theorem of $\mathcal{H}_{\text{IHM}}$ for all finite $X' \subseteq X$.

Proposition 5.4.4 (Prime Lemma)**.** *Suppose X and Y are sets of implication-free formulae. If X avoids Y then there exists a prime set G, such that $X \subseteq G$ and G avoids Y.*

Proof. Note that the deductively closed sets containing X and avoiding Y form a chain-complete poset. We can define an inflationary function which adjoins to such a set the smallest formula not already in the set which still avoids Y. By the Bourbaki-Witt theorem [Bou49, Wit51] this function has a fixpoint, which will be the desired prime set. □

The prime lemma lets us know that there are "enough" prime sets; given a set X we want to contain and a set Y we want to avoid, there is always some prime set in between.

For those interested in foundational matters, we should note that the prime lemma relies on the fact that the set of observations $\mathcal{O}$ is well-ordered. One may either invoke the axiom of choice to produce a well-ordering for arbitrary $\mathcal{O}$, or one may restrict the completeness theorem to sets $\mathcal{O}$ which are provably well-ordered (e.g., countable sets).

The next lemma makes use of the axioms of definitiveness to show that "most" of the prime sets are definite. Indeed, there is exactly one indefinite prime set; it contains just the implication-free theorems. This corresponds to the situation with ELTSs, where every "real" state is definite and the unique $\mho$ pseudostate satisfies only the valid statements.

Proposition 5.4.5 (Definite prime sets)**.** *Suppose G is a prime, implication-free set. Then either G is definite, or G is exactly the set of implication-free theorems of $\mathcal{H}_{\text{IHM}}$.*

Proof. Consider whether G contains any nontheorems. If G contains only theorems, it must nonetheless contain all the implication-free theorems because it is deductively closed. On the other hand, suppose G contains some nontheorem f. We can show that G is definite by induction on f. The case for $\top$ results in a contradiction because f is a nontheorem. Likewise $\bot$ results in a contradiction because G is prime and thus consistent. The cases for $\wedge$ and $\vee$ proceed by straightforward appeal to the induction hypothesis. In case $f = \langle o \rangle P$, we can apply some straightforward logical reasoning to learn that $[o]\bot \vee \langle o \rangle \top \in G$. From here, we apply axiom D1 to show that G is definite. In case $f = [o]P$, we apply axiom D2 to learn that $P \vee [o]\bot \vee \langle o \rangle \top \in G$. Because G is prime, we know that either $P \in G$ or $[o]\bot \vee \langle o \rangle \top \in G$.

In the first case, we can appeal to the induction hypothesis. In the second case, we appeal to axiom D1 to finish the proof. We know the case $f = P \Rightarrow Q$ cannot occur because G contains only implication-free formulae. $\square$

Note that this lemma is the source of the requirement for axioms D1 and D2 as well as the restriction to implication-free formulae.

Proposition 5.4.6 (Canonical model property). *Let f be an implication-free formula and G be a lifted, definite prime set. Then*

$$f \in G^{\bullet} \text{ iff } G \models f$$

Proof. The theorem proceeds by induction on f. The cases for $\bot$ and $\top$ are obvious. The cases for $\wedge$ and $\vee$ follow via applications of the induction hypothesis. Only the cases for $\langle o \rangle$ and $[o]$ are interesting. The case for $\langle o \rangle$ is nearly identical to the corresponding case for IK and we omit the details (cf. [Ser84]).

The forward direction for the $[o]$ case is also a straightforward application of the induction hypothesis. In the backward case, we are given that $G \models [o]P$ and we wish to prove that $[o]P \in G^{\bullet}$. Suppose $G = \mho$; then by the definition of refinement, we know that $G \sqsubseteq w$ for all worlds w. By constructing an appropriate w, we can use the fact that $G \models [o]P$ to show that $G \models P$. By the induction hypothesis $P \in G\bullet$; because $G = \mho$, P is a theorem, which implies $[o]P$ is a theorem, which means $[o]P \in G\bullet$. Now, suppose instead that $G \neq \mho$. Then we know that G is a definite prime set and we must have either $[o]\bot \in G$ or $\langle o \rangle \top \in G$. In the first case, the goal follows directly from axiom IK1. Thus suppose $\langle o \rangle \top \in G$. For the purposes of contradiction, also assume $[o]P \notin G$.

Let $X = \{F \mid [o]F \in G\}$ and let $Y = \{P\} \cup \{F \mid G \text{ avoids } \langle o \rangle F\}$. If we suppose that X avoids Y, then we can apply the prime lemma to construct a prime G' that contains X and avoids Y. If G' is definite, it is easy to show that $G \xrightarrow{o} G'$. Because $G \models [o]P$, it then follows that $G' \models P$. By the induction hypothesis, $P \in G'$. However, this is a contradiction because G' avoids Y, which contains P. If G' is not definite, then $G \xrightarrow{o} \mho$, and $\mho \models P$. By the induction hypothesis, $P \in \mho^{\bullet}$, and thus P is a theorem. But this, too, is a contradiction because if P is a theorem then $[o]P$ is a theorem and must therefore be in G.

All that remains is to show that the set X avoids Y. This involves some straightforward reasoning in the logic. The key step in this reasoning uses IK6 to decompose a disjunction appearing under $[o]$. $\square$

The completeness of implication-free theorems is an immediate corollary.

Proposition 5.4.7 (Implication-free completeness)**.** *Let f be an implication-free formula. If f is valid, then $\vdash_{\mathcal{H}} f$.*

Proof. Suppose f is valid. Then $\mho \models f$. By the canonical model property, $f \in \mho^{\bullet}$. By the definition of $\mho^{\bullet}$, we know f is a theorem. □

Unfortunately, this proposition is rather less interesting than it might appear; without implications, theorems of $\mathcal{H}_{\text{IHM}}$ consist of little more than the Boolean tautologies.

Improving the completeness result. Fortunately, we can improve the completeness result to a more interesting class of statements. Although we do not know if the completeness result can be extended to arbitrary uses of implications, we can at least handle top-level uses of implications. In other words, we can get a completeness result for sequents of implication-free statements.

Proposition 5.4.8 (Completness for $\mathcal{H}$ sequents)**.** *Let P and Q be implication-free formulae. If $P \Rightarrow Q$ is valid, then $\vdash_{\mathcal{H}} P \Rightarrow Q$.*

Proof. Suppose $P \Rightarrow Q$ is valid, but $P \Rightarrow Q$ is a nontheorem. Then clearly it is the case that $X = \{P\}$ avoids $Y = \{Q\}$. By the prime lemma, there exists a prime set G containing X which avoids Y. If G contains only theorems, then P is a theorem. By soundness P is valid. Because we assumed $P \Rightarrow Q$ is valid, Q is also valid. By the completeness of IHM theorems, Q must be a theorem, which trivially shows that $P \Rightarrow Q$ is a theorem; but this contradicts our assumption. Suppose then that G is a definite prime set; clearly $P \in G$. By the canonical model property, we have that $G \models P$. By the validity of $P \Rightarrow Q$, we know that $G \models Q$. Applying the canonical model property once more, we can show that $Q \in G$. However, this contradicts the fact that G avoids $Y = \{Q\}$. □

Characterizing canonical refinement. It turns out that refinement in the canonical model can be directly characterized as subset inclusion. This theorem is essentially an instance of adequacy, but here we can prove it directly, without requiring image-finiteness.

Proposition 5.4.9 (Subset refinement)**.** *Suppose G_1 and G_2 are implication-free, definite prime sets. Then $G_1 \subseteq G_2$ iff $G_1 \sqsubseteq G_2$.*

Proof. Consider first the backward direction. We assume $G_1 \sqsubseteq G_2$ and $P \in G_1$ for some P. We wish to show $P \in G_2$. By the canonical model property, $G_1 \models P$. The semantics of formulae are refinement-closed; thus $G_2 \models P$. A second application of the canonical model property yields $P \in G_2$, as desired.

In the other direction, it suffices to show that $\subseteq$ is a refinement relation. Then we have two proof obligations; one for the forward direction of refinement, and one for the backward direction. In the first case, we are given $G_1 \subseteq G_2$ and $G_1 \xrightarrow{o} G'_1$ and must find a G'_2 such that $G_1'^{\bullet} \subseteq G_2'^{\bullet}$ and $G_2 \xrightarrow{o} G'_2$. To do this, we employ the prime lemma with $X = G'_1 \cup \{F \mid [o]F \in G_2\}$ and $Y = \{F \mid G_2 \text{ avoids } \langle o \rangle F\}$. The reasoning which shows that X avoids Y is similar to the $\langle o \rangle$ case of the canonical model lemma and we elide the details here. For the backward direction of refinement we are given $G_1 \subseteq G_2$ and $G_2 \xrightarrow{o} G'_2$; we must find a G'_1 so that $G_1'^{\bullet} \subseteq G_2'^{\bullet}$ and and $G_1 \xrightarrow{o} G'_1$. Once again, we employ the prime lemma, this time with $X = \{F \mid [o]F \in G_1\}$ and $Y = \{F \mid G'_2 \text{ avoids } F \text{ or } G_1 \text{ avoids } \langle o \rangle F\}$. As before, we elide the details. However, note that IK6 once more plays a key role, allowing us to decompose a disjunction appearing under a $[o]$. □

5.5 A Sequent Calculus

The situation in which we find ourselves is a bit unsatisfactory. We have a completeness proof which extends only to a fragment of the logic we set out to examine. We are mollified, however, by the knowledge that implications are superfluous in the IHM models; recall that the adequacy result needed only the implication-free formulae to distinguish all distinct ELTS states. Thus, we should feel justified in the desire to expunge implication from our account and to start over from the beginning with a logic of implication-free formulae.

Obviously, if we wish to examine the proof theory of such a logic, Hilbert systems are not the correct vehicle. Without implications, we cannot employ *modus ponens*, the workhorse of the Hilbert system. Here we present an alternate account of IHM based on a Gentzen-style sequent calculus. Formulae are defined just as for the Hilbert system, except that $\Rightarrow$ is eliminated.

$$f := \bot \mid \top \mid f \wedge f \mid f \vee f \mid [o]f \mid \langle o \rangle f$$

The model theory for these formulae is defined exactly as in §5.2, except the clauses relating to $\Rightarrow$ are dropped.

The rules for the sequent system $\mathcal{G}_{\text{IHM}}$ are presented in figure 5.2. We write $\Gamma \vdash Q$ to mean that the hypotheses in Γ are sufficient to prove the conclusion Q. Γ represents a finite sequence of

$\boxed{\Gamma \vdash Q}$

$$\frac{\Gamma \vdash P \qquad P, \Gamma \vdash Q}{\Gamma \vdash Q}\ C$$

$$\frac{\Gamma' \vdash Q \quad \Gamma' \subseteq \Gamma}{\Gamma \vdash Q}\ W$$

$$\frac{}{\Gamma \vdash \top}\ \top R$$

$$\frac{}{\bot, \Gamma \vdash Q}\ \bot L$$

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}\ \wedge R$$

$$\frac{A, B, \Gamma \vdash Q}{A \wedge B, \Gamma \vdash Q}\ \wedge L$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q}\ \vee R1$$

$$\frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q}\ \vee R2$$

$$\frac{A, \Gamma \vdash Q \qquad B, \Gamma \vdash Q}{A \vee B, \Gamma \vdash Q}\ \vee L$$

$$\frac{\Gamma \vdash Q}{[o]\Gamma \vdash [o]Q}\ \Box R$$

$$\frac{P, \Gamma \vdash Q}{\langle o \rangle P, [o]\Gamma \vdash \langle o \rangle Q}\ \Diamond R$$

$$\frac{}{\langle o \rangle \bot, \Gamma \vdash Q}\ \Diamond \bot L$$

$$\frac{\langle o \rangle A, \Gamma \vdash Q \qquad \langle o \rangle B, \Gamma \vdash Q}{\langle o \rangle (A \vee B), \Gamma \vdash Q}\ \Diamond \vee L$$

$$\frac{\begin{array}{l}[o]A, \Gamma \vdash Q \qquad [o]B, \Gamma \vdash Q \\ \langle o \rangle A, \langle o \rangle B, \Gamma \vdash Q\end{array}}{[o](A \vee B), \Gamma \vdash Q}\ \Box \vee L$$

$$\frac{[o_2]\bot, \Gamma \vdash Q \qquad \langle o_2 \rangle \top, \Gamma \vdash Q}{[o_1]\bot, \Gamma \vdash Q}\ DL1$$

$$\frac{[o_2]\bot, \Gamma \vdash Q \qquad \langle o_2 \rangle \top, \Gamma \vdash Q}{\langle o_1 \rangle \top, \Gamma \vdash Q}\ DL2$$

$$\frac{\begin{array}{l}[o]\bot, \Gamma \vdash Q \qquad \langle o \rangle \top, \Gamma \vdash Q \\ P, \Gamma \vdash Q\end{array}}{[o]P, \Gamma \vdash Q}\ DL3$$

Figure 5.2: The sequent calculus $\mathcal{G}_{\text{IHM}}$

formulae. When we write $\Gamma' \subseteq \Gamma$, we mean that every formula in Γ' appears somewhere in Γ. In this way, the weakening rule simultaneously allows weakening, contraction and permutation of the context. When we write $[o]\Gamma$, we mean that the modality $[o]$ is applied to every formula in Γ. Thus, if $\Gamma = P, Q, R$ then $[o]\Gamma = [o]P, [o]Q, [o]R$.

All the rules in the left column of figure 5.2 are just the usual rules for the propositional intuitionistic sequent calculus (leaving out the rules for $\Rightarrow$). The remaining rules correspond fairly directly to axioms and rules of the Hilbert system. The rule $\Box R$ is the standard sequent calculus rule for the minimal modal logic K; it corresponds to axiom IK1 and the rule of necessitation in the Hilbert system. Rule $\Diamond R$ corresponds to axiom IK2. Likewise, the rules $\Diamond \bot L$, $\Diamond \vee L$ and $\Box \vee L$ correspond to axioms IK3, IK4 and IK6, respectively. DL1 and DL2 correspond to axiom D1, and DL3 corresponds to axiom D2. Alone of the axioms of $\mathcal{H}_{\text{IHM}}$, IK5 cannot be stated using a single top-level implication; it therefore cannot be expressed in the sequent system.

Soundness of the calculus is shown by a straightforward induction on the structure of derivations.

Definition 5.5.1 (Valid sequent)**.** *Let Γ be a finite sequence of formulae and Q a formula. We say $\Gamma \Vdash Q$ is a* valid sequent *if, for all worlds w where $w \models P$ for all $P \in \Gamma$, $w \models Q$.*

Proposition 5.5.2 (Sequent soundness). *Suppose* $\Gamma \vdash Q$. *Then* $\Gamma \Vdash Q$ *is a valid sequent.*

Furthermore, the $\mathcal{G}_{\mathrm{IHM}}$ is complete with respect to ELTS models. The proof is essentially the same as the proof for $\mathcal{H}_{\mathrm{IHM}}$; we build a canonical model, prove the canonical model property, and then exploit it to show that a derivation can be found for any valid sequent. The only real difference is that the internal reasoning logic is a sequent calculus rather than a Hilbert system. We shall not belabor the details here, but merely assert the final proposition.

Proposition 5.5.3 (Sequent completeness). *Suppose* $\Gamma \Vdash Q$ *is a valid sequent. Then* $\Gamma \vdash Q$.

Both logics we examined are sound and complete with respect to ELTS models on the fragment they have in common. As a corollary we obtain the fact that the sequent calculus and the Hilbert system prove the same sequents.

Proposition 5.5.4 (Sequent equivalence). $\Gamma \vdash Q$ *iff* $\vdash_{\mathcal{H}} (\bigwedge \Gamma) \Rightarrow Q$.

Proof-theoretic concerns. The sequent system $\mathcal{G}_{\mathrm{IHM}}$ is little more than a recasting of $\mathcal{H}_{\mathrm{IHM}}$, right down to the nearly one-to-one mapping between axioms and rules. Although this suffices for our present aim (completeness with respect to IHM models), the resulting system is rather crude from a proof-theoretic perspective.

In particular, it does not appear that $\mathcal{G}_{\mathrm{IHM}}$ enjoys cut elimination. The rules $\Diamond\bot L$, $\Diamond \vee L$, $\Box \vee L$, and $DL1-3$ do not harmonize well with their corresponding right rules. For example, consider the following simple derivation:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\bot \vdash \bot}\,\bot L}{\top, \bot \vdash \bot}\,W}{\langle o\rangle\top, [o]\bot \vdash \langle o\rangle\bot}\,\Diamond R \qquad \cfrac{}{\langle o\rangle\bot, \langle o\rangle\top, [o]\bot \vdash \bot}\,\Diamond\bot L}{\langle o\rangle\top, [o]\bot \vdash \bot}\,C$$

It does not appear that this derivation can be reduced to one without cut. Obviously, it would be preferable to find a system with better proof-theoretic properties. I leave this as a question for future work.

5.6 Related Work

Stirling [Sti87] presented a "partial bisimulation" preorder which is similar in many ways to our own refinement relation. In it, undefined behavior may be refined into arbitrary, more defined behavior.

In addition, he presents an intuitionistic variant of Hennessy-Milner logic which is adequate for partial bisimulation in the same sense as proposition 5.2.5. Milner [Mil81] and Walker [Wal90] considered similar preorders.

However, in their work, "undefined" behavior arises (only) from divergence, i.e., nontermination. Then, these preorders are best understood from a total correctness perspective: nontermination is identified with erroneous (wrong) behavior. Our work instead takes a *partial correctness* view in that we distinguish nontermination from going wrong. We feel this is a more appropriate viewpoint for our compiler correctness concerns. One generally does *not* expect that a compiler may turn a nonterminating program into a program with arbitrary behavior.

Stirling also presents a sound and complete Hilbert system, which defines an intuitionistic variant of Hennessy-Milner logic. His logic differs from ours in several respects. Most importantly, Stirling's box operator implies the absence of nontermination; this is in contrast to our system where it is the diamond operator which implies the absence of wrong behavior. As a consequence of this decision, Stirling's logic does not satisfy the rule of necessitation; thus it is not a normal modal logic. Instead, it is merely a regular modal logic.

In addition, Stirling's logic satisfies the following axiom: $[o](P \vee Q) \Rightarrow \langle o \rangle P \vee [o]Q$. This axiom is similar to, but strictly stronger than, axiom IK6. Unlike IK6, this axiom is not valid over ELTS models (neither is it valid over the birelation models of IK).

Finally, Stirling's logic is complete even with implications, whereas we currently do not know if our completeness result can be extended to cover unrestricted uses of implications.

Dunn investigated the proof and model theory of positive modal logic, the fragment of usual modal logic without implication or negation [Dun95]. The model theory Dunn investigates is the usual classical one. Dunn's proof system resembles our sequent system in some ways, and it includes rules essentially the same as $\Box R$ and $\Diamond R$. Dunn's system also includes a rule that corresponds to the axiom above from Stirling's system; as we have noted, such an axiom/rule fails to hold in ELTS models.

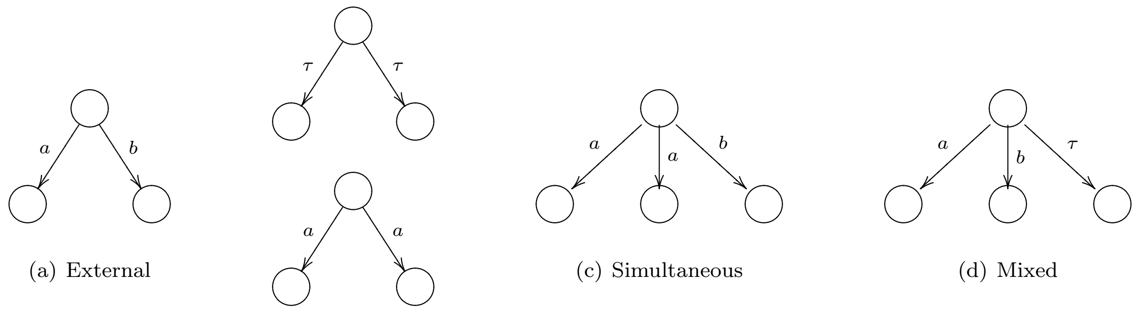
Simpson, in his thesis, presents a cut-free sequent calculus for intuitionistic modal logic [Sim94]. Simpson's presentation is rather more complicated than $\mathcal{G}_{\mathrm{IHM}}$. He attaches to each formula a label, and maintains as part of the sequent a graph that, roughly, describes the nesting structure of modalities. As a consequence, he can achieve true left and right rules for $\Diamond$ and $\Box$ rather than the mixed rules we have here, and he does not require special rules to decompose formulae "inside" modalities. Perhaps the system he presents could be modified to have the same proving power as our system, but we have not worked out the details.

Chapter 6

Choice Refinement

In this chapter, I will investigate a mode of refinement that allows program to be become more deterministic. Essentially, it allows us to resolve internal nondeterministic choices early, at compile time. This will allow us to properly treat phenomena defined to be "unspecified" by the C specification. I call the resulting refinement relation *choice refinement.* This generalization of bisimulation is orthogonal to the behavioral refinement of chapter 4, so we will use the nonextended LTSs and τLTSs throughout this chapter. Chapter 7 covers the combination of these two notions of refinement.

To understand exactly what it means to resolve internal choices, we need to understand what constitutes "internal" nondeterminism. In a labeled transition system with τ-actions, there arise four sorts of nondeterminism: external, internal, simultaneous, and mixed nondeterminism. External nondeterminism represents a choice offered to the environment, and it occurs when a state has at least two exiting arcs with distinct observable labels. A common pattern of external nondeterminism occurs on input actions. For example, inputting a character is nicely modeled by a state with a different arc for each possible character the user/environment might select. In contrast, internal nondeterminism occurs when the process itself makes a choice absent stimulus from the environment. This occurs in an LTS when a state has at least two exiting arcs with the *same* label, which may be either an observable or the special τ label. It may happen that a state exhibits both external and internal nondeterminism simultaneously. This occurs when a state has a number of observable arcs, some of which are distinct and some of which are the same. Such a state represents an atomic choice point where the environment chooses which label to follow, and the program chooses which arc with that label to follow. Internal, external, and simultaneous nondeterminism can all arise in LTSs without τ-arcs.



(a) External

(b) Internal

(c) Simultaneous

(d) Mixed

Figure 6.1: Various sorts of nondeterministic choice

The the fourth sort of nondeterminism, which can only arise in a τLTS, is what I call *mixed* nondeterminism. Mixed nondeterminism occurs when a state has both outgoing observable and τ-labeled arcs. The meaning of mixed nondeterminism is rather strange; it cannot be understood as any combination of internal and external choices. Basically, the program offers to the environment a series of observable actions. The environment can choose to select one of the offered actions, or it may refrain from doing so. If the environment refrains, the program will (after an unspecified time) instead choose one of its τ-actions to take. If the environment selects an action, the program may then choose to undertake that action by following one of its labeled arcs *or* the program may choose instead to refuse that action by following one of its τ-arcs instead.

The meaning of mixed nondeterminism is sufficiently weird that it is generally avoided. Mixed nondeterminism can arise from hiding operators in some process calculi; however, in such cases I believe that the mixed nondeterminism arises as an unintended (and mostly undesirable) consequence rather than as a deliberate design decision. When a τLTS has no mixed nondeterminism, we say it is τ-pure.

Definition 6.0.1 (τ-pure)**.** *Let $\mathcal{L}$ be a τLTS. A state x is τ-pure if it does* not *have both an outgoing τ-action and an outgoing observable action. We say $\mathcal{L}$ is τ-pure if every state in $\mathcal{L}$ is τ-pure. In other words, $\mathcal{L}$ is τ-pure if, whenever $x \xrightarrow{o} x'$, there does not exist x'' where $x \xrightarrow{\tau} x''$.*

It is tempting to restrict our attention to just the τ-pure processes. This would allow some results later in this chapter to go through without additional qualification. However, I have resisted doing so here because most of the results herein go through without the restriction to τ-purity, and because it is not clear that such a restriction is completely justified. There is at least one legitamate use for mixed nondeterminism that I can imagine, which is to model asynchronus events like POSIX signals or hardware interrupts.

For C, our primary setting of interest, all the core language features can be modeled using only internal or external nondeterminism. In particular, the unspecified evaluation order of C expressions is neatly modeled using internal choices, and interaction with the environment is handed using external choices. Simultaneous and mixed nondeterminism are not required. Even if we consider race-free multithreaded C code, I believe only internal and external choices are required. With race-free code, we can model concurrent execution as interleaving — the choice of which thread to run next is modeled as an internal choice. The one exception where I believe mixed nondeterminism might be useful is, as above, for modeling asynchronous events. It is not immediately clear to me if mixed nondeterminism is required or helpful for reasoning about racy programs with weak memory models.

In this chapter, we will be shifting views from the *possibilistic* view of nondeterminism represented by bisimulations to an *underspecification* view of nondeterminism. In the possibilistic view, the mere possibility of behaving in some way is considered observable and must be preserved by program transformations. This observation of possibilities is directly represented by the $\langle o \rangle$ modality, which corresponds to the forward direction of bisimulation. Roughly, this means that all significant[1] internal nondeterminism must be retained.

In contrast, the underspecification view of nondeterminism says that mere possibility is not observable, and program transformations are allowed to resolve the underspecification by making internal choices "early." According to this view, only those behaviors that are inevitable (rather than those merely possible) can be observed.

The difference between the possibilistic and underspecification views can also be understood as the maximal positions on a spectrum of possible fairness assumptions. The possibilistic view corresponds to a maximal fairness assumption; that is, we view the program as *cooperating* with the environment when making its internal choices to satisfy as many progress properties as possible. In contrast, the underspecification view represents a maximal *un*fairness assumption, in which we regard the internal choices as being made by an adversary. In this view, we only accept those progress properties which cannot be thwarted by an adversary's choices.

Although we do not usually think of the compiler as an adversary, the adversarial underspecification view is nonetheless appropriate for our setting. We wish to be robust to all possible compiler implementation strategies; this allows compiler writers maximal freedom to produce well-performing code so long as they follow the contract provided by the language specification.

[1] However, it might be that *apparent* internal nondeterminism may be removed by bisimulation, if the choices are indistinguishable. In other words, a choice between indistinguishable options is equal to no choice at all.

As in chapters 3 and 4, I shall first examine a strong version and then move on to a version with τ-actions.

6.1 Strong Choice Refinement

The notion of strong choice refinement I will discuss here has previously been studied in work by Bloom [Blo89], where it was called "ready simulation," and by Larsen and Skou [LS91], where it was called "$\frac{2}{3}$-bisimulation." Rather than adopting one of these names, I will instead conform to the naming scheme I have been using and call this notion "strong choice refinement." The main reason for adopting a new name is to avoid confusion; I will present a version of refinement in the opposite direction as Bloom (or Larsen and Skou). In Bloom's ready simulation, going up in the preorder involves becoming *more* nondeterministic. I prefer to orient the relation the other way, so going up in the order instead becomes *less* nondeterministic. For the same reason, the characteristic logic I will present later will be dual to the logic presented by Bloom. Although flipping the orientation of a preorder is a rather minor difference technically, I think it indicates a significant philosophical difference. Certainly it makes a large difference when considering the sorts of program properties that are preserved by transformations.

In accordance with the discussion in the previous section, strong choice refinement is a coarser relation than strong bisimulation; specifically, it weakens the "forward" direction of bisimulation and thereby preserves fewer progress properties.

Definition 6.1.1 (Strong choice refinement)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are two LTSs; let R be a binary relation between their state spaces. Let $x \xrightarrow{o} -$ be shorthand for $\exists x', x \xrightarrow{o} x'$. Then R is a* strong choice refinement *if the following hold for each $(x, y) \in R$:*

$$x \xrightarrow{o} - \;\Rightarrow\; y \xrightarrow{o} - \tag{6.1}$$

$$y \xrightarrow{o} y' \;\Rightarrow\; \exists x'.\; x \xrightarrow{o} x' \wedge (x', y') \in R \tag{6.2}$$

We write $x \sqsubseteq_C y$ when there exists a strong choice refinement R with $(x, y) \in R$.

As usual, $\sqsubseteq_C$ is reflexive, transitive and the greatest fixpoint of a monotone operator. Compared to strong bisimulation, choice refinement retains the backward direction of refinement, but weakens the forward direction. The forward direction requires only that the availability of an observable action is preserved; it says nothing about the subsequent behavior after the action is taken. Combined

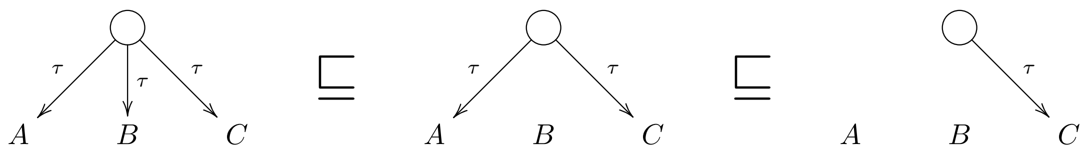


Figure 6.2: Pruning example

with the backward simulation, it essentially requires that related states offer the same actions to the environment.

Bloom argues that this relation is a natural one because it coincides with congruence for a large class of process languages, the GSOS languages [Blo89], which generalize Milner's CCS [Mil80]. Although Bloom's argument is an interesting data point in favor of ready simulation, it does not directly support my position that it properly captures the notion of reducing internal nondeterminism.

Instead, I shall prove that strong choice refinement is generated by an overly-intensional process, much as I did for strong ℧-refinement previously. This process, called *pruning*, attempts to capture as directly as possible the intuitive notion of removing internal choices.

Suppose you are given a node in an LTS with several arcs with label a; you are allowed to "prune" any of the arcs you like, so long as you leave at least one. You may choose to leave all the arcs (not resolving the internal choice at all), or remove some of them (partially resolving the choice) or to remove all but one of them (completely resolving the choice). It should be obvious that the process of choosing which arcs to discard corresponds nicely with the idea of reducing internal nondeterminism. Now imagine that we perform this process of removing arcs across an entire LTS at once — this will involving selecting a *subgraph* of the original graph, subject to the rule that each state in the subgraph must retain at least one arc of each label present in the original. See figure 6.2 for an example. The definition below formalizes this notion.

Definition 6.1.2 (LTS pruning)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are two LTSs and let $f : \mathcal{S}_2 \to \mathcal{S}_1$ be a function from the state space of $\mathcal{S}_2$ to $\mathcal{S}_1$. In the following, let y and y' range over states in $\mathcal{S}_2$. We say f is a pruning function if the following hold:*

$$f \text{ is injective} \tag{6.3}$$

$$y \xrightarrow{o} y' \;\Rightarrow\; f(y) \xrightarrow{o} f(y') \tag{6.4}$$

$$f(y) \xrightarrow{o} - \;\Rightarrow\; y \xrightarrow{o} - \tag{6.5}$$

Suppose x and y are LTS processes. We say y is a pruning of x if there exists a pruning function f between the LTSs of x and y such that $x = f(y)$. In this case we write $x \preceq_C y$.

Note that, opposite to $\mho$-expansions, the function in a pruning goes from $\mathcal{S}_2$ to $\mathcal{S}_1$.

The first two clauses of this definition basically require that $\mathcal{L}_2$ is a subgraph of $\mathcal{L}_1$ (up to renaming of the nodes). The third clause requires that at least one arc of each label is retained for each state that is selected in the subgraph. Recall that in LTSs, internal nondeterminism occurs precisely when there is more than one arc with the same label exiting a state; thus $\preceq_C$ captures the notion of resolving (some of) the internal nondeterminism in a program.

As with $\mho$-expansions, our goal is to show that pruning, considered up to bisimulation, is exactly the same relation as strong choice refinement. The following propositions are straightforward.

Proposition 6.1.3. *Suppose $x \cong y$. Then $x \sqsubseteq_C y$.*

Proof. It is easy to show that every strong bisimulation relation is also a strong choice refinement relation. □

Proposition 6.1.4. *Suppose $x \preceq_C y$. Then $x \sqsubseteq_C y$.*

Proof. It suffices to show that the relation $R \equiv \{(x, y) \mid x = f(y)\}$ is a choice refinement when f is a pruning function. □

Proposition 6.1.5. *Suppose $x\ (\cong \mathbin{⨾} \preceq_C \mathbin{⨾} \cong)\ y$. Then $x \sqsubseteq_C y$.*

Proof. Direct from the transitivity of $\sqsubseteq_C$ and the previous two propositions. □

The interesting direction is the converse, which shows that every choice refinement can be decomposed into a pruning. Just as with $\mho$-expansion, the proof strategy is to use the refinement relation itself as a template for building new LTSs with the required properties. Here I will give the constructions of these intermediate LTSs, but elide the proofs.

Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are two LTSs, and let R be a strong choice refinement between them. Q_1 is the first intermediate LTS, with state space $\mathcal{S}_1 \times \mathcal{S}_2$; that is, pairs of states from $\mathcal{L}_1$ and $\mathcal{L}_2$. The step relation for Q_1 is defined the the single rule below:

$$\frac{x \xrightarrow{o} x'}{(x, y) \xrightarrow{o} (x', y')}$$

Note that this rule completely ignores the y component of the pair. Thus, Q_1 has a rather large nondeterministic choice at each step — the state y' is chosen entirely without restriction every time Q_1 steps. Nonetheless every pair (x, y) in Q_1 is bisimilar to state x in $\mathcal{L}_1$ because the observable behavior of Q_1 is unaffected by the extra state y.

The next LTS, Q_2 restricts the state space to only those pairs $(x, y) \in R$. The step relation for Q_2 is defined by the following rule:

$$\frac{x \xrightarrow{o} x' \qquad y \xrightarrow{o} y'}{(x, y) \xrightarrow{o} (x', y')}$$

Thus, Q_2 will only step when *both* of its components step. It should be clear that Q_2 is a subgraph of Q_1. It is slightly less clear, but also true, that the Q_2 follows the pruning rule — we need to invoke the fact that every state in Q_2 is a pair in R. We can also show that every state (x, y) in Q_2 is bisimilar to state y in $\mathcal{L}_2$. These facts together allow us to prove the desired result.

Proposition 6.1.6. *Suppose $x \sqsubseteq_C y$. Then x $(\cong \fatsemi \preceq_C \fatsemi \cong)$ y.*

Proof. Because $x \sqsubseteq_C y$, there exists a choice refinement relation R with $(x, y) \in R$. Use the constructions of Q_1 and Q_2 above to complete the proof. □

Theorem 6.1.7. *For all LTS processes x and y:*

$$x \sqsubseteq_C y \qquad \textit{iff} \qquad x\ (\cong \fatsemi \preceq_C \fatsemi \cong)\ y.$$

Thus, we have a very tight connection between pruning and strong choice refinement. Hopefully, the reader can believe that pruning very closely matches the intuitive notion of resolving internal nondeterministic choices. If so, then this result should convince the reader that strong choice refinement is the right way to make pruning extensional. In particular, choice refinement is the *smallest* transitive extension of bisimulation that permits all prunings.

Corollary. *$\sqsubseteq_C$ is the smallest transitive relation containing both $\cong$ and $\preceq_C$.*

Next, I move on to the characteristic logic for strong choice refinement. This logic is similar, but dual, to Bloom's denial logic, which characterizes ready simulation [Blo89]. Denial logic has the modality $\langle o \rangle$ and atomic propositions $\mathsf{can't}(o)$, which means that the state has no o-transitions. Because strong choice refinement is the inverse of ready simulation, my logic instead has the duals: $[o]$ and atomic propositions $\mathsf{offers}(o)$, which means that some o-transition is possible.

Formally, the data inducing the characteristic logic are:

- The worlds are the LTS processes,
- the accessibility relation is strong choice refinement $\sqsubseteq_C$,
- the modes are $[o]$ for each $o \in \mathcal{O}$, and

- the atoms are $\mathsf{offers}(o)$ for each $o \in \mathcal{O}$.

$$[\![[o]]\!]_{\mathcal{M}}(P) \equiv \{w \mid \forall w'.\ w \xrightarrow{o} w' \ \Rightarrow\ w' \in P\} \tag{6.6}$$

$$[\![\mathsf{offers}(o)]\!] \equiv \{w \mid w \xrightarrow{o} -\} \tag{6.7}$$

It should be clear that the interpretations above are monotone and $\sqsubseteq_C$-closed. Soundness of the logic follows. The results below are a standard application of the techniques we have already seen in chapters 3 and 4.

Theorem 6.1.8 (Adequacy for $\mathcal{F}_C$)**.** $^{\mathsf{EM}}$ *Suppose x and y are image-finite LTS processes where $x \models f$ implies $y \models f$ for each $f \in \mathcal{F}_C$. Then $x \sqsubseteq_C y$.*

Definition 6.1.9 (Characteristic formula for $\mathcal{T}_C$)**.** *Suppose $a = \langle \mathcal{L}, z\rangle$ is an LTS process. The characteristic formula for the LTS $\mathcal{L}$ is:*

$$\mathsf{CF}(\mathcal{L}) \equiv \nu M.\ \lambda x : \mathcal{S}.$$
$$\left(\bigwedge_{\{o \ \mid\ x \xrightarrow{o} -\}} \mathsf{offers}(o) \right) \ \wedge\ \left(\bigwedge_{o \in \mathcal{O}} [o] \bigvee_{\{x' \ \mid\ x \xrightarrow{o} x'\}} M@x' \right)$$

Then the characteristic formula for the process a is $\mathsf{CF}(a) \equiv \mathsf{CF}(\mathcal{L})@z$.

Theorem 6.1.10 (Expressiveness for $\mathcal{T}_C$)**.** *Suppose a and b are LTS processes. Then:*

$$a \sqsubseteq_C b \qquad \text{iff} \qquad b \models \mathsf{CF}(a)$$

6.2 Branching Choice Refinement

Our next task is to lift the notion of choice refinement to the case where we have hidden transitions. The best way to do this is not immediately clear. Follow-up work on ready refinement has explored some of the variations (see §6.4); however, I do not believe that the proper generalization has previously been presented.

My strategy is to take the pruning notion, and the tight connection it has to resolving internal nondeterminism, as basic. Then, the goal is to try to find a relational characterization that is the smallest extension to branching bisimulation that allows all prunings. This goal I will nearly, but not quite, achieve. More details will be provided later in this section.

For now, I will simply state the definition of branching choice refinement without much motivation and will rely on later discussion to defend the definition. It turns out that the divergence-insensitive version of choice refinement is basically useless,[2] so I will skip even defining it.

Definition 6.2.1 (Stable state)**.** *Suppose $\mathcal{L}$ is a τLTS and x is a state in $\mathcal{L}$. Then x is* stable *if there does not exist any x' where $x \xrightarrow{\tau} x'$.*

A stable state is one that cannot engage in any more τ-actions. A stable state represents a process that has reached quiescence; it is either entirely halted, or is waiting for some interaction with the environment. In either case, it cannot continue to compute on its own.

Definition 6.2.2 (Branching choice refinement)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are τLTSs, and let R be a binary relation between their state spaces. Then R is a branching choice refinement if, for each $(x, y) \in R$ it satisfies the following:*

$$\begin{aligned}&y \text{ stable implies}\\ &\quad \exists x'.\ x \overset{\tau}{\Longrightarrow}_{R(-,y)} x' \wedge (x', y) \in R \wedge x' \text{ stable} \wedge\\ &\qquad (\forall o.\ x' \xrightarrow{o} - \ \Rightarrow\ y \xrightarrow{o} -)\end{aligned} \tag{6.8}$$

$$\begin{aligned}&y \xrightarrow{\alpha} y' \ \text{ implies either } \alpha = \tau \wedge (x, y') \in R, \text{ or}\\ &\quad \exists x'\ x''.\ \ x \overset{\tau}{\Longrightarrow}_{R(-,y)} x' \wedge x' \xrightarrow{\alpha} x'' \wedge (x', y) \in R \wedge (x'', y') \in R\end{aligned} \tag{6.9}$$

$$\begin{aligned}&\text{For each divergence } D \text{ in } \mathcal{L}_2 \text{ where } y \in D,\\ &\quad \text{there exists } y' \in D \text{ and } x' \text{ such that } x \xrightarrow{\tau} x' \text{ and } (x', y') \in R.\end{aligned} \tag{6.10}$$

For τLTS processes x and y, I write $x \sqsubseteq_{CB} y$ when there exists a branching choice refinement relation R with $(x, y) \in R$.

The second and third clause are just the backward direction of divergence-sensitive branching bisimulation. The first clause is unique to branching choice refinement. It basically says two things. First, it says that if y is a stable state then there exists a path x can follow to a stable state x', and that every state along the path is related to y. The second thing is that related stable states offer the same observable actions.

One thing to notice about this clause is that it collapses into the forward clause for strong choice refinement for LTSs with no τ-actions. Thus, it makes sense to call branching choice refinement a generalization of strong choice refinement. As with branching bisimulation, there is an alternate

[2]This is because the smallest trasitive relation containing $\cong_B$ and prunings is just backward simulation. Roughly, this happens because the concept of a stable state, critical to the definition of branching choice refinement, does not have a well-defined meaning when considering LTSs up to divergence-insensitive branching bisimulation.

inductive characterization of clause 6.8 (see chapter 8 for the Coq definition). Choice refinement is also closed under stuttering. Every divergence-sensitive branching bisimulation relation is also a branching choice refinement relation. The interested reader should consult the mechanized proof.

The main aim of the remainder of this section is to characterize branching choice refinement in terms of prunings. We will not be 100% successful in our endeavors, but I hope the results achieved will nonetheless help convince the reader that branching choice refinement is the correct generalization of strong choice refinement for our purposes.

The first thing we must do is tweak slightly the definition of prunings for the τLTS case. This is because in a τLTS we now have the possibility of mixed nondeterminism. As discussed in the introduction to this chapter, mixed nondeterminism occurs when a state has both outgoing τ-transitions and also at least one observable transition. The interpretation of this sort of nondeterminism is rather tricky, so it is not entirely clear what should be allowed in terms of pruning. I have made what I think is the most reasonable choice, which is to allow state with mixed nondeterminism to prune all observable transitions. This is basically because a program in a state with mixed nondeterminism always has the option of following a τ-transition and refusing to participate in any action initiated by the environment. This interpretaion falls in line with the adversarial position we are taking with respect to internal nondeterministic choices. Thus the pruning rule becomes: "pruned states must retain every transition label in the original state *unless* the state has an outgoing τ-transition, in which case the pruned state is required only to have some τ-transition." Alternately, we could just as well say: "Pruned states are stable iff the original is stable; and pruned stable states must retain every observable label from the original."

Definition 6.2.3 (τLTS pruning)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are two τLTSs and let $f : \mathcal{S}_2 \to \mathcal{S}_1$ be a function from the state space of $\mathcal{L}_2$ to $\mathcal{L}_1$. We say f is a pruning function if the following hold:*

$$f \text{ is injective} \tag{6.11}$$

$$y \xrightarrow{\alpha} y' \;\Rightarrow\; f(y) \xrightarrow{\alpha} f(y') \tag{6.12}$$

$$f(y) \xrightarrow{\alpha} - \;\Rightarrow\; y \xrightarrow{\alpha} - \vee\, y \xrightarrow{\tau} - \tag{6.13}$$

Suppose x and y are τLTS processes. We say y is a pruning of x if there exists a pruning function f from the states of y to the states of x such that $x = f(y)$. In this case we write $x \preceq_{CB} y$.

Note that states with τ-transitions are always pruned into states with at least one τ-transition, regardless of whether the state is mixed or internal (having only τ transitions). Thus, a mixed state can be pruned into an internal state, but can never become a stable state.

I would like to prove that $\sqsubseteq_{CB}$ is the same relation as $\cong_{\Delta B} \fatsemi \preceq_{CB} \fatsemi \cong_{\Delta B}$. It is easy to show that $\cong_{\Delta B} \fatsemi \preceq_{CB} \fatsemi \cong_{\Delta B}$ is included in $\sqsubseteq_{CB}$. Unfortunately, the reverse inclusion below does *not* hold in general:

$$x \sqsubseteq_{CB} y \quad \Rightarrow \quad x\;(\cong_{\Delta B} \fatsemi \preceq_{CB} \fatsemi \cong_{\Delta B})\;y \tag{6.14}$$

Later I shall present a counterexample. However, equation 6.14 is *nearly* true, in two different senses.

First, it is possible to show that $\sqsubseteq_{CB}$ is included in a slightly more generous relation $\cong_{\Delta B} \fatsemi \sqsubseteq_{CB} \fatsemi \sim_{\Delta B}$. Here $\sim_{\Delta B}$ is just the same as $\cong_{\Delta B}$ *except that* the forward clause relating to divergence is dropped. Therefore, $\sqsubseteq_{CB}$ is just slightly more generous than $\cong_{\Delta B} \fatsemi \preceq_{CB} \fatsemi \cong_{\Delta B}$; the difference is that $x \sqsubseteq_{CB} y$ will sometimes hold even if y diverges less often than any program generated by a pruning.

Second, it is possible to show that the desired statement holds for a large class of special cases. The first special case is when the program is τ-pure; that is, it has no mixed nondeterminism. The second special case is when the program has only finite divergences; that is, divergences that are expressed as cycles in the LTS graph. Therefore, all counterexamples to the desired statement are rather pathological; they must contain nontrivial interaction between mixed nondeterminism and a divergence that passes through an infinite number of distinct states. Interpreted physically, these kinds of divergences correspond to programs that consume unbounded memory (otherwise they would eventually return to the same state).

Therefore, even though I cannot prove equation 6.14 as stated, it nonetheless has a great deal of moral truth. Suppose we have two programs x and y where $x \sqsubseteq_{CB} y$ but not $x(\cong_{\Delta B} \fatsemi \preceq_{CB} \fatsemi \cong_{\Delta B})y$. Then it must be the case that both x and y exhibit mixed nondeterminism *and* it must be the case that x may diverge in a way that consumes unbounded resources *and* it must be that y *fails* to diverge in the corresponding situation, but in all other respects behaves like a pruned version of x.

From a theoretical point of view, it is rather annoying that we cannot prove equation 6.14. However, from a practical point of view it is difficult to object too strenuously if a compiler turns a program that may diverge (consuming unbounded resources) into one that does not diverge, yet still satisfies all other properties of interest. Furthermore, well-behaved programming languages tend not to use mixed nondeterminism at all; the language from chapter 9 does not, for example. The current nondeterministic semantics of C defined in CompCert (version 1.11) unfortunately has mixed nondeterminisim; however, I consider this a bug in the semantics rather than an intended feature (cf. chapter 10).

For languages without mixed nondeterminism, the desired theorem *does* hold. It is also worthwhile to note that for the special case where mixed nondeterminism does not occur, the definition of pruning for τLTSs is precisely the same as for regular LTSs.

The constructions we need to build to prove the special case results mentioned above are a bit more intricate than the constructions for strong choice refinement, but they embody the same basic idea. In the first step, we build a new τLTS out of pairs of states that (mostly) ignore the y component. In the second step, we prune the LTS to only those states related by choice refinement, and retain only those transitions that correspond to simultaneous transitions in both LTSs. However, stuttering τ-transitions and the need to preserve divergence behavior add some additional complications.

We will prove three separate results using basically the same construction, with minor tweaks. In particular, there will be one transition rule for Q_2 which I will define with an unspecified antecedent. Filling in this parameter different ways will give the three different constructions we need. The proofs involving these constructions are tedious and intricate; I will elide them entirely. Consult the mechanized proof for details.

In the following, suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are τLTSs, and let R be a branching choice refinement relation between them. We define a new τLTS Q_1 with state space $\mathcal{S}_1 \times \mathcal{S}_2$, pairs of states from $\mathcal{L}_1$ and $\mathcal{L}_2$. The step relation for Q_1 is given by the rules below:

$$\frac{x \xrightarrow{\alpha} x'}{(x,y) \xrightarrow{\alpha} (x',y')} \qquad \frac{(x,y) \in R \qquad (x,y') \in R \qquad y \xrightarrow{\tau} y' \qquad (\forall x'.\ x \xrightarrow{\tau} x' \ \Rightarrow\ (x',y') \notin R)}{(x,y) \xrightarrow{\tau} (x,y')}$$

The first rule is just the same as the step rule for the corresponding construction in the strong choice refinement case. The second rule allows the y component of the pair to step independently, but only when both y and y' are related to x *and* there is no matching step that x can take. This second rule is included because a corresponding rule will be necessary in the next construction; we must include it here to make sure we get a subgraph. The odd-looking restrictions are necessary to make sure the construction does not introduce new divergent behavior. A state (x,y) in Q_1 is (divergence-sensitive branching) bisimilar to x in $\mathcal{L}_1$.

Next we construct Q_2, which is intended to be a pruned version of Q_1. The state space of Q_2 is $\{(x,y) \mid (x,y) \in R\}$, as previously. The step relation for Q_2 is defined by the following rules:

$$\frac{x \xrightarrow{\alpha} x' \qquad y \xrightarrow{\alpha} y'}{(x,y) \xrightarrow{\alpha} (x',y')} \qquad \frac{x \xrightarrow{\tau} x' \qquad \langle\text{parameter}\rangle}{(x,y) \xrightarrow{\tau} (x',y)} \qquad \frac{y \xrightarrow{\tau} y' \qquad (\forall x'.\ x \xrightarrow{\tau} x' \ \Rightarrow\ (x',y') \notin R)}{(x,y) \xrightarrow{\tau} (x,y')}$$

It is fairly easy to see that Q_2 is a subgraph of Q_1. The remaining properties we desire about

Q_2 require us to select the ⟨parameter⟩ and make some additional assumptions.

In the first instance, we set the ⟨parameter⟩ to the trivial predicate $\top$. With this setting, it is easy to see that Q_2 is a pruning of Q_1. It is also the case that (x, y) in Q_2 is *quite nearly* bisimilar to y in $\mathcal{L}_2$. In particular, we can show both the forward and backward direction of branching bisimulation, and we can also show that divergences in $\mathcal{L}_2$ result in divergences in Q_2. However, the forward direction fails; there may be more divergences in Q_2 than exist in $\mathcal{L}_2$. Thus, this gives us the first result I mentioned above, where $\sqsubseteq_C$ is included in $\cong_{\Delta B} \fatsemi \preceq_{CB} \fatsemi \sim_{\Delta B}$.

In the next instance, we set the ⟨parameter⟩ to be the predicate $x \neq x'$. Thus, Q_2 allows x to step independently along τ transitions as long as it does not follow τ self loops. With this setting, it is possible, with a little work, to show that Q_2 is a pruning of Q_1. It is also the case that (x, y) in Q_2 is bisimilar to y in $\mathcal{L}_2$, provided we make the assumption that the only divergences in $\mathcal{L}_1$ are self-loops. This gets us most of the way to handling the special case with finite divergences. To finish up, we need an additional construction that quotients a τLTS by branching bisimulation. In this construction, we take a τLTS and build a new one with the $\cong_{\Delta B}$-equivalence classes as the new states. This quotient τLTS is bisimilar to the original in the obvious way. However, the quotient LTS has the property that all finite divergences are collapsed into self-loops. In other words, every cycle in the LTS graph is collapsed into a single state. Using these two constructions, we can then show that every LTS with only finite divergences satisfies equation 6.14.

In the final instance, we must set the ⟨parameter⟩ to a more sophisticated predicate. For x to τ-step to x' in a pair with y, we require that x' be along the *shortest path to a shared stable offer* with y. Basically, this predicate only holds when y is a stable state, and when taking the step from x to x' is the fastest way to get to a stable state related to y that offers the same observable actions. Such a shortest path always exists provided that y is stable and $(x, y) \in R$. This is sufficient to show that Q_2, with this parameter, is a pruning of Q_1. To show that Q_2 bisimulates with $\mathcal{L}_2$ requires us to further assume that $\mathcal{L}_2$ has no mixed nondeterminism; that is, whenever y takes some observable action, it must be stable. This allows us to prove that equation 6.14 holds whenever y has no mixed nondeterminism.

The following theorems formally state the results discussed above.

Theorem 6.2.4. *Suppose x and y are τLTS processes with $x \sqsubseteq_{CB} y$. If x has only finite divergences, then $x \; (\cong_{\Delta B} \fatsemi \preceq_{CB} \fatsemi \cong_{\Delta B}) \; y$.*

Note in particular that theorem 6.2.4 holds in the following special cases: when x is a finite-state τLTS; when x is a weakly image-finite τLTS; and when x converges everywhere.

Theorem 6.2.5. *Suppose x and y are τLTS processes with y τ-pure. Then $x \sqsubseteq_{CB} y$ implies $x\ (\cong_{\Delta B} \fatsemi \preceq_{CB} \fatsemi \cong_{\Delta B})\ y$.*

With these results proved, we now have a pretty good idea of where to look for counterexamples. A counterexample must have an input program with mixed nondeterminism and an infinite divergence, and the output program must have mixed nondeterminism and fail to diverge. Consider the τLTSs in figure 6.3; I believe (although I have not formally proved) that they form a counterexample to equation 6.14.

Diagrams (a) and (b) are graphical representations of the initial part of infinite LTSs, which proceed according to the established patterns. LTS (b) is a pruning of (a) that prunes off all the numbered observational arcs. With this done, all the numbered states become indistinguishable, so (b) is bisimilar to (c). Then (d) is obviously a pruning of (c). This is sufficent to establish that LTS (a) is refined by (d). However, I believe it is impossible to arrange a single pruning between them.

The main point is that each numbered node in LTS (a) offers either action a or action b, but never both at the same state. If you are at an even state, you will have to step further down the spine to get to a state offering b — in order to do this, you will have to reject forever the possibility of taking one of the numbered actions. This is the main point of the counterexample; any bisimilar LTS will have to retain this basic property. This is sufficent to require that any pruning which has the property that every state either halts or offers both a and b must retain the entire spine (all the numbered states). Such a pruning must therefore have a divergence, but LTS (d) clearly does not.

Therefore, this counterexample shows that the relation $\cong_{\Delta B} \fatsemi \preceq_{CB} \fatsemi \cong_{\Delta B}$ fails to be transitive! Thus, it was too much to hope for it to be equal to $\sqsubseteq_{CB}$, which we required, at a minimum, to be transitive.

Perhaps instead it is the case that $\sqsubseteq_{CB}$ is equal to the transitive closure of $\cong_{\Delta B} \fatsemi \preceq_{CB} \fatsemi \cong_{\Delta B}$. Certainly, taking the transitive closure deals with the counterexample from figure 6.3. However, it is not immediately clear how, or if, the additional options we get by taking the transitive closure can be used in a generic proof. Having briefly examined the question, all I can say is that the answer is nonobvious. Thus I state the following as an item for future work.

Conjecture. *Suppose $x \sqsubseteq_{CB} y$. Then $x\ (\cong_{\Delta B} \fatsemi \preceq_{CB} \fatsemi \cong_{\Delta B})^+\ y$.*

It is not hard to see that this conjecture is equivalent to stating that $x \sqsubseteq_{CB} y$ is the smallest transitive relation containing both $\cong_{\Delta B}$ and $\preceq_{CB}$.

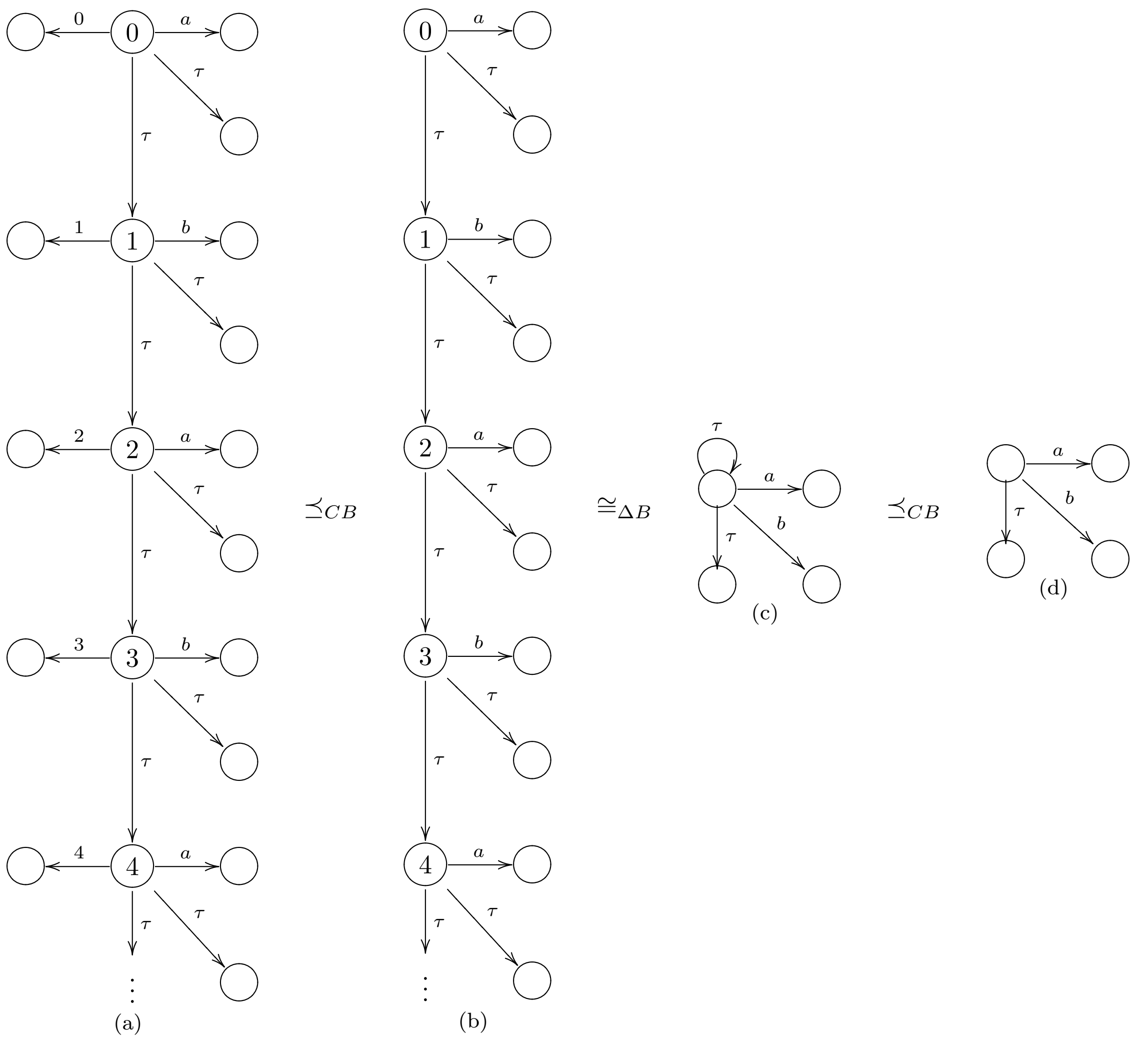


Figure 6.3: Counterexample to equation 6.14

6.3 Back-and-forth Choice Refinement

Our next step is to develop a back-and-forth version of choice refinement over process histories. As with branching bisimulation, this will allow us to find a characteristic logic that is expressive. Throughout this section, I will use the ideas and notation developed in §3.3.

Definition 6.3.1 (Stable history). *Suppose $\mathcal{L}$ is a τLTS and let h be a process history in $\mathcal{L}$. Then we say h is stable if the current state of h is stable.*

Definition 6.3.2 (Back-and-forth choice refinement). *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are τLTSs and R is a binary relation between their histories. Then R is a back-and-forth choice refinement if the following hold for each $(g, h) \in R$:*

$$\begin{aligned} &h \textit{ stable implies} \\ &\quad \exists g'.\ g \overset{\tau}{\Longrightarrow} g' \wedge g' \textit{ stable } \wedge (g', h) \in R \wedge \\ &\quad\quad (\forall o.\ g' \xrightarrow{o} - \ \Rightarrow\ h \xrightarrow{o} -) \end{aligned} \tag{6.15}$$

$$\forall \alpha\ h'.\ h \overset{\alpha}{\Longrightarrow} h' \rightarrow \exists g'. g \overset{\alpha}{\Longrightarrow} g' \wedge (g', h') \in R \tag{6.16}$$

$$\forall \alpha\ g'.\ g' \overset{\alpha}{\Longrightarrow} g \rightarrow \exists h'. h' \overset{\alpha}{\Longrightarrow} h \wedge (g', h') \in R \tag{6.17}$$

$$\forall \alpha\ h'.\ h' \overset{\alpha}{\Longrightarrow} h \rightarrow \exists g'. g' \overset{\alpha}{\Longrightarrow} g \wedge (g', h') \in R \tag{6.18}$$

$$\begin{aligned} &\textit{For each history divergence } D \textit{ where } h \in D, \\ &\quad \textit{there exists } h' \in D \textit{ and } g' \textit{ where } g \xrightarrow{\tau} g' \textit{ and } (g', h') \in R \end{aligned} \tag{6.19}$$

For τLTS process histories g and h, we write $g \sqsubseteq_{CBF} h$ if there exists a back-and-forth choice refinement R with $(g, h) \in R$.

Back-and-forth choice refinement is essentially just the same as back-and-forth bisimulation except that: the forward direction/forward in time clause is dropped; the forward divergence clause is dropped; and the forward choice clause is added instead.

Back-and-forth choice refinement is related to branching choice refinement in precisely the same way that back-and-forth bisimulation is related to branching bisimulation, via the curr and rel relation transformers (see theorem 3.3.13).

Now we are ready to define the characteristic logic for back-and-forth choice refinement. It is remarkably similar to the logic for strong choice refinement. In particular, we lose the forward diamond modality $\langle \alpha \rangle$ and instead get atomic observations $\mathsf{offers}(o)$. We also lose the forward divergence modality Δ. However, we get a new modality, written $\bigcirc P$, that I pronounce "post P."

The meaning of $\bigcirc P$ is: every stable state reachable via a τ-path from the current state satisfies P. In other words, if and when the program reaches quiescence, P will hold. $\bigcirc P$ is the modality of (partial) postconditions; hence the name. The meaning of the $\mathsf{offers}(o)$ atomic proposition is similar: every stable state τ-reachable from the current state offers action o. Note that if a program diverges absolutely (i.e., has no reachable stable states), then $\bigcirc P$ and $\mathsf{offers}(o)$ both hold vacuously. Therefore, they are both statements with a partial-correctness flavor. Both $\bigcirc P$ and $\mathsf{offers}(o)$ can be upgraded to total correctness statements by conjoining them with $\nabla\bot$, which expresses that the program will certainly reach a stable state in finite time. Conversely, the statement $\bigcirc\bot$ expresses absolute divergence — there are no τ-reachable stable states.

This gives us a firm grasp on what sorts of progress properties are preserved by choice refinement; they are basically properties that can be interpreted as partial or total postconditions and the offering of actions to the environment. Note that for LTSs without any τ actions, offers collapses into the offers predicate for strong choice refinement. Furthermore, $\bigcirc P$ holds iff P holds in the absence of τ actions (because every state is stable in that case).

Formally, the data needed to specify the characteristic logic for back-and-forth choice refinement are as follows:

- The worlds are τLTS process histories,
- the accessibility relation is $\sqsubseteq_{CBF}$,
- the modes are as given below, and
- the atoms are $\mathsf{offers}(o)$ for each $o \in \mathcal{O}$.

$$\mathcal{M} := [o] \mid \bigcirc \mid \overleftarrow{[o]} \mid \overleftarrow{\langle o\rangle} \mid \nabla$$

$$[\![[o]]\!]_{\mathcal{M}}(P) \equiv \{h \mid \forall h'.\ h \overset{o}{\Longrightarrow} h' \ \Rightarrow\ h' \in P\} \tag{6.20}$$

$$[\![\overleftarrow{[o]}]\!]_{\mathcal{M}}(P) \equiv \{h \mid \forall h'.\ h' \overset{o}{\Longrightarrow} h \ \Rightarrow\ h' \in P\} \tag{6.21}$$

$$[\![\overleftarrow{\langle o\rangle}]\!]_{\mathcal{M}}(P) \equiv \{h \mid \exists h'.\ h' \overset{o}{\Longrightarrow} h \ \wedge\ h' \in P\} \tag{6.22}$$

$$[\![\nabla]\!]_{\mathcal{M}}(P) \equiv \{h \mid \forall D. D \text{ is a divergence } \Rightarrow h \in D \Rightarrow (\exists h'.\ h' \in D \wedge h' \in P)\} \tag{6.23}$$

$$[\![\bigcirc]\!]_{\mathcal{M}}(P) \equiv \{h \mid \forall h'.\ h \overset{\tau}{\Longrightarrow} h' \ \Rightarrow\ h' \text{ stable } \ \Rightarrow\ h' \in P\} \tag{6.24}$$

$$[\![\mathsf{offers}(o)]\!]_{\mathcal{A}} \equiv \{h \mid \forall h'.\ h \overset{\tau}{\Longrightarrow} h' \ \Rightarrow\ h' \text{ stable } \ \Rightarrow\ h' \xrightarrow{o} -\} \tag{6.25}$$

These definitions are monotone and $\sqsubseteq_{CBF}$-closed; soundness of the characteristic logics follows.

Before moving on, it is interesting to note that offers and $\bigcirc P$ are definable in the characteristic logic of branching bisimulation (cf. chapter 3):

$$\begin{aligned} \mathsf{offers}(o) &= [\tau](\Delta\top \vee \langle o\rangle\top) && (6.26) \\ \bigcirc P &= [\tau](\Delta\top \vee \langle\tau\rangle P) && (6.27) \end{aligned}$$

This gives us a clear picture of what happens, in terms of the logic, when we move from bisimulation to choice refinement. We can no longer write unrestricted uses of the modalities $\langle\alpha\rangle$ or Δ, but only those that fit the specific patterns given above. Roughly, we can only state progress properties that are preserved by $[\tau]$ — that is, preserved by internal computation.

Following the standard recipe, we can show the propositional logic is adequate. With only a little ingenuity we can also find a characteristic formula, showing that the μ-calculus logic is expressive.

Theorem 6.3.3 (Adequacy for $\mathcal{F}_{CBF}$)**.** $^{\mathsf{EM}}$ *Suppose g and h are weakly image-finite τLTS process histories where $g \models f$ implies $h \models f$ for each $f \in \mathcal{F}_{CBF}$. Then $g \sqsubseteq_{CBF} h$.*

Definition 6.3.4 (Characteristic formula for $\mathcal{T}_{CBF}$)**.** *Suppose $h = \langle\mathcal{L}, z\rangle$ is a τLTS process history. The characteristic formula for the LTS $\mathcal{L}$ is:*

$$\begin{aligned} &\mathsf{CF}(\mathcal{L}) \equiv \nu X.\ \lambda h. \\ &\quad \left(\bigcirc \bigvee_{\{h' \mid h \overset{\tau}{\Longrightarrow} h' \,\wedge\, h' \ \mathit{stable}\}} \left(X@h' \ \wedge \bigwedge_{\{o \mid h' \xrightarrow{o} -\}} \mathsf{offers}(o)\right)\right) \wedge \\ &\quad \left(\bigwedge_{\alpha} [\alpha] \bigvee_{\{h' \mid h \overset{\alpha}{\Longrightarrow} h'\}} X@h'\right) \wedge \left(\bigwedge_{\alpha} \bigwedge_{\{h' \mid h' \overset{\alpha}{\Longrightarrow} h\}} \overleftarrow{\langle\alpha\rangle}\, X@h'\right) \wedge \\ &\quad \left(\bigwedge_{\alpha} \overleftarrow{[\alpha]} \bigvee_{\{h' \mid h' \overset{\alpha}{\Longrightarrow} h\}} X@h'\right) \wedge \left(\nabla \left(\bigvee_{\{h' \mid h \xrightarrow{\tau} h'\}} X@h'\right)\right) \end{aligned}$$

Then the characteristic formula for the process history h is $\mathsf{CF}(h) \equiv \mathsf{CF}(\mathcal{L})@z$.

Theorem 6.3.5 (Expressiveness for $\mathcal{T}_{CBF}$)**.** *Suppose g and h are τLTS process histories. Then:*

$$g \sqsubseteq_{CBF} h \qquad \text{iff} \qquad h \models \mathsf{CF}(g)$$

6.4 Related Work

The most closely related work is the ready simulation of Bloom et al. [BIM95, Blo89] and Larsen and Skou's study of $\frac{2}{3}$-bisimulaion [LS91]. Bloom's primay concern seems to have been to challenge bisimulation on the grounds that it allows unrealistic testing seniraios — that is, bisimulation allows the observer too much power to distinguish programs. In support of this argument, Bloom shows that fully-abstract operational semantics for GSOS languages coincide with ready equivalence, rather than bisimulation equivalence. He also shows that full abstraction for bisimulation seems to require "global testing" operations with unnatural semantics.

Larsen and Skou attempt to refute this objection by considering *probabilistic* LTSs; among other things, they show that ordinary bisimulation is testable (in a sense they make precise) when a probabilistic model of nondeterminism is adopted. Roughly, this means that a sufficent amount of testing can decrease the possibility of incorrectly concluding that a program satisfies a property arbitrarily low. The difference in mindset can be summarised by saying that Larsen and Skou adopted a stochastic, *nonadversarial* model of internal nondeterminism, whereas Bloom adopted an adverserial model. The difference in conclusions flow from this difference in mindset.

At any rate, the aims of this previous work were quite different than my present aims. I am not so much interested in what observations of programs are allowed "by nature," but rather I have asked what is the smallest extensional relation I can find that allows the resolution of internal nondeterministic choices. Theorem 6.1.7 shows that strong choice refinement is that relation. It is interesting that it coincides with (the inverse of) ready simulation, which was developed with different aims in mind.

Bloom does not study the generalization of ready simulation to the case where hidden actions are allowed, but instead lists it as an interesting avenue for future work. As stated by Bloom et al. [BIM95], "Silent moves make the philosophy as well as the mathematics trickier; it is no longer clear what the right questions are." For our present concerns, I have stated that I think the right question is: "What does it mean resolve internal nondeterminism?" The related work mentioned below has been asking different questions — unsurprisingly, this work does not seem to contain the answer to my question.

One generalization of ready simulation is the *refusal simulation* of Ulidowski [Uli92]. Ulidowski shows that his refusal simulation is equivalant to equivalence under a testing scenario he calls copy+refusal testing. Refusal simulation is strictly weaker than my branching choice refinement because the branching properties of the backward part of the refinement correspond to a global test, which is disallowed by the copy+refusal test scenario. Refusal simulation also differs in how it handles divergence.

Lüttgen and Volger [LV10] have developed an adaptation of ready simulation to *logic LTSs*, a variant of LTSs that enables the definition of logical operators, like conjunction, over LTS structures. Their definition uses the extra structure avaliable in logic LTSs, so their definition is obviously not equivalant to mine. However, even if the parts specific to logic LTS are removed, I believe their definition is distinct (weaker, or perhaps incomparable) to my own. It is interesting to note that Lüttgen and Volger restrict their attention to the τ-pure LTSs; that is, the LTSs that have no mixed nondeterminism.

Ingólfsdóttir studied yet another generalization of ready simulation [Ing97], which she calls the *readiness preorder*. Again, it differs from mine; in particular, divergent processes are interpreted as satisfying no interesting properties. I further suspect it is weaker than branching choice refinement even for nondivergent processes. Although Ingólfsdóttir does not assume τ-purity, she does make the weaker assumption that every hidden step preserves the ability to perform an observable action.

The notion of pruning I consider is similar in some ways to the refinement mappings of Abadi and Lamport [AL91]. Both basically require that one state machine be a subgraph of another and additionally satisfy some liveness property. Their setup differs sufficently from the LTS setup I use that the notions are difficult to directly compare.

Chapter 7

Behavioral Choice Refinement

In this chapter, I tie together the threads from chapters 4 and 6, showing how both behavioral and choice refinement can be combined together into a single notion. The combination should be unsurprising. The main idea is to apply the may/must step paradigm to choice refinement; alternately, we weaken the forward direction $\mho$-refinement in the same way we weakened bisimulation to get choice refinement.

By now the pattern of results is well-established. Few additional issues of interest arise from the combination of behavioral and choice refinement. Therefore, in this chapter I will present key definitions and theorems, but little discussion or proof.

7.1 Strong Choice Behavioral Refinement

We begin, as usual, with the strong theory, moving on to the branching theory later.

Definition 7.1.1 (Strong choice $\mho$-refinement)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are ELTSs, and R is a binary relation between their lifted state spaces. Then R is a strong choice $\mho$-refinement if the following hold for each $(x, y) \in R$:*

$$\forall o.\ x \xrightarrow[\mathit{must}]{o} - \ \Rightarrow\ y \xrightarrow[\mathit{must}]{o} - \tag{7.1}$$

$$\forall o\ y'.\ y \xrightarrow[\mathit{may}]{o} y' \ \Rightarrow\ \exists x'.\ x \xrightarrow[\mathit{may}]{o} x' \wedge (x', y') \in R \tag{7.2}$$

Let x and y be ELTS processes. Then we write $x \sqsubseteq_{\mho C} y$ when there exists a strong choice behavioral refinement R with $(x, y) \in R$.

As usual, $\sqsubseteq_{\mho C}$ is reflexive and transitive. As with $\sqsubseteq_{\mho}$, the wrong state $\mho$ is the bottom element of the preorder.

Using a combination of the constructions from chapters 4 and 6, we can show that $\sqsubseteq_{\mho C}$ is the smallest transitive relation that contains $\cong_{\mho}$, $\mho$-expansions, and prunings.

The characteristic logic for $\sqsubseteq_{\mho C}$ is given by the following data:

- The worlds are the ELTS processes,
- the accessibility relation is $\sqsubseteq_{\mho C}$,
- the modes are $[o]$ for each $o \in \mathcal{O}$, and
- the atoms are $\mathsf{offers}(o)$ for each $o \in \mathcal{O}$.

$$[\![[o]]\!]_{\mathcal{M}}(P) \equiv \{x \mid \forall x'.\ x \xrightarrow[\text{may}]{o} x' \ \Rightarrow\ x' \in P\} \tag{7.3}$$

$$[\![\mathsf{offers}(o)]\!]_{\mathcal{A}} \equiv \{x \mid x \xrightarrow[\text{must}]{o} -\} \tag{7.4}$$

Definition 7.1.2 (Characteristic formula for $\mathcal{T}_{\mho C}$)**.** *Suppose $a = \langle \mathcal{L}, z\rangle$ is an ELTS process. The characteristic formula for the ELTS $\mathcal{L}$ is:*

$$\mathsf{CF}(\mathcal{L}) \equiv \nu M.\ \lambda x : \mathcal{S}.$$
$$\left(\bigwedge_{\{o \,\mid\, x \xrightarrow[\text{must}]{o} -\}} \mathsf{offers}(o) \right) \wedge \left(\bigwedge_{o \in \mathcal{O}} [o] \bigvee_{\{x' \,\mid\, x \xrightarrow[\text{may}]{o} x'\}} M@x' \right)$$

Then the characteristic formula for the process a is $\mathsf{CF}(a) \equiv \mathsf{CF}(\mathcal{L})@z$.

Soundness, adequacy and expressiveness follow for this characteristic logic via simple modifications of the corresponding results from earlier chapters.

7.2 Branching Behavioral Choice Refinement

Now we are ready to lift behavioral choice refinement to the setting with τ-actions. For the most part, we apply the standard formula to choice refinement: forward directions are stated using must stepping and backward directions are stated using may stepping. However, the forward clause of branching choice refinement does not neatly fit into the pattern. There is an unusual nesting in this clause — we first ask if y steps (is y stable?), and then say something about x stepping. This is an inversion of the usual order where we first know that x steps and have to show that y steps. Does the usual rule apply? Which style of stepping should we use?

In particular, which stepping relation should we use when we define what it means to be stable? Ultimately, the question comes down to whether we should consider the wrong state $\mho$ to be stable. The answer is, a little unintuitively, that we should consider $\mho$ stable, and therefore stability should be defined with respect to the must-step relation. This is true despite the fact that $\mho$ may be refined to a state that takes additional τ-steps.

Why should this be the case? The answer is most easily understood by considering how this decision impacts the characteristic logic; in particular the semantics of the $\mathsf{offers}(o)$ atom means that there is only one correct way to define stability. The offers atom makes some particular informative claim about the state being judged; *clearly* it should not be a tautology. Recall that its intuitive meaning is that every τ-reachable stable state should offer action o. We wish to retain the property about $\mho$ from chapter 4, which is that $\mho$ satisfies only the tautologies. We are therefore *forced* to consider $\mho$ a stable state; otherwise $\mathsf{offers}(o)$ would hold vacuously on $\mho$. This decision also has the pleasing effect of making it so that $\bigcirc P$ holds on an unsafe state (one where $\mho$ is τ-reachable) only if P is a tautology.

We can also consider the alternate, inductive definition of the forward choice clause (which I have not defined here; consult chapter 8 or the mechanized proof). If we state the inductive version using must-steps everywhere (as is appropriate to a forward clause), it turns out to be equivalent to the version defined below.

Finally, with this definition, we can prove that $\bigcirc$ and offers satisfy equations corresponding to 6.26 and 6.27, when these equations are interpreted using the characteristic logic from chapter 4.

Definition 7.2.1 (ELTS stability)**.** *Suppose $\mathcal{L}$ is an ELTS; let x be a lifted state in $\mathcal{L}$. We say x is* stable *if there does not exist any x' where $x \xrightarrow[must]{\tau} x'$.*

Definition 7.2.2 (Branching behavioral choice refinement)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are τELTSs, and let R be a binary relation between their lifted state spaces. Then R is a branching behavioral choice refinement if, for each $(x, y) \in R$ it satisfies the following:*

$$\begin{aligned} &y \textit{ stable implies} \\ &\quad \exists x'.\ x \overset{\tau}{\underset{may}{\Longrightarrow}}_{R(-,y)} x' \wedge (x', y) \in R \wedge x' \textit{ stable} \wedge \\ &\qquad (\forall o.\ x' \xrightarrow[must]{o} - \ \Rightarrow\ y \xrightarrow[must]{o} -) \end{aligned} \tag{7.5}$$

$$\begin{aligned} &y \xrightarrow[\text{may}]{\alpha} y' \text{ implies either } \alpha = \tau \wedge (x, y') \in R, \text{ or} \\ &\quad \exists x'\ x''.\ x \underset{\text{may}}{\overset{\tau}{\Longrightarrow}}_{R(-,y)} x' \wedge x' \xrightarrow[\text{may}]{\alpha} x'' \wedge (x', y) \in R \wedge (x'', y') \in R \end{aligned} \tag{7.6}$$

$$\begin{aligned} &\textit{For each may-divergence } D \textit{ in } \mathcal{L}_2 \textit{ where } y \in D, \\ &\quad \textit{there exists } y' \in D \textit{ and } x' \textit{ such that } x \xrightarrow[\text{may}]{\tau} x' \textit{ and } (x', y') \in R. \end{aligned} \tag{7.7}$$

For τLTS processes x and y, I write $x \sqsubseteq_{\mho CB} y$ when there exists a branching behavioral choice refinement relation R with $(x, y) \in R$.

As before, $\sqsubseteq_{\mho CB}$ is reflexive and transitive.

Given the unresolved status of similar conjectures for (divergence-sensitive) branching behavioral refinement and branching choice refinement, I have not yet made any progress on proving or disproving the following conjecture. Nonetheless, I suspect it is true, or is in some sense morally true.

Conjecture. *$\sqsubseteq_{\mho CB}$ is the smallest transitive relation containing $\cong_{\Delta B}$, $\mho$-expansions and choice prunings.*

Resolving the corresponding questions for $\sqsubseteq_{\mho \Delta B}$ and $\sqsubseteq_{CB}$ will, I strongly suspect, lead directly to a proof or disproof of this conjecture.

7.3 Back-and-forth Behavioral Choice Refinement

The back-and-forth version of behavioral choice refinement is an unsurprising application of ideas from the previous section and §6.3.

Definition 7.3.1 (Stable history)**.** *Suppose $\mathcal{L}$ is a τELTS and let h be a process history in $\mathcal{L}$. Then we say h is stable if the current state of h is stable. In other words, h is stable if it has no must-extensions.*

Definition 7.3.2 (Back-and-forth behavioral choice refinement)**.** *Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are τELTSs and R is a binary relation between their histories. Then R is a back-and-forth choice refinement if the following hold for each $(g, h) \in R$:*

$$\begin{aligned} &h \textit{ stable implies} \\ &\quad \exists g'.\ g \underset{\text{may}}{\overset{\tau}{\Longrightarrow}} g' \wedge g' \textit{ stable } \wedge (g', h) \in R \wedge \\ &\qquad (\forall o.\ g' \xrightarrow[\text{must}]{o} - \ \Rightarrow\ h \xrightarrow[\text{must}]{o} -) \end{aligned} \tag{7.8}$$

$$\forall \alpha\ h'.\ h \underset{\mathit{may}}{\overset{\alpha}{\Longrightarrow}} h' \to \exists g'. g \underset{\mathit{may}}{\overset{\alpha}{\Longrightarrow}} g' \wedge (g', h') \in R \tag{7.9}$$

$$\forall \alpha\ g'.\ g' \underset{\mathit{may}}{\overset{\alpha}{\Longrightarrow}} g \to \exists h'. h' \underset{\mathit{may}}{\overset{\alpha}{\Longrightarrow}} h \wedge (g', h') \in R \tag{7.10}$$

$$\forall \alpha\ h'.\ h' \underset{\mathit{may}}{\overset{\alpha}{\Longrightarrow}} h \to \exists g'. g' \underset{\mathit{may}}{\overset{\alpha}{\Longrightarrow}} g \wedge (g', h') \in R \tag{7.11}$$

$$\begin{gathered}\textit{For each history may-divergence } D \textit{ where } h \in D,\\ \textit{there exists } h' \in D \textit{ and } g' \textit{ where } g \underset{\mathit{may}}{\overset{\tau}{\longrightarrow}} g' \textit{ and } (g', h') \in D\end{gathered} \tag{7.12}$$

For τLTS process histories g and h, we write $g \sqsubseteq_{\mho CBF} h$ if there exists a back-and-forth behavioral choice refinement R with $(g, h) \in R$.

The usual results hold: $\sqsubseteq_{\mho CBF}$ is reflexive, transitive, and related to the branching version of choice behavioral refinement in the same way as the other back-and-forth refinements.

The data needed to specify the characteristic logic for back-and-forth behavioral choice refinement are as follows:

- The worlds are τELTS process histories,
- the accessibility relation is $\sqsubseteq_{\mho CBF}$,
- the modes are as given below, and
- the atoms are $\mathsf{offers}(o)$ for each $o \in \mathcal{O}$.

$$\mathcal{M} := [o] \mid \bigcirc \mid \overleftarrow{[o]} \mid \overleftarrow{\langle o \rangle} \mid \nabla$$

$$[\![[o]]\!]_{\mathcal{M}}(P) \equiv \{h \mid \forall h'.\ h \underset{\text{may}}{\overset{o}{\Longrightarrow}} h' \ \Rightarrow\ h' \in P\} \tag{7.13}$$

$$[\![\overleftarrow{[o]}]\!]_{\mathcal{M}}(P) \equiv \{h \mid \forall h'.\ h' \underset{\text{may}}{\overset{o}{\Longrightarrow}} h \ \Rightarrow\ h' \in P\} \tag{7.14}$$

$$[\![\overleftarrow{\langle o \rangle}]\!]_{\mathcal{M}}(P) \equiv \{h \mid \exists h'.\ h' \underset{\text{may}}{\overset{o}{\Longrightarrow}} h \ \wedge\ h' \in P\} \tag{7.15}$$

$$[\![\nabla]\!]_{\mathcal{M}}(P) \equiv \{h \mid \forall D. D \text{ is a may-divergence } \Rightarrow h \in D \Rightarrow (\exists h'.\ h' \in D \wedge h' \in P)\} \tag{7.16}$$

$$[\![\bigcirc]\!]_{\mathcal{M}}(P) \equiv \{h \mid \forall h'.\ h \underset{\text{may}}{\overset{\tau}{\Longrightarrow}} h' \ \Rightarrow\ h' \text{ stable } \ \Rightarrow\ h' \in P\} \tag{7.17}$$

$$[\![\mathsf{offers}(o)]\!]_{\mathcal{A}} \equiv \{h \mid \forall h'.\ h \underset{\text{may}}{\overset{\tau}{\Longrightarrow}} h' \ \Rightarrow\ h' \text{ stable } \ \Rightarrow\ h' \underset{\text{must}}{\overset{o}{\longrightarrow}} -\} \tag{7.18}$$

These definitions are monotone and $\sqsubseteq_{\mho CBF}$-closed; soundness of the characteristic logics follows.

Theorem 7.3.3 (Adequacy for $\mathcal{F}_{\mho CBF}$)**.** $^{\mathsf{EM}}$ *Suppose g and h are weakly image-finite τELTS process histories where $g \models f$ implies $h \models f$ for each $f \in \mathcal{F}_{\mho CBF}$. Then $g \sqsubseteq_{\mho CBF} h$.*

Definition 7.3.4 (Characteristic formula for $\mathcal{T}_{\mho CBF}$)**.** *Suppose $h = \langle \mathcal{L}, z \rangle$ is a τLTS process history. The characteristic formula for the ELTS $\mathcal{L}$ is:*

$$
\begin{aligned}
\mathsf{CF}(\mathcal{L}) \equiv \nu X.\ \lambda h.& \\
&\left(\bigcirc \bigvee_{\{h' \ \mid\ h \xRightarrow[\mathit{may}]{\tau} h' \ \wedge\ h' \ \mathit{stable}\}} \left(X@h' \ \wedge \bigwedge_{\{o \ \mid\ h' \xrightarrow[\mathit{must}]{o} -\}} \mathsf{offers}(o) \right) \right) \wedge \\
&\left(\bigwedge_{\alpha} [\alpha] \bigvee_{\{h' \ \mid\ h \xRightarrow[\mathit{may}]{\alpha} h'\}} X@h' \right) \wedge \left(\bigwedge_{\alpha} \bigwedge_{\{h' \ \mid\ h' \xRightarrow[\mathit{may}]{\alpha} h\}} \overleftarrow{\langle\alpha\rangle}\ X@h' \right) \wedge \\
&\left(\bigwedge_{\alpha} \overleftarrow{[\alpha]} \bigvee_{\{h' \ \mid\ h' \xRightarrow[\mathit{may}]{\alpha} h\}} X@h' \right) \wedge \left(\nabla \left(\bigvee_{\{h' \ \mid\ h \xrightarrow[\mathit{may}]{\tau} h'\}} X@h' \right) \right)
\end{aligned}
$$

Then the characteristic formula for the process history h is $\mathsf{CF}(h) \equiv \mathsf{CF}(\mathcal{L})@z$.

Theorem 7.3.5 (Expressiveness for $\mathcal{T}_{\mho CBF}$)**.** *Suppose g and h are τELTS process histories. Then:*

$$g \sqsubseteq_{\mho CBF} h \qquad \text{iff} \qquad h \models \mathsf{CF}(g)$$

7.4 Related Work

Beyond the related work already listed in chapters 4 and 6, the other main body of related work is Hoare's Communicating Sequential Processes (CSP) [Hoa85].

One of the main organizing principles in CSP is the refinement of nondeterministic processes. The usual order used in CSP is the failures/divergences refinement ordering. The semantics of CSP is defined denotationally, using sets of *failures* and *divergences*. Both failure (inability to take some action requested by the environment) and divergence (interpreted as a catastrophic failure) are considered undesirable outcomes; a refined process is defined to be one with fewer failures and divergences.

The notion of failure/divergence refinement captures many of the same ideas as both behavioral refinement and choice refinement. The intuition is quite similar: "better" programs are those that go wrong less often, and are more deterministic.

Although the intuition is similar, the mathematical machinery employed in CSP is quite different from what I have employed here. Hoare endows CSP with a denotational semantics based on failures and divergences, whereas I use the operational paradigm based on labeled transition systems.

Nonetheless, there are a number of interesting points of similarity.

The worst CSP process, called CHAOS, diverges and fails on any trace —it is basically equivalent to $\mho$. Going upward in the failures/divergences order also allows one to resolve internal nondeterminism, as with choice refinement. However, the treatment of divergence is quite different. In CSP, recursive processes are interpreted using CPO least-fixpoint semantics. The partial order used is failure/divergence refinement; this means that nonterminating (divergent) processes are interpreted as the least element of the order, CHAOS. As I have discussed in previous chapters, I do not believe that this is the appropriate interpretation of divergence in the setting of compiler correctness.

Setting aside this difference, it may be interesting to examine how failure/divergence refinement is related to behavioral choice refinement if we consider only converging processes. I suspect that behavioral choice refinement (interpreted over a suitable operational semantics of CSP and when restricted to converging processes) will either coincide with the failure/divergence ordering or be strictly finer.

Chapter 8

An Interface for Compiler Proofs

Suppose the reader is convinced by the preceeding chapters that I have found good and useful notions for compiler correctness and he wishes to use these notions in his own correctness proof. The previous chapters contained a lot of definitions and proofs, most of which are not needed if one is only interested in adopting and applying the ideas of behavioral and choice refinement. In this chapter, I present a concrete and (mostly) minimal interface that contains just the bare essentials needed to apply the theory. This interface is sufficent for all the proofs in the case study of chapter 9 as well as the proofs relating to CompCert appearing in chapter 10.

Unlike previous chapters, I will be presenting this interface directly from the formal proof, using the native syntax of Coq. The following is a verbatim listing (file `iface.v`) from the formal proof, with some light syntax sugar applied for typesetting. Inline comments explain what the various pieces do.

```
Require Import Relations. (* Standard library module *)

Module Type COMPILER_INTERFACE.

(** This typeclass is a simple organizational trick that allows
    me to get close to the informal mathematical practice
    of saying: fix in advance a set O of observations. *)
Class ObservationSystem :=
  { O : Type; observations_inhabited : O }.

(** The inverse of a relation. *)
Definition inv {X Y} (R:X → Y → Prop) : Y → X → Prop
  := fun y x ⇒ R x y.
```

```

(** The lift of a type.  This type is essentially the same as option,
   but with different names for the constructors to avoid confusion. *)
Inductive lift (X:Type) :=
  | into : ∀ (x:X), lift X
  | mho  : lift X.
Implicit Arguments into [[X]].

(** Defines when a lifted state [q] contains an actual state [x]. *)
Definition contains X (q:lift X) (x:X) :=
  match q with
  | into q' ⇒ q' = x
  | mho ⇒ False
  end.
Implicit Arguments contains.

(** Some basic facts about contains.  *)
Axiom contains_uniq : ∀ X (q:lift X),
  ∀ x y, contains q x → contains q y → x = y.

Axiom contains_eq : ∀ X (x y:lift X),
  contains x = contains y → x = y.

Axiom contains_same : ∀ X (x y:lift X) (x' y':X),
  contains x x' → contains y y' → x' = y' → x = y.

Definition lift_map A B (f:A → B) (x:lift A) : lift B :=
  match x with
  | into x' ⇒ into (f x')
  | mho ⇒ mho _
  end.
Implicit Arguments lift_map.

(** The defininition of and a few simple facts about paths.
    This definition of paths is slightly different from the
    one presented earlier in that it does not require the
    final state in the path to satisfy the property [P].
    This definition is slightly more convenient. *)
Section paths.
  Variables X:Type.
  Variable R:X → X → Prop.

  Inductive path_where (P:X → Prop) : X → X → Prop :=
  | path_nil : ∀ x, path_where P x x
  | path_cons : ∀ x1 x2 x3,
            P x1 → R x1 x2 →
            path_where P x2 x3 →
            path_where P x1 x3.

  Axiom path_rt : ∀ P x1 x2,
    path_where P x1 x2 → clos_refl_trans X R x1 x2.

  Axiom path_cons_right : ∀ P x1 x2 x3,
    path_where P x1 x2 →
    P x2 → R x2 x3 →
    path_where P x1 x3.

  Axiom path_trans : ∀ P x1 x2 x3,
    path_where P x1 x2 →
    path_where P x2 x3 →
    path_where P x1 x3.

  Axiom path_inv_right : ∀ P x1 x2,
    path_where P x1 x2 →
    x1 = x2 ∨
    (∃ x',
      path_where P x1 x' ∧
      P x' ∧ R x' x2).

  Axiom path_where_incl : ∀ (P1 P2: X → Prop) x1 x2,
    path_where P1 x1 x2 →
    (∀ x, P1 x → P2 x) →
    path_where P2 x1 x2.
End paths.

(** Now we get to the meat of the interface.  Here, we define
    tau-ELTSs and the notions of refinement we want.  This
    is all done in a section to make it easy for all the
    definitions to refer to the [ObservationSystem]
    declared at the beginning. *)
Section compiler_interface. Context {Obs:ObservationSystem}.

```

```
(** Coq does not provide useful induction schemes for the
    inductive refinements below, so we turn off automatic
    generation of induction schemes and do it manually. *)
Unset Elimination Schemes.

  (** This record defines what I call a tau-ELTS in the
      main body of the thesis.  [None] represents
      tau actions, and [Some x] represent observable action [x]. *)
  Record LTS :=
    { state : Type
    ; steps : state → option O → lift state → Prop
    }.

  Definition lstate X := lift (state X).

  (** May and must stepping are defined in a slightly different
      (but equivalent) way to the main body of the thesis. *)
  Definition must_step (X:LTS) (s:lstate X) o (s':lstate X) :=
    ∃ s_, contains s s_ ∧ steps X s_ o s'.

  Definition may_step (X:LTS) (s:lstate X) o (s':lstate X) :=
    (∀ s_, contains s s_ → steps X s_ o s') ∧
    (∀ s'_, contains s' s'_ → ∃ s_, contains s s_).

  Definition must_step_star (X:LTS) :=
    clos_refl_trans (lstate X) (fun s s' ⇒ must_step X s None s').

  Definition may_step_star (X:LTS) :=
    clos_refl_trans (lstate X) (fun s s' ⇒ may_step X s None s').

  (** A few frequently-useful facts about may and must stepping. *)
  Axiom must_may_step : ∀ X s o s',
    must_step X s o s' →
    may_step X s o s'.

  Axiom may_step_step : ∀ A a o a',
    may_step A (into a) o a' →
    steps A a o a'.

  Axiom must_may_step_star : ∀ X x y,
    must_step_star X x y →
    may_step_star X x y.

  (** A [transition_diagram] is the body of an operator on relations.
      All the refinements I define below are explicitly built as
      the Knaster-Tarski greatest fixpoint of such an operator. *)
  Definition transition_diagram :=
    ∀ (X Y:LTS) (R:lstate X → lstate Y → Prop), lstate X → lstate Y → Prop.

  (** [td_and] allows us to combine two transition diagrams by conjunction. *)
  Definition td_and (T1 T2:transition_diagram) : transition_diagram :=
    fun X Y R x y ⇒ T1 X Y R x y ∧ T2 X Y R x y.

  (** A relation [R] is a refinement of type [T] if every pair in
      the relation is also in [T] applied to [R]. *)
  Definition refinement (T:transition_diagram) X Y (R:lstate X → lstate Y → Prop)
      :=
    ∀ x y, R x y → T X Y R x y.

  (** A relation [R] is a bisimulation of type [T] iff it is a refinement
      and its inverse is a refinement. *)
  Definition bisimulation (T:transition_diagram) X Y (R:lstate X → lstate Y → Prop)
       :=
    ∀ x y, R x y → T X Y R x y ∧ T Y X (inv R) y x.

  (** [x] is refined by [y] if there is a refinement relation
      that relates them. This definition constructs the
      greatest fixpoint of the operator [T] *)
  Definition refines T X Y x y :=
    ∃ R, refinement T X Y R ∧ R x y.

  (** [x] is bisimilar by [y] if there is a bisimulation relation
      that relates them. This definition constructs the
      greatest fixpoint of the operator [T] conjoined
      with its inverse. *)
  Definition bisimilar T X Y x y :=
    ∃ R, bisimulation T X Y R ∧ R x y.

  (** Bisimilarity is symmetric, by construction. *)
  Axiom bisimilar_sym : ∀ T X Y x y,
    bisimilar T X Y x y → bisimilar T Y X y x.

```

```
(** The next few propositions allow us to
    break apart and put together compound refinements. *)
Axiom refinement_and : ∀ T1 T2 X Y R,
  refinement T1 X Y R →
  refinement T2 X Y R →
  refinement (td_and T1 T2) X Y R.

Axiom and_refinement : ∀ T1 T2 X Y R,
  refinement (td_and T1 T2) X Y R →
  refinement T1 X Y R ∧ refinement T2 X Y R.

Axiom bisimulation_and : ∀ T1 T2 X Y R,
  bisimulation T1 X Y R →
  bisimulation T2 X Y R →
  bisimulation (td_and T1 T2) X Y R.

Axiom and_bisimulation : ∀ T1 T2 X Y R,
  bisimulation (td_and T1 T2) X Y R →
  bisimulation T1 X Y R ∧ bisimulation T2 X Y R.

(** The inductive version of branching behavioral refinement. *)
Inductive branch_forward_ind X Y R : lstate X → lstate Y → Prop :=
  branch_forward_ind_intro : ∀ x y,
    (∀ o x', must_step X x o x' →
        (o = None ∧ R x' y ∧ branch_forward_ind X Y R x' y) ∨
        ∃ y', ∃ y'',
          path_where _ (fun a b ⇒ must_step Y a None b) (R x) y y' ∧
          must_step Y y' o y'' ∧ R x y' ∧ R x' y'') →
    branch_forward_ind X Y R x y.

Inductive branch_backward_ind X Y R : lstate X → lstate Y → Prop :=
  branch_backward_ind_intro : ∀ x y,
    (∀ o y', may_step Y y o y' →
        (o = None ∧ R x y' ∧ branch_backward_ind X Y R x y') ∨
        ∃ x', ∃ x'',
          path_where _ (fun a b ⇒ may_step X a None b) (fun x ⇒ R x y) x x' ∧
          may_step X x' o x'' ∧ R x' y ∧ R x'' y') →
    branch_backward_ind X Y R x y.

(** The induction schemes for branching behavioral refinement *)
Axiom branch_forward_ind_ind
```

```
  : ∀ (X Y : LTS) (R P : lstate X → lstate Y → Prop),
     (∀ (x : lstate X) (y : lstate Y),
      (∀ (o : option O) (x′ : lstate X),
       must_step X x o x′ →
       o = None ∧ R x′ y ∧ P x′ y ∨
       (∃ y′ : lstate Y,
          ∃ y′′ : lstate Y,
            path_where (lstate Y)
              (fun a b : lstate Y ⇒ must_step Y a None b)
              (R x) y y′ ∧ must_step Y y′ o y′′ ∧ R x y′ ∧ R x′ y′′)) →
      P x y) →
     ∀ (l : lstate X) (l0 : lstate Y),
     branch_forward_ind X Y R l l0 → P l l0.


Axiom branch_backward_ind_ind
  : ∀ (X Y : LTS) (R P : lstate X → lstate Y → Prop),
     (∀ (x : lstate X) (y : lstate Y),
      (∀ (o : option O) (y′ : lstate Y),
       may_step Y y o y′ →
       o = None ∧ R x y′ ∧ P x y′ ∨
       (∃ x′ : lstate X,
          ∃ x′′ : lstate X,
            path_where (lstate X)
              (fun a b : lstate X ⇒ may_step X a None b)
              (fun x0 : lstate X ⇒ R x0 y) x x′ ∧
            may_step X x′ o x′′ ∧ R x′ y ∧ R x′′ y′)) →
      P x y) →
     ∀ (l : lstate X) (l0 : lstate Y),
     branch_backward_ind X Y R l l0 → P l l0.



(** The inductive version of branching behavioral refinement,
    without stuttering.  These are called inductive
    pre−branching behavioral refinements in the body of the thesis. *)
Inductive branch_forward_ind_no_stutter X Y R : lstate X → lstate Y → Prop :=
  branch_forward_ind_no_sutter_intro : ∀ x y,
    (∀ o x′, must_step X x o x′ →
        (o = None ∧ R x′ y ∧ branch_forward_ind_no_stutter X Y R x′ y) ∨
        ∃ y′, ∃ y′′,
          must_step_star Y y y′ ∧
          must_step Y y′ o y′′ ∧ R x y′ ∧ R x′ y′′) →
    branch_forward_ind_no_stutter X Y R x y.

Inductive branch_backward_ind_no_stutter X Y R : lstate X → lstate Y → Prop :=
  branch_backward_ind_no_stutter_intro : ∀ x y,
    (∀ o y′, may_step Y y o y′ →
        (o = None ∧ R x y′ ∧ branch_backward_ind_no_stutter X Y R x y′) ∨
        ∃ x′, ∃ x′′,
          may_step_star X x x′ ∧
          may_step X x′ o x′′ ∧ R x′ y ∧ R x′′ y′) →
    branch_backward_ind_no_stutter X Y R x y.

(** The induction schemes for non-stuttering branching
    behavioral refinement. *)
Axiom branch_forward_ind_no_stutter_ind
  : ∀ (X Y : LTS) (R P : lstate X → lstate Y → Prop),
     (∀ (x : lstate X) (y : lstate Y),
      (∀ (o : option O) (x′ : lstate X),
       must_step X x o x′ →
       o = None ∧ R x′ y ∧ P x′ y ∨
       (∃ y′ : lstate Y,
          ∃ y′′ : lstate Y,
            must_step_star Y y y′ ∧
            must_step Y y′ o y′′ ∧ R x y′ ∧ R x′ y′′)) →
      P x y) →
     ∀ (l : lstate X) (l0 : lstate Y),
     branch_forward_ind_no_stutter X Y R l l0 → P l l0.

Axiom branch_backward_ind_no_stutter_ind
  : ∀ (X Y : LTS) (R P : lstate X → lstate Y → Prop),
     (∀ (x : lstate X) (y : lstate Y),
      (∀ (o : option O) (y′ : lstate Y),
       may_step Y y o y′ →
       o = None ∧ R x y′ ∧ P x y′ ∨
       (∃ x′ : lstate X,
          ∃ x′′ : lstate X,
            may_step_star X x x′ ∧
            may_step X x′ o x′′ ∧ R x′ y ∧ R x′′ y′)) →
      P x y) →
     ∀ (l : lstate X) (l0 : lstate Y),
     branch_backward_ind_no_stutter X Y R l l0 → P l l0.

```

```
(** The transition diagram for inductive branching behavioral refinement is
    just the conjunction of the forward and backward directions. *)
Definition branch_ind : transition_diagram :=
  td_and branch_forward_ind branch_backward_ind.

(** Likewise for the non-stuttering version. *)
Definition branch_ind_no_stutter : transition_diagram :=
  td_and branch_forward_ind_no_stutter branch_backward_ind_no_stutter.

(** Once we take the greatest fixpoint, the stuttering and non-stuttering
    refinements are the same. *)
Axiom branch_ind_stutter_eq : ∀ X Y x y,
  refines branch_ind X Y x y ↔ refines branch_ind_no_stutter X Y x y.

Axiom bisim_branch_ind_stutter_eq : ∀ X Y x y,
  bisimilar branch_ind X Y x y ↔ bisimilar branch_ind_no_stutter X Y x y.

(** Inductive branching behavioral refinement is a preorder. *)
Axiom refines_ind_refl : ∀ X x,
  refines branch_ind X X x x.

Axiom refines_ind_trans : ∀ X Y Z x y z,
  refines branch_ind X Y x y →
  refines branch_ind Y Z y z →
  refines branch_ind X Z x z.

(** Inductive branching behavioral bisimulation is an equivalence. *)
Axiom bisim_ind_refl : ∀ X x,
  bisimilar branch_ind X X x x.

Axiom bisim_ind_trans : ∀ X Y Z x y z,
  bisimilar branch_ind X Y x y →
  bisimilar branch_ind Y Z y z →
  bisimilar branch_ind X Z x z.

(** These refinements are fixpoints of their generating operators. *)
Axiom refines_ind_refinement : ∀ X Y,
  refinement branch_ind X Y (refines branch_ind X Y).

Axiom refines_ind_no_stutter_refinement : ∀ A B,
  refinement branch_ind_no_stutter A B (refines branch_ind_no_stutter A B).
```

```

(** [x] enables the observation [o] if there is some transition
    from [x] producing [o] *)
Definition enables {X:LTS} (x:lstate X) (o:option O) :=
  ∃ x′, must_step X x o x′.

(** This is the inductive presentation of the forward direction of
    choice refinement.  It is equal to the version presented in
    the main body of the thesis (up to the excluded middle). *)
Inductive choice_forward (X Y:LTS) (R:lstate X → lstate Y → Prop) (y:lstate Y) :
  lstate X → Prop :=
  | choice_forward_intro : ∀ x,
      R x y →
      (∀ o, enables x o →
        enables y o ∨ enables y None ∨
          (∃ x′, must_step X x None x′ ∧ choice_forward X Y R y x′)) →
      choice_forward X Y R y x.

(** The induction scheme for the forward direction of choice refinement. *)
Axiom choice_forward_ind
  : ∀ (X Y : LTS) (R : lstate X → lstate Y → Prop)
     (y : lstate Y) (P : lstate X → Prop),
     (∀ x : lstate X,
      R x y →
      (∀ o, enables x o →
        enables y o ∨ enables y None ∨
        ∃ x′ : lstate X, must_step X x None x′ ∧ P x′) →
      P x) → ∀ s : lstate X, choice_forward X Y R y s → P s.

(** We needed to state the inductive definition with the
    desired order of arguements reversed, so this definition flips them. *)
Definition choice_forward′ X Y R x y := choice_forward X Y R y x.

(** Choice refinement is the forward choice direction conjoined
    with the backward direction of behavioral refinement. *)
Definition choice : transition_diagram :=
  td_and choice_forward′ branch_backward_ind.

(** There is also a non-stuttering version. *)
Definition choice_no_stutter : transition_diagram :=
  td_and choice_forward′ branch_backward_ind_no_stutter.
```

```

(** The non−stuttering version is the same, once we take the fixpoint. *)
Axiom stuttering_choice_refines_eq : ∀ X Y x y,
  refines choice_no_stutter X Y x y ↔ refines choice X Y x y.

(** Behavioral choice refinement is a preorder. *)
Axiom refines_choice_refl : ∀ A x,
  refines choice A A x x.

Axiom choice_refines_trans : ∀ X Y Z x y z,
  refines choice X Y x y →
  refines choice Y Z y z →
  refines choice X Z x z.

(** Both versions are the fixpoints of their generating operators. *)
Axiom refines_choice_refinement : ∀ A B,
  refinement choice A B (refines choice A B).

Axiom refines_choice_no_stutter_refinement : ∀ A B,
  refinement choice_no_stutter A B (refines choice_no_stutter A B).

(** Behavioral refinement implies choice refinement. *)
Axiom refines_branch_ind_choice : ∀ X Y x y,
  refines branch_ind X Y x y →
  refines choice X Y x y.

(** The remaining definitions and propositions are useful for reasoning
    about undefined behavior. *)
Definition void_lts : LTS := Build_LTS Empty_set (fun _ _ _ ⇒ False).

Axiom refines_ind_mho : ∀ X Y y,
  refines branch_ind X Y (mho _) y.

Axiom refines_ind_mho_step_star : ∀ A B x,
  refines branch_ind A B x (mho _) →
  must_step_star A x (mho _).

Axiom refines_ind_mho_tau : ∀ X Y x o x′,
  refines branch_ind X Y x (mho _) →
  may_step X x o x′ →
  refines branch_ind X Y x′ (mho _).
```

```

Definition reachable' (X:LTS) : lstate X → lstate X → Prop :=
  clos_refl_trans _ (fun a b ⇒ ∃ o, may_step X a o b).

Axiom refines_ind_mho_star : ∀ X Y x x',
  refines branch_ind X Y x (mho _) →
  reachable' X x x' →
  refines branch_ind X Y x' (mho _).

Axiom choice_refines_mho_step_star : ∀ A B x,
  refines choice A B x (mho _) →
  must_step_star A x (mho _).

Axiom refines_mho_forward_no_stutter : ∀ A B C (R:lstate A → lstate B → Prop) a,
  (∀ x, refines branch_ind A C x (mho _) → R x (mho _)) →
  refines branch_ind A C a (mho _) →
  branch_forward_ind_no_stutter A B R a (mho _).

Inductive goes_wrong (X:LTS) : lstate X → Prop :=
  | go_wrong_now : goes_wrong X (mho _)
  | go_wrong_later : ∀ x,
        (∃ x', steps X x None x') →
        (∀ o x', steps X x o x' → o = None ∧ goes_wrong X x') →
        goes_wrong X (into x).

Axiom goes_wrong_ind
   : ∀ (X : LTS) (P : lstate X → Prop),
     P (mho (state X)) →
     (∀ x : state X,
       (∃ x', steps X x None x') →
      (∀ (o : option O) (x' : lift (state X)),
       steps X x o x' → o = None ∧ P x') →
      P (into x)) → ∀ l : lstate X, goes_wrong X l → P l.

Axiom refines_ind_mho_goes_wrong : ∀ A B x,
  refines branch_ind A B x (mho _) →
  goes_wrong A x.

Axiom refines_ind_goes_wrong : ∀ A B x y,
  refines branch_ind A B x y →
  goes_wrong B y → goes_wrong A x.

  Axiom prove_branch_ind_bisim : ∀ A B R,
    refinement branch_forward_ind_no_stutter A B R →
    refinement branch_forward_ind_no_stutter B A (inv R) →
    (∀ x y, R x y → goes_wrong B y → goes_wrong A x) →
    (∀ x y, R x y → goes_wrong A x → goes_wrong B y) →
    (∀ x y, R x y → bisimilar branch_ind A B x y).

End compiler_interface.

Hint Resolve @must_may_step.
End COMPILER_INTERFACE.

(** These are imports from the main proof development.  We will
    use these to impliement the module type just defined. *)
Require prelim.
Require lts_ref.
Require branch_ref.
Require choice_beh_ref.

(** The interface we defined above is literally just a subset
    of the overall development. *)
Module CompilerInterface <: COMPILER_INTERFACE.
  Include prelim.
  Include branch_ref.BRANCH_REF.
  Include choice_beh_ref.CHOICE_BEH_REF.
End CompilerInterface.

(** For some reason, Coq doesn't allow the previous module
    to be matched opaquely, but it will allow the following
    definition, which does match opaquely.  I don't know why. *)
Module CI : COMPILER_INTERFACE := CompilerInterface.
```

Chapter 9

A Case Study

Most of this thesis has, thus far, been devoted to presenting particular notions of refinement and examining their properties. Among other things, I have attempted to argue that these notions are minimal (relate as few programs as possible). However, I have thus far presented only abstract evidence arguing that these refinement notions are large enough to allow interesting program transformations. In this chapter, I will present a simple programming language that I shall use as a vehicle to argue for the usefulness of the refinements presented in the previous chapters of this thesis. In addition, two of the transformations presented in this chapter can be seen as proof-of-concept implementations of the enhancements to CompCert I suggest in chapter 10.

First, I shall present two separate operational semantics for the same language: one given in a conventional abstract machine style and another given in a compositional style. I shall prove the equivalence of these two semantics, demonstrating that the refinements I present allow considerable flexibility when rearranging computations. Although the program syntax does not change in this particular case, we can nonetheless regard passing from one operational semantics to another as a kind of program transformation. In this case, the transformation is quite significant, and demonstrates the general ability to transform computations. The additional expressive power granted by refinements do not much come into play for this transformation, but the proof at least shows that they do not get in the way.

Second, I will present a procedure for performing scope extrusion and prove it correct with respect to behavioral refinement. Recall from chapter 1 that scope extrusion was one of the troublesome program transformations we wanted to allow. Here I will concretely demonstrate how behavioral refinement allows precisely this transformation.

Third, I will show how we can specialize an unspecified expression evaluation order into a particular evaluation strategy (left-to-right evaluation, in this case). This specialization exercises the capabilities of choice refinement. The combination of the two passes can then be proved correct with respect to behavioral choice refinement, which includes the features of both sorts of refinement.

Taken together, I hope these proofs will suffice to convince the reader that behavioral and choice refinement are *useful* notions for compiler correctness, as well as being (in some sense) minimal.

9.1 The WHILE-C Language

The programming language I present here is a simple statement language with while loops and local variable declarations. This language is a modification of a standard WHILE language to contain some of the exotic features of C. I call this language "WHILE-C."

WHILE-C contains just enough features to exhibit two of the problematic program transformations discussed in the introduction. The language has only two significant syntactic sorts: expressions and statements. See figure 9.1 for the syntax of the language. There are no types, arrays, records, pointers, or dynamic allocation. The language is Turing-complete, but only barely — local variables can take on values in $\mathbb{Z}$ (rather than machine integers) which allows one to store unbounded data in a single variable.

The syntax of WHILE-C is given in figure 9.1. Throughout this chapter, I will use the metavariable v to refer to variables (members of a countable infinite set $\mathcal{V}$), n to refer to integer constants, $\oplus$ to refer to binary operators, e to refer to expressions, s to refer to statements, and o to refer to observable events. It will sometimes be useful to refer to a syntactic category that has been lifted by adding the additional wrong value $\mho$. I will add a hat to the appropriate metavariable to indicate such lifting. For example, $\hat{e}$ refers to something that may either be an expression or $\mho$. As in previous chapters, I will use metavariable α to refer to a step label, which may either be the hidden action, τ, or an observable action.

In most respects, the semantics I will give this language should be unsurprising; however, some points are worth highlighting. Notice that the available binary operations include integer division and modulus. Although it will not impact the transformations presented below in any interesting way, these operations already give us some undefined behavior. In particular, division by 0 produces undefined behavior, which will be formally modeled using the $\mho$ state.

Also notice that expressions contain assignments. This is modeled after C, where expressions may produce side-effects. The value of an assignment expression, like in C, is the value that was

Variables	v	$\in$	$\mathcal{V}$	
Numeric Constants	n	$\in$	$\mathbb{Z}$	
Binary Operators	$\oplus$	$\in$	**op**	
		$::=$	$+ \mid - \mid * \mid / \mid \mathsf{mod} \mid == \mid != \mid < \mid > \mid <= \mid >=$	
Expressions	e	$\in$	**expr**	
		$::=$	n	constant integer
			$\mid e_1 \oplus e_2$	binary operator
			$\mid e_1 \; ? \; e_2 \; : \; e_3$	conditional expression
			$\mid v$	variable reference
			$\mid v \; := \; e$	variable assignment
Statements	s	$\in$	**stmt**	
		$::=$	e	expressions
			$\mid s_1 \; ; \; s_2$	statement sequencing
			$\mid \mathsf{local}\ v\ \mathsf{in}\ s$	local variable declaration
			$\mid \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$	if/then/else
			$\mid \mathsf{while}\ e\ \mathsf{do}\ s$	while loop
Variable state	z	$\in$	**varstate**	
		$::=$	$\bot \mid \mathbb{Z}$	
Observations	o	$\in$	$\mathcal{O}$	
		$::=$	$\mathsf{val}(n) \mid \mathsf{read}(v, z) \mid \mathsf{write}(v, z)$	

Figure 9.1: Abstract syntax for WHILE-C

written. Evaluating an assignment has the side-effect of writing the value into the named variable. Further, and again like C, I will assign a semantics to expressions that leaves the order of evaluations unspecified. In the simple language presented here, this really only matters at the evaluation of binary operators, but a richer language would have other opportunities for nondeterminism, such as assigning to array lvalues and the arguments to function calls. There is no distinction between integer values and boolean values. The integer 0 is considered false, and any integer other than 0 is considered true. The comparison operators always return 0 for false and 1 for true. The conditional operator $e_1?e_2 : e_3$ behaves just like the corresponding operator in C; expression e_1 is evaluated and then either e_2 or e_3 is evaluated, depending on whether the value of e_1 was 0 or nonzero. The other expression is not evaluated, and causes no side-effects.

Notice that the local variable declaration contains a substatement; this statement is the scope of the declared variable. This is in contrast to C (the C'99 and C'11 standards in particular), where a declaration is a stand-alone statement and its scope extends as far to the right as possible.[1] The formulation I use here is technically simpler, but there is an easy translation so no expressivity is lost.[2] I impose no requirement that variables appearing in expressions be bound by local variable declarations. If an unbound variable is read or written, it is treated as an access to a global variable and produces an observable side effect (similar to the way CompCert treats access to global volatile variables). There is no declaration for global variables — every variable is considered a valid global variable unless it has been locally bound.

Finally, note that there is no skip statement. However, every expression is itself a statement, so any side-effect free expression can play the role of a skip. In particular, the constant expression 0 plays this role nicely.

As mentioned in the introduction to this chapter, I will give WHILE-C two different operational semantics. However, these semantics will share some properties in common. In particular, they will both be defined as extended labeled transition systems with the observation set as defined in figure 9.1. A $\mathsf{val}(n)$ observation is produced when a statement halts; n is an integer value obtained from evaluating the statement. A $\mathsf{read}(v, z)$ observation corresponds to a program reading global variable v and getting value z. Likewise, $\mathsf{write}(v, z)$ indicates a program has written value z into global variable v. The special symbol $\bot$ represents the indeterminate value of an uninitialized variable; reading or writing this indeterminate value results in going wrong.[3]

[1] However, note that C'90 (also known as ANSI C) declarations may only appear at the beginning of a new scope.

[2] Additional issues may arise if one wishes to consider the interaction with unstructured control flow in C, like goto or certain uses of the case statement (e.g., Duff's device). I do not consider such issues here.

[3] However, note there is no way to generate a $\mathsf{write}(v, \bot)$ event in the language.

$$\boxed{e \xrightarrow[\text{redex}]{\alpha} \hat{e}}$$

$$\frac{}{v \xrightarrow[\text{redex}]{\mathsf{read}(v,n)} n} \qquad \frac{}{v \xrightarrow[\text{redex}]{\mathsf{read}(v,\bot)} \mho} \qquad \frac{}{(v \ := \ n) \xrightarrow[\text{redex}]{\mathsf{write}(v,n)} n}$$

$$\frac{n \neq 0}{(n \ ? \ e_1 \ : \ e_2) \xrightarrow[\text{redex}]{\tau} e_1} \qquad \frac{n = 0}{(n \ ? \ e_1 \ : \ e_2) \xrightarrow[\text{redex}]{\tau} e_2} \qquad \frac{}{(n_1 \ \oplus \ n_2) \xrightarrow[\text{redex}]{\tau} \mathsf{evalop}(\oplus, n_1, n_2)}$$

$$\boxed{\mathsf{isredex}(e)}$$

$$\frac{}{\mathsf{isredex}(v)} \qquad \frac{}{\mathsf{isredex}(v \ := \ n)} \qquad \frac{}{\mathsf{isredex}(n \ ? \ e_1 \ : \ e_2)} \qquad \frac{}{\mathsf{isredex}(n_1 \ \oplus \ n_2)}$$

$$\begin{array}{llll} \text{Contexts} & C & ::= \square & \text{context hole} \\ & & \mid C \ \oplus \ e & \text{left op. context} \\ & & \mid e \ \oplus \ C & \text{right op. context} \\ & & \mid v \ := \ C & \text{assignment context} \\ & & \mid C \ ? \ e_1 : e_2 & \text{conditional context} \end{array} \qquad \begin{array}{rcl} \square[e] & \equiv & e \\ (C \ \oplus \ e')[e] & \equiv & (C[e]) \ \oplus \ e' \\ (e' \ \oplus \ C)[e] & \equiv & e' \ \oplus \ (C[e]) \\ (v \ := \ C)[e] & \equiv & v \ := \ (C[e]) \\ (C \ ? \ e_1 : e_2)[e] & \equiv & (C[e]) \ ? \ e_1 : e_2 \end{array}$$

$$\begin{array}{llll} \text{Expression states} & es & \in & \textbf{exprstate} \\ & & ::= & \mathsf{done} \ \mid \ e \ \mid \ (C, \ e) \end{array}$$

$$\boxed{es \xrightarrow{\tau} \hat{es}}$$

$$\frac{e = C[e'] \qquad \mathsf{isredex}(e')}{e \xrightarrow{\tau} (C, \ e')} \qquad \frac{e \xrightarrow[\text{redex}]{\alpha} e'}{(C, \ e) \xrightarrow{\alpha} C[e']} \qquad \frac{e \xrightarrow[\text{redex}]{\alpha} \mho}{(C, \ e) \xrightarrow{\alpha} \mho} \qquad \frac{}{n \xrightarrow{\mathsf{val}(n)} \mathsf{done}}$$

Figure 9.2: Expression semantics

9.1.1 Expression Semantics

Both the abstract machine semantics and the compositional semantics I present below rely on the same semantics for expressions. Conceptually, the expression semantics are a straightforward nondeterministic small-step semantics. Evaluation proceeds by nondeterministically selecting some redex and reducing it. Another redex is selected, etc, until the expression finally is reduced to just a constant. Some redexes will produce observable events when evaluated, so the nondeterminism in expression evaluation is observable at a fine-grain level beyond just the effect nondeterminism may have on the eventual value. See figure 9.2 for the rules defining how to reduce redexes. The judgment $e \xrightarrow[\text{redex}]{\alpha} e'$ means that the redex e reduces in one step to e' producing observable α, which may either be an actual event in $\mathcal{O}$, or it may be the hidden event τ, representing silent computation.

In the redex rule for binary operations, the function $\mathsf{evalop}(\oplus, n_1, n_2)$ computes the value of the binary operation $\oplus$ when applied to arguments n_1 and n_2. The details are entirely unsurprising, and I will not include them here, except to note as before that division by 0 results in $\mho$.

The evaluation contexts define where it is permitted to select a redex. The contexts include both sides of a binary operator, the rvalue of an assignment, and the conditional value of a conditional expression. The function $C[e]$ means to plug the expression e into the evaluation context C. This function is defined by the clauses appearing in figure 9.2.

Now we are ready to define the ELTS for expression evaluation. There are three states expression evaluation can be in. The done state is a placeholder state that is entered after we emit the $\mathsf{val}(n)$ observation that signals we have completed evaluation. An expression e indicates that we are ready to select the next redex. A pair (C, e) indicates we have selected redex e out of the context C and are ready to reduce the redex. Note that it takes two steps to reduce a redex. First, the expression is decomposed into a context and the redex; then the redex is reduced in a second step. One might think that we could compress this down and perform both the selection and reduction in one step. However, this is not a good idea! Doing so leads to a system with mixed nondeterminism, where it is no longer clear who is making which decisions. With the system as stated, it is clear that which redex to select next is always an internal choice. However, if we tried to do everything in one step, the environment would also have a hand in selecting what redex gets reduced next. This is not what we want to model. Thus it is important that redex selection and reduction occur in separate steps. Otherwise, we would be unable to prove that the specialization to a determinstic evaluation order produces a choice refinement.

The reader may be surprised to see that *every* variable read and write in the expression semantics behaves like a global variable, producing an observable event. This is not a bug; it is a feature. There is no way, just from examining the syntax of an expression, to tell if a variable is local or global. One must examine the entire statement context of an expression to determine that. By writing the expression semantics to treat every variable as though it were global, we push off the details of how to handle local variable binding to the statement semantics, where such details ought to be handled. We are able to do this because side effects are communicated via LTS observations rather than, e.g., causing state updates in a local variable store. In the next two sections, we will see two different ways of handling these issues.

9.1.2 Abstract Machine Semantics

In this section I will present a small-step semantics of WHILE-C statements that is roughly based on the operational semantics of Cminor, as used by the Verified Software Toolchain [App11], which goes back to Appel and Blazy's formulation [AB07]. Current CompCert versions use a similar style of operational semantics for all the "structured" languages in the stack: CompCert C, Clight, C#minor and Cminor [Ler09b].

This semantics is a standard presentation of an abstract machine for an imperative language. The states of the operational semantics are *configurations* that contain information about what expressions and statements to evaluate, a memory that maps locations to values, and a local variable environment that maps variable names to locations. The memory is represented by a function from $\mathbb{N}$ to $\mathsf{varstate}$, together with a list of previously allocated locations, the domain of the memory. The machine begins execution with the domain empty and every location mapped to the indeterminate value $\bot$. Throughout execution, we maintain the invariant that every location not in the domain maps to $\bot$. When we enter the scope of a new local variable, we (nondeterministically) select a fresh location in the memory and bind the variable name to that location in the local variable environment; thus uninitialized local variables map to $\bot$, as expected. When we exit the scope of a local variable, the name is unbound, but the memory is left unchanged. The local variable environment is managed in a stack discipline. When the scope of a local variable is entered, a new binding is pushed onto the top of the stack; this binding is popped when the scope is exited. Variable lookups go from the top of the stack downward — this gives the expected behavior when local variable bindings are shadowed by inner bindings.

The main data structure that manages program control flow is the *control continuation*. It is a stack of directives telling the machine what to do next, once the current unit of work is completed. The machine alternates between evaluating expressions and manipulating the control stack. When the machine needs to evaluate an expression (e.g., because it appears directly as a statement, or as the test expression in a while statement), it uses the expression semantics defined in the previous section as a subroutine. Reads and writes to variables generated by the expression evaluation are either passed through (because the variable is global) or intercepted and used to manipulate the memory (for local variables). The rules that intercept local reads and writes convert the external nondeterminism in expression evaluation into hidden *deterministic* computation that manipulates the configuration memory. Figure 9.3 contains the definition of configurations and figures 9.4 and 9.5 define the step relation for the abstract machine. Let $\mathcal{M}$ represent the abstract machine transition system so defined. The the initial configuration for a statement s is defined by:

$$\mathit{initconfig}(s) \equiv \langle [\,],\ \lambda \ell.\ \bot,\ [\,],\ [\mathsf{kstmt}\ s],\ 0\rangle. \tag{9.1}$$

Control continuations	κ	$\in$	$\mathbf{kctl}$
		$::=$	$\mathsf{kstmt}\ s$
			$\mid \mathsf{kwhile}\ e\ s$
			$\mid \mathsf{kif}\ s_1\ s_2$
			$\mid \mathsf{kenv}\ v$
Memory domains	*dom*	$\in$	$\mathsf{list}\ \mathbb{N}$
Memories	*mem*	$\in$	$\mathbb{N} \to \mathbf{varstate}$
Variable environements	*env*	$\in$	$\mathsf{list}\ (\mathcal{V} \times \mathbb{N})$
Control stack	*stk*	$\in$	$\mathsf{list}\ \mathbf{kctl}$
Machine configurations	M	$\in$	$\mathsf{list}\ \mathbb{N}\ \times\ (\mathbb{N} \to \mathbf{varstate})\ \times\ \mathsf{list}\ (\mathcal{V} \times \mathbb{N})\ \times$ $\mathsf{list}\ \mathbf{kctl}\ \times \mathbf{exprstate}$

Figure 9.3: WHILE-C machine configurations

Then the ELTS process that gives the semantics of a statement s is given by:

$$\langle \mathcal{M}, \mathit{initconfig}(s) \rangle \tag{9.2}$$

The machine runs by alternating between evaluating expressions and manipulating the control stack. The rules of figure 9.4 are concerned with evaluating expressions. Each rule "calls" into the expression evaluation transition system to reduce expressions. Silent reductions are carried out silently, and reads and writes to variables not appearing in the local variable environment produce the same observables as the expression evaluation transition. When an expression wants to read a local variable, the value is instead looked up in the configuration memory, and the observable is converted to a silent action. Likewise when a local variable is assigned, the configuration memory is updated and a silent action is produced. For each situation, there are two separate rules depending on whether the expression evaluation goes wrong or not. In the formal proof, these distinct cases are handled together in one case at the expense of some auxiliary definitions.

The rules of figure 9.5 only fire when the current expression has been reduced to a value, and which rule is selected depends on what is on the control stack. If the stack is empty then the machine terminates, outputting the final value. If the top element of the control stack is a kwhile, then we have been evaluating the test expression of a while loop. If the expression is false (that is, 0), then we pop the while loop and continue down the stack. If the test is true, we push a new instance of the loop body and test expression onto the stack. If the top element of the stack is a kif, we have been evaluating the test expression of an if/then/else command, and we select either the then

$$\mathsf{lookup} \in \big(\mathsf{list}\ (\mathcal{V} \times \mathbb{N}) \times \mathcal{V}\big) \to \mathsf{option}\ \mathbb{N}$$

$$\begin{aligned}
\mathsf{lookup}([\,], v) &\equiv \mathsf{None} \\
\mathsf{lookup}([(v', n)] \bullet l, v) &\equiv \begin{cases} \mathsf{Some}\ n & v = v' \\ \mathsf{lookup}(l, v) & v \neq v' \end{cases}
\end{aligned}$$

$\boxed{M \xrightarrow{\alpha} \hat{M}}$

$$\frac{es \xrightarrow{\tau} es'}{\langle dom,\ mem,\ env,\ stk,\ es\rangle \xrightarrow{\tau} \langle dom,\ mem,\ env,\ stk,\ es'\rangle}\ \text{expr compute}$$

$$\frac{es \xrightarrow{\tau} \mho}{\langle dom,\ mem,\ env,\ stk,\ es\rangle \xrightarrow{\tau} \mho}\ \text{expr compute } \mho$$

$$\frac{es \xrightarrow{\mathsf{read}(v,z)} es' \qquad \mathsf{lookup}(env, v) = \mathsf{None}}{\langle dom,\ mem,\ env,\ stk,\ es\rangle \xrightarrow{\mathsf{read}(v,z)} \langle dom,\ mem,\ env,\ stk,\ es'\rangle}\ \text{global read}$$

$$\frac{es \xrightarrow{\mathsf{read}(v,z)} \mho \qquad \mathsf{lookup}(env, v) = \mathsf{None}}{\langle dom,\ mem,\ env,\ stk,\ es\rangle \xrightarrow{\mathsf{read}(v,z)} \mho}\ \text{global read } \mho$$

$$\frac{es \xrightarrow{\mathsf{read}(v,z)} es' \qquad \mathsf{lookup}(env, v) = \mathsf{Some}\ i \qquad mem(i) = z}{\langle dom,\ mem,\ env,\ stk,\ es\rangle \xrightarrow{\tau} \langle dom,\ mem,\ env,\ stk,\ es'\rangle}\ \text{local read}$$

$$\frac{es \xrightarrow{\mathsf{read}(v,z)} \mho \qquad \mathsf{lookup}(env, v) = \mathsf{Some}\ i \qquad mem(i) = z}{\langle dom,\ mem,\ env,\ stk,\ es\rangle \xrightarrow{\tau} \mho}\ \text{local read } \mho$$

$$\frac{es \xrightarrow{\mathsf{write}(v,z)} es' \qquad \mathsf{lookup}(env, v) = \mathsf{None}}{\langle dom,\ mem,\ env,\ stk,\ es\rangle \xrightarrow{\mathsf{write}(v,z)} \langle dom,\ mem,\ env,\ stk,\ es'\rangle}\ \text{global write}$$

$$\frac{es \xrightarrow{\mathsf{write}(v,z)} \mho \qquad \mathsf{lookup}(env, v) = \mathsf{None}}{\langle dom,\ mem,\ env,\ stk,\ es\rangle \xrightarrow{\mathsf{write}(v,z)} \mho}\ \text{global write } \mho$$

$$\frac{es \xrightarrow{\mathsf{write}(v,z)} es' \qquad \mathsf{lookup}(env, v) = \mathsf{Some}\ i}{\langle dom,\ mem,\ env,\ stk,\ es\rangle \xrightarrow{\tau} \langle dom,\ mem[i \mapsto z],\ env,\ stk,\ es'\rangle}\ \text{local write}$$

$$\frac{es \xrightarrow{\mathsf{write}(v,z)} \mho \qquad \mathsf{lookup}(env, v) = \mathsf{Some}\ i}{\langle dom,\ mem,\ env,\ stk,\ es\rangle \xrightarrow{\tau} \mho}\ \text{local write } \mho$$

Figure 9.4: WHILE-C abstract machine semantics

$$\frac{}{\langle \mathit{dom},\ \mathit{mem},\ \mathit{env},\ [\,],\ n\rangle \xrightarrow{\mathsf{val}(n)} \langle \mathit{dom},\ \mathit{mem},\ \mathit{env},\ [\,],\ \mathsf{done}\rangle}\ \text{terminate}$$

$$\frac{n = 0}{\langle \mathit{dom},\ \mathit{mem},\ \mathit{env},\ [\mathsf{kwhile}\ e\ s] \bullet \mathit{stk},\ n\rangle \xrightarrow{\tau} \langle \mathit{dom},\ \mathit{mem},\ \mathit{env},\ \mathit{stk},\ n\rangle}\ \text{while false}$$

$$\frac{n \neq 0}{\begin{array}{l}\langle \mathit{dom},\ \mathit{mem},\ \mathit{env},\ [\mathsf{kwhile}\ e\ s] \bullet \mathit{stk},\ n\rangle \xrightarrow{\tau} \\ \qquad \langle \mathit{dom},\ \mathit{mem},\ \mathit{env},\ [\mathsf{kstmt}\ s, \mathsf{kstmt}\ e, \mathsf{kwhile}\ e\ s] \bullet \mathit{stk},\ n\rangle\end{array}}\ \text{while true}$$

$$\frac{n = 0}{\begin{array}{l}\langle \mathit{dom},\ \mathit{mem},\ \mathit{env},\ [\mathsf{kif}\ s_1\ s_2] \bullet \mathit{stk},\ n\rangle \xrightarrow{\tau} \\ \qquad \langle \mathit{dom},\ \mathit{mem},\ \mathit{env},\ [\mathsf{kstmt}\ s_2] \bullet \mathit{stk},\ n\rangle\end{array}}\ \text{if false}$$

$$\frac{n \neq 0}{\begin{array}{l}\langle \mathit{dom},\ \mathit{mem},\ \mathit{env},\ [\mathsf{kif}\ s_1\ s_2] \bullet \mathit{stk},\ n\rangle \xrightarrow{\tau} \\ \qquad \langle \mathit{dom},\ \mathit{mem},\ \mathit{env},\ [\mathsf{kstmt}\ s_1] \bullet \mathit{stk},\ n\rangle\end{array}}\ \text{if true}$$

$$\frac{}{\begin{array}{l}\langle \mathit{dom},\ \mathit{mem},\ [(v, l)] \bullet \mathit{env},\ [\mathsf{kenv}\ v] \bullet \mathit{stk},\ n\rangle \xrightarrow{\tau} \\ \qquad \langle \mathit{dom},\ \mathit{mem},\ \mathit{env},\ \mathit{stk},\ n\rangle\end{array}}\ \text{pop env}$$

$$\frac{}{\langle \mathit{dom},\ \mathit{mem},\ \mathit{env},\ [\mathsf{kstmt}\ e] \bullet \mathit{stk},\ n\rangle \xrightarrow{\tau} \langle \mathit{dom},\ \mathit{mem},\ \mathit{env},\ \mathit{stk},\ e\rangle}\ \text{stmt expr}$$

$$\frac{}{\begin{array}{l}\langle \mathit{dom},\ \mathit{mem},\ \mathit{env},\ [\mathsf{kstmt}\ (s_1;\ s_2)] \bullet \mathit{stk},\ n\rangle \xrightarrow{\tau} \\ \qquad \langle \mathit{dom},\ \mathit{mem},\ \mathit{env},\ [\mathsf{kstmt}\ s_1,\ \mathsf{kstmt}\ s_2] \bullet \mathit{stk},\ n\rangle\end{array}}\ \text{stmt seq}$$

$$\frac{l \notin \mathit{dom}}{\begin{array}{l}\langle \mathit{dom},\ \mathit{mem},\ \mathit{env},\ [\mathsf{kstmt}\ (\mathsf{local}\ v\ \mathsf{in}\ s)] \bullet \mathit{stk},\ n\rangle \xrightarrow{\tau} \\ \qquad \langle [l] \bullet \mathit{dom},\ \mathit{mem},\ [(v, l)] \bullet \mathit{env},\ [\mathsf{kstmt}\ s,\ \mathsf{kenv}\ v] \bullet \mathit{stk},\ n\rangle\end{array}}\ \text{stmt local}$$

$$\frac{}{\begin{array}{l}\langle \mathit{dom},\ \mathit{mem},\ \mathit{env},\ [\mathsf{kstmt}\ (\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2)] \bullet \mathit{stk},\ n\rangle \xrightarrow{\tau} \\ \qquad \langle \mathit{dom},\ \mathit{mem},\ \mathit{env},\ [\mathsf{kif}\ s_1\ s_2] \bullet \mathit{stk},\ e\rangle\end{array}}\ \text{stmt ifte}$$

$$\frac{}{\begin{array}{l}\langle \mathit{dom},\ \mathit{mem},\ \mathit{env},\ [\mathsf{kstmt}\ (\mathsf{while}\ e\ \mathsf{do}\ s)] \bullet \mathit{stk},\ n\rangle \xrightarrow{\tau} \\ \qquad \langle \mathit{dom},\ \mathit{mem},\ \mathit{env},\ [\mathsf{kwhile}\ e\ s] \bullet \mathit{stk},\ e\rangle\end{array}}\ \text{stmt while}$$

Figure 9.5: WHILE-C abstract machine semantics (continued)

branch or the else branch, depending on the value of the expression. If the top stack element is a kenv, then we are exiting the scope of a local variable, so we pop the top element off the local variable environment and continue evaluating the stack. It is an invariant of the abstract machine that the variables listed in kenv will match the top name in the local variable environment. Because WHILE-C has no pointers, the memory location associated with the popped name is now garbage and can never be referenced again.

The remaining rules handle the final possibility, which occurs when the top element of the control stack is a kstmt. To determine what to do, we must examine the contained statement, so there is one rule for each statement form. Most of these are completely straightforward, but the rule for the local statement deserves particular mention. The rule for local allocates a new memory location not already in the domain of the memory and adds it to the domain. It also pushes a new entry onto the local variable environment, binding the name to that location. Finally, it pushes the inner statement to be evaluated on top of the kenv which will later pop variable binding. Notice that the location l is chosen nondeterministically — any location not already in the memory domain is acceptable. The domain is a finite list, so some suitable memory location can always be found. Also notice that the memory is not updated; it continues to contain the indeterminate value $\bot$. Thus, new local variables begin their lifetime uninitialized.

Invariants. Although not specified directly in the definition of states, the abstract machine maintains a number of invariants that turn out to be important later. Informally, these are: the control stack remains in sync with the environment stack; all addresses in the environment are in the memory domain; the environment stack contains no duplication of addresses; and all addresses not in the memory domain contain $\bot$.

The predicate defining agreement between the control stack and the environment is defined by the rules below:

$$\boxed{\mathsf{agree}(\mathit{stk}, \mathit{env})}$$

$$\frac{}{\mathsf{agree}([\,], [\,])} \qquad \frac{\mathsf{agree}(\mathit{stk}, \mathit{env})}{\mathsf{agree}([\mathsf{kenv}\ v] \bullet \mathit{stk}, [(v, l)] \bullet \mathit{env})}$$

$$\frac{\mathsf{agree}(\mathit{stk}, \mathit{env})}{\mathsf{agree}([\mathsf{kstmt}\ s] \bullet \mathit{stk}, \mathit{env})} \qquad \frac{\mathsf{agree}(\mathit{stk}, \mathit{env})}{\mathsf{agree}([\mathsf{kif}\ s_1\ s_2] \bullet \mathit{stk}, \mathit{env})} \qquad \frac{\mathsf{agree}(\mathit{stk}, \mathit{env})}{\mathsf{agree}([\mathsf{kwhile}\ e\ s] \bullet \mathit{stk}, \mathit{env})}$$

Essentially agree just requires that the kenvs on the control stack mention all the variables in the environment in the correct order. This property is maintained because the environment is handled via a stack discipline and because variable scope is rigidly defined by the lexical structure of statements.

The invariants of the abstract machine are formally defined by the *invariants* predicate below:

$$\begin{aligned} &\mathit{invariants}(\langle \mathit{dom}, \mathit{mem}, \mathit{env}, \mathit{stk}, \mathit{es}\rangle) \equiv \\ &\qquad \mathsf{agree}(\mathit{stk}, \mathit{env}) \\ &\qquad \wedge\ \mathit{nodups}(\mathit{env}) \\ &\qquad \wedge\ \big(\forall (v, l) \in \mathit{env}.\ l \in \mathit{dom}\big) \\ &\qquad \wedge\ \big(\forall l \notin \mathit{dom}.\ \mathit{mem}(l) = \bot\big) \end{aligned} \tag{9.3}$$

The *nodups* predicate simply asserts that the *env* stack never duplicates addresses. The initial state of the abstract machine satisfies *invariants* and every step maintains the invariants.

9.1.3 A Compositional Semantics

In this section, I will present an alternate semantics for statements. It will be an operational semantics, like the abstract machine; however, this semantics will be defined compositionally over the program syntax. Thus it is *also* a denotational semantics. The main idea is that each statement corresponds to an operation that “glues together” the ELTSs of its substatements into a larger ELTS that represents the statement as a whole. In the next section, I will show that the two semantics are $\cong_{\mho\Delta B}$-equivalent.

Note on formalization. *In the formal proof, I everywhere use the inductive versions refinements and bisimulations that are presented in chapter 8. However, in this chapter I will use the notation for the primary versions discussed in preceding chapters. That is, I will mention $\cong_{\mho\Delta B}$ instead of $\cong_{\mho IB}$, etc. The notions are equivalent up to the axiom of the excluded middle, so this notational fib is well-justified.*

The meaning of a statement s is given by $[\![s]\!]$, an ELTS process defined by recursion on the structure of statements.

Definition 9.1.1 (Denotation of statements)**.**

$$[\![e]\!] \equiv \langle \mathcal{E}, e\rangle \tag{9.4}$$

$$[\![s_1;\ s_2]\!] \equiv \mathit{seq}([\![s_1]\!], [\![s_2]\!]) \tag{9.5}$$

$$[\![\mathsf{local}\ x\ \mathsf{in}\ s]\!] \equiv \mathit{local}(x, [\![s]\!]) \tag{9.6}$$

$$[\![\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2]\!] \equiv \mathit{ifte}([\![e]\!], [\![s_1]\!], [\![s_2]\!]) \tag{9.7}$$

$$[\![\mathsf{while}\ e\ \mathsf{do}\ s]\!] \equiv \mathit{while}([\![e]\!], [\![s]\!]) \tag{9.8}$$

Here $\mathcal{E}$ refers to the expression semantics defined in §9.1.1. Obviously, this definition is not especially illuminating unless one also knows the definitions of *seq*, *local*, etc. The formal definitions will come shortly; but first, I will provide some intuition for how each operator works.

The simplest is the process $\mathit{seq}(a, b)$. It begins by simulating the process a, beginning at its initial state. When (and if) process a produces a $\mathsf{val}(n)$ observable, $\mathit{seq}(a, b)$ produces a τ and steps to the initial state of b. It then simulates b until it produces a $\mathsf{val}(n)$ observable, which it then produces itself. In other words, $\mathit{seq}(a, b)$ simulates a until it terminates (if ever), and then simulates b; the eventual value produced by b (if any) is reproduced.

The other sequential control operators (while and if/then/else) are similar. While first simulates the test expression process. If it produces a 0 value, the while process terminates producing 0. Otherwise it simulates its body and then loops back to the beginning. If/then/else evaluates its test expression process and then selects which branch to simulate based on the produced value. For all the sequential control operators, any read or write observables produced by any subprocess are directly reproduced, ensuring side-effects take place as they should.

The $\mathit{local}(x, a)$ process is of a rather different character. Basically, we want to simulate the process a, *except that* we want to intercept any reads or writes targeting the variable x and handle them locally. This is handled by simulating the process a *in parallel* with a small helper process that maintains the state of the variable x. Internal communication between a and the x variable process are then hidden from the surrounding context. This construction is quite similar to the way one might use a π-calculus process to simulate a mutable variable.

In order to reduce later proof burden, I define all the sequential control operators as instances of a single, generic construction. Sequencing, if/then/else, and while loops all emerge by appropriate setting of the parameters. This design has the pleasant property that adding further sequential control operators (e.g., do/while, for loops, one-sided if statements, structured case statements, etc.) would be completely straightforward. The correctness of scope extrusion (cf. §9.3) is proved generically with respect to arbitrary sequential control structures, so that proof would also be easy to adapt.

The basic idea is that control structures are in one of several *control states*; each control state is associated with an ELTS process. When the control structure enters a new control state, it begins simulating the associated ELTS. When (and if) the simulated process halts, the return value determines which control state to enter next. Each control structure has a distinguished start state and termination state. The control states and the connections between them are basically a control-flow graph for the given control structure. Nesting control structures inside each other gives rise to

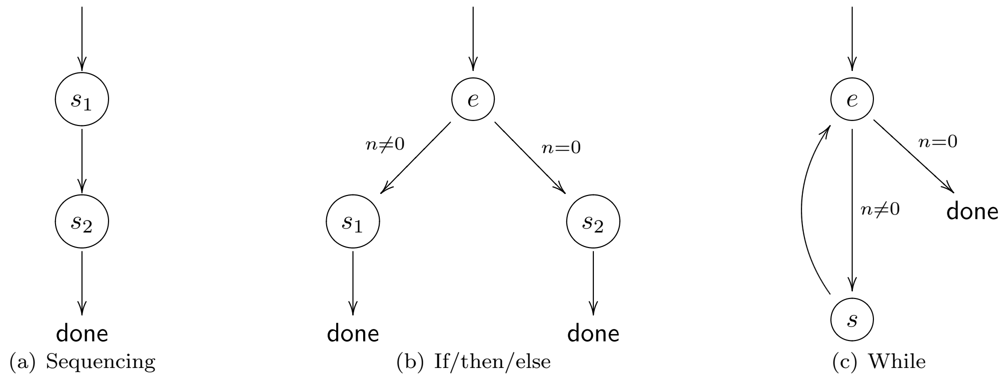


Figure 9.6: Sequential control graphs

more complicated control-flow graphs.

To define a control structure, one must provide a set $\mathcal{C}$ of control states, a distinguished state $\mathit{init} \in \mathcal{C}$, and a transfer function $\mathit{transfer} : (\mathcal{C} \times \mathbb{Z}) \to (\mathcal{C} + \mathbb{Z})$. When a simulated ELTS halts (producing n) in control state c, the value of $\mathit{transfer}(c, n)$ says what to do next. If *transfer* returns a new state c', then control transfers to that control state; if it returns a value n', then that value is emitted and the control structure halts. Finally, one must give a function $\mathcal{L}(c)$ that assigns an ELTS process to each control state c. Let $\mathcal{S}(c)$ be a function that associates the state space of $\mathcal{L}(c)$ to c. Then the states of the control structure are dependent pairs of controls states and states from the simulated ELTS corresponding to that control state, or a distinguished terminated state:

$$\mathcal{S}_{\mathsf{ctl}} \equiv \{\mathsf{done}\} \cup (\Sigma c : \mathcal{C}.\ \mathcal{S}(c)) \tag{9.9}$$

Let t range over $\mathcal{S}_{\mathsf{ctl}}$. Then the step relation of a control structure is defined by these rules:

$\boxed{t \xrightarrow{\alpha} \hat{t}}$

$$\frac{x \xrightarrow{\alpha} x' \qquad \alpha \neq \mathsf{val}(-)}{\langle c, x\rangle \xrightarrow{\alpha} \langle c, x'\rangle} \qquad \frac{x \xrightarrow{\alpha} \mho \qquad \alpha \neq \mathsf{val}(-)}{\langle c, x\rangle \xrightarrow{\alpha} \mho}$$

$$\frac{x \xrightarrow{\mathsf{val}(n)} x' \qquad \mathit{transfer}(c, n) = c'}{\langle c, x\rangle \xrightarrow{\tau} \langle c', \mathcal{L}(c')\rangle} \qquad \frac{x \xrightarrow{\mathsf{val}(n)} x' \qquad \mathit{transfer}(c, n) = n'}{\langle c, x\rangle \xrightarrow{\mathsf{val}(n')} \mathsf{done}}$$

The initial state of the control structure is then $\langle \mathit{init}, \mathcal{L}(\mathit{init})\rangle$. Let $\mathit{ctl}(\mathcal{C}, \mathit{transfer}, \mathcal{L})$ refer to the control structure process generated in this way.

The diagrams in figure 9.6 show the parameters needed to define sequencing, if/then/else and while loops. Formally, we define the control states and transfer functions as below:

$$\mathcal{C}_{\text{seq}} \equiv \{A, B\} \qquad \mathit{transfer}_{\text{seq}}(A, n) \equiv B$$

$$\mathit{transfer}_{\text{seq}}(B, n) \equiv n$$

$$\mathcal{C}_{\text{ifte}} \equiv \{E, A, B\} \qquad \mathit{transfer}_{\text{ifte}}(E, n) \equiv \begin{cases} A & n \neq 0 \\ B & n = 0 \end{cases}$$

$$\mathit{transfer}_{\text{ifte}}(A, n) \equiv n$$

$$\mathit{transfer}_{\text{ifte}}(B, n) \equiv n$$

$$\mathcal{C}_{\text{while}} \equiv \{E, A\} \qquad \mathit{transfer}_{\text{while}}(E, n) \equiv \begin{cases} A & n \neq 0 \\ 0 & n = 0 \end{cases}$$

$$\mathit{transfer}_{\text{while}}(A, n) \equiv E$$

Then we can define the statement operators as:

$$\mathit{seq}(a, b) \equiv \mathit{ctl}(\mathcal{C}_{\text{seq}}, \mathit{transfer}_{\text{seq}}, [A \mapsto a, B \mapsto b]) \tag{9.10}$$

$$\mathit{ifte}(e, a, b) \equiv \mathit{ctl}(\mathcal{C}_{\text{ifte}}, \mathit{transfer}_{\text{ifte}}, [E \mapsto e, A \mapsto a, B \mapsto b]) \tag{9.11}$$

$$\mathit{while}(e, a) \equiv \mathit{ctl}(\mathcal{C}_{\text{while}}, \mathit{transfer}_{\text{while}}, [E \mapsto e, A \mapsto a]) \tag{9.12}$$

Thus, the generic control structure construction gives the definitions of *seq*, *ifte*, and *while*. All that remains is to define *local*. As I mentioned above, the local variable statement is built by putting the substatement in parallel with a gadget that keeps track of the state of the local variable.

Given a variable name v, we build the variable gadget ELTS for that variable (called $\mathit{var}(v)$) with **varstate** as the state space and with the following rules defining the transition relation:

$$\boxed{z \xrightarrow{\alpha} \hat{z}}$$

$$\frac{}{\bot \xrightarrow{\mathsf{write}(v,\bot)} \mho} \qquad \frac{}{n \xrightarrow{\mathsf{write}(v,n)} n}$$

$$\frac{}{z \xrightarrow{\mathsf{read}(v,\bot)} \mho} \qquad \frac{}{z \xrightarrow{\mathsf{read}(v,n)} n}$$

The initial state of a new variable gadget is the indeterminate value $\bot$.

Notice there is an apparent inversion of sense relating to the observables. The write observable is produced when we look up the value of the variable, and a read observable is produced when we set the value of the variable. This is because the variable gadget communicates via the complementary observables to the program. When placed in parallel, complementary events synchronize, so the gadget needs to communicate with a write event when the program is performing a variable read and vice versa.

With the variable gadget defined, all we need to do now is define the parallel operator. What I actually will define below is a parallel operator with hiding. It would be more general to define separate parallel and hiding operators; however, I have cut a corner here to make things a little easier on myself.

Before defining the parallel operator, I first need to make precise the notion of complementary observables I mentioned above. There is little more to it than already discussed; the rules below define complementary observables at a variable v:

$\boxed{\mathsf{complement}(v, o_1, o_2)}$

$$\frac{}{\mathsf{complement}\big(v, \mathsf{read}(v,n), \mathsf{write}(v,n)\big)} \qquad \frac{}{\mathsf{complement}\big(v, \mathsf{write}(v,n), \mathsf{read}(v,n)\big)}$$

Notice that the $\mathsf{val}(n)$ observation has no complement. $\mathsf{complement}(v,\alpha,\beta)$ only holds if α and β are a read/write pair on the variable v.

Given two ELTSs $\mathcal{L}_1$ and $\mathcal{L}_2$ and a variable v, we define the parallel composition (with hiding on v) using pairs of lifted states from $\mathcal{L}_1$ and $\mathcal{L}_2$ as the state space: $\mathcal{S}_{\mathsf{par}} \equiv \mathcal{S}_1^{\mho} \times \mathcal{S}_2^{\mho}$. Let p range over states in $\mathcal{S}_{\mathsf{par}}$. The rules below define the parallel stepping operation:

$\boxed{p \xrightarrow{\alpha} \hat{p}}$

$$\frac{a \xrightarrow{\alpha} a' \qquad a \neq \mho \qquad v\#\alpha}{(a,b) \xrightarrow{\alpha} (a',b)} \qquad \frac{b \xrightarrow{\beta} b' \qquad b \neq \mho \qquad v\#\beta}{(a,b) \xrightarrow{\beta} (a,b')} \qquad \frac{a = \mho \ \vee \ b = \mho}{(a,b) \xrightarrow{\tau} \mho}$$

$$\frac{a \xrightarrow{\alpha} a' \qquad b \xrightarrow{\beta} b' \qquad a \neq \mho \qquad b \neq \mho \qquad \mathsf{complement}(v,\alpha,\beta)}{(a,b) \xrightarrow{\tau} (a',b')}$$

The notation $v\#\alpha$ means that the variable v does not appear in the event α. In particular α is not a read or a write on the variable v. The initial state of the parallel operator is the pair containing the initial states of the subordinate transition systems. Let $par(v,a,b)$ represent the ELTS process that puts a in parallel with b while hiding on v.

Aside. *When it comes to defining a parallel operator over ELTSs, it turns out that there are some nontrivial design decisions to make regarding how to handle $\mho$. The version I have defined above is what I call the "delayed failure" semantics. This is because there may be an indefinite delay between when one of the subprocesses goes wrong and when the overall parallel process goes wrong; indeed,* *the parallel process may decide to diverge while running the other process or even to safely halt (!) and never notice that one side has gone wrong. The two readily apparent alternatives are "eager failure" and "isolated failure." In eager failure, the entire parallel process goes wrong as soon as either subprocess goes wrong; no delay is allowed. In isolated failure, the overall process goes wrong only if* both *subprocesses go wrong.*

The delayed failure semantics is a bit strange; certainly it defies my intuition about how to combine the failure of parallel processes. It seems, however, that eager failure does not behave as well as delayed failure; I was unable to prove that the parallel operator with eager failure is a congruence for $\sqsubseteq_{\mho\Delta B}$*. For the particular semantics I am interested in, both eager and delayed failure semantics give the same result because the variable state gadget will always go wrong at the same moment as the statement it is paired with, so the issue is not especially pressing.*

Isolated failure seems like a viable alternative that might be more suited than delayed failure to modeling systems that have strong process isolation, like a pure message passing system or distributed computing settings (hence the name).

Now we have everything we need to define the *local* operator, which combines the variable gadget with the parallel operator:

$$local(v, a) \equiv par(v, \langle var(v), \bot\rangle, a) \tag{9.13}$$

This is the last of the statement operators we needed to define the semantics of statements.

Congruence. When defining a denotational semantics based on bisimulation methods, it is not enough just to give a denotation function defined on the structure of the syntax. One must also show that the operator corresponding to each syntactic construct is a *congruence*. That is, if equivalent subterms are placed in identical contexts, the overall programs are equivalent. We will also be interested in congruence with respect to refinements — this will allow us to replace a program fragment with a refinement and know that the overall program is itself refined.

It is tedious, but not especially difficult, to verify that the operators defined above are congruences for $\cong_{\mho\Delta B}$, $\sqsubseteq_{\mho\Delta B}$ and $\sqsubseteq_{\mho CB}$.

Proposition 9.1.2. *Suppose $\mathcal{C}$ is a set of control state and transfer is a control state transfer function. Further, let $\mathcal{L}_1(-)$ and $\mathcal{L}_2(-)$ be functions from control states to processes. Then, for* *each relation* $\sim\, \in \{\cong_{\mho\Delta B}, \sqsubseteq_{\mho\Delta B}, \sqsubseteq_{\mho C B}\}$, *the following implication holds:*

$$\big(\forall c \in \mathcal{C}.\ \mathcal{L}_1(c) \sim \mathcal{L}_2(c)\big) \qquad \textit{implies} \qquad \mathit{ctl}(\mathcal{C}, \mathit{transfer}, \mathcal{L}_1) \sim \mathit{ctl}(\mathcal{C}, \mathit{transfer}, \mathcal{L}_2) \tag{9.14}$$

Proposition 9.1.3. *For each relation* $\sim\, \in \{\cong_{\mho\Delta B}, \sqsubseteq_{\mho\Delta B}, \sqsubseteq_{\mho C B}\}$, *the following implication holds:*

$$a \sim b \quad \textit{and} \quad c \sim d \qquad \textit{implies} \qquad \mathit{par}(v, a, c) \sim \mathit{par}(v, b, d) \tag{9.15}$$

These facts are sufficient to show that all the statement operators are congruences.

Proposition 9.1.4. *For each relation* $\sim\, \in \{\cong_{\mho\Delta B}, \sqsubseteq_{\mho\Delta B}, \sqsubseteq_{\mho C B}\}$, *the following implications hold:*

$$a \sim b \quad \textit{and} \quad c \sim d \qquad \textit{implies} \qquad \mathit{seq}(a, c) \sim \mathit{seq}(b, d) \tag{9.16}$$

$$e_1 \sim e_2 \quad \textit{and} \quad a \sim b \quad \textit{and} \quad c \sim d \qquad \textit{implies} \qquad \mathit{ifte}(e_1, a, c) \sim \mathit{ifte}(e_2, b, d) \tag{9.17}$$

$$a \sim b \qquad \textit{implies} \qquad \mathit{local}(v, a) \sim \mathit{local}(v, b) \tag{9.18}$$

$$e_1 \sim e_2 \quad \textit{and} \quad a \sim b \qquad \textit{implies} \qquad \mathit{while}(e_1, a) \sim \mathit{while}(e_2, b) \tag{9.19}$$

These congruence facts will be important later when we prove the correctness of scope extrusion; they will allow us to do scope extrusion via a series of bottom-up program rewrites. They are also key to the proof that specializing to a deterministic evaluation order for expressions produces a choice refinement.

9.2 Equivalence of the Semantics

I have now presented an operational semantics of statements in two significantly different styles. The first is a conventional presentation via an abstract machine. The second is a compositional semantics where the shape of program states is determined by the syntax of the program itself. Both styles have their advantages. The abstract machine semantics is very concrete, which makes it easier (in some ways) to understand, and it seems to lend itself well to global program transformations. On the other hand, the compositional semantics is considerably more well-suited to reasoning about local program transformations, as we shall see in the section on scope extrusion.

In this section, I will show that (if we expend enough effort) we can have our cake and eat it too. Although given in quite different styles, the abstract machine and the compositional semantics are $\cong_{\mho\Delta B}$-equivalent. This means we can freely use whichever semantics we prefer to prove some transformation; because the semantics are equivalent we can pass between in either direction without any loss. Of course, nothing comes for free — the proof that these two semantics are equivalent is long, tedious and fairly difficult, even for such a simple language. In this section, I will give the high-level ideas behind the proof; as always, consult the proof scripts for all the gory details.

To prove $\cong_{\mho\Delta B}$-equivalence, we must first state a relation between the state spaces of the processes we want to relate and prove that it is a $\mho$-refinement in both directions. Because the state-space of the compositional semantics is defined by recursion on statements, the bisimulation relation will have to be as well. Given a statement s, we define a relation $\mathit{matchstate}(s, M, x)$ between configurations M and states x from the state-space of the process $[\![s]\!]$.

$$\begin{aligned}
&\mathit{matchstate}(e, \langle \mathit{dom}, \mathit{mem}, \mathit{env}, \mathit{stk}, e_1\rangle, e_2) \equiv\\
&\qquad (\mathit{stk} = [\mathsf{kstmt}\ e] \wedge (\exists n, e_1 = n) \wedge e_2 = e)\\
&\qquad \vee\ (\mathit{stk} = [\,] \wedge \mathit{env} = [\,] \wedge e_1 = e_2)
\end{aligned}$$

$$\begin{aligned}
&\mathit{matchstate}(s_1; s_2, \langle \mathit{dom}, \mathit{mem}, \mathit{env}, \mathit{stk}, e\rangle, \langle A, a\rangle) \equiv\\
&\qquad \big(\mathit{stk} = [\mathsf{kstmt}\ (s_1; s_2)] \wedge (\exists n, e = n) \wedge a = [\![s_1]\!]\big)\\
&\qquad \vee\ \big(\exists \mathit{stk}'.\ \mathit{stk} = \mathit{stk}' \bullet [\mathsf{kstmt}\ s_2] \wedge \mathit{matchstate}(s_1, \langle \mathit{dom}, \mathit{mem}, \mathit{env}, \mathit{stk}', e\rangle, a)\big)
\end{aligned}$$

$$\begin{aligned}
&\mathit{matchstate}(s_1; s_2, \langle \mathit{dom}, \mathit{mem}, \mathit{env}, \mathit{stk}, e\rangle, \langle B, b\rangle) \equiv\\
&\qquad \mathit{matchstate}(s_2, \langle \mathit{dom}, \mathit{mem}, \mathit{env}, \mathit{stk}, e\rangle, b)
\end{aligned}$$

$$\mathit{matchstate}(s_1; s_2, \langle \mathit{dom}, \mathit{mem}, \mathit{env}, \mathit{stk}, e\rangle, \mathsf{done}) \equiv \big(\mathit{stk} = [\,] \wedge e = \mathsf{done}\big)$$

$$\begin{aligned}
&\mathit{matchstate}(\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2, \langle \mathit{dom}, \mathit{mem}, \mathit{env}, \mathit{stk}, e_1\rangle, \langle E, e_2\rangle) \equiv\\
&\qquad \big(\mathit{stk} = [\mathsf{kstmt}\ (\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2)] \wedge (\exists n, e_1 = n) \wedge e_2 = e\big)\\
&\qquad \vee\ \big(\mathit{stk} = [\mathsf{kif}\ s_1\ s_2] \wedge e_1 = e_2\big)
\end{aligned}$$

$$\begin{aligned}
&\mathit{matchstate}(\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2, \langle \mathit{dom}, \mathit{mem}, \mathit{env}, \mathit{stk}, e\rangle, \langle A, a\rangle) \equiv\\
&\qquad \mathit{matchstate}(s_1, \langle \mathit{dom}, \mathit{mem}, \mathit{env}, \mathit{stk}, e\rangle, a)
\end{aligned}$$

$$\begin{aligned}
&\mathit{matchstate}(\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2, \langle \mathit{dom}, \mathit{mem}, \mathit{env}, \mathit{stk}, e\rangle, \langle B, b\rangle) \equiv\\
&\qquad \mathit{matchstate}(s_2, \langle \mathit{dom}, \mathit{mem}, \mathit{env}, \mathit{stk}, e\rangle, b)
\end{aligned}$$

$$\mathit{matchstate}(\textsf{if } e \textsf{ then } s_1 \textsf{ else } s_2, \langle \mathit{dom}, \mathit{mem}, \mathit{env}, \mathit{stk}, e\rangle, \textsf{done}) \equiv \big(\mathit{stk} = [\,] \wedge e = \textsf{done}\big)$$

$$\begin{aligned}
&\mathit{matchstate}(\textsf{while } e \textsf{ do } s, \langle \mathit{dom}, \mathit{mem}, \mathit{env}, \mathit{stk}, e_1\rangle, \langle E, e_2\rangle) \equiv \\
&\quad \big(\mathit{stk} = [\textsf{kstmt } (\textsf{while } e \textsf{ do } s)] \wedge (\exists n, e_1 = n) \wedge e_2 = e\big) \\
&\quad \vee\ \big(\mathit{stk} = [\textsf{kwhile } e\ s] \wedge e_1 = e_2\big) \\
&\quad \vee\ \big(\mathit{stk} = [\,] \wedge e_1 = e_2 = 0\big)
\end{aligned}$$

$$\begin{aligned}
&\mathit{matchstate}(\textsf{while } e \textsf{ do } s, \langle \mathit{dom}, \mathit{mem}, \mathit{env}, \mathit{stk}, e\rangle, \langle A, a\rangle) \equiv \\
&\quad \exists \mathit{stk}'.\ \mathit{stk} = \mathit{stk}' \bullet [\textsf{kstmt } e, \textsf{kwhile } e\ s] \wedge \mathit{matchstate}(s, \langle \mathit{dom}, \mathit{mem}, \mathit{env}, \mathit{stk}', e\rangle, a)
\end{aligned}$$

$$\mathit{matchstate}(\textsf{while } e \textsf{ do } s, \langle \mathit{dom}, \mathit{mem}, \mathit{env}, \mathit{stk}, e\rangle, \textsf{done}) \equiv \big(\mathit{stk} = [\,] \wedge e = \textsf{done}\big)$$

$$\begin{aligned}
&\mathit{matchstate}(\textsf{local } v \textsf{ in } s, \langle \mathit{dom}, \mathit{mem}, \mathit{env}, \mathit{stk}, e\rangle, (z, a)) \equiv \\
&\quad \big(\mathit{stk} = [\textsf{kstmt } (\textsf{local } v \textsf{ in } s)] \wedge (\exists n.\ e = n) \wedge a = [\![s]\!] \wedge z = \bot\big) \\
&\quad \vee \Big(\exists \mathit{stk}'\ \mathit{env}'\ l.\ \mathit{stk} = \mathit{stk}' \bullet [\textsf{kenv } v] \wedge \mathit{env} = \mathit{env}' \bullet [(v, l)] \wedge \\
&\qquad \Big(\big(a = \mho \wedge \langle \mathit{dom}, \mathit{mem}, \mathit{env}, \mathit{stk}, e\rangle \sqsubseteq_{\mho\Delta B} \mho\big) \\
&\qquad\quad \vee \big(z \neq \mho \wedge a \neq \mho \wedge \mathit{mem}(l) = z \wedge \mathit{matchstate}(s, \langle \mathit{dom}, \mathit{mem}, \mathit{env}', \mathit{stk}', e\rangle, a)\big)\Big)\Big) \\
&\quad \vee \big(\mathit{stk} = [\,] \wedge (\exists n.\ e = n \vee e = \textsf{done}) \wedge \\
&\qquad\quad z \neq \mho \wedge a \neq \mho \wedge \mathit{matchstate}(s, \langle \mathit{dom}, \mathit{mem}, \mathit{env}, \mathit{stk}, e\rangle, a)\big)
\end{aligned}$$

Now we are ready to state the main results of this section.

Lemma 9.2.1. *Let R_s be a family of relations indexed by statements s and defined by:*

$$\begin{aligned}
R_s \equiv \Big\{(\hat{M}, \hat{a}) \mid \quad & \big(\hat{M} = \mho \wedge \hat{a} \sqsubseteq_{\mho\Delta B} \mho\big) \\
\vee\ & \big(\hat{a} = \mho \wedge \hat{M} \sqsubseteq_{\mho\Delta B} \mho\big) \\
\vee\ & \big(\hat{a} \neq \mho \wedge \hat{M} \neq \mho \wedge \mathit{invariants}(\hat{M}) \wedge \mathit{matchstate}(s, \hat{M}, \hat{a})\big)\Big\}
\end{aligned} \tag{9.20}$$

For each s, R_s is an inductive pre-branching $\mho$-refinement. Likewise, R_s^{-1} is an inductive pre-branching $\mho$-refinement.

Proof. The main branch of the proof (where each side is non-$\mho$) direction goes by induction on s. Each inductive case is a significant proof obligation in its own right. I elide the lengthy, but largely routine, subproofs. □

Lemma 9.2.2. *For all statements s,*

$$\mathit{matchstate}(s, \mathit{initconfig}(s), [\![s]\!]). \tag{9.21}$$

Proof. By induction on s. Each case is straightforward from the definitions. □

Theorem 9.2.3. *For each statement s,*

$$\langle \mathcal{M}, \mathit{initconfig}(s)\rangle \quad \cong_{\mho\Delta B} \quad [\![s]\!]. \tag{9.22}$$

Proof. R_s from lemma 9.2.1 is an inductive pre-branching $\mho$-bisimulation, so its stuttering closure $\mathsf{ST}(R_s)$ is an inductive branching $\mho$-bisimulation. Lemma 9.2.2 suffices to show $(\mathit{initconfig}(s), [\![s]\!]) \in \mathsf{ST}(R_s)$, which completes the proof. □

Full abstraction? Is it fair to characterize the result in this section as a full abstraction result? The answer depends a little bit on how much one is willing to stretch the definitions involved. If one takes "full abstraction" very widely to mean an equivalence between some operational semantics and some denotational semantics, then yes, this result is a full abstraction result.

However, I think such an interpretation stretches the meaning of the term too far. Full abstraction, rendered a little more formally, usually means that programs with distinct denotations can be placed in some *closing* program context such that the combined programs have different observable behavior. The restriction to closing contexts is the critical factor that, in my accounting, makes my equivalence result different from a full abstraction result.

To see why, consider the following program fragment:

$$x := 5; \; x := 5$$

This statement writes the value 5 into the variable x twice in succession. According to the semantics of the previous two sections, this program is quite distinct from the following, which only writes to x once:

$$x := 5$$

However, no *closing* context of WHILE-C can distinguish these two programs because: the only binding operation in WHILE-C creates local variables, and writing to a local variable is an idempotent operation; and WHILE-C is a purely sequential programming language, so no program can interleave between the two writes and observe a difference between the programs that way.

To get a full abstraction result (as they are usually understood) one would have to increase the distinguishing power of program contexts by adding additional constructs like: a variable binding construct for which writing is not idempotent (e.g., channels); or parallel composition. It is not immediately clear what constructs would suffice. In the alternative, one could weaken the denotational semantics in such a way that the two program fragments above are indistinguishable.

Before moving on, it is worthwhile to point out that the above considerations about full abstraction are not just theoretical musings. An optimizing compiler might well eliminate redundant stores, justifying such a translation by the fact that no program context can distinguish. However, if done injudiciously, such a translation will be incorrect in the presence of shared-memory multithreading or if applied to locations corresponding to memory-mapped I/O ports (for which storing may cause side-effects).

This latter concern is especially interesting because there is no way (to my knowledge) to write a race-free C program that distinguishes between the above program fragments. However, if the variable `x` is declared as a volatile variable, the C specification requires that both writes be retained as-is. This is because memory-mapped hardware (which `volatile` variables abstract) *can* distinguish them. Thus, the C spec requires compilers to make more distinctions than can be explained by closing program contexts. Such a situation occurs whenever one decides that the external environment has powers beyond those of the constructs of the programming language under consideration. In such situations, a full abstraction result is hopeless.

9.3 Correctness of Scope Extrusion

In order to do scope extrusion correctly, we must sometimes do variable renaming in order to avoid variable capture. Therefore, a nontrivial portion of the correctness of scope extrusion involves proving, along the way, the correctness of α-conversion for locally bound variables.

To help get a handle on the variable-binding issues that arise, I adopt some techniques inspired by the nominal approach to variable binding [GP02, Pit03]. In contrast to the work by Gabbay and Pitts, I develop only the bare essentials of the theory necessary to carry out the present proofs and my treatment of support is somewhat different.

The primary idea from nominal logic I will be using is the idea of a “nominal type.” A nominal type is just some set (or type) equipped with: an equivalence relation, $\approx$; a variable permutation operator, $\cdot$; and a support operator, supp. If σ is a variable permutation, and x is some element of a nominal type, then $\sigma \cdot x$ is the application of the permutation σ to x. We write $\mathsf{supp}(x)$ to mean the support of x.

The idea behind the variable permutation operator is that it applies some permutation to all the variables appearing in an object — both bound and free variables are subject to permutations. The support of an object is a set of variables that are "significant" to that object. Support is a generalization of the set of free variables.

One of the important innovations of the nominal approach is the idea both syntactic objects and *semantic* objects can be nominal types. In other words, the ideas of permutation operators and support are sufficiently abstract that they can apply just as well to the denotation of a program as to its syntax. In this approach, it makes sense to state and prove statements like $\sigma \cdot [\![s]\!] \approx [\![\sigma \cdot s]\!]$. Basically, this formula means that if we permute the variables in a program and then calculate its denotation, we get the same thing as if we first calculate the denotation, and then modify the denotation by the permutation.

Definition 9.3.1 (Variable permutations)**.** *A variable permutation is a bijection from the set of variables to itself. More precisely, it is a pair of functions* $\langle f, g \rangle$ *with* $f, g : \mathcal{V} \to \mathcal{V}$ *where* $f \circ g = \mathsf{id}$ *and* $g \circ f = \mathsf{id}$*. I use the metavariable* σ *to range over permutations. I write* $\sigma \circ \sigma'$ *for the composition of permutations,* σ^{-1} *for the inverse of a permutation and* 1 *for the identity permutation. Let* $v_1 \leftrightarrow v_2$ *be the permutation that swaps* v_1 *with* v_2 *and leaves all other variables unaffected.*

Definition 9.3.2 (Nominal types)**.** *Let* X *be a set of of values. Let* $\approx$ *be an equivalence relation, let* $\cdot : \mathsf{perm} \times X \to X$ *be a permutation application operator, and let* $\mathsf{supp} : X \to \wp(\mathbf{var})$ *be a support operator. Then* $\langle X, \approx, \cdot, \mathsf{supp} \rangle$ *is a nominal type if it satisfies the following:*

$$1 \cdot x = x \tag{9.23}$$

$$(\sigma \circ \sigma') \cdot x = \sigma \cdot (\sigma' \cdot x) \tag{9.24}$$

$$(\forall v \in \mathsf{supp}(x).\ \sigma(v) = v) \qquad \textit{implies} \qquad \sigma \cdot x \approx x \tag{9.25}$$

$$v \in \mathsf{supp}(x) \qquad \textit{iff} \qquad \sigma(v) \in \mathsf{supp}(\sigma \cdot x) \tag{9.26}$$

When x *and* y *are elements drawn from (possibly distinct) nominal types, we write* $x \# y$ *to mean that* x *and* y *have disjoint support. That is,* $x \# y$ *iff* $\mathsf{supp}(x) \cap \mathsf{supp}(y) = \emptyset$.

Definition 9.3.3 (Nominal variables)**.** *Variables are endowed with the obvious nominal structure,* *using = for the equivalence relation:*

$$\sigma \cdot v \equiv \sigma(v) \qquad \mathsf{supp}(v) = \{v\}$$

I make this definition primarily so that I can write down $v\#x$, which means that v is not in the support of x.

Definition 9.3.4 (Nominal expressions)**.** *The following definitions supply a nominal structrure to the expressions of WHILE-C, using = for equivalence:*

$$\begin{aligned}
\sigma \cdot n &\equiv n \\
\sigma \cdot (e_1 \ \oplus \ e_2) &\equiv (\sigma \cdot e_1) \ \oplus \ (\sigma \cdot e_2) \\
\sigma \cdot (e_1 \ ? \ e_2 \ : \ e_3) &\equiv (\sigma \cdot e_1) \ ? \ (\sigma \cdot e_2) \ : \ (\sigma \cdot e_3) \\
\sigma \cdot v &\equiv \sigma(v) \\
\sigma \cdot (v := e) &\equiv \sigma(v) := (\sigma \cdot e)
\end{aligned}$$

$$\begin{aligned}
\mathsf{supp}(n) &\equiv \emptyset \\
\mathsf{supp}(e_1 \ \oplus \ e_2) &\equiv \mathsf{supp}(e_1) \cup \mathsf{supp}(e_2) \\
\mathsf{supp}(e_1 \ ? \ e_2 \ : \ e_3) &\equiv \mathsf{supp}(e_1) \cup \mathsf{supp}(e_2) \cup \mathsf{supp}(e_3) \\
\mathsf{supp}(v) &\equiv \{v\} \\
\mathsf{supp}(v := e) &\equiv \{v\} \cup \mathsf{supp}(e)
\end{aligned}$$

The required properties of $\cdot$ and supp *are easily proved by induction.*

Definition 9.3.5 (Nominal statements)**.** *The following definitions supply a nominal structure to the statements of WHILE-C using $\approx_\alpha$ (defined below) for equivalence.*

$$\begin{aligned}
\sigma \cdot (s_1; s_2) &\equiv (\sigma \cdot s_1; \sigma \cdot s_2) \\
\sigma \cdot (\mathsf{local}\ v\ \mathsf{in}\ s) &\equiv \mathsf{local}\ \sigma(v)\ \mathsf{in}\ (\sigma \cdot s) \\
\sigma \cdot (\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2) &\equiv \mathsf{if}\ (\sigma \cdot e)\ \mathsf{then}\ (\sigma \cdot s_1)\ \mathsf{else}\ (\sigma \cdot s_2) \\
\sigma \cdot (\mathsf{while}\ e\ \mathsf{do}\ s) &\equiv \mathsf{while}\ (\sigma \cdot e)\ \mathsf{do}\ (\sigma \cdot s)
\end{aligned}$$

$$\begin{aligned}
\mathsf{supp}(s_1; s_2) &\equiv \mathsf{supp}(s_1) \cup \mathsf{supp}(s_2) \\
\mathsf{supp}(\mathsf{local}\ v\ \mathsf{in}\ s) &\equiv \mathsf{supp}(s) - \{v\} \\
\mathsf{supp}(\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2) &\equiv \mathsf{supp}(e) \cup \mathsf{supp}(s_1) \cup \mathsf{supp}(s_2) \\
\mathsf{supp}(\mathsf{while}\ e\ \mathsf{do}\ s) &\equiv \mathsf{supp}(e) \cup \mathsf{supp}(s)
\end{aligned}$$

Definition 9.3.6 (α-equivalence). *Two statements x and y are α-equivalent (written $x \approx_\alpha y$) if there exists a permutation σ such that:*

$$\sigma(v) = v \quad \textit{for all} \quad v \in \mathsf{supp}(x) \tag{9.27}$$

$$\sigma \cdot x = y \tag{9.28}$$

Basically, two statements are α-equivalent if the bound variables of one can be permuted so that the resulting statement is equivalent to the other. Using the properties of nominal types and permutations, one can show that $\approx_\alpha$ is an equivalence relation.

Definition 9.3.7 (Nominal observations). *The following definitions supply the observable events of WHILE-C with a nominal structure, using $=$ for equivalence.*

$$\begin{array}{ll} \sigma \cdot (\mathsf{val}(n)) \equiv \mathsf{val}(n) & \mathsf{supp}(\mathsf{val}(n)) \equiv \emptyset \\ \sigma \cdot (\mathsf{read}(v,z)) \equiv \mathsf{read}(\sigma(v),z) & \mathsf{supp}(\mathsf{read}(v,z)) \equiv \{v\} \\ \sigma \cdot (\mathsf{write}(v,z)) \equiv \mathsf{write}(\sigma(v),z) & \mathsf{supp}(\mathsf{write}(v,z)) \equiv \{v\} \end{array}$$

So far, all the nominal definitions we have seen have been entirely straightforward definitions over syntactic objects. Next, however, we are going to provide nominal structures over labeled transition systems and LTS processes.

Definition 9.3.8 (Nominal τELTS). *Suppose $\mathcal{L}$ is a τELTS with actions from $\mathcal{O}$, and let σ be a variable permutation Then we can build a new LTS, $\mathcal{L}'$ with the same state space as $\mathcal{L}$, and with the transition relation defined below:*

$$\frac{x \xrightarrow{\alpha}_{\mathcal{L}} x'}{x \xrightarrow{\sigma\cdot\alpha}_{\mathcal{L}'} x'}$$

Then the nominal structure is given by the following, using $=$ as equivalence.

$$\sigma \cdot \mathcal{L} \equiv \mathcal{L}' \tag{9.29}$$

$$\mathsf{supp}(\mathcal{L}) \equiv \{v \mid \exists x\ \alpha\ x'.\ x \xrightarrow{\alpha} x' \land v \in \mathsf{supp}(\alpha)\} \tag{9.30}$$

Basically, applying a permutation to an LTS just permutes all the variables in the transition labels. The support of an LTS is all the variables appearing in any transition; however, this is usually much too large to be of any use. In particular, note that the support of the expression evaluation LTS includes every variable. Instead, we are usually only interested in the variables appearing in the labels of transitions that can actually be reached from some particular state. This is what the nominal structure on LTS processes gives us.

Definition 9.3.9 (Nominal τELTS processes)**.** *Suppose $\langle \mathcal{L}, x\rangle$ is a τELTS process. Then, the nominal structure on $\langle \mathcal{L}, x\rangle$ is given by the following, using $\cong_{\mho\Delta B}$ for equivalence.*

$$\sigma \cdot \langle \mathcal{L}, x\rangle \equiv \langle \sigma \cdot \mathcal{L}, x\rangle \tag{9.31}$$

$$\mathsf{supp}\big(\langle \mathcal{L}, x\rangle\big) \equiv \big\{v \mid \exists x' \ \alpha \ x''.\ x \longrightarrow^* x' \wedge x' \xrightarrow{\alpha} x'' \wedge v \in \mathsf{supp}(\alpha)\big\} \tag{9.32}$$

The notation $x \longrightarrow^* x'$ means that x' is reachable from x by some sequence of τ or observable steps.

Note that the nominal structure over processes moves to using $\cong_{\mho\Delta B}$ for equivalence, rather than $=$. This is necessary because we have restricted the support set to include only variables occurring in reachable actions. A permutation might well change variables in some unreachable part of the LTS, so bisimilarity is the best equivalence we can get.

Now that we have the nominal framework in place, we can state and prove a stronger congruence fact for local variables. This version of the congruence lemma will allow us to rename variables.

Proposition 9.3.10. *Suppose v_1 and v_2 are variables, and p_1 and p_2 are τELTS processes with $v_1 \# v_2$, $v_1 \# p_2$ and $v_2 \# p_1$. Then, for each relation $\sim \in \{\cong_{\mho\Delta B}, \sqsubseteq_{\mho\Delta B}, \sqsubseteq_{\mho CB}\}$, the following implication holds:*

$$p_1 \sim (v_1 \leftrightarrow v_2) \cdot p_2 \qquad \textit{implies} \qquad \textit{local}(v_1, p_1) \sim \textit{local}(v_2, p_2) \tag{9.33}$$

Basically, this lemma tells us that if p_1 is refined by p_2 (up to swapping the variables v_1 and v_2), then when the variables are bound separately, the two programs are in the refinement relation. Note that this proof is done entirely at the level of τELTSs — no program syntax is involved.

Now we are ready to prove some important facts about the denotations of statements. The first relates the support of a statement to the support of its denotation, and the second shows that the denotation function respects the application of permutations.

Proposition 9.3.11. *For all statements s,*

$$\mathsf{supp}(\llbracket s \rrbracket) \subseteq \mathsf{supp}(s). \tag{9.34}$$

Proof. By induction on s. In the base case we need to show that the actions of the expression evaluation LTS only contain actions that appear in the syntax of the evaluated expression. In the inductive cases, we need to show that the actions produced by the larger construct are generated by actions of inner constructs. □

This proposition lets us know that all the variables which might occur in reachable transitions are variables appearing free in s. In some cases, the inclusion is strict; for example, a variable might appear free in dead code, which can never actually be executed.

Proposition 9.3.12. *For all permutations σ and statements s,*

$$[\![\sigma \cdot s]\!] \cong_{\mho\Delta B} \sigma \cdot [\![s]\!] \tag{9.35}$$

Proof. By induction on s. At each stage, we need to show that permutations push down through each of the denotation operators for statements, and we need to use congruence properties of proposition 9.1.4 to apply the induction hypothesis. In the case for local, we need to use the strengthened congruence from proposition 9.3.10 if the variable is changed by the permutation. $\square$

Corollary.

$$s_1 \approx_\alpha s_2 \qquad \textit{implies} \qquad [\![s_1]\!] \cong_{\mho\Delta B} [\![s_2]\!]$$

Now we are just about ready to move on to the heart of the scope extrusion proof. The main idea of the proof is that we want to prove a series of one-step local scope extrusion lemmas. That is, we will show that local variable declarations can be "pulled out" of all the other sequential control constructs (subject to variable freshness conditions). At the end, I will show that we can combine the one-step rewrites into a global program rewrite that pulls all the local variable declarations to the top level.

To properly state the one-step scope extrusion lemmas, we need some auxiliary concepts.

Definition 9.3.13 (Indeterminite crashing)**.** *We say $\mathcal{L}$ has indeterminate crashing if, whenever $x \xrightarrow{o} x'$ where o is a read or write operation that uses the indeterminate value $\bot$, it is the case that $x' \sqsubseteq_{\mho\Delta B} \mho$.*

Basically, this means that if the program encounters the indeterminate value, it must eventually go wrong.

Definition 9.3.14 (Read receptive)**.** *We say $\mathcal{L}$ is read-receptive if, whenever $x \xrightarrow{\mathsf{read}(v,z)} x'$, then for all z' there exists x'' such that $x \xrightarrow{\mathsf{read}(v,z')} x''$.*

A read-receptive LTS is one that treats read observations like input; whenever it stands ready to accept some read action on variable v, it stands ready to accept *every* read action on v.

The expression evaluation LTS has indeterminate crashing and is read-receptive. Furthermore, each of the statement constructors preserves each of these properties. Thus, the denotation of every statement has them.

Proposition 9.3.15. *For each statement s, $[\![s]\!]$ has indeterminate crashing and is read-receptive.*

The core of the scope extrusion argument is the following proposition, which states that local declarations can be pulled out of each statement constructor.

Proposition 9.3.16. *Each of the following holds, subject to the assumption that each named process has indeterminate crashing and is read-receptive.*

$$v\#q \quad \textit{implies} \quad \textit{seq}(\textit{local}(v,p),q) \sqsubseteq_{\mho\triangle B} \textit{local}(v,\textit{seq}(p,q)) \tag{9.36}$$

$$v\#p \quad \textit{implies} \quad \textit{seq}(p,\textit{local}(v,q)) \sqsubseteq_{\mho\triangle B} \textit{local}(v,\textit{seq}(p,q)) \tag{9.37}$$

$$v\#e \quad \textit{and} \quad v\#q \quad \textit{implies} \quad \textit{ifte}(e,\textit{local}(v,p),q) \sqsubseteq_{\mho\triangle B} \textit{local}(v,\textit{ifte}(e,p,q)) \tag{9.38}$$

$$v\#e \quad \textit{and} \quad v\#p \quad \textit{implies} \quad \textit{ifte}(e,p,\textit{local}(v,q)) \sqsubseteq_{\mho\triangle B} \textit{local}(v,\textit{ifte}(e,p,q)) \tag{9.39}$$

$$v\#e \quad \textit{implies} \quad \textit{while}(e,\textit{local}(v,p)) \sqsubseteq_{\mho\triangle B} \textit{local}(v,\textit{while}(e,p)) \tag{9.40}$$

All the refinements above are instances of a generic refinement proof that applies to all sequential control operators. Using the fact that the support of statements includes the support of their denotation, we can then prove the following version of the above proposition that references syntax.

Proposition 9.3.17.

$$v\#s_2 \quad \textit{implies} \quad [\![(\mathsf{local}\ v\ \mathsf{in}\ s_1);s_2]\!] \sqsubseteq_{\mho\triangle B} [\![\mathsf{local}\ v\ \mathsf{in}\ (s_1;s_2)]\!] \tag{9.41}$$

$$v\#s_1 \quad \textit{implies} \quad [\![s_1;(\mathsf{local}\ v\ \mathsf{in}\ s_2)]\!] \sqsubseteq_{\mho\triangle B} [\![\mathsf{local}\ v\ \mathsf{in}\ (s_1;s_2)]\!] \tag{9.42}$$

$$\begin{aligned} v\#e \quad \textit{and} \quad v\#s_2 \quad \textit{implies} \quad & [\![\mathsf{if}\ e\ \mathsf{then}\ (\mathsf{local}\ v\ \mathsf{in}\ s_1)\ \mathsf{else}\ s_2]\!] \\ & \sqsubseteq_{\mho\triangle B} \quad [\![\mathsf{local}\ v\ \mathsf{in}\ (\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2)]\!] \end{aligned} \tag{9.43}$$

$$\begin{aligned} v\#e \quad \textit{and} \quad v\#s_1 \quad \textit{implies} \quad & [\![\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ (\mathsf{local}\ v\ \mathsf{in}\ s_2)]\!] \\ & \sqsubseteq_{\mho\triangle B} \quad [\![\mathsf{local}\ v\ \mathsf{in}\ (\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2)]\!] \end{aligned} \tag{9.44}$$

$$\begin{aligned} v\#e \quad \textit{implies} \quad & [\![\mathsf{while}\ e\ \mathsf{do}\ (\mathsf{local}\ v\ \mathsf{in}\ s)]\!] \\ & \sqsubseteq_{\mho\triangle B} \quad [\![\mathsf{local}\ v\ \mathsf{in}\ (\mathsf{while}\ e\ \mathsf{do}\ s)]\!] \end{aligned} \tag{9.45}$$

I believe all the above statements, except the statement for while, could be strengthened to use $\cong_{\mho\triangle B}$ rather than $\sqsubseteq_{\mho\triangle B}$. For scope extrusion, we need only the refinement version, so I have not attempted the stronger proofs.

To prove the correctness of scope extrusion, we need one final auxiliary concept: the binding of a sequence of variables.

Definition 9.3.18 (Multiple variable binding)**.** *Given a list of variables l, let* $\mathsf{local}\ l\ \mathsf{in}\ s$ *be syntax sugar for the process of binding each of the variables in l in order.*

$$\mathsf{local}\ [\,]\ \mathsf{in}\ s \equiv s \tag{9.46}$$

$$\mathsf{local}\ ([v] \bullet l)\ \mathsf{in}\ s \equiv \mathsf{local}\ v\ \mathsf{in}\ (\mathsf{local}\ l\ \mathsf{in}\ s) \tag{9.47}$$

Until now, we have treated variables as an abstract set. However, the algorithm for scope extrusion explicitly exploits the fact that variables are isomorphic to the natural numbers (in fact, in the formal proof, they are simply defined to be the natural numbers). The algorithm starts by first calculating the set of variables appearing in a statement and then finds the largest variable, in terms of their enumeration. Now, whenever the scope extrusion algorithm needs to choose a fresh variable, it just chooses the next variable in the sequence. This is a simple way to ensure that the algorithm always chooses a fresh variable.

The main body of the algorithm takes a variable cutoff and statement containing local variable declarations; it returns a new cutoff, a list of variables and a new statement not containing any local variable declarations. If we bind all the variables in the list at the top of the new statement, we get a refinement of the original statement.

Definition 9.3.19 (Main scope extrusion algorithm)**.** *The following clauses define the main body of the scope extrusion algorithm by recursion on the shape of statements.*

$$\mathit{extrude}(v_0,\ e) \equiv (v_0, [\,], e) \tag{9.48}$$

$$\begin{aligned}\mathit{extrude}(v_0,\ s_1; s_2) \equiv\ & \mathsf{let}\ (v_1, l_1, s_1') = \mathit{extrude}(v_0, s_1)\ \mathsf{in} \\ & \mathsf{let}\ (v_2, l_2, s_2') = \mathit{extrude}(v_1, s_2)\ \mathsf{in} \\ & \quad (v_2,\ l_1 \bullet l_2,\ s_1'; s_2')\end{aligned} \tag{9.49}$$

$$\begin{aligned}\mathit{extrude}(v_0,\ \mathsf{local}\ x\ \mathsf{in}\ s) \equiv\ & \mathsf{let}\ (v_1, l_1, s') = \mathit{extrude}(v_0, s)\ \mathsf{in} \\ & \quad (1 + v_1,\ [v_1] \bullet l_1,\ (x \leftrightarrow v_1) \cdot s')\end{aligned} \tag{9.50}$$

$$\begin{aligned} \mathit{extrude}(v_0,\ \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2) \equiv\ & \mathsf{let}\ (v_1, l_1, s_1') = \mathit{extrude}(v_0, s_1)\ \mathsf{in} \\ & \mathsf{let}\ (v_2, l_2, s_2') = \mathit{extrude}(v_1, s_2)\ \mathsf{in} \\ & \quad (v_2,\ l_1 \bullet l_2,\ \mathsf{if}\ e\ \mathsf{then}\ s_1'\ \mathsf{else}\ s_2') \end{aligned} \tag{9.51}$$

$$\begin{aligned} \mathit{extrude}(v_0,\ \mathsf{while}\ e\ \mathsf{do}\ s) \equiv\ & \mathsf{let}\ (v_1, l_1, s') = \mathit{extrude}(v_0, s)\ \mathsf{in} \\ & \quad (v_1,\ l_1,\ \mathsf{while}\ e\ \mathsf{do}\ s') \end{aligned} \tag{9.52}$$

With the exception of the case for local, each case of the above algorithm just passes into each substatement recursively, collecting together the extruded variables and passing along the allocation cutoff variable. In the case for local, we first extrude the variables of the substatement. Then, the current cutoff variable is used as a new fresh variable, and is swapped into the statement in place of the original variable. Finally, this new variable is placed in the list of extruded variables and the cutoff is incremented.

Proposition 9.3.20. *Let s be a statement and v_0 be a variable where $\forall x \in \mathsf{supp}(s).\ x < v_0$. Let $(v', l, s') = \mathsf{extrude}(v_0, s)$. Then all of the following hold:*

$$v_0 \leq v' \tag{9.53}$$

$$\forall x \in l.\ v_0 \leq x < v' \tag{9.54}$$

$$\forall x \in \mathsf{supp}(s').\ x \in l \vee x \in \mathsf{supp}(s) \tag{9.55}$$

$$s' \textit{ contains no instances of } \mathsf{local} \tag{9.56}$$

$$[\![s]\!] \sqsubseteq_{\mho \Delta B} [\![\mathsf{local}\ l\ \mathsf{in}\ s']\!] \tag{9.57}$$

Proof. The proof proceeds by induction on s. Much of the proof is just bookkeeping to make sure we can satisfy the necessary variable freshness side-conditions. To prove the interesting clause (equation 9.57) we rely on the local scope extrusion properties from proposition 9.3.17, the congruence properties from proposition 9.1.4 and the variable-swapping congruence for local variable binding from proposition 9.3.10. □

Finally, we complete the algorithm by first calculating the starting value of the cutoff variable and show the correctness of the overall pass.

Definition 9.3.21 (Scope extrusion)**.** *The final scope extrusion algorithm is defined as below:*

$$\mathit{scopeExtrusion}(s) \ \equiv\ \mathsf{let}\ (v', l, s') = \mathit{extrude}\big(1 + \mathsf{max}\big(\mathsf{supp}(s)\big),\ s\big)\ \mathsf{in}\ (l, s') \tag{9.58}$$

Theorem 9.3.22 (Correctness of scope extrusion)**.** *Let* $(l, s') = \mathit{scopeExtrusion}(s)$*. Then:*

$$s' \textit{ contains no instances of } \mathsf{local} \tag{9.59}$$

$$[\![s]\!] \sqsubseteq_{\mho\Delta B} [\![\mathsf{local}\ l\ \mathsf{in}\ s']\!] \tag{9.60}$$

Discussion. Recall from chapter 1 that scope extrusion for this sort of language cannot be proved correct with respect to an equivalence. The problem comes in when we consider looping constructs. In particular, the refinement in equation 9.45 is strict. If we had not found the correct way to weaken bisimulation equivalence, it would not be possible to show the correctness of this pass.

Deep in the guts of the formal proof, it is possible to find the precisely the place where this power is used. Basically, we need to show that the indeterminate variable state $\bot$ refines every non-$\mho$ variable state. This works because reading the $\bot$ value goes wrong and writing over $\bot$ behaves exactly the same as writing over some defined value. Thus, when some looping construct reenters the scope of its body for the second and following iterations, we know that the state of any extruded variables are refined by the intederminate value $\bot$ — this is the germ of the argument.

Although the particular proof method I have used here might or might not be appropriate in CompCert's setting, I believe that this proof shows that it should be possible, in principle, to formally verify the scope extrusion pass that currently occurs in CompCert's unverified wrapper code (see chapter 10).

9.4 Correctness of Left-to-right Evaluation

In this section, I will show that fixing a particular evaluation strategy for expressions is an allowable transformation under behavioral choice refinement. For concreteness, I will chose a standard left-to-right evaluation order, but a similar argument should go through for any reasonable choice of evaluation strategy.

The first step is to define a new transition system for evaluating expressions that follows the left-to-right strategy. One way to do this would be to take the same definition as from figure 9.2, but restrict the evaluation contexts so that the right-hand side of an operator is only evaluable if the left-hand side is a value. Doing this results in a *pruning* of the original LTS, which therefore results in a choice refinement.

However, to make the result less trivial, I will instead be presenting the left-to-right evaluation semantics of expressions in an alternate style, using recursive "search" rules in the style of Plotkin

$$\boxed{e \xrightarrow[\text{LR}]{\alpha} \hat{e}}$$

$$\frac{e_1 \xrightarrow[\text{LR}]{\alpha} e_1'}{(e_1 \ \oplus \ e_2) \xrightarrow[\text{LR}]{\alpha} (e_1' \ \oplus \ e_2)} \qquad \frac{e_1 \xrightarrow[\text{LR}]{\alpha} \mho}{(e_1 \ \oplus \ e_2) \xrightarrow[\text{LR}]{\alpha} \mho} \qquad \frac{e_2 \xrightarrow[\text{LR}]{\alpha} e_2'}{(n \ \oplus \ e_2) \xrightarrow[\text{LR}]{\alpha} (n \ \oplus \ e_2')} \qquad \frac{e_2 \xrightarrow[\text{LR}]{\alpha} \mho}{(n \ \oplus \ e_2) \xrightarrow[\text{LR}]{\alpha} \mho}$$

$$\frac{e_1 \xrightarrow[\text{LR}]{\alpha} e_1'}{(e_1 \ ? \ e_2 \ : \ e_3) \xrightarrow[\text{LR}]{\alpha} (e_1' \ ? \ e_2 \ : \ e_3)} \qquad \frac{e_1 \xrightarrow[\text{LR}]{\alpha} \mho}{(e_1 \ ? \ e_2 \ : \ e_3) \xrightarrow[\text{LR}]{\alpha} \mho}$$

$$\frac{}{(n_1 \ \oplus \ n_2) \xrightarrow[\text{LR}]{\tau} \mathsf{evalop}(\oplus, n_1, n_2)} \qquad \frac{n \neq 0}{(n \ ? \ e_2 \ : \ e_3) \xrightarrow[\text{LR}]{\tau} e_2} \qquad \frac{n = 0}{(n \ ? \ e_2 \ : \ e_3) \xrightarrow[\text{LR}]{\tau} e_3}$$

$$\frac{}{v \xrightarrow[\text{LR}]{\mathsf{read}(v,\bot)} \mho} \qquad \frac{}{v \xrightarrow[\text{LR}]{\mathsf{read}(v,n)} n} \qquad \frac{e \xrightarrow[\text{LR}]{\alpha} e'}{(v := e) \xrightarrow[\text{LR}]{\alpha} (v := e')} \qquad \frac{}{(v := n) \xrightarrow[\text{LR}]{\mathsf{write}(v,n)} n}$$

LR expression states	et	$\in$	**lrexprstate**
		$::=$	$e \mid \mathsf{lrdone}$

$$\boxed{et \xrightarrow{\alpha} \hat{et}}$$

$$\frac{e \xrightarrow[\text{LR}]{\alpha} e'}{e \xrightarrow{\alpha} e'} \qquad \frac{e \xrightarrow[\text{LR}]{\alpha} \mho}{e \xrightarrow{\alpha} \mho} \qquad \frac{}{n \xrightarrow{\mathsf{val}(n)} \mathsf{lrdone}}$$

Figure 9.7: Left-to-right evaluation semantics for expressions

[Plo81] instead of the previous Wright/Felleisen [WF94] style using evaluation contexts. See figure 9.7 for the rules of left-to-right evaluation; let $\mathcal{E}_{LR}$ stand for the τELTS so defined.

Note that the recursively-defined step relation $\xrightarrow[\text{LR}]{}$ is used to define an outer nonrecursive step relation. This is because expression evaluation must only terminate and produce the $\mathsf{val}(n)$ observable event when we have reached a value at the top level. Adding this rule to the recursive $\xrightarrow[\text{LR}]{}$ would be incorrect, as it would cause the relation to produce $\mathsf{val}(n)$ observables in the middle of evaluations.

The first thing to notice about this semantics is that it is internally deterministic. That is, the resulting state of a step is uniquely determined by the starting state and the produced observable.

Proposition 9.4.1. *For all expressions e, e_1, e_2 and observable α, $e \xrightarrow[LR]{\alpha} e_1$ and $e \xrightarrow[LR]{\alpha} e_2$ implies $e_1 = e_2$.*

Proof. By induction on e and inversion on the $\xrightarrow[\text{LR}]{}$ relation. □

To prove that left-to-right evaluation is a choice refinement of nondeterministic evaluation, we use the basic idea mentioned above. We identify those evaluation contexts that correspond to left-to-right evaluation and show that every step of $\underset{\text{LR}}{\longrightarrow}$ can be understood as reducing a redex in a left-to-right context.

Definition 9.4.2 (Left-to-right contexts)**.** *An evaluation context C is a left-to-right context if every time it passes to the right-hand side of a binary operator, the left-hand side is a value. More formally:*

$\boxed{\mathsf{islr}(C)}$

$$\frac{}{\mathsf{islr}(\Box)} \qquad \frac{\mathsf{islr}(C)}{\mathsf{islr}(C \;\oplus\; e)} \qquad \frac{\mathsf{islr}(C)}{\mathsf{islr}(n \;\oplus\; C)} \qquad \frac{\mathsf{islr}(C)}{\mathsf{islr}(v \;:=\; C)} \qquad \frac{\mathsf{islr}(C)}{\mathsf{islr}(C \;?\; e_1 \;:\; e_2)}$$

Next, we need to know that every step of $\underset{\text{LR}}{\longrightarrow}$ corresponds to reducing a redex in some left-to-right context. Furthermore, reducing inside a left-to-right context generates a step in $\underset{\text{LR}}{\longrightarrow}$. Finally, we need to show that if an expression has some redex, then it has a redex in leftmost position.

Proposition 9.4.3. *When $e \xrightarrow[LR]{\alpha} e'$ there exist unique C, e_0 and e_0' such that:*

$$\mathsf{islr}(C) \tag{9.61}$$

$$e_0 \xrightarrow[redex]{\alpha} e_0' \tag{9.62}$$

$$e = C[e_0] \tag{9.63}$$

$$(e' = \mho \wedge e_0' = \mho) \vee e' = C[e_0'] \tag{9.64}$$

Proof. By induction on the derivation of $e \xrightarrow[\text{LR}]{\alpha} e'$. □

Proposition 9.4.4. *Suppose C is a left-to-right context and $e \xrightarrow[redex]{\alpha} e'$. Then either $e' = \mho$ and $C[e] \xrightarrow[LR]{\alpha} \mho$, or $e' \neq \mho$ and $C[e] \xrightarrow[LR]{\alpha} C[e']$.*

Proof. By induction on the context C. □

Proposition 9.4.5. *Suppose $e = C[e_0]$ and e_0 is a redex. Then there exists C' and e_0' such that $e = C'[e_0']$, C' is a left-to-right context, and e_0' is a redex.*

Proof. By induction on e. □

Definition 9.4.6 (Refinement relation for left-to-right evaluation)**.** *The following rules define a choice refinement relation between lifted* **exprstate***s and lifted* **lrexprstate***s.*

$\boxed{\mathsf{lrmatch}(\hat{es}, \hat{et})}$

$$\frac{}{\mathsf{lrmatch}(\mho, \mho)} \qquad \frac{}{\mathsf{lrmatch}(\mathsf{done}, \mathsf{lrdone})} \qquad \frac{e = e'}{\mathsf{lrmatch}(e, e')} \qquad \frac{\mathsf{isredex}(e_0) \qquad \mathsf{islr}(C) \qquad C[e_0] = e'}{\mathsf{lrmatch}((C, e_0), e')}$$

Proposition 9.4.7. $\mathsf{lrmatch}$ *is a branching behavioral choice refinement between* $\mathcal{E}$ *and* $\mathcal{E}_{LR}$.

Proof. Straightforward from propositions 9.4.3, 9.4.4 and 9.4.5. □

Proposition 9.4.8. *For expressions* e, $\langle \mathcal{E}, e\rangle \sqsubseteq_{CB} \langle \mathcal{E}_{LR}, e\rangle$.

Now we can define an alternate semantics of statements that uses left-to-right expression evaluation rather than nondeterministic evaluation. Because the semantic operators are all congruences for choice refinement it is easy to show that this new semantics of statements is a choice refinement of the original one.

Definition 9.4.9 (Left-to-right denotation of statements)**.**

$$[\![e]\!]_{LR} \equiv \langle \mathcal{E}_{LR}, e\rangle \tag{9.65}$$

$$[\![s_1;\ s_2]\!]_{LR} \equiv \mathit{seq}([\![s_1]\!]_{LR}, [\![s_2]\!]_{LR}) \tag{9.66}$$

$$[\![\mathsf{local}\ v\ \mathsf{in}\ s]\!]_{LR} \equiv \mathit{local}(v, [\![s]\!]_{LR}) \tag{9.67}$$

$$[\![\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2]\!]_{LR} \equiv \mathit{ifte}([\![e]\!]_{LR}, [\![s_1]\!]_{LR}, [\![s_2]\!]_{LR}) \tag{9.68}$$

$$[\![\mathsf{while}\ e\ \mathsf{do}\ s]\!]_{LR} \equiv \mathit{while}([\![e]\!]_{LR}, [\![s]\!]_{LR}) \tag{9.69}$$

Theorem 9.4.10. *For all statements* s, $[\![s]\!] \sqsubseteq_{CB} [\![s]\!]_{LR}$

Proof. By induction on s. In the base cases, use proposition 9.4.8 to show that the refinement for expressions. In each inductive case, use the congruence facts from proposition 9.1.4. □

I have not done the proof, but I expect it would be straightforward to show a corresponding fact for the abstract machine semantics. That is, if you replace the nondeterministic evaluation LTS with the left-to-right evaluaion LTS everywhere in the definition of the abstract machine, you should get a new abstract machine that is a choice refinement of the original.

Chapter 10

Applying Refinement to the CompCert Compiler

It is a significant design goal of the Verified Software Toolchain (VST) project [App11] to connect a stack of program logics and static analysis tools to the formally verified CompCert C compiler [Com] to achieve end-to-end, foundational correctness proofs. One of the main motivators of the work in this thesis is the desire to better understand the software interface between a compiler and its clients. In this chapter, I will explain how we can apply the refinement ideas developed in the previous chapters to CompCert and make some suggestions for improvements.

Recall from chapter 1 that I discussed the correctness statement from CompCert as of Leroy's 2006 overview paper [Ler06] (cf. theorem 1.1.4). At that time, the correctness statement for CompCert applied only to safe, terminating programs, and only discussed the return value of the `main` function. Unfortunately, such limitations are unacceptable for the main application of interest for a verified C compiler: embedded control software. Such control software is typically long-running, interactive software — we are primarily interested in how this software interacts with its environment rather than what values it computes.

Subsequent to the 2006 paper, the CompCert development team modified the correctness statement to lift the restriction to terminating programs, and so that it additionally discusses a trace of "external events." These external events include things like direct access to memory-mapped hardware ports (via `volatile` global variables) and calls to external functions (essentially, system calls). In version of the correctness theorem, there are essentially three sorts of possible program behavior: a program may terminate, exhibiting a finite trace of external events; a program may run forever, exhibiting a possibly infinite trace of external events; or a program may go wrong, exhibiting a trace of events up until then. Then the correctness theorem says that the behavior exhibited by the source program is exhibited by the compiled program. Under the assumption that both the program and the environment are deterministic, this is sufficent to constrain the behavior of the compiled program.

This event trace model provided a greater degree of observability of the reactive behavior of programs, and was more appropriate for reasoning about long-running, reactive software. This version of CompCert is discussed in an overview paper [Ler09a] and in considerably more detail in a companion journal article [Ler09b].

Although a significant improvement, the event trace model was also not entirely satisfactory. In particular, the VST project is interested in using CompCert to compile multithreaded C code and reason about the resulting programs. It is not immediately obvious whether or not the traces model is sufficiently powerful to allow one to reason about the composition of multiple threads. In general, it is the case that linear-time notions of program equivalence/refinement (such as those based on trace sets) are weaker than branching-time notions of equivalence/refinement, such as those based on bisimulation [Pnu85]. Fundamentally, it is not entirely clear that the CompCert traces model is the strongest theorem we can achieve.

I have argued in previous chapters that behavioral and choice refinement are the strongest extensional notions of program refinement that allow for the arbitrary interpretation of undefined behavior and the resolution of internal nondeterminism, respectively. In the remainder of this chapter, I shall present in some detail the current state of CompCert's top-level correctness statement, and compare it to the refinement notions I have developed. The current correctness statement (as of CompCert version 1.10) is based on certain kinds of forward and backward simulation relations. The form of these simulations is based in part on ideas that emerged from discussions between myself and the CompCert development team, and in part on the measured backward simulations from CompCertTSO [ŠVZN$^+$11].

Before I go into the details, here is a summary of the results. The vast majority of the CompCert stack (from `Cstrategy` down to `Asm`) is correct with respect to divergence-sensitive branching behavioral refinement; no changes to the CompCert code base were required to show this result. I have already argued that this refinement relation is the strongest reasonable theory that allows the expansion of undefined behavior; thus, this result indicates that the vast majority of CompCert is proved correct in the strongest reasonable manner.

However, there are two additional places where I believe CompCert should be improved. The first has to do with the very top level of the verified CompCert core, which defines the nondeterministic semantics of CompCert's flavor of C. The first pass of the compiler is actually a "virtual" pass; no program transformation is done, but the same program is interpreted first with a nondeterministic semantics that captures the unspecified behavior of C, and next with a deterministic semantics that fixes a particular evaluation order. The choice refinement developed in chapter 6 was developed primarily to characterize the correctness of this virtual pass. However, there is a subtle error in the statement of the nondeterministic semantics for C that makes it impossible to show that the fixed evaluation strategy is a choice refinement. Fortunately, the bug should be easy to fix, at which point I believe a choice refinement will be provable.

The second improvement I suggest is more involved. The majority of CompCert is proved correct in the Coq theorem prover. However, there is an unverified wrapper that performs a number of important tasks. To understand the context for the improvement I suggest, it is important to understand how CompCert is divided into the verified core and the unverified wrapper. When the CompCert executable is invoked on a source file, the first thing that happens is that the system C preprocessor is invoked. Obviously, the system CPP executable is unverified. After preprocessing, the resulting source file is parsed, typechecked and elaborated into a type-annotated abstract syntax tree — these steps are also unverified. At this point, the syntax tree is *almost* ready to be passed off to the verified core of the compiler; however, before that happens a small number of unverified source-to-source translation passes occur. After these passes, the verified core translates from the high-level C syntax tree to an assembly language syntax tree. The unverified wrapper then takes over again to print the assembly syntax tree, which is then passed to the system assembler and linker.

Three of the unverified source-to-source passes are optional, controlled by command-line flags; they implement language extensions (an alternate packing algorithm for `struct`s, bitfields, and `struct`s as return values). Two of the passes, however, are mandatory. These two passes lift nested block scopes up to function scope level and rename program identifiers to avoid clashes. Although this scope extrusion pass is fairly simple (as program transformations go), the correctness argument for it is sufficiently subtle that I believe it is worthwhile to consider moving this pass from the unverified wrapper into the verified core. This reduces the possibility that some extremely subtle bug lurks somewhere in this transformation pass.

Such concerns about the unverified wrapper are not just theoretical hand-wringing. Yang et al. [YCER11] applied Csmith, a bug-finding technique based on generating random programs, to a battery of different compiler implementations, including CompCert (version 1.6). They found several bugs in CompCert, all stemming from either the unverified wrapper or from errors in the compiler specification. It is a testament to the strength of CompCert's verified core that Yang et al. found no bugs in CompCert's "middle end," where the majority of errors were found in competing compilers. Although the CompCert development team has fixed the bugs uncovered by Csmith, it is still worthwhile, I think, to reduce the amount and the complexity of the unverified wrapper code as much as possible.

The operational semantics of each of the intermediate languages in the CompCert stack are specified in a way similar, but not identical, to the extended labeled transition systems over which behavioral and choice refinement operate. Thus, we need a few definitions to inject the semantics of CompCert languages into my setting. The first thing we need to do is to fix the set of observable events we will be using. These events are based closely on CompCert's event model.[1]

Listing 10.1: CompCert Events (from `Events.v`)

```
Inductive eventval: Type :=
  | EVint: int → eventval
  | EVfloat: float → eventval
  | EVptr_global: ident → int → eventval.

Inductive event: Type :=
  | Event_syscall: ident → list eventval → eventval → event
  | Event_vload: memory_chunk → ident → int → eventval → event
  | Event_vstore: memory_chunk → ident → int → eventval → event
  | Event_annot: ident → list eventval → event.

Definition trace := list event.

Definition E0 : trace := nil.
```

The type `eventval` defines the type of data that can be communicated with the environment: 32-bit integer machine words; IEEE 754 floating-point values; and pointers calculated as offsets from global static data. Note that pointers to global static data are referenced by identifier and offset — this means that no raw pointer values can "escape" to the environment. This restriction allows the compiler to reason about compilation passes that rearrange memory layout, but it also means that the event model cannot express interactions with the environment that are characteristic of realistic operating systems. I discuss this issue in more detail in §11.3.

The `event` type describes a single communication with the environment. A system call event

[1]All code excerpts from CompCert are taken from version 1.10.

(`Event_syscall id args ret`) is used to indicate that the program made a system call to function `id` with arguments `args` and that the function returned value `ret`. An `Event_vload` event is used to indicate a load from a global volatile location, and `Event_vstore` indicates a store to a global volatile location. In C, volatile locations are generally used to model memory-mapped hardware devices. Loads and stores to such locations may cause the hardware to produce observable side-effects, so the C specification requires such loads and stores to be preserved by the program as-is. CompCert ensures this by making volatile memory accesses part of the observable events of a program run. The `memory_chunk` argument indicates the width of the load or store (8, 16 or 32 bits), and whether the access is interpreted as a signed int, unsigned int, or floating point value. The `Event_annot` event is used to allow the programmer to insert annotations in their program, injecting internal values that would otherwise be unobservable into the program's observable events. A list of events is called a `trace`, and `E0` indicates the empty trace.

To use CompCert's events in my setting, I need to add one additional observation relating to program termination. In CompCert, final program states are associated with an exit code, which is considered observable. I prefer not to treat final states any differently from other states, so I instead model exit codes as a distinguished program observation that only occurs at most once, just before a program terminates.

Listing 10.2: Observation Structure for CompCert

```
Inductive compcert_obs :=
  | obs_event : event → compcert_obs
  | obs_terminate : int → compcert_obs.

Instance compcert_observations : ObservationSystem :=
  {| O := compcert_obs; observations_inhabited := obs_terminate Int.zero |}.
```

For the rest of this chapter, whenever I refer to an observable event, I mean the type `compcert_obs` which contains all of CompCert's events as well as my special program termination event.

The next thing I must do is adapt CompCert's notion of small step program semantics to ELTSs. One difference is that CompCert's step relations produce a `trace` (that is, a list of events) rather than a single event. I believe this is a misfeature that arises as a historical oddity rather than a core design decision. The vast majority of the steps in all the languages in CompCert's stack return 0 (corresponding to a τ action) or 1 event. The only exceptions occur in two reduction rules for the deterministic semantics of C (`Cstrategy.v`) having to do with the semantics of volatile variables. It would be desirable to modify those two places to return at most 1 event per step. Unfortunately, such a change is not entirely straightforward — it impacts the translation from Cstrategy to Clight, one of the trickiest parts of the overall compiler correctness proof. I am informed[2] that previous attempts to make this change have run into considerable difficulty.

Another difference is the handling of initial states. In CompCert, initial states are specified as a distinguished set together with the step relation. In my setting, I have instead assumed that there is some function mapping program syntax to an ELTS process — thus, I assume that initial states are *calculated* rather than *characterized*. Finally, CompCert implicitly defines the states that go wrong as those states that have no successors in the step relation and that are not final states, rather than using an explicit ℧ state.

The following listing defines a generic small-step operational semantics in the CompCert stack.

Listing 10.3: CompCert language semantics (from `Smallstep.v`)

```
Record semantics : Type := Semantics {
  state: Type;
  funtype: Type;
  vartype: Type;
  step : Genv.t funtype vartype → state → trace → state → Prop;
  initial_state: state → Prop;
  final_state: state → int → Prop;
  globalenv: Genv.t funtype vartype
}.
```

Given a `semantics`, I can generate an ELTS in the following way.

Listing 10.4: Building an ELTS from a `semantics`

```
Inductive match_ev : option compcert_obs → list event → Prop :=
  | match_ev_none : match_ev None nil
  | match_ev_some : ∀ e, match_ev (Some (obs_event e)) (e::nil).

Definition immed_safe (sem:semantics) (x:Smallstep.state sem) :=
  (∃ i, Smallstep.final_state sem x i)
  ∨
  (∃ t, ∃ x', Smallstep.step sem (globalenv sem) x t x').

Inductive semantics_lts_state (sem:semantics) :=
  | run  : Smallstep.state sem → semantics_lts_state sem
  | done : semantics_lts_state sem.

```

[2]Personal communications with Xavier Leroy.

```
Inductive semantics_lts_step (sem:semantics) :
  semantics_lts_state sem → option O → lift (semantics_lts_state sem) → Prop :=

  | sem_lts_step_ok : ∀ x o t x′,
      match_ev o t →
      Smallstep.step sem (globalenv sem) x t x′ →
      semantics_lts_step sem (run sem x) o (into (run sem x′))

  | sem_lts_step_wrong : ∀ x,
      ¬immed_safe sem x →
      semantics_lts_step sem (run sem x) None (mho _)

  | sem_lts_step_term : ∀ x i,
      final_state sem x i →
      semantics_lts_step sem (run sem x) (Some (obs_terminate i)) (into (done _)).

Definition semantics_LTS (sem:semantics) : LTS :=
  Build_LTS (semantics_lts_state sem) (semantics_lts_step sem).
```

`match_ev` shows how the observable events I use match up to the traces used by CompCert. A CompCert program state is immediately safe if it is final or if it can take some step. A state that is not immediately safe is *stuck*, and stuck states correspond to $\mho$. To do termination in the style I prefer, I need to add an additional, distinguished terminating state — `semantics_lts_state` therefore extends the state space with a new state named `done`. Finally, the `semantics_lts_step` definition builds the ELTS step relation. Every step from the CompCert `semantics` is turned directly into a step in the ELTS by simply matching the event. On the other hand, if the CompCert state is stuck, then the ELTS takes a τ step to $\mho$. Finally, if the CompCert state is a final state, it takes a single additional step to the `done` state, producing the termination event containing the program's exit code. Finally, `semantics_LTS` bundles together the state space with its step relation, producing an ELTS from a `semantics`.

As of CompCert version 1.10, the correctness lemma for each pass of the main part of the compiler is a forward simulation. In general, proving only a forward simulation is too weak because it does not guarantee that the compilation pass does not introduce new behaviors. However, in the special case where the target language is internally deterministic, forward simulations *are* enough because the the target program has only one possible behavior at each step. As I shall show below, we can make this argument completely formal by showing that, under the right assumptions, CompCert's forward simulations induce branching behavioral refinements over the associated ELTS.

Listing 10.5: CompCert forward simulation (from `Smallstep.v`)

```
Record forward_simulation (L1 L2: semantics) : Type :=
  Forward_simulation {
    fsim_index: Type;
    fsim_order: fsim_index → fsim_index → Prop;
    fsim_order_wf: well_founded fsim_order;
    fsim_match_states :> fsim_index → state L1 → state L2 → Prop;
    fsim_match_initial_states:
      ∀ s1, initial_state L1 s1 →
      ∃ i, ∃ s2, initial_state L2 s2 ∧ fsim_match_states i s1 s2;
    fsim_match_final_states:
      ∀ i s1 s2 r,
      fsim_match_states i s1 s2 → final_state L1 s1 r → final_state L2 s2 r;
    fsim_simulation:
      ∀ s1 t s1', Step L1 s1 t s1' →
      ∀ i s2, fsim_match_states i s1 s2 →
      ∃ i', ∃ s2',
          (Plus L2 s2 t s2' ∨ (Star L2 s2 t s2' ∧ fsim_order i' i))
       ∧ fsim_match_states i' s1' s2';
    fsim_symbols_preserved:
      ∀ id, Genv.find_symbol (globalenv L2) id = Genv.find_symbol (globalenv L1) id
  }.
```

The main elements of this record are the simulation relation itself, `fsim_match_states` and the simulation fact, `fsim_simulation`. In this definition, `Plus` refers to taking at least one step, and `Star` refers to taking 0 or more steps.

Note that `fsim_match_states` takes a third index component; this additional component is used to ensure divergence-sensitivity. The index type is equipped with a well-founded order and the simulation diagram is written in such a way that whenever the source program takes a stuttering step, the index must go down in the well-founded order. `fsim_simulation` is written in the style of weak bisimulation, rather than in the style of branching simulation from §3.2. In general, weak bisimulation is coarser than branching bisimulation; however, under the right determinacy assumption (as stated below), this simulation diagram is sufficient to prove a branching-style simulation.

The assumptions we need in order to get a behavioral refinement from CompCert's forward simulations are listed below.

Listing 10.6: Receptiveness and determinacy (from `Smallstep.v`)

```
Definition single_events (L: semantics) : Prop :=
  ∀ s t s′, Step L s t s′ → (length t ≤ 1)%nat.

Record receptive (L: semantics) : Prop :=
  Receptive {
    sr_receptive: ∀ s t1 s1 t2,
      Step L s t1 s1 → match_traces (globalenv L) t1 t2 → ∃ s2, Step L s t2 s2;
    sr_traces:
      single_events L
  }.

Record determinate (L: semantics) : Prop :=
  Determinate {
    sd_determ: ∀ s t1 s1 t2 s2,
      Step L s t1 s1 → Step L s t2 s2 →
      match_traces (globalenv L) t1 t2 ∧ (t1 = t2 → s1 = s2);
    sd_traces:
      single_events L;
    sd_initial_determ: ∀ s1 s2,
      initial_state L s1 → initial_state L s2 → s1 = s2;
    sd_final_nostep: ∀ s r,
      final_state L s r → Nostep L s;
    sd_final_determ: ∀ s r1 r2,
      final_state L s r1 → final_state L s r2 → r1 = r2
  }.
```

The `match_traces` relation (not shown here), defines what it means for two traces to have the same "shape." Basically, they need to have pairwise matching events, and two events match if they are of the same type and have identical output components. In particular, they can only differ in the result of doing a volatile load or in the result of performing a system call. A receptive semantics is one where, whenever a state stands ready to accept some input event, it is willing to accept every input event of the same shape. A determinate semantics is one that has no internal nondeterminism, and also satisfies some basic sanity properties (e.g., final states cannot step).

If we assume that the input language is receptive, that it is decidable if a state is stuck or not, and that the output language is determinate, then we can prove that a CompCert forward simulation induces a behavioral refinement. The following listing summarizes the formal proof of this fact.

Listing 10.7: Forward simulation induce behavioral refinements

```
Definition immed_safe_dec (L:semantics) :=
  ∀ x:Smallstep.state L, immed_safe L x ∨ ¬immed_safe L x.

Section forward_sim_to_beh_refine.
  Variables L1 L2 : semantics.
  Variable fsim : forward_simulation L1 L2.

  Hypothesis safe_dec : immed_safe_dec L1.
  Hypothesis L1recep  : receptive L1.
  Hypothesis L2determ : determinate L2.

  Definition fsim_refine x y :=
      match x, y with
      | mho, _ ⇒ True
      | into (run x), mho ⇒ ∃ q, ∃ y,
           ¬immed_safe L2 y ∧ fsim_match_states fsim q x y
      | into (run x), into (run y) ⇒
          ∃ y′, (Star L2 y E0 y′ ∨ Star L2 y′ E0 y) ∧
          ∃ q, fsim_match_states fsim q x y′
      | into done, into done ⇒ True
      | _, _ ⇒ False
      end.

  Lemma compcert_refinement :
    refinement branch_ind_no_stutter
       (semantics_LTS L1) (semantics_LTS L2) fsim_refine.
  Proof. ··· Qed. (* proof omitted *)

  Lemma matching_states_refine :
    ∀ q x y, fsim q x y →
      refines branch_ind (semantics_LTS L1) (semantics_LTS L2)
         (into (run L1 x)) (into (run L2 y)).
  Proof. ··· Qed. (* proof omitted *)

  Lemma initial_states_refine :
    ∀ i1, initial_state L1 i1 → ∃ i2, initial_state L2 i2 ∧
      refines branch_ind (semantics_LTS L1) (semantics_LTS L2)
         (into (run L1 i1)) (into (run L2 i2)).
  Proof. ··· Qed. (* proof omitted *)
End forward_sim_to_beh_refine.
```

In the main case of the `fsim_refine` relation (where both `x` and `y` are running states) we require that `y` be *in the vicinity* of some state matching `x`. That is, `y` can either reach via τ-steps, or is reachable via τ-steps from, some state that is related to `x`. Such a loose relation only works because we assumed that `L2` is internally deterministic.

By including the top-level CompCert correctness theorem, we can show that the main compiler stack, from `Cstrategy` down to `Asm` is correct with respect to behavioral refinement.

Listing 10.8: Main correctness theorem

```
Require Cstrategy.
Require Asm.
Require Compiler.

Let C p := atomic (Cstrategy.semantics p).
Let A p := Asm.semantics p.

Theorem CompCert_main_pass_correctness :
  ∀ p tp, Compiler.transf_c_program p = Errors.OK tp →
    ∀ i1, initial_state (C p) i1 → ∃ i2, initial_state (A tp) i2 ∧
      refines branch_ind (semantics_LTS (C p)) (semantics_LTS (A tp))
         (into (run (C p) i1)) (into (run (A tp) i2)).
Proof.  ···  Qed.  (* proof omitted *)
```

Note there is one tiny wrinkle here; because the language defined in `Cstrategy` will sometimes produce more than one event in a step, it must be coerced to a semantics that produces at most one event each step. The `atomic` construction takes an arbitrary semantics and produces a new one that does at most one event each step. With this little modification, the main theorem is a straightforward application of the main CompCert theorem and the fact that forward simulations induce behavioral refinements, together with the determinacy and receptiveness facts about `Cstrategy` and `Asm`.

This is excellent news! It means that, under the event model defined above, most of the CompCert stack is proved correct with respect to the strongest notion of program refinement that is appropriate for this setting.

It would also be desirable to show that the compiled program generated by the verified portion of CompCert's compiler stack is a choice refinement of the nondeterministic semantics of C. This cannot be done, however, with the current operational semantics as defined in `Csem`. There is a subtle error in the way the nondeterministic semantics of C are specified, as I shall explain below, that prevents us from proving the desired theorem. The bug, fortunately, has an easy fix. I believe that it will be straightforward to prove choice refinement once the bug is fixed.

Currently, CompCert proves that there exists a backward simulation from `Cstrategy` (C with a fixed evaluation order) to `Csem` (C with unspecified evaluation order). This notion of backward simulation is similar to the measured backward simulations of CompCertTSO (cf. definition 1.1.6), but stronger because it also requires some progress properties. However, it is strictly weaker than choice refinement.

Listing 10.9: CompCert backward simulations (from `Smallstep.v`)

```
Definition safe (L: semantics) (s: state L) : Prop :=
 ∀ s′, Star L s E0 s′ →
 (∃ r, final_state L s′ r) ∨ (∃ t, ∃ s′′, Step L s′ t s′′).

Record backward_simulation (L1 L2: semantics) : Type :=
 Backward_simulation {
   bsim_index: Type;
   bsim_order: bsim_index → bsim_index → Prop;
   bsim_order_wf: well_founded bsim_order;
   bsim_match_states :> bsim_index → state L1 → state L2 → Prop;
   bsim_initial_states_exist:
     ∀ s1, initial_state L1 s1 → ∃ s2, initial_state L2 s2;
   bsim_match_initial_states:
     ∀ s1 s2, initial_state L1 s1 → initial_state L2 s2 →
     ∃ i, ∃ s1′, initial_state L1 s1′ ∧ bsim_match_states i s1′ s2;
   bsim_match_final_states:
     ∀ i s1 s2 r,
     bsim_match_states i s1 s2 → safe L1 s1 → final_state L2 s2 r →
     ∃ s1′, Star L1 s1 E0 s1′ ∧ final_state L1 s1′ r;
   bsim_progress:
     ∀ i s1 s2,
     bsim_match_states i s1 s2 → safe L1 s1 →
     (∃ r, final_state L2 s2 r) ∨
     (∃ t, ∃ s2′, Step L2 s2 t s2′);
   bsim_simulation:
     ∀ s2 t s2′, Step L2 s2 t s2′ →
     ∀ i s1, bsim_match_states i s1 s2 → safe L1 s1 →
     ∃ i′, ∃ s1′,
        (Plus L1 s1 t s1′ ∨ (Star L1 s1 t s1′ ∧ bsim_order i′ i))
      ∧ bsim_match_states i′ s1′ s2′;
   bsim_symbols_preserved:
     ∀ id, Genv.find_symbol (globalenv L2) id = Genv.find_symbol (globalenv L1) id
 }.
```

Backward simulations are much like forward simulations, except that there are additional `safe` assumptions in various places and there is an additional progress property. As compared to the progress property implied by choice refinement, this progress property is strictly weaker. Choice refinement requires the preservation of all stable offers, whereas these backward simulations require only that *some* action be offered.

Unsurprisingly, the bug in the nondeterministic semantics of C exercises this particular difference. The issue has to do with the semantics of expression evaluation. The evaluation relation for expressions is written in the Wright/Felleisen style, [WF94] using evaluation contexts. The relevant part of the operational semantics is reproduced below.

Listing 10.10: Nondeterministic evaluation of expressions (from `Csem.v`)

```
Inductive estep: state → trace → state → Prop :=

  | step_lred: ∀ C f a k e m a' m',
      lred e a m a' m' →
      context LV RV C →
      estep (ExprState f (C a) k e m)
         E0 (ExprState f (C a') k e m')

  | step_rred: ∀ C f a k e m t a' m',
      rred a m t a' m' →
      context RV RV C →
      estep (ExprState f (C a) k e m)
          t (ExprState f (C a') k e m')

  | step_call: ∀ C f a k e m fd vargs ty,
      callred a fd vargs ty →
      context RV RV C →
      estep (ExprState f (C a) k e m)
         E0 (Callstate fd vargs (Kcall f e C ty k) m)

  | step_stuck: ∀ C f a k e m K,
      context K RV C → ¬(imm_safe e K a m) →
      estep (ExprState f (C a) k e m)
         E0 Stuckstate.
```

The troublesome case is `step_rred` — note that this is the only evaluation rule that might produce a nonempty trace. The problem arises because this rule both selects the context in which it will do the reduction and also reduces the redex in a single step. This is the wrong thing to do when reducing redexes may have observable side-effects! Consider the case where we have a binary operator (addition, say) applied to two global volatile variables, `x` and `y`. That is, we are reducing the expression $\texttt{x} + \texttt{y}$, where `x` and `y` are both global volatile integer variables. Because of the form of this rule (where we select the redex and reduce it all at once) what happens is that we offer to the environment an *external* choice. In other words, this rule allows the environment to pick which volatile variable it wants to provide a value for first. This is not what we intended — instead we intended that the program gets to chose what part of the expression it wants to evaluate next, and *then* we allow the environment to provide a value for the selected variable.

The semantics of `Cstrategy` always selects the leftmost-innermost redex when performing reduction, so when reducing the expression $\texttt{x} + \texttt{y}$, it only allows the environment to select values for `x` first. Thus, the semantics in `Cstrategy` is not a choice refinement of `Csem` because choice refinements are not allowed to reduce external nondeterminism, but only internal nondeterminism.

This problem is solved by breaking the reduction of redexes into two steps. First, a redex is selected via an internal choice; then, in a second step, the redex is reduced, possibly producing an observable side-effect. This models the desired situation, where the program is always fully in charge of which redex gets selected. This pattern is demonstrated in the expression reduction semantics of the case study language of chapter 9.

I strongly suspect that, if this issue were repaired, it would be straightforward to show that the repaired nondeterministic semantics is choice refined by the deterministic semantics of `Cstrategy`. Because behavioral refinement implies behavioral choice refinement, this would imply that the entire verified CompCert stack, from `Csem` down to `Asm`, is a correct program transformation with respect to behavioral choice refinement.

Recap and recommendations. Under CompCert's current model of external events, one can show that the main verified CompCert stack (from `Cstrategy` to `Asm`) is correct with respect to inductive branching behavioral refinement. With a minor fix to the nondeterministic expression evaluation semantics, it should be possible to show that the entire verified stack (from `Csem` to `Asm`) is correct with respect to inductive branching choice refinement.

In addition to the already-discussed fix to the expression evaluation semantics, I also recommend that CompCert's basic definition of `semantics` be changed so that atomic steps produce `option event` rather than `list event`. Such a setup corresponds more directly to standard practice for small-step semantics with effects. Furthermore, the restriction to producing at most one event every step is required as a side-condition in both `receptive` and `determinate`.

Finally, and with the understanding that this is a rather more substantial change, I would recommend that the scope extrusion pass that is currently done in the unverified wrapper code be pushed into the verified core. Although the scope extrusion pass itself is not so complicated, the reasons for its correctness are sufficiently subtle that I believe they are worth spelling out.

Chapter 11

Future Work and Conclusions

11.1 Acceptable Failures

In this thesis, we examined two sorts of refinement: refinement with respect to undefined behavior, and refinement with respect to unspecified behavior. There is a third sort of refinement along these same lines that is useful for reasoning about resource exhaustion.

Consider the issue of dynamic memory allocation — when it comes to reasoning about realistic systems software, we somehow have to come to grips with the fact that real hardware has only a finite amount of memory. In the context of C, this means that `malloc` calls may fail and that we may exhaust stack space by making deeply nested function calls. There are three basic approaches to dealing with this problem: we may explicitly reason about resource bounds; we may sidestep the problem by pretending we have infinite resources; or we may introduce a system of *acceptable failures*.

We have already seen each of these approaches taken in the compiler correctness theorems from chapter 1. The CLI short stack (cf. definition 1.1.1) and the Verisoft C0 compiler (cf. definition 1.1.5) both use correctness statements that explicitly define a link between the memory resources used by the source and target computations. CompCert, on the other hand, uses an unbounded model of memory where allocations always succeed, modeling infinite resources. Finally, CompCertTSO, in effect, uses a model based on acceptable failures.

In the acceptable failures model, one designates a class of bad events that can happen but are outside the control of the program — these events are considered acceptable failures. Acceptable failures are dual to going wrong. When a program goes wrong it performs some erroneous action

— an action that it could have avoided. In contrast, an acceptable failure is some bad event that could *not* have been prevented by the program, due to some circumstance beyond its control.

If we consider user-mode programs running on a shared multiprocessing operating system, it might make sense to decide that running out of memory is an acceptable failure. Ultimately, the operating system is in charge of managing the physical memory and the user process can only request memory from the OS. If a request is denied, it might well be because some other process has hogged all the memory, or our process may have been assigned a low priority, etc. In any event, the user process can only request more memory — if the OS fails to grant the request, the user process is not at fault. In contrast, if we consider hard real-time embedded software, we might instead decide that running out of memory is an *unacceptable failure* (i.e., is the same as going wrong) because the process is running directly on the bare hardware and is entirely in charge of its own resource management. How we make this decision determines whether or not we will have to explicitly reason about computational resources. If resource exhaustion is the same as going wrong, we must prove our program will only use an amount of resources less than some bound fixed in advance.

Of course, the idea of acceptable failures can be used to model a variety of events besides out of memory conditions, such as: hardware failures; power outage; kill signals from the operating system; and resource exhaustion of other sorts (file descriptors, process identifiers, available sockets, etc). Regardless of exactly which events we want to consider acceptable failures, the formal mechanism we can use to model them is the same. Just as with going wrong, we add a new distinguished state that represents experiencing an acceptable failure. I use the symbol $\lightning$, which I pronounce "oops." The main idea behind $\lightning$ is that it is dual to $\mho$. Where as $\mho$ is the bottom of its refinement order, $\lightning$ is the top of its refinement order. Any program at all can be refined into one that experiences acceptable failure. Somewhat perversely, it means we consider "going oops" a good outcome.

As a concrete example, consider a programming language with an unbounded memory (like in CompCert) such that allocation always succeeds. We would like to be able to instead interpret programs in a more realistic memory model, which has some specific bound on allocation (e.g., there are at most 2^{32} virtual addresses). In the bounded memory model, suppose we want to allocate a block of memory, but we have run out of space and cannot. If we interpret this event as stepping to the oops state, $\lightning$, then we are allowed to translate from the unbounded memory model to the bounded one. By analogy to behavioral refinement, this sort of refinement makes formal the idea that "the compiled program behaves the same as the source program until and unless it runs out of memory/the power shuts off/the hardware fails/etc."

Acceptable failures are a powerful tool, but that power comes at a price. If not used carefully, acceptable failure refinement can weaken a compiler correctness theorem to the point of unusability. Assume, for example, we use acceptable failure as I suggest above to reason about out of memory conditions in a bounded memory model. If we allow the compiler to "know" that there is a bound on memory, it can, in theory, exploit this knowledge to turn any program into one that simply exhausts memory. For example, consider the following:[1]

```
int main() { return main() + 1; }
```

This program will, in any bounded memory model, eventually allocate enough stack frames to exhaust all available memory and go oops. Under acceptable failures refinement in a bounded memory model, this program would be an acceptable translation of any program whatever! Obviously, this is not what we wanted.

Perhaps a workable compromise can be arranged whereby the main body of the compiler uses an unbounded memory model (as in CompCert) but at the very end, after the program code has been generated, we reinterpret the program in a bounded memory model. Because the compiler is proved correct with an unbounded memory, it cannot use the "escape hatch" of purposfully exhausting memory. However, acceptable failures refinement still allows us to get out of the unbounded memory into a realistic memory model.

Although I believe this compromise is workable, it is still not entirely satisfactory. We would like for the end-to-end correctness theorem to say everything we want to know, including that the compiler has not played any dirty tricks. If our end-to-end statement includes acceptable failures refinement, it is not obvious that this can be achived. Unfortunately, it is still not entirely clear what is the best method for dealing with the issue of resource exhausion, but perhaps acceptable failures refinement has a role to play.

I have not yet worked out the details of refinement with respect to acceptable failures, but I expect the resulting theory to be pretty straightforward and to interact well with behavioral and choice refinement.

11.2 Statewise Properties

Throughout this thesis, I have restricted the observations that can be made so that one can only differentiate program states by the actions they can perform; the identity of states has not been observable in any way. However, one might like to have a way to observe some properties of states directly.

[1] Thanks to Xavier Leroy for suggesting this example.

Suppose that, in addition to the set $\mathcal{O}$ of observable events, there is also a set $\mathcal{A}$ of atomic observations. Then, when defining transition systems, we include a function that maps each state to the set of atomic properties it satisfies. The resulting systems are called "doubly labeled transition systems," or (if we label *only* the states) "Kripke structures."

It is fairly easy to modify the definition of bisimulation so that it also accounts for atomic state properties — one simply adds a clause that requires related states to have identical atomic properties.

I believe it would be fairly straightforward to add atomic statewise observations to all the systems of bisimulation and refinement found in this thesis by adopting the Kripke Modal Transition Systems of Huth et al. [HJS01]. The characteristic logics would likewise be straightforward to modify by adding the atomic state properties directly as new atomic propositions in the logic.

This enhancement would allow a very nice embedding of Hoare-style program logics into the modal characteristic logics found in this thesis. Suppose P and Q are assertions built up from basic logical operations and atomic state properties. Then the Hoare tuple of partial correctness can be defined directly as:

$$\{P\}c\{Q\} \qquad \equiv \qquad [\![c]\!] \models P \Rightarrow \bigcirc Q \tag{11.1}$$

Recall that $\bigcirc$ from chapter 6 is called the modality of postconditions, and $\bigcirc Q$ means that every τ-reachable stable state satisfies Q. Thus, $P \Rightarrow \bigcirc Q$ means that, if P holds on the current state, the Q holds on every reachable stable state. This is precisely the semantic meaning of the Hoare tuple of partial correctness.

Likewise, the Hoare tuple of total correctness is rendered:

$$[P]c[Q] \qquad \equiv \qquad [\![c]\!] \models P \Rightarrow \nabla\bot \wedge \bigcirc Q \tag{11.2}$$

This version additionally requires that the program converge if P holds, and this is exactly the usual meaning of total correctness.

One could imagine taking this idea a step further by requiring that the states of an LTS also be equipped with separation algebras [DHA09]. Under some assumptions about how the separation algebra and the transition relation interact, this should allow one to develop a nice theory of branching-time temporal logic with spatial separation into which the Hoare tuples of separation logic could be embedded.

11.3 Pointer Escape

I claimed in chapter 10 that the majority of CompCert is already proved correct with respect to the strongest notion of refinement available. However, CompCert's correctness statement is not as complete as we would like it to be, because the external events model is too restrictive. Recall that CompCert's event model allows only a very limited form of pointer value to be communicated with the environment — only global static data can be referenced. Furthermore, the current state of the memory is not accessible to the external world via trace events. This means that very important classes of system calls cannot be represented at all. For example, the POSIX file API communicates by reading and writing into buffers; without the ability to pass pointers and access memory, CompCert's external system call model cannot express these functions.

This also means it we cannot use CompCert's current event model to reason about shared-memory multithreading or separate compilation. The reason is essentially the same — we need to be able to pass pointers and memory state across the external function boundary. For multithreaded programs, we need to expose memory state to the environment so that concurrency primitives (like lock acquire and release) can be modeled using the external function call interface. For separate compilation, we need functions defined in separate compilation units to be able to communicate via standard function calls. Otherwise, it would be impossible to, for example, define a data structure library in a separate file, or in a statically linked or shared library. Such a restriction would destroy the main method for achieving modularity in C.

However, we cannot simply allow arbitrary pointer values and memory state to "escape" into the environment. The reason raw pointers are hidden from the environment is that some of CompCert's compilation passes rearrange memory layouts (e.g., by spilling register locations), so the raw pointer values produced from the source program may not be equal to the pointer values of the compiled program. Likewise, the raw memory state is hidden because the source and compiled memory images will not be identical.

I believe the correct approach to this problem is to allow pointers and memory state to be observable at external events, but to weaken our notions of bisimulation/refinement so that we must view the actual values of pointers abstractly. This is similar to how bisimulation forces us to view the identity of *states* abstractly; states cannot be distinguished except by how they behave. Likewise, we should not be able to distinguish pointers that point to indistinguishable areas of memory.

I believe that an approach based on *environmental bisimulation* will allow us to tackle this pointer escape issue. Environmental bisimulation [SKS07, KLS11] enhances ordinary bisimulations by adding a third component to bisimulation relations. This third component abstractly captures the state of knowledge of the environment. In this setting, the actions taken by related processes do not have to be *identical*, but are rather *related* to each other in a way determined by the knowledge of the environment.

For our domain of interest, environment's knowledge has to do with previously escaped pointer values. As the environment observes output from a program, it learns about the pointer values that get leaked. When the environment returns control to the program, its reply may be constructed using the knowledge about pointers it has previously acquired; however, it may not construct arbitrary pointers "out of thin air." By using the technique of environmental bisimulation, it should be possible to allow pointers to escape into the environment, but require that they be treated as black boxes. This prevents the environment from examining the bit pattern of pointers, and essentially only allows doing pointer arithmetic and dereferencing.

Summi [Sum09] has presented a notion of environmental bisimulation for an ML-style language with explicit allocation and deallocation. Summi's bisimulation is defined directly over the syntax of programs, rather than abstractly over LTSs, and he does not examine the question of characteristic logics for these types of bisimulations. Environmental bisimulations have been applied to a variety of similar higher-order languages based on the λ-calculus [SKS07].

This previous work has all been done over the syntax of particular varieties of some high-level core language. To my knowledge, no attempt has yet been made to define environmental bisimulations abstractly over generic notions of small-step operational semantics using structures like labeled transition systems. I believe that defining a generic notion of environmental bisimulation over LTS-like structures is a worthwhile goal for future research. Especially, I find the question of determining what a characteristic logic for environmental bisimulation should look like quite interesting. In contrast to all the logics seen in this thesis, we would need modes describing the behavior of processes that do not make direct reference to the identity of transition labels. It is not immediately clear what is the best way to do this.

I suspect that a suitably generalized notion of environmental bisimulation would lend itself well to combination with the ideas of behavioral and choice refinement, giving a comprehensive solution to the pointer escape problem.

11.4 Relational Properties

Throughout this thesis, I have emphasized the importance of understating what properties are preserved by a compiler correctness theorem. The characteristic logics presented here precisely capture the observations that can be made about a program. However, all the properties thus captured are *unary* program properties — that is, they are properties of a single program run.

Sometimes we are instead interested in *relational* program properties, which reference two or more program runs simultaneously. For example, parametricity [Rey83] and noninterference properties [SM03] are most naturally stated as relations on pairs of program runs. Benton has argued the case that relational properties should be considered seriously when thinking about the correctness of program transformations [Ben04].

Some relational properties can be *factored* into unary properties [Ber11], but it does not seem possible for all relational properties. Refinements, in particular, can destroy relational properties that hold over source-level programs.

Consider the case of noninterference, a popular way to formally define the absence of certain kinds of information flow through a software system. Richard Forster, in his dissertation [For99], defined a notion of noninterference over nondeterministic processes that generalizes the usual notions (which only make sense for deterministic processes). However, he notes that refinements with respect to nondeterministic choices (in the style of CSP) can destroy noninterference relations. Roughly, this is because two processes that stand in a noninterference relation can be compiled in ways that make different choices regarding how to resolve internal nondeterminism. These differences in implementation choices can generate an information channel that did not exist before. Essentially the same argument applies to choice refinement as defined in chapter 6. I suspect behavioral refinement would likewise cause problems with noninterference properties.

However, all is not lost. If we restrict ourselves to processes that are maximal under refinement, we can preserve all "reasonable" relational properties through compilation. For behavioral and choice refinement, the maximal processes are those that are safe and deterministic.

Consider some relational property R over pairs of processes. R is a "reasonable" property if it is closed under $\cong_{\mho\Delta B}$ bisimulation. That is, we require that whenever $(x, y) \in R$, $x \cong_{\mho\Delta B} x'$ and $y \cong_{\mho\Delta B} y'$, it is the case that $(x', y') \in R$. Essentially, R should not be more discriminating than $\cong_{\mho\Delta B}$.

Suppose we have a compiler T, which is proved correct with respect to branching behavioral choice refinement, $\sqsubseteq_{\mho CB}$. Further, suppose x and y are safe, deterministic programs and $(x, y) \in R$

for some reasonable R. We compile x and y, so that $T(x) = x'$ and $T(y) = y'$. T has been proved correct so $x \sqsubseteq_{\mho CB} x'$ and $y \sqsubseteq_{\mho CB} y'$. However, because x and y are safe and deterministic, they are thus *maximal*, which means we can strengthen the relationships to $x \cong_{\mho\Delta B} x'$ and $y \cong_{\mho\Delta B} y'$. Because R is reasonable, we can deduce that $(x', y') \in R$, as desired.

The restriction to safe and deterministic programs is a bit disappointing. However, nothing prevents us from reasoning directly about relational properties that are closed under refinement; for properties like these, the safe and deterministic restriction does not apply. For information-flow properties in particular, it seems like the restrictions are necessary; information security properties do not (and should not!) apply to programs that might go wrong, and can be damaged by the resolution of internal nondeterminism. This is simply an inherent characteristic of information security properties, and not a weakness of the refinement-based approach to compiler correctness.

Therefore, even though I do not treat relational properties directly in this work, we can nonetheless get a handle on what relational properties are be preserved, and under what conditions. An interesting future direction for research would be to survey existing relational properties from the published literature and see if they can fit into the pattern described above. Additionally, it would be interesting to see if characteristic logics for relational properties could be developed — this would help to more directly illuminate which relational properties are preserved by compilation.

11.5 Concluding Remarks

In this thesis I have been concerned with the question of compiler correctness. In particular, I have tried to definitively state the form that a compiler correctness theorem ought to take when we are concerned with compiling languages like C. In summary, my approach as been thus: find the smallest extensional (i.e., containing bisimulation) preorder on program meaning (interpreted as small-step operational semantics) that has a nice theory and allows the program transformations of interest.

This approach has the pleasant feature that it clearly highlights the trade-off between allowing more program transformations versus preserving program properties. For example, when moving from branching bisimulation to behavioral refinement, we gain the ability to expand undefined behavior, but lose the ability to reason classically about program behavior. Likewise, when moving from branching bisimulation to choice refinement, we gain the ability to resolve internal nondeterminism in the program at the price of restricting the form of progress properties that are preserved by compilation. The strength of this approach is also demonstrated by the possible extensions discussed in this chapter. All the extensions should be possible to implement in an orthogonal way, without much changing the basic setup.

My motivating setting for this work is C, and the CompCert compiler in particular. For this setting, I believe the correct notion is the behavioral choice refinement of chapter 7, perhaps enhanced in the ways discussed in this chapter. The discussions of chapters 9 and 10 demonstrate that behavioral choice refinement can be successfully applied to the task of verifying a C compiler. Furthermore, the preceding chapters have argued that behavioral choice refinement is the weakest extensional refinement that can properly handle the notions of undefined and unspecified behavior, as found in the C specification [ISO11].

This puts us in the excellent position of knowing *what* it is we want to prove, *why* it is the right thing to prove, and *how* to go about proving it.

Bibliography

[AB07] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step C minor. In *Proc. Intl. Conference on Theorem Proving in Higher-Order Logics (TPHOLS)*, volume 4732 of *LNCS*, pages 5–21. Springer, 2007.

[Abr91] Samson Abramsky. A domain equation for bisimulation. *Journal of Information and Computation*, 92(2):161–218, 1991.

[AFV01] Luca Aceto, Wan Fokkink, and Chris Verhoef. Structural operational semantics. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 197–292. Elsevier, 2001.

[AILS12] Luca Aceto, Anna Ingólfsdóttir, Paul Blain Levy, and Joshua Sack. Characteristic formulae for fixed-point semantics: a general framework. *Mathematical Structures in Computer Science*, 22:125–173, 2012.

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[App11] Andrew W. Appel. Verified software toolchain. In *Proc. European Symposium on Programming Languages and Systems (ESOP)*, volume 6602 of *LNCS*, pages 1–17. Springer, 2011.

[BB85] Errett Bishop and Douglas Bridges. *Constructive Analysis*. Springer, 1985.

[Ben04] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 14–25. ACM, 2004.

[Ber11] Lennart Beringer. Relational decomposition. In *Proc. Interactive Theorem Proving (ITP)*, volume 7215 of *LNCS*, pages 369–389. Springer, 2011.

[BHMY89] William R. Bevier, Warren A. Hunt, J. Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, pages 441–428, 1989.

[BIM95] Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced. *Journal of the ACM*, 42(1):232–268, 1995.

[Blo89] Bard Bloom. *Ready Simulation, Bisimulation, and the Semantics of CCS-Like Languages*. PhD thesis, Massachusetts Institute of Technology, 1989.

[Bou49] Nicolas Bourbaki. Sur le théorème de Zorn. *Archiv der Mathematik*, 2(6):434–437, 1949.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[Com] The CompCert formally verified C compiler. http://compcert.inria.fr.

[Coq] The Coq proof assistant. http://coq.inria.fr.

[CP90] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In *Proc. Intl. Conference on Computer Logic (COLOG-88)*, volume 417 of *LNCS*, pages 50–66. Springer, 1990.

[DFH95] Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications*, volume 902 of *LNCS*, pages 124–138. Springer, 1995.

[DHA09] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *Proc. Asian Symposium on Programming Languages and Systems (APLAS)*, volume 5904 of *LNCS*, pages 161–177. Springer, 2009.

[DNV95] Rocco De Nicola and Frits Vaandrager. Three logics for branching bisimulation. *Journal of the ACM*, 42(2):458–487, 1995.

[Dun95] J. Michael Dunn. Positive modal logic. *Studia Logica*, 55:301–317, 1995.

[FBU09] Dario Fischbein, Victor Braberman, and Sebastian Uchitel. A sound observational semantics for modal transition systems. In *Proc. Intl. Colloquium on Theoretical Aspects of Computing (ICTAC)*, volume 5684 of *LNCS*, pages 215–230. Springer, 2009.

[Fel91] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

[For99] Richard Forster. *Non-interference properties for nondeterministic processses.* PhD thesis, University of Oxford, 1999.

[GP02] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.

[HAZ08] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *Proc. European Symposium on Programming Languages and Systems (ESOP)*, volume 4960 of *LNCS*, pages 353–367. Springer, 2008.

[HJS01] Michael Huth, Radha Jagadeesan, and David A. Schmidt. Modal transition systems: A foundation for three-valued program analysis. In *Proc. European Symposium on Programming Languages and Systems (ESOP)*, volume 2028 of *LNCS*, pages 155–169. Springer, 2001.

[HM80] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In *Proc. Intl. Colloquium on Automata, Languages and Programming (ICALP)*, volume 85 of *LNCS*, pages 299–309. Springer, 1980.

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32:137–161, 1985.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[HS85] Matthew Hennessy and Colin Stirling. The power of the future perfect in program logics. *Information and Control*, 67:23–52, 1985.

[Ing97] Anna Ingólfsdóttir. Weak semantics based on lighted button pressing experiments. In *Selected Papers from the 10th Intl. Workshop on Computer Science Logic (CSL)*, volume 1258 of *LNCS*, pages 226–243. Springer, 1997.

[ISO11] *ISO/IEC 9899:2011 – Programming languages – C.* ISO, 2011.

[Kau88] Matt Kaufmann. A user's manual for an interactive enhancement to the Boyer-Moore theorem prover. Technical Report 19, 1988. Computational Logic, Inc.

[Kle52] Stephen C. Kleene. *Introduction to Metamathematics.* Van Nostrand, 1952.

[KLS11] Vasileios Koutavas, Paul Blain Levy, and Eijiro Sumii. From applicative to environmental bisimulation. *ENTCS*, 276(0):215–235, 2011. Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII).

[Koz83] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[Kri63] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[Lar90] Kim G. Larsen. Modal specifications. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 232–246. Springer, 1990.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 42–54. ACM, 2006.

[Ler09a] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[Ler09b] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[LP08] Dirk Leinenbach and Elena Petrova. Pervasive compiler verification – from verified programs to verified systems. *ENTCS*, 217:23–40, 2008.

[LS91] Kim G. Larsen and Arne Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, 1991.

[LT88] Kim G. Larsen and Bent Thomsen. A modal process logic. In *Proc. Symposium on Logic in Computer Science (LICS)*, pages 203–210. IEEE, 1988.

[LV10] Gerald Lüttgen and Walter Vogler. Ready simulation for concurrency: It's logical! *Information and Computation*, 208(7):845–867, 2010.

[Mar73] Per Martin-Löf. An intuitionistic theory of types: predicative part. *Logic Colloquium*, pages 73–118, 1973.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

[Mil81] Robin Milner. A modal characterisation of observable machine-behaviour. In *Proc. Colloquium on Trees in Algebra and Programming (CAAP)*, volume 112 of *LNCS*, pages 25–34. Springer, 1981.

[MO97] Markus Müller-Olm. *Modular Compiler Verification*, volume 1283 of *LNCS*. Springer, 1997.

[MO98] Markus Müller-Olm. Derivation of characteristic formulae. *ENTCS*, 18:159–170, 1998. MFCS'98 Workshop on Concurrency.

[NV07] Sumit Nain and Moshe Y. Vardi. Branching vs. linear time: semantical perspective. In *Proc. Intl. Conference on Automated Technology for Verification and Analysis (ATVA)*, volume 4762 of *LNCS*, pages 19–34. Springer, 2007.

[Par81] David Park. Concurrency and automata on infinite sequences. In *Proc. Theoretical Computer Science*, volume 104 of *LNCS*, pages 167–183. Springer, 1981.

[PE88] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 199–208. ACM, 1988.

[Pit03] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics, 1981.

[Pnu85] Amir Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *Proc. Incl. Colloquium on Automata, Languages and Programming (ICALP)*, volume 194 of *LNCS*, pages 15–32. Springer, 1985.

[Pol98] Robert Pollack. How to believe a machine-checked proof. In Giovanni Sambin and Jan Smith, editors, *Twenty-Five years of constructive type theory*, volume 36 of *Oxford Logic Guides*, pages 205–220. Oxford, 1998.

[Rey83] John C. Reynolds. Types, abstraction, and parameteric polymorphism. *Information Processing*, pages 513–523, 1983.

[San12] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction.* Cambridge University Press, 2012.

[Ser84] Gisèle Fischer Servi. Axiomatizations for some intuitionsitic modal logics. *Rendiconti del Seminario Mathematico*, 42(3):179–194, 1984.

[Sim94] Alex Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic.* PhD thesis, University of Edinburgh, 1994.

[SKS07] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. In *Proc. Symposium on Logic in Computer Science (LICS)*. IEEE, 2007.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *Selected Areas in Communications*, (1):5–19, 2003.

[Sti87] Colin Stirling. Modal logics for communicating systems. *Theoretical Computer Science*, 49:311–347, 1987.

[Sum09] Eijiro Sumii. A theory of non-monotone memory (or: contexts for free). In *Proc. European Symposium on Programming Languages and Systems (ESOP)*, volume 5502 of *LNCS*, pages 237–251. Springer, 2009.

[ŠVZN$^+$11] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Relaxed-memory concurrency and verified compilation. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 43–54. ACM, 2011.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Uli92] Irek Ulidowski. Equivalences on observable processes. In *Proc. Symposium on Logic in Computer Science (LICS)*, pages 148–159. IEEE, 1992.

[vG93] Rob J. van Glabbeek. The linear time - branching time spectrum II. In *Proc. Intl. Conference on Concurrency Theory (CONCUR)*, volume 715 of *LNCS*, pages 66–81. Springer, 1993.

[vGLT09] Rob J. van Glabbeek, Bas Luttik, and Nikola Trčka. Branching bisimilarity with explicit divergence. *Fundamenta Informaticae*, 93(4):371–392, 2009.

[vGW96] Rob J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43:613–618, 1996.

[Wal90] David J. Walker. Bisimulation and divergence. *Information and Computation*, 85(2):202–241, 1990.

[Wer97] Benjamin Werner. Sets in types, types in sets. In *Proc. Symposium on Theoretical Aspects of Computer Science (TACS)*, volume 1281 of *LNCS*, pages 530–546. Springer, 1997.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[Wit51] Ernst Witt. Beveisstudien zum Staz von M. Zorn. *Mathematische Nachrichten*, 4:434–438, 1951.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294. ACM, 2011.

[You89] William D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5:493–518, 1989.

[ZNMZ12] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 427–440. ACM, 2012.