

A PROOF THEORY FOR LOOP-PARALLELIZING
TRANSFORMATIONS

CHRISTIAN JAMES BELL

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: DAVID P. WALKER

JUNE 2014

© Copyright by Christian James Bell, 2014.

All rights reserved.

Abstract

The microprocessor industry has embraced multicore architectures as the new dominant design paradigm. Harnessing the full power of such computers requires writing multithreaded programs, but regardless of whether one is writing a program from scratch or porting an existing single-threaded program, concurrency is hard to implement correctly and often reduces the legibility and maintainability of the source code. Single-threaded programs are easier to write, understand, and verify.

Parallelizing compilers offer one solution by automatically transforming sequential programs into parallel programs. Assisting the programmer with challenging tasks like this (or other optimizations), however, causes compilers to be highly complex. This leads to bugs that add unexpected behaviors to compiled programs in ways that are very difficult to test. Formal compiler verification adds a rigorous mathematical proof of correctness to a compiler, which provides high assurance that successfully compiled programs preserve the behaviors of the source program such that bugs are not introduced. However, no parallelizing compiler has been formally verified.

We lay the groundwork for verified parallelizing compilers by developing a general theory to prove the soundness of parallelizing transformations. Using this theory, we prove the soundness of a framework of small, generic transformations that compose together to build optimizations that are correct by construction. We demonstrate it by implementing several classic and cutting-edge loop-parallelizing optimizations: DOALL, DOACROSS, and Decoupled Software Pipelining. Two of our main contributions are the development and proof of a general parallelizing transformation and a transformation that coinductively folds a transformation over a potentially nonterminating loop, which we compose together to parallelize loops. Our third contribution is an exploration of the theory behind the correctness of parallelization, where we consider the preservation of nondeterminism and develop bisimulation-based proof techniques. Our proofs have been mechanically checked by the Coq Proof Assistant.

Acknowledgements

I thank my advisor, David Walker, for his receptiveness and patience with my ideas, for providing guidance and excellent feedback on my writing and presentation skills, and for suggesting that I verify a parallelizing optimization (and allowing me to follow the rabbit down the hole). I also thank Andrew Appel, who inspired me to mechanize my proofs, was always able and willing to engage me in deep technical conversations, and for inviting me to join him in jazz improvisational lessons. As well, my thanks go to David August and his Liberators (a.k.a. graduate students) for answering all my questions about their parallelizing compiler and various optimizations. I am also grateful to our graduate student coordinator, Melissa Lawson, who put in an exceptional effort toward making our department run smoothly and was always there to guide me through each stage of my degree.

I thank my siblings, Nelson and Jordan, for instilling in me the belief that I always needed to catch up with them – a masterful feat (among many, more humorous, tricks they pulled on me), considering that this even extended into areas in which they held no interest. Some of my earliest memories are of my sister teaching me long division with bright, plastic, magnetic numbers. I owe more than can be expressed to my parents, Karen and David, for indulging in my various curiosities, from padlocks to building robots (i.e. stringing tin cans together with wire and connecting a battery – I was only five...) to computers; especially for putting up with me disassembling their electronics and for the hamburger I put in the VCR (again, only five...). I thank Uncle Gary for feeding my interest in electronics by giving me his old electronics books and tools. To Herb and Ernie: thanks for trusting Ana and me with your home.

I would also like to recognize those who contributed to a healthy, fun, and mostly productive research/office environment: Rob Dockins, Lennart Beringer, Aquinas Hobor, Matthew Meola, Cole Schlesinger, and Gordon Stewart. In addition to random lunch discussions about politics, whiskey, and beer, they patiently listened as my ideas

progressed from incoherent ramblings to articulated (and publishable!) ideas about parallelization. I am especially grateful to Rob for our discussions on bisimulations.

I must thank my Princeton University friends – Justin Brown, Sam Taylor, Jeff Dwoskin, Matthew Meola, Jebro Lit, June Young, Rodolfo Rios-Zertuche Rios-Zertuche, Noah Jafferis, Seth Dorfman, Jess Hawthorne, David Liao, Divjot Sethi – who made Princeton a wonderful place to live, and for their dedicated efforts to go behind my back to ensure marital bliss. They found me a wife (“Hey! You shared an office with CJ; you two seem to get along really well...”), celebrated my (fake) bachelor party at Hooters, and conspired to marry us by the Powers vested in the Chair of the Princeton University Graduate College House Committee (managing to convince the residents and staff of said college that it was legitimate); only upon affixing a top hat to my head, playing “Maneater,” and forming the aisle to commence the ceremony did they bother to tell me. Only to later kidnap my bride, shred, dissolve, and explode our Graduate College House Certificate of Marriage – just so we could have a legally-recognized marriage in the State of Washington several months later. It happened so fast that I am unable to assign proper blame where deserved. But I distinctly recall that all of the suspects wore tuxedos. With friends like these, it’s always Thursday somewhere.

And finally, I am eternally grateful for the emotional and logistical support of my wife, Ana; my best friend and love. Ever since we met on our first day of graduate school, she was always willing to go on an adventure, share a story, or watch a movie, and continued to stand by me throughout the happy and difficult times that followed.

This research is funded in part by NSF grants OCI-1047879 and 1016937. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF

In memory of my mother.

Contents

Abstract	iii
Acknowledgements	iv
1 Introduction	1
1.1 Related work: parallelizing optimizations	3
1.2 Related work: formal verification of compilers	5
1.3 Nondeterminism	9
1.4 Mechanized proofs	12
1.5 Thesis structure	12
2 A Transformation Framework for Loop Parallelization	14
2.1 Assertions	16
2.2 Example: DOALL parallelization	19
2.2.1 Parallelize the loop body	21
2.2.2 Folding parallelism into the whole loop	22
2.2.3 Loop schema & folding	23
2.2.4 Bounded loops	24
2.2.5 Combining parallelism	27
2.2.6 Loop parallelization	28
2.3 Example: Decoupled software pipelining	30
2.3.1 The DSWP transformation	32

2.3.2	Phase 1: Decoupling dependencies	33
2.3.3	Phase 2: Parallelizing the loop body	37
2.3.4	Phase 3: Loop folding	38
2.4	Example: DOACROSS parallelization	42
2.4.1	Phase 1: Decoupling	45
2.4.2	Phase 2: Parallelizing the loop body	48
2.4.3	Phase 3: Loop folding	50
2.4.4	Phase 4: Maximize concurrency	52
2.5	Conclusion & related work	55
2.5.1	Remarks on loop folding	55
2.5.2	Remarks on parallelization	56
2.5.3	Strengthening the assertion logic	57
2.5.4	Finite communication channels	57
2.5.5	Existing proof of DSWP	58
2.5.6	Existing verified parallelizing optimizations	60
2.5.7	Automation	61
2.5.8	Conclusion	62
2.6	Summary of framework rewrite rules	62
3	Foundations of Parallelization	67
3.1	Introductory example	69
3.2	CCS-Seq	74
3.2.1	The parallelization transformation	77
3.3	Contrasimulation	79
3.4	Delayed observations	85
3.5	Proof of parallelization	88
3.5.1	Preliminary definitions	88
3.5.2	Proof	90

3.6	Conclusion	94
3.6.1	Coq proof development	94
4	A Semantic Framework	95
4.1	Technical development	96
4.1.1	Machine state	96
4.1.2	Expressions	100
4.1.3	Predicates	100
4.1.4	Iteration bounds	102
4.1.5	Program syntax	104
4.1.6	Labels	106
4.1.7	Operational semantics	107
4.1.8	Free variables	109
4.2	Program equivalence	111
4.2.1	Bisimilarity and contrasimilarity	111
4.3	Congruence	113
4.3.1	Supporting lemmas	113
4.3.2	Algebraic	114
4.3.3	Compositional	115
4.4	Loop folding	119
4.4.1	Loop decomposition	120
4.4.2	The proof of loop folding	124
4.5	Parallelization	131
4.6	Hoare triples	134
4.7	Coq Proof Development	135
5	Conclusion & Future Work	137
5.1	Concluding remarks	137

5.1.1	Loop folding	137
5.1.2	Parallelization	138
5.1.3	Loop parallelization	139
5.1.4	Coq proofs	140
5.2	Future work	140
5.2.1	Divergence sensitivity	140
5.2.2	Incorporation into an existing verified compiler	141
	Bibliography	142

Chapter 1

Introduction

For most of the history of computer science, programmers could count on the fact that year after year, hardware advances would make virtually all their applications run faster. Unfortunately, this is no longer the case. Despite continuing increases in chip transistor counts, hardware designers are unable to produce new single-processor chips that perform significantly better than their predecessors. Consequently, the microprocessor industry has embraced multicore architectures as the new dominant design paradigm.

The shift to multicore processors has significant ramifications for application developers. No longer do these developers – and their substantial investment in existing programs used by scientists, banks, industries, and consumers – get increasing performance for free with each new generation of processors. They must find a way to parallelize their [sequential] applications in order to exploit the many cores now supplied by chip manufacturers.

This is a daunting prospect for even new applications; parallelism is often at odds with legibility and maintainability, and requires expertise in order to implement correctly. One must take great care to avoid introducing concurrency errors when transforming a well-tested, successful application. A solution is to use emerging com-

pilers technology to automatically, partly automatically, or mostly manually manage the parallelization, allowing programmers to take advantage of the increasing number of cores found in modern CPUs with little additional effort. These optimizations free the programmer from dealing with the inherent complexities of writing multithreaded code directly and bring new vigor to a large base of existing sequential source code by the simple act of recompilation.

Automated optimizations come with a high payoff, but at a cost – compilers are large and complex, even without parallelizing optimizations, and are prone to subtle bugs that are difficult to track. In consumer-facing applications, it can be worth the cost. But for critical systems, like medical devices, satellites, banking, security, and transportation, failure is unacceptable. Thus it is common practice to prove the correctness of these programs by hand, by using model-checking tools, or by carefully inspecting the generated machine code. There is no room for additional risk, so compiler optimizations are simply turned off.

Advances in machine-checked proofs and many man-hours have given rise to *verified compilers*, which provide a very high assurance that the compiler does not contain bugs (or at least does not introduce bugs into compiled programs). These compilers, and particularly the optimizations they can perform, are currently the focus of active research. For parallelizing optimizations, the primary focus of research has been to develop better program analysis techniques to determine when and where they may be applied, some of which has been formally verified. However, the formal justification of the program transformations that such optimizations apply has received relatively little attention – only a handful of published papers prove the soundness of any kind of parallelization transformation, and no verified compiler currently implements one.

Thesis: We develop a theory for proving the soundness of loop parallelization, and thus study the intersection of verified compilers and parallelizing optimizations. To

apply this theory, we create a framework of local, generic, and sound rewrite rules that programmers or compilers may compose together in order to parallelize programs in a way that is correct by construction. We demonstrate this framework by implementing instances of classic parallelizing optimizations: DOALL, DOACROSS, and decoupled software pipelining (DSWP) [29, 17, 28]. Our main technical results are:

- A bisimulation-style relation that holds for parallelization (and loop folding) in the presence of internal choice.
- A proof of soundness for a general transformation that can parallelize sequential programs and also combine parallel programs together, where the programs may have nontrivial termination properties and may communicate over shared queues.
- A proof of soundness for a general loop transformation, which uses a combining function (i.e. a program transformation) to fold all of the iterations of a loop together, and which does not require the loop to terminate. When composed with a transformation that combines parallel programs, loop folding is powerful enough to implement each of the classic loop parallelizing transformations (DOALL, DOACROSS, and DSWP).
- A formalization of the theorems and proofs appearing in this thesis that is checked by the Coq Proof Assistant [21].

1.1 Related work: parallelizing optimizations

Parallelizing optimizations have a long history and there are many techniques, such as vectorization, transactional memory, and loop parallelism. Banerjee et al. [2] and Wolfe [45] thoroughly review the classic optimizations.

We study optimizations that rewrite single loops into multiple loops that run concurrently. A canonical example targets loops with no loop-carried dependencies

(i.e. between iterations), called DOALL loops, by dividing the iterations of a sequential DOALL loop into multiple parallel loops that run independently of each other, called DOALL parallelization. These were first developed in the early Fortran compilers [29, 45], and many strategies have arisen to enable them, ranging from programmer-assisted (e.g. `doall` loop constructs in the language) to advanced static analyses that automatically identify dependencies between iterations.

When applicable, DOALL parallelism is very effective. However, proving that iterations are independent is hard (generally undecidable) and, in many programs of interest, the iterations are not necessarily independent. To enable more applications of DOALL parallelism, loop-carried dependencies can sometimes be eliminated by reorganizing the loop structure. Techniques such as polyhedral methods, loop skewing, and loop distribution attempt to do just that. But they are foiled by complicated control flow, shared memory, uncounted loops (i.e. loops with no index variable to count each iteration), and nonterminating loops.

DOACROSS extends DOALL parallelization and resolves some of the above issues by inserting synchronization, in order to respect loop-carried dependencies, so that the iterations take turns accessing shared resources [17]. Increased parallelism comes from the portions of iterations that do not rely on loop-carried dependencies, which overlap in time. In the worst case, the iterations effectively run in sequence. But even in the ideal case, a slowdown in one iteration, perhaps due to a cache miss or synchronization latency, will delay all subsequent iterations in every thread. As a result, DOACROSS suffers a *multiplicative* slowdown (with respect to the number of iteration executed) as communication latency increases.

A third approach is to exploit pipelined parallelism in loops, which is the premise of DSWP [27, 28]. DSWP breaks a loop body into a sequence of tasks with acyclic dependencies, forming a pipeline, then runs each stage of the pipeline in parallel, where each stage communicates dependencies to the next using shared queues. Because

the parallel stages have acyclic dependencies that are buffered, they do not run in lock-step; a slowdown in one stage does not necessarily delay the other stages. Thus communication latency only *adds* to the overall running time. An added advantage of DSWP is that it effectively reorganizes the dependencies so that other optimizations are easier to apply to each stage. For example, parallel-stage decoupled software pipelining (PS-DSWP) first performs DSWP, and then attempts to apply DOALL parallelization on the resulting stages [34].

1.2 Related work: formal verification of compilers

A compiler is formally verified if the source and compiled (target) programs are proven to be within a relation such as behavioral equivalence or semantic refinement. Formal verification of a compiler provides a very high assurance that the compiler does not contain bugs, or at least does not introduce bugs into compiled programs; especially when the proofs are machine-checked using a tool such as the Coq Proof Assistant or Isabelle [21, 31]. Proof-carrying code, translation validation, and compiler verification are some general approaches that have been used to verify a compiler.

There is a delicate balance between choosing a weak or strong correctness criterion. If too weak, then the compiler may introduce or remove termination, divergence, or crashing behaviors; but proof automation and adding new features is easier. If too strong, then the proof becomes more difficult or tedious, and the compiler may not be allowed to apply certain optimizations or be able to satisfy the criterion at all. Another challenge is to find a relation that is compositional, which allows modular proofs of correctness and separate compilation of modules. Some common criteria that are used to verify compilers, in order of weak to strong, are semantic equivalence, trace equivalence, and variations on bisimulation such as weak bisimulation and branching

bisimulation (Sangiorgi [36] is a good introduction for these relations and we define several of them in chapter 3).

Proof-carrying code (PCC) is a mechanism to generate a *certificate*, a formal and independently-checkable proof, along with the compiled program to prove that it satisfies certain kinds of specifications [24]. Usually, PCC is used to ensure that the resulting machine code is type- and memory- safe, but not necessarily equivalent to the source program. When running the program, the user or system can make sure that the program satisfies the specification (type or memory safety) by running a validator on the accompanying certificate. Only the validator, which is typically much smaller and less complex than the compiler, needs to be either trusted or formally verified.

Translation validation is a technique that can work independently of the compiler (although the compiler may also provide hints): it takes the source and compiled programs and attempts to automatically construct a proof that they are equivalent [32]. It has been used to prove the correctness of parts of the GCC compiler and of optimization frameworks [25, 18]. However, translation validation tends to be limited by lack of information (some of which the compiler could provide), is not easily extended, and is not modular. This last issue has been partially addressed by parameterized equivalence checking [18]. The main advantage of translation validation is that it can be used with any existing compiler with little or no modification to the compiler.

Verified compilers are proved correct by construction: compilation either results in a correct program or it fails to compile. As depicted in Fig. 1.1, a verified compiler translates a source program into successive stages of intermediate languages until finally being translated into the target (compiled) language, which is typically an assembly language, machine code, or bytecode. Optimizations may be performed at each translation step and also within the same intermediate language. Each transla-

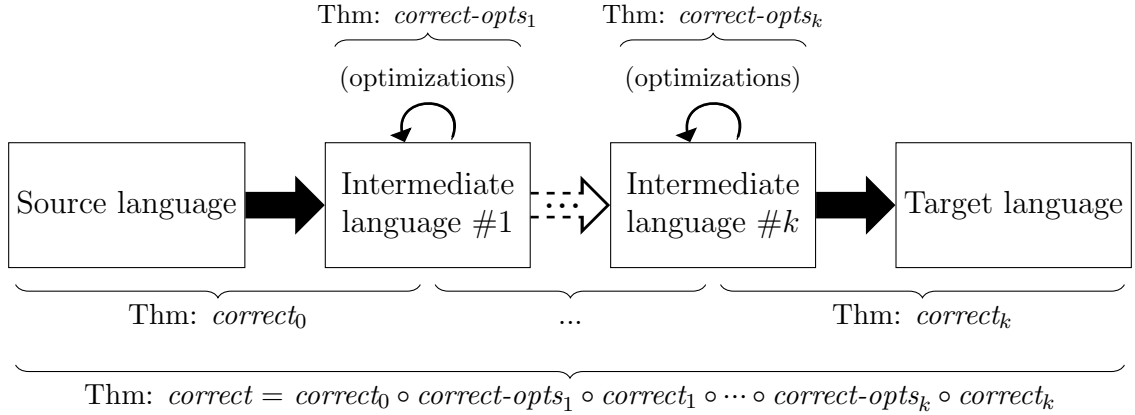


Figure 1.1: Structure of a verified compiler. The source language, for example, could be C, Java, ML, or Haskell, and the target language could be assembly, bytecode, or machine code.

tion is proven to preserve a correctness criteria, and then the proofs of correctness are composed together (e.g. via transitivity) to form an end-to-end proof of correctness between all source and compiled programs. Well-known examples of verified compilers include CompCert (and variations such as CompCertTSO, and XCERT), Piton, Vellvm, and Bedrock [20, 40, 41, 23, 46, 9].

CompCert supports a substantial subset of the C language, including recursive functions, pointers, structs, and function pointers. Like regular compilers, it translates a C source program into successive stages of intermediate languages until it finally generates PowerPC, ARM, or x86 assembly code. However, each translation is proved correct with respect to a *forward simulation* (i.e., the target simulates the source); by transitivity, a forward simulation holds at the top level between the input C program and generated assembly code. Specifically, CompCert uses *weak simulation*, which is equivalent to weak bisimulation in this context because the target program is deterministic.

There are two main efforts to add support for compiling concurrent programs with CompCert: the Verified Software Toolchain (VST) and CompCertTSO [1, 40]. In order to minimize changes to CompCert, VST treats threading primitives as external

function calls and uses a strong memory model. It also uses a cooperative threading semantics that yields upon calling each synchronization instruction. Because they only consider data-race free programs, this is equivalent to a preemptive interleaving semantics. In contrast, CompCertTSO diverges further from CompCert by adopting a relaxed memory model directly and adding threading primitives to the underlying semantics (rather than treat them as external function calls).

Vellvm mechanically formalizes the sequential semantics of LLVM [46]. Like CompCert, it is developed in the Coq Proof Assistant and the correctness criterion for the transformations (e.g. SoftBound) and optimizations they implement is a simulation. The advantage of targeting LLVM is that it is used by many existing compilers developed in both academia and industry.

The Bedrock Structured Programming System has more specific goals than the previously mentioned compilers: to compile low-level systems software, such as operating systems or drivers, that are paired with Hoare logic specifications [9]. Bedrock programs are written directly in Coq, using its built-in “Notation” system, and are evaluated into a low-level Bedrock bytecode. Instead of providing a high-level source language with structured control flow, Bedrock provides an extensible macro facility that generate bytecode. Many control structures and high-level language features can be implemented as macros, and it is these macros that are verified by the compiler.

The correctness criterion of Bedrock is to preserve the Hoare logic specification of a program. Although generally much weaker than the simulation/bisimulation criteria in use by other verified compilers, it is roughly equivalent when the programs are deterministic and guaranteed to terminate; it is often considered good enough. An advantage of using the weaker correctness criterion is that extending Bedrock with new macros is much easier than adding new language features or optimizations to the other verified compilers, and this allows most of the proofs to be automated.

1.3 Nondeterminism

A program may have many behaviors, and which ones manifest can depend on interactions with other programs, the operating system, user input, hardware events, or thread scheduling. When it is impractical or impossible to know which behavior a program or its environment will exhibit, we can abstract the set of possibilities as an arbitrary choice. This is an example of *nondeterminism*, which is when the choice between behaviors is not fully specified; it may be stochastic, defined later (for example, when compiled modules are linked together), or simply unknown.

To make reasoning about the behavior of a large system more manageable, we break it into smaller components and reason about them in isolation (i.e. *locally*). Then we compose proofs about the smaller units together into a statement about the entire system. When focusing on a single unit, such as a process or subprogram, we assume that the other units obey some given specification. However, the specifications of the units may not tell us everything about how they interact with the unit of focus, so we use nondeterminism to model everything that is not specified about the other units. When nondeterminism models a choice that an observer makes – for example, from the operating system, other units, or user – then we call it an *external choice*.

Internal choice is nondeterministic behavior that the local unit may exhibit; i.e. choices that are not controlled by the observer. Some examples can include memory allocation (`malloc()`) and pseudorandom number generation (`rand()`), although a conservative compiler may always treat them as external choices. Internal choice also exists in some programming languages (such as C) as *unspecified behavior*, which we also call *compiler choice* because the compiler may refine the choice statically (during compilation) or dynamically (at runtime). A common example of compiler choice is the order in which expressions may be evaluated between sequence points in C.

The key strength of bisimulation-style relations are that they generally preserve nondeterminism and thus enable the modular reasoning described above. However,

we have discovered that none of the relations used by existing verified compilers hold for the parallelization of programs that contain internal choice (even for very simple programs). We know of two solutions.

Because the only form of internal choice used by current verified compilers is compiler choice, the first solution is to refine all internal choice before parallelization. Then it will preserve a weak bisimulation. A second solution is to find an alternate program equivalence that allows parallelization of programs with internal choice.

While a compiler *may* refine compiler choice, the first solution *forces* it to do so. It is plausible that a compiler would prefer to refine compiler choice after parallelization, or even preserve it in the compiled program. For example, loop peeling is an optimization where one or more iterations of a loop are executed outside (before or after) the loop. For each peeled iteration and the loop itself, compiler choice may be refined differently. Whether loop peeling is applied and how compiler choice is refined depends on the nature of the loop, but the loop parallelization that we explore in this thesis might dramatically change the nature of the loop (especially DSWP) such that a compiler can optimally apply loop peeling and refinement only after parallelization.

Furthermore, it is not clear whether all forms of internal choice should be compiler choice; some might instead be *program choice*, which is internal choice that cannot be refined by the compiler. A potential source of program choice is `rand()` because, intuitively, neither the observer nor compiler should be able to influence the stochastic nondeterminism found within a program. As compiler choice, the compiler would be allowed to change the probability distribution of `rand()` by hardcoding the result to 0. Although the compiler may conservatively treat `rand()` as an external choice, doing so disallows optimizations that would commute `rand()` – e.g. transforming `x := rand(); y := rand()` into `y := rand(); x := rand()`.

For example, existing research shows that allowing operations such as `rand()` and `malloc()` to commute, i.e. treating these calls as internal choice, is instrumental

to getting good parallel speedup [33]. Testing eight benchmark programs with their parallelizing compiler, they achieved a geomean speedup of 1.5x on eight cores, but were able to increase this to 5.7x by allowing such operations to commute. Without this flexibility, the compiler was unable to parallelize half of the test programs.

In this thesis, we treat `rand()` as a program choice, but note that we do so only for demonstration purposes; our definitions of program equivalence also allow the probability distribution to be changed¹.

Another potential source of program choice is the thread scheduler, particularly if we are to perform parallelizing optimizations. If the thread schedule were an external choice, then the observer could trivially distinguish between a program that uses one thread and a (parallelized) program that uses two threads by choosing to schedule just one of the threads; this would rule out parallelizing optimizations.

However, it is unclear whether the thread schedule should be a compiler or program choice. The compiler cannot arbitrarily refine the thread schedule (i.e. remove potential thread interleavings), else nearly all programs that use synchronization could be compiled into programs that deadlock. But we are not aware of a (mainstream) language specification that explicitly states whether or not a compiler may otherwise remove behaviors by refining the schedule. If it can, then the compiler may transform `x := 1 || x := 2` into `x := 2` (assuming that the assignments are atomic).



Regardless of whether the thread schedule is a program or compiler choice, there is an advantage in finding a program equivalence that allows internal choice to be preserved. If a compiler chooses the first solution in order to preserve a weak bisimulation and also accepts multithreaded source programs, then it faces the burden of (effec-

¹We propose two ways to resolve this issue in a way that enables optimizations involving `rand()`. First, we could allow the observer to see each invocation of `rand()`, but not its result – this would allow commuting consecutive calls to `rand()` while preventing them from being combined (which can change the probability distribution if we are not careful). Or we could use a variation of probabilistic bisimulation that accounts for stochastic choices and allows them to be combined and commuted while preserving their distribution [19, 39].

tively) linearizing source programs – and proving that no deadlocks are introduced by the linearization – before it can apply parallelization.

Thus we have developed *eventual similarity* and identify *contrasimilarity* [13] as alternate relations, based on bisimilarity, that hold for parallelizing programs with internal choice and have many of the nice properties that the well-known bisimulations have: a coinductive proof method and preservation of nondeterminism. This allows verified parallelizing compilers to support program choice in source programs, to target multithreaded source programs without having to linearize them, and to wait until after the parallelization phase to refine compiler choice. Furthermore, we prove that these alternate relations are equivalent to bisimulation when the programs do not contain internal choice. Thus the results in this thesis are directly applicable to compilers that do choose to refine all internal choice first.

1.4 Mechanized proofs

We base all results in this thesis on proofs that are mechanically checked using the Coq Proof Assistant. Chapter 3 is fully checked. The results in chapter 4 and the soundness of the transformation rules presented in chapters 2 and 4 are *mostly* formalized and proven, but some of the results are been split between two Coq proof developments. Section 2.6 identifies which rewrite rules have been proved sound and in which context. Throughout this thesis, we either conclude each proof with  or (if we have elided the proof) annotate each lemma with  if there is a proof in Coq.

1.5 Thesis structure

Our rewriting framework is presented in chapter 2, where we motivate our parallelization and loop folding transformations by using them to implement instances of DOALL, DSWP, and DOACROSS optimizations. The remainder of our thesis is de-

voted to proving the soundness of our framework. In chapter 3, we work with a model language based on the Calculus of Communicating Systems to study parallelization in order to find a strong notion of bisimulation that holds (eventual similarity and contrasimilarity). The results from this chapter are based on our conference paper, *Certifiably Sound Parallelizing Transformations* [3]. In chapter 4, we apply what we learned from the previous chapter in order to formalize the imperative language from chapter 2, and then prove the soundness of loop folding, parallelization, and several other rewrite rules in the framework. Proving the soundness of the rewrite rules shows that the optimizations based on our framework are correct by construction.

Chapter 2

A Transformation Framework for Loop Parallelization

We demonstrate two of our main contributions:

- a loop-folding transformation, which folds a “combining transformation” over potentially nonterminating loops;
- a parallelizing transformation, capable of parallelizing iterations and combining them together;

and introduce our high-level proof strategy by using a framework of rewrite rules to apply loop parallelization to a few example programs.

Using a framework of rewrite rules on a high-level structured language, as opposed to proving optimizations directly and/or targeting a backend intermediate language of the compiler (such as a control flow graph), allows us to break the steps of our proofs into smaller components that are easier to understand. Other advantages of this approach are that it is compositional, so that new optimizations can be created simply by finding new ways of combining the rules, and because it is easier to integrate with the front-end of existing verified compilers (which we discuss as a topic of future work in Section 5.2.2). Although we limit ourselves to using rewrite rules for each

$t ::=$	$x := e$	assign the value of e to x
	$x := \text{rand } V$	assign a random value from V to x
	skip	do nothing
	if e then t_1 else t_2	branch; if e then execute t_1 , otherwise t_2
	while e do t	loop; iterate t while e is true
	$\{P\}$	assertion; is stuck iff P does not hold on the state
	$t_1; t_2$	instruction sequencing
	$[P_1] t_1 \parallel [P_2] t_2$	run t_1 in parallel with t_2
	send $a e$	enqueue the value of e in channel a
	$x := \text{recv } a$	dequeue a value from channel a into variable x

Figure 2.1: Program syntax

transformation in this chapter, this does not preclude combining this framework with other techniques, such as a Hoare logic analysis or translation validation.

The target language for loop parallelization in this thesis is listed in Fig. 2.1; it is a small imperative language that has most of the basic features one would expect, such as local variables, assignment, if-statements, loops, and concurrency. We also add asynchronous thread communication through shared queues, or *channels*, which we use to synchronize iterations and to pass values between threads in order to satisfy dependencies as we apply DOACROSS and DSWP. However, we forgo some common features that are unnecessary to demonstrate loop parallelization: functions, a heap, and allocation/scoping of local variables.

Instruction **send** $a e$ sends the value of expression e into channel a , and instruction $x := \text{recv } a$ blocks until it can receive the next available value from channel a , and then assigns it to variable x . The assertion and fork-join parallelism statements will be explained in Section 2.1.

When running a program, the state consists of a set of channels (a mapping from channel names to queues of values) and the set of threads being executed. Each thread has a local state consisting of an environment (a partial mapping from variable names to values) and current instruction. We consider two programs equivalent if, for any initial state, there is an observational congruence between them. That is,

two programs are equivalent if an adversary (the observer) cannot tell them apart, where the adversary may observe the final state of each program (and thus whether it terminated) and may interact with each program by communicating over any channel that is not private.

For this chapter, we use a congruence between program terms, \equiv , to state that two programs are equivalent for any initial state. An example rewrite rule is T-SkipL, which replaces any occurrence of `skip;t` in a program with `t` (for any `t`):

$$\frac{}{\text{skip}; t \equiv t} \text{T-SkipL} \clubsuit$$

Although presented in an axiomatic style, we give our rewrite rules a semantic definition in chapter 4. In other words, all rewrite rules presented in this chapter are just theorems, and we annotate each with \clubsuit when we have proved it in Coq. A summary of the core rewrite rules in our framework and whether they have been proved in Coq is given in Section 2.6.

2.1 Assertions

Assertions ensure that the current state satisfies a predicate. If the predicate is satisfied by the current state, then the assertion does nothing and steps to the next instruction. Otherwise, the assertion is stuck. If an assertion is satisfied by some state, then it must be satisfied by any extension of the state with more channels or variables. As we apply parallelization to the example programs in this chapter, assertions play a key role in justifying each step.

We write the predicates using separation logic, which is a convenient way to state that blocks of code are independent, in order to support parallelization [35]. Because

our language is limited to channels and variables (and has no heap), we use a separation logic that defines both variables and channels as resources [6]. However, our use of separation logic is primarily limited to stating the preconditions of transformations. We will also avoid formal proofs to discharge the entailments that result from transformation side-conditions in this thesis. (Thus we can use a variables-as-resources style separation logic without having to deal with the additional complexity that it often adds to proving entailment.)

A selection of predicate forms we will use in this chapter are:

$$\begin{array}{l}
 P ::= \text{ true } \mid \text{ false } \mid \neg P \mid e_1 = e_2 \mid e \mid P_1 \wedge P_2 \mid P_1 \star P_2 \mid \text{ owns } x \\
 \mid \text{ owns } a \mid \text{ cansend } a \mid \text{ canrecv } a \mid a = [e_1, \dots, e_n] \mid \dots
 \end{array}$$

Predicate $P_1 \star P_2$ holds when the current state can be broken into two substates, having disjoint channels and variables, such that P_1 holds on one substate and P_2 holds on the other. To assert ownership of program variable x so that no other thread may read from or write to it, we use $\text{owns } x$; for ownership of both variables x and y , we assert $\text{owns } x \star \text{owns } y$, which we abbreviate as $\text{owns } x, y$; etc. Because substates must be disjoint, writing $\text{owns } x \star \text{owns } x$ is equivalent to false . Given expressions e_1 and e_2 , predicate $e_1 = e_2$ holds when they both evaluate to the same value. Expressions can also be directly lifted to predicates; predicate e is equivalent to $e = \text{true}$. We will explain forms $\text{owns } a$, $\text{cansend } a$, $\text{canrecv } a$, and $a = [e_1, \dots, e_n]$ later in Section 2.3; the remaining predicate forms are standard. When one predicate implies another for all states, we write $P_1 \vdash P_2$; we write $P_1 \dashv\vdash P_2$ when they are equivalent.

Many transformations use assertions as guards on the program state. For example,

$$\frac{}{\{x = e\}; x := e \equiv \{x = e\}} \text{T-AssertAssign}$$

is a transformation that replaces an assertion stating that variable x is equal to expression e , followed by an assignment from e to x , with just the assertion; i.e., it eliminates a redundant assignment. We can also make inferences from assignments:

$$\frac{}{x := e \equiv x := e; \{x = e\}} \text{T-AssignAssert}$$

can be used to satisfy the guards of subsequent transformations that require x and e to be equal.

We rule out the possibility of inserting false assertions by allowing termination to be observed (i.e. termination must be preserved by \equiv); an assertion that is not always true may become stuck, and thus will not preserve all termination properties of the program. This also means that we may assert `false` wherever there is dead code (e.g., after an infinite loop or crash) because the assertion will never run. If we can assert `false`, then we can insert or remove any subsequent program code:

$$\frac{}{\{\text{false}\}; t \equiv \{\text{false}\}} \text{T-AssertFalse} \clubsuit$$

We can always prove “true”, thus:

$$\frac{}{\{\text{true}\} \equiv \text{skip}} \text{T-AssertTrue} \clubsuit$$

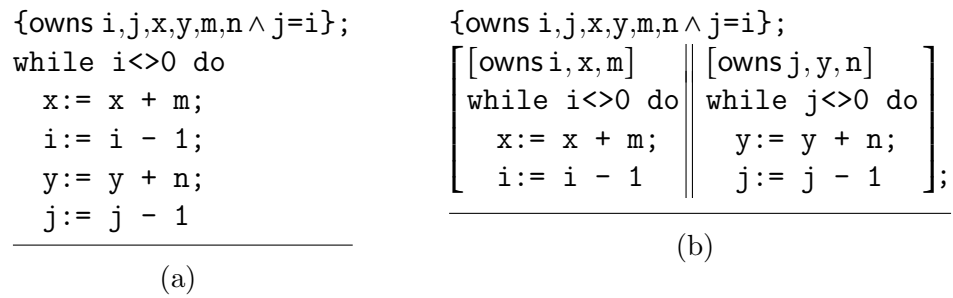


Figure 2.2: Demonstration of loop parallelization.

Manipulation of assertions can be done by rewrite rules alone, but we can also use analyses based on Hoare logic via the following rule:

$$\frac{\vdash\{P\} t \{Q\}}{\{P\}; t \equiv \{P\}; t; \{Q\}} \text{T-Hoare}$$

The converse of T-Hoare does not hold because safety is not implied by assertions.

Guarded threads. Threads have form $[P_1] t_1 \parallel [P_2] t_2$ and compose two programs, t_1 and t_2 , in parallel (i.e. to be run concurrently). However, t_1 and t_2 cannot claim ownership to the same resources, so the threads are guarded by predicates P_1 and P_2 . In order to fork into two threads, $P_1 \star P_2$ must hold; after each thread is created, P_1 will hold for the initial state of t_1 , and P_2 will hold for the initial state of t_2 . If the guards cannot be satisfied, then the program will get stuck.

2.2 Example: DOALL parallelization

Figure 2.2 presents a simple example of DOALL parallelization. A defining feature of a DOALL loop is that each iteration is independent of the other iterations; in Fig. 2.2a, this is approximated by having the loop contain two independent tasks that can be run in parallel: one that modifies x and i , and one that modifies y and j .

```

{owns i,j,x,y,m,n ∧ j = i};
while i<>0 do
  {owns i,j,x,y,m,n ∧ j=i};
  if i<>0 then
    x:= x + m;
    i:= i - 1;
  if j<>0 then
    y:= y + n;
    j:= j - 1

```

Figure 2.3: Separating the loop body into two tasks.

Each program is guarded by an initial assertion that defines a list of distinct variables that the program will use and that ensures i and j are equal.

Despite its simplicity, Fig. 2.2 is not trivial to parallelize because it will not terminate if i is initially less than 0. Thus we will be unable to prove correctness by inducting on the number of iterations that execute.

We first split the loop body into two tasks, where the first task increments x and decrements i , and the second task increments y and decrements j . Then we parallelize the two tasks in the body of the loop and finally apply parallelization to the whole loop.

To split the tasks, we first infer a loop invariant, $\text{owns } i,j,x,y,m,n \wedge j = i$, by the following rules:

$$\frac{\{e \wedge P\}; t \equiv \{e \wedge P\}; t; \{P\}}{\{P\}; \text{while } e \text{ do } t \equiv \text{while } e \text{ do } (\{e \wedge P\}; t); \{-e \wedge P\}} \text{T-WhileAssertI} \quad \clubsuit$$

$$\frac{}{\{e\}; \text{if } e \text{ then } t_1 \text{ else } t_2 \equiv \{e\}; t_1} \text{T-TrueIf} \quad \clubsuit$$

We use T-WhileAssertI to move the assertion into the loop; the side condition that the assertion must continue to hold at the end of the iteration is easy to prove by

T-Hoare and standard Hoare logic. The tasks are finally split into two `if` statements, respectively guarded by `i<>0` and `j<>0`, using T-TrueIf.

2.2.1 Parallelize the loop body

Parallelizing the loop body is performed by just one rule:

$$\begin{array}{c}
 P_1 : t_1 \Downarrow \\
 \text{freechans } t_1 \cup \text{freechans } t_2 = \emptyset \\
 \text{writes } t_1 \cap \text{freevars } t_2 = \text{freevars } t_1 \cap \text{writes } t_2 = \emptyset \\
 \frac{\begin{array}{l} \{P_1 \star P_2\}; t_1 \equiv [P_1] t_1 \parallel [P_2] \text{skip} \\ \{P_1 \star P_2\}; t_2 \equiv [P_2] t_2 \parallel [P_1] \text{skip} \end{array}}{\{P_1 \star P_2\}; t_1; t_2 \equiv [P_1] t_1 \parallel [P_2] t_2} \text{T-SeqParA} \clubsuit
 \end{array}$$

The transformation guard, $P_1 \star P_2$, ensures that the state (e.g. variables) can be divided into two disjoint substates satisfying P_1 and P_2 , respectively. Once parallelized, P_1 and P_2 guard threads t_1 and t_2 , respectively, and it is no longer necessary to explicitly assert $P_1 \star P_2$. We use P_1 and P_2 to list what t_1 and t_2 *can* do, and the free variables/channels of t_1 and t_2 to state what they *cannot* do.¹ We explain the judgments in order:

- $P_1 : t_1 \Downarrow$ – if P_1 holds on an initial state, then any future state that t_1 may reach will be able to terminate.
- $\text{freechans } t_1 \cup \text{freechans } t_2 = \emptyset$ – The set of channels that t_1 and t_2 use are disjoint. (For now, we assume that the threads cannot communicate/synchronize with each other.)
- $\text{writes } t_1 \cap \text{freevars } t_2 = \text{freevars } t_1 \cap \text{writes } t_2 = \emptyset$ – t_1 cannot write to variables that are accessed by t_2 , and t_2 cannot write to variables that are accessed by t_1 .

¹We could instead place such restrictions on variable and channel usage via Hoare triples. (Assertions cannot state such restrictions because \equiv does not imply safety.)

```

{owns i,j,x,y,m,n ∧ j=i};
while i<>0 do
  {owns i,j,x,y,m,n ∧ j=i};
  [ [owns i, x, m] || [owns j, y, n]
    if i<>0 then   if j<>0 then
      x:= x + m;   y:= y + n;
      i:= i - 1    j:= j - 1 ] ]

```

Figure 2.4: Parallelized loop body

- $\{P_1 \star P_2\}; t_1 \equiv [P_1] t_1 \parallel [P_2] \text{skip}$ – The behavior of t_1 does not change between whether we give it access to resources that are associated with P_2 or take them away (by putting them in another thread). In other words, t_1 cannot attempt to use any variables or channels that are specified by P_2 . (This is essentially a “frame rule” for transformations, and lemma 4.32 shows how to prove this transformation using a Hoare triple).
- $\{P_1 \star P_2\}; t_2 \equiv [P_2] t_2 \parallel [P_1] \text{skip}$ – likewise, t_2 cannot attempt to use any variables or channels that are specified by P_1 .

To parallelize the loop body, we apply T-SeqParA to Fig. 2.3 by choosing thread-guards $P_1 = \text{owns } i, x, m$ and $P_2 = \text{owns } j, y, n$. This divides the variables between the threads. The conditions for T-SeqParA are met because the threads use disjoint variables, will always terminate, and do not use any channel communication. We do not parallelize the loop invariant assertion because neither thread-guard may, by itself, assert $j=i$ due to ownership of the referenced variables being split between them. Figure 2.4 is the result of parallelizing the loop body.

2.2.2 Folding parallelism into the whole loop

We will now parallelize the whole loop, resulting in Fig. 2.2b. Let t_1 and t_2 represent the left and right threads, respectively, of the parallelized loop body in Fig. 2.4. Conceptually, we first unfold the loop by peeling every iteration:

`while i <> 0 do (t1 || t2) ≡ (t1 || t2); (t1 || t2); (t1 || t2); (t1 || t2); (t1 || t2); ...`

And then we combine the iterations together:

$$\begin{aligned}
&\equiv (t_1; t_1 \parallel t_2; t_2); (t_1 \parallel t_2); (t_1 \parallel t_2); (t_1 \parallel t_2); \dots \\
&\equiv (t_1; t_1; t_1 \parallel t_2; t_2; t_2); (t_1 \parallel t_2); (t_1 \parallel t_2); \dots \\
&\quad \vdots \\
&\equiv (t_1; t_1; \dots) \parallel (t_2; t_2; \dots) \\
&\equiv \text{while } i \neq 0 \text{ do } t_1 \parallel \text{while } j \neq 0 \text{ do } t_2.
\end{aligned}$$

The result is akin to folding a list. We explain, in four parts, how to apply this idea toward transforming Fig. 2.4 into Fig. 2.2b. The first is a loop transformation that, by itself, does not know about parallelization (Section 2.2.3). Instead, it abstracts the iteration-combining transformation into a *loop schema*. The second part introduces a language extension to describe the intermediate number of iterations shown in the process above; an explicitly-bounded while loop (Section 2.2.4). The third introduces a parallelizing transformation that we will use to combine parallel iterations together (Section 2.2.5). And finally, we piece these together to complete the DOALL parallelization in Section 2.2.6.

2.2.3 Loop schema & folding

A loop schema is defined as (f, e, P) , where $f[z]$ is the program that results from combining any z iterations, e is the loop condition, P is the loop invariant, and the following properties hold:

- if P holds, it will continue to hold after an iteration;
- if the loop condition is false, then the loop is effectively terminated;
- if the loop terminates, then the loop condition will be false; and

- two combined iterations, composed in sequence, may be combined together.

Because $f[1]$ corresponds to the loop body, the original loop is equivalent to `while e do $f[1]$` . The result of the transformation is the combination of f for an unbounded number of iterations: $f[\infty]$. Thus we have derived the following rule, where n is positive, u is either finite or ∞ , and $n \sqcap u$ is roughly equivalent to $n + u$:

$$\frac{\begin{array}{l} \{P\}; f[1] \equiv \{P\}; f[1]; \{P\} \\ \forall u. \{-e \wedge P\}; f[u] \equiv \{-e \wedge P\} \\ \{P\}; f[\infty] \equiv \{P\}; f[\infty]; \{-e \wedge P\} \\ \forall n \geq 1, u. \{P\}; f[n \sqcap 0]; f[u] \equiv \{P\}; f[n \sqcap u] \end{array}}{\{P\}; \text{while } e \text{ do } f[1] \equiv \{P\}; f[\infty]} \quad \text{T-FoldLoop} \quad \clubsuit$$

To prove the rule sound, we do not actually peel every iteration at once because there will be an unknown (and possibly infinite) number of them. Instead, we peel the loops only as needed while the source and target of the transformation execute; we give a proof in Section 4.4.

2.2.4 Bounded loops

To represent a sequence of up to z iterations, we extend `while` loops to be parameterized by a bound on the maximum number of iterations that it may execute:

`while e max z do t .`

If a loop reaches a finite bound, then it is forced to terminate even if the loop condition is still true. A normal [unbounded] loop, `while e do t` , is equivalent to

`while e max ∞ do t`. A loop with an iteration bound of 1 is equivalent to an `if`-statement and a loop with a bound of 0 is equivalent to `skip`:

$$\frac{}{\text{while } e \text{ max } 1 \text{ do } t \equiv \text{if } e \text{ then } t} \text{T-WhileIf}_{\blacktriangleright}$$

$$\frac{}{\text{while } e \text{ max } 0 \text{ do } t \equiv \text{skip}} \text{T-WhileZero}_{\blacktriangleright}$$

There is also a third kind of iteration bound, which we call an *iteration barrier*, that has form $n \square u$. After n iterations, `while e max $n \square u$ do t` will “pause” (i.e. become stuck) rather than allow the remaining u iterations to run. (But if e becomes false, it will terminate.) We write t_{\blacktriangleright} to resume all loops in program t ; in other words, to replace all occurrences of $n \square u$ with $n + u$. Finally, we restrict \equiv from making any assumptions about whether a loop will pause at an iteration barrier by requiring every equivalence to continue to hold after resuming all loops:

$$\frac{t_1 \equiv t_2}{t_{1\blacktriangleright} \equiv t_{2\blacktriangleright}} \text{T-Resume}_{\blacktriangleright}$$

To use T-FoldLoop, the user must prove $f[n \square 0]; f[u] \equiv f[n \square u]$ instead of $f[n]; f[u] \equiv f[n + u]$. (By T-Resume, the former equivalence implies the latter.) In other words, to prove the combining transformation, the user must provide an equivalence that does not assume $f[n]$ can terminate after the (finite) n iterations.

There are two reasons behind our use of iteration barriers. The first is the existence of degenerate loop schemas that satisfy $f[n]; f[u] \equiv f[n + u]$, but do not hold for `while e do $f[1] \equiv f[\infty]$` . The second is that iteration barriers provide an extensional

way to dissect the looping behavior of f (so that it is treated as a black box). We explain these reasons in more detail in Section 4.4.

Iteration barriers also provide a convenient way to write programs that are parameterized by holes (f) to be later filled with iteration bounds ($f[z]$). We define $f[z]$ as the substitution of every occurrence of $n \square u$ in f with $n+z+u$. Thus resuming loops, $t \blacktriangleright$, is equal to $t[0]$. (Both t and f denote the same kinds of programs, but we use f to indicate that we are using it as a loop schema.)

A very simple loop schema can be defined by **while-max** for any t , e , and P :

Proposition 2.1. *If $\{P\};t \equiv \{P\};t;\{P\}$, then $(\text{while } e \text{ max } \square \text{ do } t, e, P)$ is a loop schema (see lemma 4.25),*

where we abbreviate $0 \square 0$ as \square . The termination properties – that e is a loop condition such that $\neg e$ implies and is implied by termination – trivially hold by:

$$\frac{}{\{-e\}; \text{while } e \text{ max } z \text{ do } t \equiv \{-e\}} \text{T-FalseWhile} \blacktriangleright$$

$$\frac{}{\text{while } e \text{ do } t \equiv \text{while } e \text{ do } t; \{-e\}} \text{T-WhileFalse} \blacktriangleright$$

To prove the last property, combining iterations for a **while** loop, we first define an addition operation, $z \diamond u$, which adds finite or infinite iterations to the end of bound z . If z is finite or ∞ , then \diamond is equivalent to $+$; otherwise, $(n' \square u') \diamond u = n' \square (u' + u)$ for $z = n' \square u'$. Combining iterations for a **while** loop is analogous to loop splitting:

$$\frac{}{\text{while } e \text{ max } (z \diamond u) \text{ do } t \equiv (\text{while } e \text{ max } z \text{ do } t); (\text{while } e \text{ max } u \text{ do } t)} \text{T-Split}_{\diamond} \blacktriangleright$$

The programmer is restricted to writing programs with unbounded loops. Finite bounds and iteration barriers are only introduced by T-FoldLoop when the user must prove the combining transformation. Through transformation rules such as T-Split $_{\diamond}$, the user can construct combining transformations to prove complicated loop schemas (such as parallel loops) without reasoning directly about whether the iteration bounds are finite, infinite, or barriers.

2.2.5 Combining parallelism

We need a combining transformation for parallelized iterations, which we get by generalizing T-SeqParA. If the following conditions hold for any t_1, t_2, t_3 , and t_4 :

- $[P_1] t_1 \updownarrow [P_2] t_2 - t_1$ and t_2 *coterminate*: if we run $[P_1] t_1 \parallel [P_2] t_2$ and t_1 terminates, then t_2 is able to terminate without being observed. If, instead, t_2 terminates first, then t_1 can terminate without being observed. Cotermination is also satisfied when neither program terminates. When we (later) present a parallelizing transformation that allows threads to communicate, they may coordinate termination to satisfy cotermination as well.
- *precise* P_3 and *precise* $P_4 - P_3$ and P_4 unambiguously divide resources (channels) between themselves. In other words, $P_3 \star (A \wedge B) \dashv\vdash (P_3 \star A) \wedge (P_3 \star B)$ for all A and B , and likewise for P_4 [26].
- $\text{freechans}(t_1; t_3) \cup \text{freechans}(t_2; t_4) = \emptyset$ – again, we assume for now that no communication channels are used.
- $\text{writes}(t_1; t_3) \cap \text{freevars}(t_2; t_4) = \emptyset$ – neither t_1 nor t_3 can write to a variable that is accessed by t_2 or t_4 .
- $\text{freevars}(t_1; t_3) \cap \text{writes}(t_2; t_4) = \emptyset$ – neither t_2 nor t_4 can write to a variable that is accessed by t_1 or t_3 .

<pre>{owns i,j,x,y,m,n ∧ j=i}; while i<>0 do f[1];</pre>	<pre>{owns i,j,x,y,m,n ∧ j=i}; f[∞]</pre>
(a)	(b)

Figure 2.5: Parallelizing the loop

- $\{P_1\}; t_1 \equiv \{P_1\}; t_1; \{P_3\}$ – assuming P_1 holds in the initial state, t_1 is equivalent to $t_1; \{P_3\}$; this means that P_3 , which is the guard for t_3 , will always hold after t_1 terminates.
- $\{P_2\}; t_2 \equiv \{P_2\}; t_2; \{P_4\}$ – the termination of t_2 ensures that the guard for t_4 (i.e., P_4) will hold.

Then we may combine the parallel programs:

$$\frac{
\begin{array}{l}
[P_1] t_1 \updownarrow [P_2] t_2 \quad \text{precise } P_3 \quad \text{precise } P_4 \\
\text{freechans}(t_1; t_3) \cup \text{freechans}(t_2; t_4) = \emptyset \\
\text{writes}(t_1; t_3) \cap \text{freevars}(t_2; t_4) = \emptyset \\
\text{freevars}(t_1; t_3) \cap \text{writes}(t_2; t_4) = \emptyset \\
\{P_1\}; t_1 \equiv \{P_1\}; t_1; \{P_3\} \\
\{P_2\}; t_2 \equiv \{P_2\}; t_2; \{P_4\}
\end{array}
}{
([P_1] t_1 \parallel [P_2] t_2); ([P_3] t_3 \parallel [P_4] t_4) \equiv [P_1] (t_1; t_3) \parallel [P_2] (t_2; t_4)
} \text{T-SeqParB}$$

In effect, T-SeqParB parallelizes t_1 with t_4 and t_2 with t_3 in one step. It implies T-SeqParA by choosing either $t_2 = t_3 = \text{skip}$ for $t_1; t_4 \equiv t_1 \parallel t_4$ or $t_1 = t_4 = \text{skip}$ for $t_2; t_3 \equiv t_2 \parallel t_3$, depending on whether t_1 or t_3 communicates over a public channel.

2.2.6 Loop parallelization

We transform Fig. 2.4 into Fig. 2.2 using T-FoldLoop and the loop schema

$$(f, i < > 0, \text{owns } i, j, x, y, m, n \wedge j = i),$$

where

$$f[z] = \left[\begin{array}{l} [\text{owns } i, x, m] \\ \text{while } i <> 0 \text{ max } z \text{ do} \\ \quad x := x + m; \\ \quad i := i - 1 \end{array} \parallel \left[\begin{array}{l} [\text{owns } j, y, n] \\ \text{while } j <> 0 \text{ max } z \text{ do} \\ \quad y := y + n; \\ \quad j := j - 1 \end{array} \right] \right].$$

Figure 2.4 is equivalent to Fig. 2.5a by T-WhileIf. By T-FoldLoop, Fig. 2.5a is equivalent to Fig. 2.5b, which is equivalent to the fully parallelized loop, Fig. 2.2b. Then we show that $(f, i <> 0, \dots)$ has the properties of a loop schema. Proving that $i <> 0$ implies and is implied by termination of $f[1]$ is straightforward using T-FalseWhile and T-WhileFalse.

The loop invariant property cannot be proved by propagating the assertion forward through the parallelized body because ownership of i and j is split. However, it suffices to prove the loop invariant for the (sequential) loop body of Fig. 2.2a because we have already proved it equivalent to $f[1]$.

The combining transformation is the most interesting property because it uses T-SeqParB to combine the iterations together. By T-SeqParB and then T-Split $_{\diamond}$:

$$\begin{aligned} & \{ \text{owns } i, j, x, y, m, n \wedge j=i \}; \\ & f[n \square 0]; f[u] \\ \\ = & \{ \text{owns } i, j, x, y, m, n \wedge j=i \}; && \text{by def. of } f \\ & \left[\begin{array}{l} [\text{owns } i, x, m] \\ \text{while } i <> 0 \text{ max } n \square 0 \text{ do} \\ \quad x := x + m; \\ \quad i := i - 1 \end{array} \parallel \left[\begin{array}{l} [\text{owns } j, y, n] \\ \text{while } j <> 0 \text{ max } n \square 0 \text{ do} \\ \quad y := y + n; \\ \quad j := j - 1; \end{array} \right] \right]; \\ & \left[\begin{array}{l} [\text{owns } i, x, m] \\ \text{while } i <> 0 \text{ max } u \text{ do} \\ \quad x := x + m; \\ \quad i := i - 1 \end{array} \parallel \left[\begin{array}{l} [\text{owns } j, y, n] \\ \text{while } j <> 0 \text{ max } u \text{ do} \\ \quad y := y + n; \\ \quad j := j - 1 \end{array} \right] \right] \end{aligned}$$

$$\begin{aligned}
&\equiv \{ \text{owns } i, j, x, y, m, n \wedge j=i \}; && \text{by T-SeqParB} \\
&\left[\begin{array}{l} [\text{owns } i, x, m] \\ \text{while } i <> 0 \text{ max } n \square 0 \text{ do} \\ \quad x := x + m; \\ \quad i := i - 1; \\ \text{while } i <> 0 \text{ max } u \text{ do} \\ \quad x := x + m; \\ \quad i := i - 1 \end{array} \parallel \left[\begin{array}{l} [\text{owns } j, y, n] \\ \text{while } j <> 0 \text{ max } n \square 0 \text{ do} \\ \quad y := y + n; \\ \quad j := j - 1; \\ \text{while } j <> 0 \text{ max } u \text{ do} \\ \quad y := y + n; \\ \quad j := j - 1 \end{array} \right] \right. \\
&\equiv \{ \text{owns } i, j, x, y, m, n \wedge j=i \}; && \text{by T-Split}_{\diamond} \\
&\left[\begin{array}{l} [\text{owns } i, x, m] \\ \text{while } i <> 0 \text{ max } n \square u \text{ do} \\ \quad x := x + m; \\ \quad i := i - 1 \end{array} \parallel \left[\begin{array}{l} [\text{owns } j, y, n] \\ \text{while } j <> 0 \text{ max } n \square u \text{ do} \\ \quad y := y + n; \\ \quad j := j - 1 \end{array} \right] \right. \\
&= \{ \text{owns } i, j, x, y, m, n \wedge j=i \}; && \text{by def. of } f \\
&\quad f[n \square u].
\end{aligned}$$

The side conditions of T-SeqParB are satisfied:

- The first pair of threads, bounded by $n \square 0$, coterminate. In other words, if either thread terminates, then $i=0$ and $j=0$, so both will terminate.
- Neither thread uses channels.
- The threads use disjoint local variables.
- The first left thread preserves its guard at termination, and thus satisfies the (same) guard of the next left thread.
- The first right thread preserves its guard at termination, and thus satisfies the (same) guard of the next right thread.

2.3 Example: Decoupled software pipelining

Figure 2.6 is a program that approximates π via Monte Carlo simulation. The first line is an assertion that defines the variables, channels, and their initial states that we will use to justify transforming the program. In short, the assertion holds for any

```

{owns i,j,x,y,z,m * a=[] * b=[] ^ i=N ^ j=i>0};
while i>0 do
  i:= i - 1;
  x:= rand[0..1];
  y:= rand[0..1];
  z:= x*x + y*y;
  if z<1 then
    m:= m + 1;
send pi (4*m/N);
j:= i>0

```

Figure 2.6: Approximating π

state where i and j are initialized to N and $i>0$, respectively, and which can be divided into three substates that define disjoint variables and channels. The first substate has exclusive (read & write) ownership of the variables, the second has empty channel a , and the third has empty channel b . By stating that the buffer of a channel is empty, the second and third components of the assertion imply that the program has full ownership of a and b ; i.e. the two channels are private. (An assertion may not refer to the contents of a public channel because it may change at any time.)

The program consists of a loop that runs for N iterations, where each iteration generates a point with x and y coordinates distributed uniformly at random between 0 and 1. This point is then checked to determine if it is within a circle of unit radius. If so, the program increments a counter (m) that tracks the number of generated points that fell within the circle. When the loop completes, the program estimates π by using the fraction of points generated inside the circle as an approximation for the relative area of the circular sector with respect to the enclosing square of edge length one. The estimate is made observable by sending the value to channel pi .

Finally, we set j to $i>0$ at the end of the program because we will use j for a loop condition, equivalent to $i>0$, in the parallelized program. Without this assignment in the source program, the parallelized program would have to restore the original value of j in order to use it for its own purposes because, in our program equivalence

model, an observer may inspect the value of all variables (and the contents of all channels) once the program terminates. Preemptively including `j` (and channels `a` and `b`) in the source program like this is necessary only because the language does not support scoping. (With such support, we would use transformations to insert temporary variables and channels rather than including them in the source program.)

Notably, the process of correctly transforming a program does not necessarily require the program itself to be safe or correct. For example, the initial assertion in Fig. 2.6 is just strong enough to support our transformations, but the program may get stuck on the last line (i.e. crash) because the assertion does not define channel `pi`. Another potential bug in the program is that `m` may not be initially equal to 0, causing the value sent over `pi` to be meaningless. Although “bad” programs may not be desirable, they will be faithfully preserved by the transformations.

2.3.1 The DSWP transformation

We apply a DSWP-style pipelined parallelizing optimization by breaking the program into two *tasks*. The first task generates a random point and computes the distance from the origin, and the second task increments `m` whenever it determines membership in the circle. In the resulting program, the tasks run in parallel loops that correspond to two stages of a pipelined operation. The first task/stage sends the value of the loop-condition and distance from the origin to the second task/stage through a channel.

Figure 2.7 is the result of this parallelization, where each loop is preceded by a guard to ensure that the variables and channels are properly divided between the two threads. The left thread requires write access to variables `i`, `x`, and `y`, and the ability to send values over channels `a` and `b`. The right thread owns variables `j`, `x`, and `m`, and has the ability to receive values from channels `a` and `b`. Finally, we have removed the assignment of `i>0` to `j` because it is now being set within the right loop to convey the loop condition.

```

{owns i,j,x,y,z,m * a=[] * b=[] ^ i=N ^ j=i>0};
[ [owns i,x,y * cansend a,b] || [owns j,z,m * canrecv a,b]
while i>0 do                               while j do
  i:= i - 1;                               j:= recv b;
  x:= rand[0..1];                          z:= recv a;
  y:= rand[0..1];                          if z<1 then
  send a (x*x+y*y);                        m:= m + 1
  send b (i>0)                               ];
send pi (4*m/N)

```

Figure 2.7: DSWP parallelization of Fig. 2.6

We prove the correctness of this transformation in three phases: phase 1 (Section 2.3.2) decouples the dependencies of the two tasks, phase 2 (Section 2.3.3) parallelizes the tasks in the body of the loop, and phase 3 (Section 2.3.4) folds the parallelism into the entire loop.

2.3.2 Phase 1: Decoupling dependencies

Our first step is to decouple the tasks by expressing data and control-flow dependencies explicitly using channels. A *data dependency* exists between two tasks if a task relies on a variable that the other has modified. A *control-flow dependency* exists when the execution of one task depends on the control-flow choices of the other task; these are generated by `if`-, `while`-, and `recv`- statements.

To decouple data dependencies, we first ensure that each task uses a disjoint set of local variables by inserting new variables as needed; in this example, the tasks already use disjoint variables. Then we ensure that the dependencies are transmitted between tasks by replacing certain assignments with `send` and `recv` instructions on empty channels.

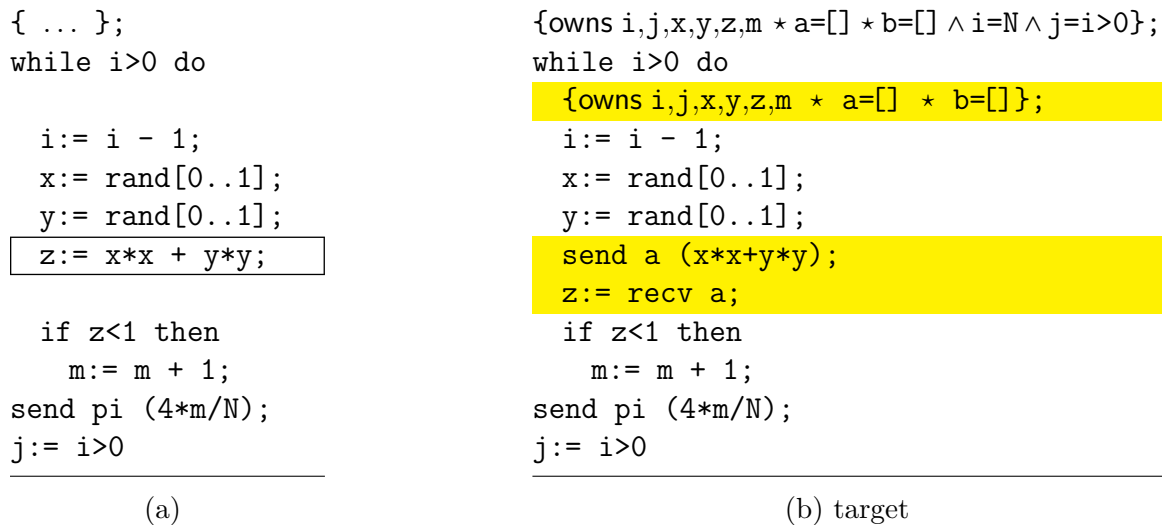


Figure 2.8: Decoupling z via a by transforming Fig. 2.6 into (b) (step 1/4)

We achieve this using rewrite rule T-AssignSendRecv, which transforms an assignment into a pair of send and receive instructions.

$$\frac{}{\{a=[]\}; x:= e \equiv \{a=[]\}; \text{send } a \ e; x:= \text{recv } a} \text{T-AssignSendRecv}$$

It uses an assertion to state that the channel must be empty, which also implies that the channel is private. Without the assertion, the equivalence might fail for two reasons. First, if the channel is not empty, then `recv` will retrieve the oldest value in the channel, not the value just sent. Second, if the channel is not private, then an observer may steal the value between `send` and `recv` (causing `recv` to block) or it may inject a different value to be received into x .

In Fig. 2.8b, we first infer a loop invariant stating that channels a and b are empty at the beginning of each iteration. Task two has a data dependency on task one through variable z . For this particular example, we do not need to create a redundant variable for z because it is only used by task two. We transmit the dependency from task one through channel a to task two by using T-AssignSendRecv. It replaces the

```

{ ... };
while i>0 do
  { ... };
  i:= i - 1;
  x:= rand[0..1];
  y:= rand[0..1];
  send a (x*x+y*y);

  z:= recv a;
  if z<1 then
    m:= m + 1;
send pi (4*m/N);
j:= i>0

```

(a)

```

{owns i,j,x,y,z,m * a=[] * b=[] ^ i=N ^ j=i>0};
while i>0 do
  {owns i,j,x,y,z,m * a=[] * b=[] ^ j=i>0};
  i:= i - 1;
  x:= rand[0..1];
  y:= rand[0..1];
  send a (x*x+y*y);
  j:= i>0;
  z:= recv a;
  if z<1 then
    m:= m + 1;
send pi (4*m/N);

```

(b) target

Figure 2.9: Duplicating loop condition $i>0$ via j by transforming Fig. 2.8b into (b) (step 2/4)

assignment to z with a `send` instruction that produces $x*x+y*y$ followed by a `recv` instruction to consume the value into variable z .

In Fig. 2.9b, we move the assignment to j into the loop using T-AssignCom, T-AssertAssign, and T-WhileAssign.

$$\frac{\text{writes } t \cap (\{x\} \cup \text{freevars } e) = \emptyset}{x := e; t \equiv t[e/x]; x := e} \text{ T-AssignCom}$$

$$\frac{x \notin \text{reads } t \cup \text{freevars } e_1 \cup \text{freevars } e_2}{\text{while } e_1 \text{ do } t; x := e_2 \equiv \text{while } e_1 \text{ do } (t; x := e_2); x := e_2} \text{ T-WhileAssign}$$

T-AssignCom commutes an assignment around a subprogram when the subprogram does not also write to the same variable ($t[e/x]$ denotes the substitution of any read from x in program t with expression e). T-WhileAssign moves an assignment that appears after a loop into the loop when the variable does not appear in the assignment expression, loop body, or loop condition. Then we strengthen the loop invariant by

```

{ ... };
while i>0 do
  { ... };
  i:= i - 1;
  x:= rand[0..1];
  y:= rand[0..1];
  send a (x*x+y*y);
  j:= i>0;

  z:= recv a;
  if z<1 then
    m:= m + 1;
send pi (4*m/N)

```

(a)

```

{owns i,j,x,y,z,m * a=[] * b=[] ^ i=N ^ j=i>0};
while i>0 do
  {owns i,j,x,y,z,m * a=[] * b=[] ^ j=i>0};
  i:= i - 1;
  x:= rand[0..1];
  y:= rand[0..1];
  send a (x*x+y*y);
  send b (i>0);
  j:= recv b;
  z:= recv a;
  if z<1 then
    m:= m + 1;
send pi (4*m/N)

```

(b) target

Figure 2.10: Decoupling j via b by transforming Fig. 2.9b into (b) (step 3/4)

```

{ ... };
while i>0 do
  { ... };

  i:= i - 1;
  x:= rand[0..1];
  y:= rand[0..1];
  send a (x*x+y*y);
  send b (i>0);

  j:= recv b;
  z:= recv a;
  if z<1 then
    m:= m + 1;
send pi (4*m/N)

```

(a)

```

{owns i,j,x,y,z,m * a=[] * b=[] ^ i=N ^ j=i>0};
while i>0 do
  {owns i,j,x,y,z,m * a=[] * b=[] ^ j=i>0};
  if i>0 then
    i:= i - 1;
    x:= rand[0..1];
    y:= rand[0..1];
    send a (x*x+y*y);
    send b (i>0);
  if j then
    j:= recv b;
    z:= recv a;
    if z<1 then
      m:= m + 1;
send pi (4*m/N)

```

(b) target

Figure 2.11: Decoupling control flow by transforming Fig. 2.10b into (b) (step 4/4)

adding condition $j=i>0$ so that it is nearly identical to the initial assertion (but where the value of i is not known).

In Fig. 2.10b, we prepare to separate the control-flow dependency by introducing `send/recv` instructions that transmit the loop condition over channel b to variable j (T-AssignSendRecv). (We infer that b is empty at this point from the loop invariant.)

Then we use T-WhileAssertI to infer that $i > 0$ and j are both true within the loop in order to insert two `if`-statements via T-TrueIf. This intercepts the control dependence from the `while`-loop to its body, in Fig. 2.11b, fully dividing the body into the two tasks. Although the `if`-statements are currently redundant, they will allow us to parallelize the loop in phase 3 (Section 2.3.4) without directly reasoning about the termination condition of the current loop.

2.3.3 Phase 2: Parallelizing the loop body

We first parallelized a loop body in Section 2.2.1 using T-SeqParA. But since our current example uses channels, we introduce a more general rule:

$$\begin{array}{c}
P_1 : t_1 \Downarrow \\
\text{recvs } t_1 \cap \text{recvs } t_2 = \emptyset \\
P_1 * P_2 \vdash (\text{owns}(\text{freechans } t_1) \vee \text{owns}(\text{freechans } t_2)) * \text{true} \\
\text{writes } t_1 \cap \text{freevars } t_2 = \text{freevars } t_1 \cap \text{writes } t_2 = \emptyset \\
\{P_1 * P_2\}; t_1 \equiv [P_1] t_1 \parallel [P_2] \text{skip} \\
\{P_1 * P_2\}; t_2 \equiv [P_2] t_2 \parallel [P_1] \text{skip} \\
\hline
\{P_1 * P_2\}; t_1; t_2 \equiv [P_1] t_1 \parallel [P_2] t_2
\end{array}
\quad \text{T-SeqParC}$$

We explain the new judgments in order:

- $\text{recvs } t_1 \cap \text{recvs } t_2 = \emptyset$ – the set of channels that t_1 receives from, $\text{recvs } t_1$, is disjoint from the set of channels that t_2 receives from. In other words, t_1 and t_2 cannot both receive from the same channel. Doing so would allow t_2 to steal values destined for t_1 in the parallelized program, possibly causing t_1 to block.
- $P_1 * P_2 \vdash (\text{owns}(\text{freechans } t_1) \vee \text{owns}(\text{freechans } t_2)) * \text{true}$ – any channels used by t_1 (or alternately, t_2) are jointly owned by t_1 and t_2 . In other words, one of the threads may only communicate with the other thread; not with an observer or other threads in the surrounding context. Otherwise, we would be able to parallelize `send a 0` with `send b 1` for observable channels a and b , but doing

```

{owns i,j,x,y,z,m * a=[] * b=[] ^ i=N ^ j=i>0};
while i>0 do
  {owns i,j,x,y,z,m * a=[] * b=[] ^ j=i>0};
  [ [owns i,x,y * cansend a,b] || [owns j,z,m * canrecv a,b] ]
  [ if i>0 then
    i:= i - 1;
    x:= rand[0..1];
    y:= rand[0..1];
    send a (x*x+y*y);
    send b (i>0)
  || if j then
    j:= recv b;
    z:= recv a;
    if z<1 then
      m:= m + 1
  ];
send pi (4*m/N)

```

Figure 2.12: Parallelized loop body

so would result in different observed traces between the parallel and sequential programs. (Joint ownership arises from `owns` being derived from $P_1 * P_2$, and $(\dots * \text{true})$ allows $P_1 * P_2$ to specify even more channels or variables than stated by $\text{owns}(\text{freechans } t_1) \vee \text{owns}(\text{freechans } t_2)$.)

To parallelize the loop body, we apply T-SeqParC to Fig. 2.11b by choosing thread-guards $P_1 = \text{owns } i, x, y * \text{cansend } a, b$ and $P_2 = \text{owns } j, z, m * \text{canrecv } a, b$. This divides the variables between the threads and splits access to channels `a` and `b` such that the left thread only sends (`cansend a, b`) and the right thread only receives (`canrecv a, b`).

We do not parallelize the assertion of the loop invariant because neither thread-guard may, by itself, assert that the channel contents are empty or that $j=i>0$ due to ownership of the referenced channels and variables being split between them. (See Section 2.5.3 on how we might overcome this.) The conditions for T-SeqParC are met because the threads use disjoint variables, will always terminate, do not receive from the same channel, and do not communicate over any observable channel. Figure 2.12 is the result of parallelizing the loop body.

2.3.4 Phase 3: Loop folding

We transform Fig. 2.12 into Fig. 2.7 using T-FoldLoop and the loop schema

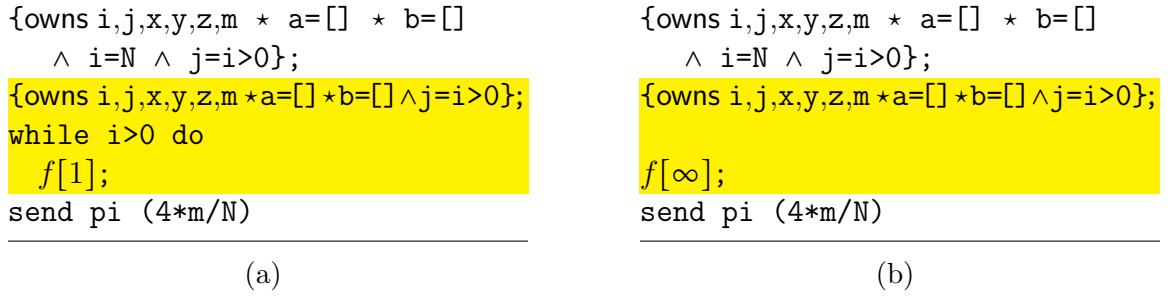


Figure 2.13: Parallelizing the loop

$$(f, i>0, \text{owns } i,j,x,y,z,m * a=[] * b=[] \wedge j=i>0),$$

where

$$f[z] = \left[\begin{array}{l} [\text{owns } i,x,y * \text{cansend } a,b] \\ \text{while } i>0 \text{ max } z \text{ do} \\ \quad i := i - 1; \\ \quad x := \text{rand}[0..1]; \\ \quad y := \text{rand}[0..1]; \\ \quad \text{send } a \text{ (} x*x+y*y \text{);} \\ \quad \text{send } b \text{ (} i>0 \text{)} \end{array} \parallel \left[\begin{array}{l} [\text{owns } j,z,m * \text{canrecv } a,b] \\ \text{while } j \text{ max } z \text{ do} \\ \quad j := \text{recv } b; \\ \quad z := \text{recv } a; \\ \quad \text{if } z < 1 \text{ then} \\ \quad \quad m := m + 1 \end{array} \right].$$

Figure 2.12 is equivalent to Fig. 2.13a by T-WhileIf and T-WhileAssertI. By T-FoldLoop, Fig. 2.13a is equivalent to Fig. 2.13b, which is equivalent to Fig. 2.7. Then we show that $(f, i>0, \dots)$ has the properties of a loop schema. Proving that $i>0$ implies and is implied by termination of $f[1]$ is straightforward using T-FalseWhile and T-WhileFalse.

The loop invariant, particularly that a and b will be empty upon termination of $f[1]$, cannot be proved directly because the assertion logic is not strong enough to reason about how the channels are used between threads. However, it suffices to prove the loop invariant for the (sequential) loop body of Fig. 2.9b because we have

already proved it equivalent to $f[1]$. This is fairly straightforward, particularly by first converting the pair of `send/recv` instructions back into assignment statements.

The last property is the combining transformation. In Section 2.2.5, we used T-SeqParB to combine parallelized iterations, but it does not support communication channels. Thus we introduce our final, more general, parallelization rule:

$$\frac{\begin{array}{l} [P_1] t_1 \Downarrow [P_2] t_2 \quad \text{precise } P_3 \quad \text{precise } P_4 \\ \text{recvs}(t_1; t_3) \cap \text{recvs}(t_2; t_4) = \emptyset \\ P_1 \star P_2 \vdash \text{owns}(\text{freechans}(t_2; t_4)) \star \text{true} \\ \text{writes}(t_1; t_3) \cap \text{freevars}(t_2; t_4) = \emptyset \\ \text{freevars}(t_1; t_3) \cap \text{writes}(t_2; t_4) = \emptyset \\ \{P_1\}; t_1 \equiv \{P_1\}; t_1; \{P_3\} \\ \{P_2\}; t_2 \equiv \{P_2\}; t_2; \{P_4\} \end{array}}{([P_1] t_1 \parallel [P_2] t_2); ([P_3] t_3 \parallel [P_4] t_4) \equiv [P_1] (t_1; t_3) \parallel [P_2] (t_2; t_4)} \text{T-SeqPar} \clubsuit$$

We explain the new judgments in order:

- $\text{recvs}(t_1; t_3) \cap \text{recvs}(t_2; t_4) = \emptyset$ – neither t_1 nor t_3 receive from the same (public or private) channels as t_2 and t_4 .
- $P_1 \star P_2 \vdash \text{owns}(\text{freechans}(t_2; t_4)) \star \text{true}$ – t_2 and t_4 never communicate over an observable channel; any channel used by either must be fully owned by $P_1 \star P_2$.

This allows private communication with t_1 and t_3 .

T-SeqPar implies T-SeqParC by choosing either $t_2 = t_3 = \text{skip}$ for $t_1; t_4 \equiv t_1 \parallel t_4$ or $t_1 = t_4 = \text{skip}$ for $t_2; t_3 \equiv t_2 \parallel t_3$, depending on whether t_1 or t_3 communicates over a public channel. Thus T-SeqPar trivially implies T-SeqParA and T-SeqParB as well.

By T-SeqPar and then T-Split $_{\diamond}$:

$$\begin{array}{l} \{\text{owns } i, j, x, y, z, m \star a = [] \star b = [] \wedge j = i > 0\}; \\ f[n \square 0]; f[u] \end{array}$$

$$\begin{aligned}
&= \{ \text{owns } i, j, x, y, z, m * a=[] * b=[] \wedge j=i>0 \}; && \text{by def. of } f \\
&\left[\begin{array}{l} [\text{owns } i, x, y * \text{cansend } a, b] \\ \text{while } i>0 \text{ max } n \square 0 \text{ do} \\ \dots \end{array} \parallel \left[\begin{array}{l} [\text{owns } j, z, m * \text{canrecv } a, b] \\ \text{while } j \text{ max } n \square 0 \text{ do} \\ \dots \end{array} \right] \right]; \\
&\left[\begin{array}{l} [\text{owns } i, x, y * \text{cansend } a, b] \\ \text{while } i>0 \text{ max } u \text{ do} \\ \dots \end{array} \parallel \left[\begin{array}{l} [\text{owns } j, z, m * \text{canrecv } a, b] \\ \text{while } j \text{ max } u \text{ do} \\ \dots \end{array} \right] \right]; \\
&\equiv \{ \text{owns } i, j, x, y, z, m * a=[] * b=[] \wedge j=i>0 \}; && \text{by T-SeqParC} \\
&\left[\begin{array}{l} [\text{owns } i, x, y * \text{cansend } a, b] \\ \text{while } i>0 \text{ max } n \square 0 \text{ do} \\ \dots \\ \text{while } i>0 \text{ max } u \text{ do} \\ \dots \end{array} \parallel \left[\begin{array}{l} [\text{owns } j, z, m * \text{canrecv } a, b] \\ \text{while } j \text{ max } n \square 0 \text{ do} \\ \dots \\ \text{while } j \text{ max } u \text{ do} \\ \dots \end{array} \right] \right]; \\
&\equiv \{ \text{owns } i, j, x, y, z, m * a=[] * b=[] \wedge j=i>0 \}; && \text{by T-Split}_{\diamond} \\
&\left[\begin{array}{l} [\text{owns } i, x, y * \text{cansend } a, b] \\ \text{while } i>0 \text{ max } n \square u \text{ do} \\ \dots \end{array} \parallel \left[\begin{array}{l} [\text{owns } j, z, m * \text{canrecv } a, b] \\ \text{while } j \text{ max } n \square u \text{ do} \\ \dots \end{array} \right] \right]; \\
&= \{ \text{owns } i, j, x, y, z, m * a=[] * b=[] \wedge j=i>0 \}; && \text{by def. of } f \\
&f[n \square u].
\end{aligned}$$

The side conditions of T-SeqPar are satisfied:

- The first pair of threads, bounded by $n \square 0$, coterminate. If the left thread (conditioned on $i>0$) terminates, then $\leq n$ values were sent on both local channels a and b ; either exactly n iterations were made, or the last value sent on b is **false**. The right thread (conditioned on j) can receive as many values without blocking, at which point it will have either made n iterations or the value received into j will be **false** and it will terminate. If the right thread terminates, then $\leq n$ values were received. Because the loop invariant is that both a and b are empty, it must be the case that as many values were sent from the left thread. Either n iterations were made or **false** was sent on channel b (in which case $\neg i>0$), so the left thread must also terminate.

```

{owns i,x,y,z,m,i',x',y',z',m' * a=[] * b=[] ^ i=N ^ i'=i};
while i>0 do
  i:= i - 1;
  x:= rand[0..1];
  y:= rand[0..1];
  z:= x*x + y*y;
  if z<1 then
    m:= m + 1;
send pi (4*m/N);
i':= 0; x':= 0; y':= 0; z':= 0; m':= 0

```

Figure 2.14: Approximating π ; this is the same program as Fig. 2.6, but with minor adjustments to allow us to introduce duplicate variables.

- The left threads (conditioned on $i>0$) never receive values (and thus do not receive from the same channels as the right threads).
- No threads communicate over observable (public) channels.
- The threads use disjoint local variables.
- The first left thread preserves its guard at termination, and thus satisfies the (same) guard of the next left thread.
- The first right thread preserves its guard at termination, and thus satisfies the (same) guard of the next right thread.

2.4 Example: DOACROSS parallelization

Our final example is to apply DOACROSS parallelization to (nearly) the same program that we worked with in the preceding section, listed in Fig. 2.14. Like DOALL parallelization, DOACROSS divides the iterations of a sequential loop into multiple parallel loops. However, the iterations need not be independent – we add synchronization to coordinate access to shared resources. In particular, we introduce pairs of `send/recv` instructions so that the threads strictly alternate their access of the shared resources.

```

{owns i,x,y,z,m,i',x',y',z',m' * a=[] * b=[] ^ i=N ^ i'=i};
[ [owns i, x, y, z, m *
  canrecv a * cansend b
  if i>0 do
    x:= rand[0..1];
    y:= rand[0..1];
    z:= x*x + y*y;
  while i>0 do
    i:= i - 1;
    if z<1 then
      m:= m + 1;
    send b m;
    if i>0 then
      i:= i - 1;
      if i>0 do
        x:= rand[0..1];
        y:= rand[0..1];
        z:= x*x + y*y;
      m:= recv a
  ] || [ [owns i', x', y', z', m' *
  cansend a * canrecv b
  while i'>0 do
    i':= i' - 1;
    if i'>0 then
      x':= rand[0..1];
      y':= rand[0..1];
      z':= x'*x' + y'*y';
    m':= recv b;
    if i'>0 then
      i':= i' - 1;
      if z'<1 then
        m':= m' + 1;
      send a m'
  ] ];
send pi (4*m/N);
i':= 0; x':= 0; y':= 0; z':= 0; m':= 0

```

Figure 2.15: DOACROSS parallelization of Fig. 2.14

Figure 2.15 lists the final result of our application of DOACROSS. To parallelize the program, we introduce temporary variables i' , x' , y' , z' , and m' to duplicate variables i , x , y , z , and m , respectively. (Because the language does not support scoping, we define the variables in the initial assertion and set them to 0 at the end.)

The shared resource is the counter. When the left thread has exclusive access, it is stored in m , and when the right thread has exclusive access, it is stored in m' . Initially, the left thread has access to the counter, and each time that it is done using the resource, it releases access by sending its value into channel b ; to reacquire the resource, it receives from channel a . Likewise, the right thread acquires access by receiving from channel b and releases access by sending into channel a . It is important for each thread to do as much computation as possible without requiring access to the

```

{owns i,x,y,z,m,i',x',y',z',m' * a=[] * b=[] ^ i=N ^ i'=i};
while i>0 do
  i:= i - 1;
  x:= rand[0..1];
  y:= rand[0..1];
  z:= x*x + y*y;
  if z<1 then
    m:= m + 1;
  if i>0 then
    i:= i - 1;
    x:= rand[0..1];
    y:= rand[0..1];
    z:= x*x + y*y;
    if z<1 then
      m:= m + 1;
send pi (4*m/N);
i':= 0; x':= 0; y':= 0; z':= 0; m':= 0

```

Figure 2.16: Unroll the loop by T-Unroll.

shared resource in order to maximize concurrency, thus we have mostly minimized the amount of code that appears between each `recv` and subsequent `send`.

We implement DOACROSS in four phases: decoupling, parallelization, loop folding, and maximizing concurrency. In the first phase, we decouple odd iterations from even iterations by making them use disjoint variables and using `send/recv` to pass the counter from odd to even via channel `b` and even to odd via `a`. Then we parallelize each odd iteration with its subsequent even iteration in phase 2, followed by folding this parallelism into the whole loop in phase 3. Finally, phase 4 attempts to maximize concurrency by reordering instructions so that most of the odd computation can be performed in parallel with the even computation.

```

{owns i,x,y,z,m,i',x',y',z',m' * a=[] * b=[] ^ i=N ^ i'=i};
while i>0 do
  {owns i,x,y,z,m,i',x',y',z',m' * a=[] * b=[] ^ i'=i};
  i:= i - 1;
  x:= rand[0..1];
  y:= rand[0..1];
  z:= x*x + y*y;
  if z<1 then
    m:= m + 1;
  m' := m;
  i' := i;
  {i'=i};
  if i'>0 then
    i' := i'-1;
    x' := rand[0..1];
    y' := rand[0..1];
    z' := x'*x' + y'*y';
    if z'<1 then
      m' := m'+1;
    m := m';
    i := i';
  send pi (4*m/N);
i' := 0; x' := 0; y' := 0; z' := 0; m' := 0

```

Figure 2.17: Make odd and even iterations use disjoint variables.

2.4.1 Phase 1: Decoupling

Shown in Fig. 2.16, we start by *unrolling* the loop using the following rule:

$$\overline{\text{while } e \text{ do } t} \equiv \overline{\text{while } e \text{ do } (t; \text{if } e \text{ then } t)} \quad \text{T-Unroll} \quad \blacktriangleright$$

The first half of the resulting loop computes odd iterations and the second half computes even iterations.

Then in Fig. 2.17, we introduce duplicate (prime) variables so that the even iterations use different variables than the odd iterations. To support this, we also introduce assignments from m to m' and from i to i' when transitioning from odd to

```

{owns i,x,y,z,m,i',x',y',z',m' * a=[] * b=[] ^ i=N ^ i'=i};
while i>0 do
  {owns i,x,y,z,m,i',x',y',z',m' * a=[] * b=[] ^ i'=i};
  i:= i - 1;
  x:= rand[0..1];
  y:= rand[0..1];
  z:= x*x + y*y;
  if z<1 then
    m:= m + 1;
    send b m;
    m':= recv b;
    i':= i' - 1;
  {i'=i};
  if i'>0 then
    i':= i' - 1;
    x':= rand[0..1];
    y':= rand[0..1];
    z':= x'*x' + y'*y';
    if z'<1 then
      m':= m' + 1;
      send a m';
      m:= recv a;
      i:= i - 1;
    send pi (4*m/N);
    i':= 0; x':= 0; y':= 0; z':= 0; m':= 0

```

Figure 2.18: Decoupling the dependency between i and i' , and between m and m' .

even iterations, and from m' to m and from i' to i when transitioning from even to odd iterations. By adding assignments $i' := i$ and $i := i'$, we can introduce assertions to state that i' and i are equal in between odd and even iterations in order to simplify later reasoning.

In Fig. 2.17, we replace $i' := i$ with $i' := i' - 1$ and $i := i'$ with $i := i - 1$ to remove the dependency between i and i' . To remove the dependency between m and m' , we instead introduce `send/recv` pairs using T-AssignSendRecv because the counter will be modified by both odd and even iterations in a nontrivial way. We use two different channels, `a` and `b`, for this to avoid having the odd and even iterations

```

{owns i,x,y,z,m,i',x',y',z',m' * a=[] * b=[] ^ i=N ^ i'=i};
while i>0 do
  {owns i,x,y,z,m,i',x',y',z',m' * a=[] * b=[] ^ i'=i};
  if i>0 then
    i:= i - 1;
    x:= rand[0..1];
    y:= rand[0..1];
    z:= x*x + y*y;
    if z<1 then
      m:= m + 1;
    send b m;
  if i'>0 then
    m' := recv b;
    i' := i' - 1;
  {i'=i};
  if i'>0 then
    i' := i' - 1;
    x' := rand[0..1];
    y' := rand[0..1];
    z' := x'*x' + y'*y';
    if z'<1 then
      m' := m' + 1;
    send a m';
  if i>0 then
    m:= recv a;
    i:= i - 1;
  send pi (4*m/N);
i' := 0; x' := 0; y' := 0; z' := 0; m' := 0

```

Figure 2.19

both receive from the same channel, as this would be incompatible with T-SeqPar when we later attempt to parallelize them.

Finally, we split up the control flow between odd and even iterations in Fig. 2.19. Because we introduced assignments between m , m' , i , and i' in the previous step, the odd and even iterations are now broken into two parts each that are interleaved. The first part of the odd iteration computes x , y , and z , and then releases the counter; the first part of the even iteration decrements i' and acquires the counter. This is followed by the second part of the even iteration, its computation and release of the

counter, and then the second part of the odd iteration, to decrement i and reacquire the counter. We separate these parts by surrounding them by if-statements.

The first two if-statements are introduced by T-TrueIf because $i > 0$ and $i' = i$ hold at the beginning of the loop. We obtain the last if-statement by T-TrueIf and:

$$\frac{}{\{-e\}; \text{if } e \text{ then } t_1 \text{ else } t_2 \equiv \{-e\}; t_2} \text{T-FalseIf} \quad \clubsuit$$

$$\frac{}{\text{if } e \text{ then } (t_1; t_3) \text{ else } (t_2; t_3) \equiv \text{if } e \text{ then } t_1 \text{ else } t_2; t_3} \text{T-IfBack} \quad \clubsuit$$

First, we use T-TrueIf to surround the assignment to m and i by an if-statement conditioned on $i > 0$, and we then use T-FalseIf to introduce the same if-statement into the false-branch of the preexisting if-statement. We use T-IfBack to move the newly-introduced if-statements to just after the preexisting if-statement.

2.4.2 Phase 2: Parallelizing the loop body

We apply T-SeqParC to parallelize the first parts of the odd and even iterations. Then we apply T-SeqParC again to parallelize the second parts of the odd and even iterations, followed by applying rule:

$$\frac{}{[P_1] t_1 \parallel [P_2] t_2 \equiv [P_2] t_2 \parallel [P_1] t_1} \text{T-ParCom} \quad \clubsuit$$

to swap the threads so that the left and right threads correspond to the odd and even iterations, respectively. Parallelization holds because the odd and even iterations use disjoint variables, all channels are private, they do not receive from the same channel, and the relevant parts always terminate. This results in Fig. 2.20.

```

{owns i,x,y,z,m,i',x',y',z',m' * a=[] * b=[] ^ i=N ^ i'=i};
while i>0 do
  {owns i,x,y,z,m,i',x',y',z',m' * a=[] * b=[] ^ i'=i};
  [
  [owns i,x,y,z,m *
  canrecv a * cansend b]
  if i>0 then
    i:= i - 1;
    x:= rand[0..1];
    y:= rand[0..1];
    z:= x*x + y*y;
    if z<1 then
      m:= m + 1;
    send b m
  ] ||
  [owns i',x',y',z',m' *
  cansend a * canrecv b]
  if i'>0 then
    m':= recv b;
    i':= i' - 1
  ];
  {i'=i};
  [
  [owns i,x,y,z,m *
  canrecv a * cansend b]
  if i>0 then
    m:= recv a;
    i:= i - 1
  ] ||
  [owns i',x',y',z',m' *
  cansend a * canrecv b]
  if i'>0 then
    i':= i' - 1;
    x':= rand[0..1];
    y':= rand[0..1];
    z':= x'*x' + y'*y';
    if z'<1 then
      m':= m' + 1;
    send a m'
  ];
  send pi (4*m/N);
  i':= 0; x':= 0; y':= 0; z':= 0; m':= 0

```

Figure 2.20: Both parts of the odd and even iterations have been parallelized

Finally, we remove assertion $\{i'=i\}$ and apply T-SeqPar to combine the two parallel sections so that the odd and even iterations are performed by the single left and right threads, respectively. This results in Fig. 2.21. The preconditions of T-SeqPar are satisfied because they use disjoint variables, all channels are private, they do not receive from the same channel, and the first part of the odd and even iterations coterminate. Cotermination holds in one direction because the first part of the odd iteration always terminates, and in the direction because channel b is initially empty and either both $i>0$ and $i'>0$ are false, or else running `send b m` (when the odd part terminates) ensures that `m':= recv b` will not block.

```

{owns i,x,y,z,m,i',x',y',z',m' * a=[] * b=[] ^ i=N ^ i'=i};
while i>0 do
  {owns i,x,y,z,m,i',x',y',z',m' * a=[] * b=[] ^ i'=i};
  [
  [owns i, x, y, z, m *
  canrecv a * cansend b]
  if i>0 then
    i:= i - 1;
    x:= rand[0..1];
    y:= rand[0..1];
    z:= x*x + y*y;
    if z<1 then
      m:= m + 1;
    send b m;
  if i>0 then
    m:= recv a;
    i:= i - 1
  ]
  [owns i', x', y', z', m' *
  cansend a * canrecv b]
  if i'>0 then
    m':= recv b;
    i':= i' - 1;
  if i'>0 then
    i':= i' - 1;
    x':= rand[0..1];
    y':= rand[0..1];
    z':= x'*x' + y'*y';
    if z'<1 then
      m':= m' + 1;
    send a m'
  ];
send pi (4*m/N);
i':= 0; x':= 0; y':= 0; z':= 0; m':= 0

```

Figure 2.21: The two parallel sections are combined.

2.4.3 Phase 3: Loop folding

Folding parallelism into the loop proceeds like Section 2.3.4. Before folding, however, we first nest the second if-statement of each thread inside their preceding if-statements by rules T-IfBack and T-FalseIf (Fig. 2.22). Then we apply T-FoldLoop using schema:

$$(f, i>0, \text{owns } i, x, y, z, m, i', x', y', z', m' * a=[] * b=[] \wedge i'=i),$$

where

```

{owns i,x,y,z,m,i',x',y',z',m' * a=[] * b=[] ^ i=N ^ i'=i};
while i>0 do
  {owns i,x,y,z,m,i',x',y',z',m' * a=[] * b=[] ^ i'=i};
  [
  [owns i, x, y, z, m *
  canrecv a * cansend b] || [owns i', x', y', z', m' *
  cansend a * canrecv b]
  if i>0 then
    i:= i - 1;
    x:= rand[0..1];
    y:= rand[0..1];
    z:= x*x + y*y;
    if z<1 then
      m:= m + 1;
      send b m;
      if i>0 then
        m:= recv a;
        i:= i - 1
    if i'>0 then
      m':= recv b;
      i':= i' - 1;
      if i'>0 then
        i':= i' - 1;
        x':= rand[0..1];
        y':= rand[0..1];
        z':= x'*x' + y'*y';
        if z'<1 then
          m':= m' + 1;
          send a m'
  ];
send pi (4*m/N);
i':= 0; x':= 0; y':= 0; z':= 0; m':= 0

```

Figure 2.22: The last if-statement of each thread is nested in the first if-statements.

$$f[z] = \left[\begin{array}{l} \text{owns } i, x, y, z, m * \\ \text{canrecv } a * \text{cansend } b \\ \text{while } i > 0 \text{ max } z \text{ do} \\ \quad i := i - 1; \\ \quad x := \text{rand}[0..1]; \\ \quad y := \text{rand}[0..1]; \\ \quad z := x*x + y*y; \\ \quad \text{if } z < 1 \text{ then} \\ \quad \quad m := m + 1; \\ \quad \text{send } b \text{ m}; \\ \quad \text{if } i > 0 \text{ then} \\ \quad \quad m := \text{recv } a; \\ \quad \quad i := i - 1 \end{array} \right] \parallel \left[\begin{array}{l} \text{owns } i', x', y', z', m' * \\ \text{cansend } a * \text{canrecv } b \\ \text{while } i' > 0 \text{ max } z \text{ do} \\ \quad m' := \text{recv } b; \\ \quad i' := i' - 1; \\ \quad \text{if } i' > 0 \text{ then} \\ \quad \quad i' := i' - 1; \\ \quad \quad x' := \text{rand}[0..1]; \\ \quad \quad y' := \text{rand}[0..1]; \\ \quad \quad z' := x'*x' + y'*y'; \\ \quad \quad \text{if } z' < 1 \text{ then} \\ \quad \quad \quad m' := m' + 1; \\ \quad \quad \text{send } a \text{ m}' \end{array} \right].$$

The result of loop folding is listed in Fig. 2.23. To use T-FoldLoop, we must prove that $(f, i > 0, \dots)$ has the properties of a loop schema, which is true for the same reasons found in Section 2.3.4. Proving that $i > 0$ implies and is implied by termination of $f[1]$ is straightforward using T-FalseWhile and T-WhileFalse.

```

{owns i,x,y,z,m,i',x',y',z',m' * a=[] * b=[] ^ i=N ^ i'=i};
[ [owns i, x, y, z, m *
  canrecv a * cansend b ] | [owns i', x', y', z', m' *
  cansend a * canrecv b ]
while i>0 do
  i:= i - 1;
  x:= rand[0..1];
  y:= rand[0..1];
  z:= x*x + y*y;
  if z<1 then
    m:= m + 1;
  send b m;
  if i>0 then
    m:= recv a;
    i:= i - 1
while i'>0 do
  m' := recv b;
  i' := i' - 1;
  if i'>0 then
    i' := i' - 1;
    x' := rand[0..1];
    y' := rand[0..1];
    z' := x'*x' + y'*y';
    if z'<1 then
      m' := m' + 1;
    send a m'
];
send pi (4*m/N);
i' := 0; x' := 0; y' := 0; z' := 0; m' := 0

```

Figure 2.23: The result of folding parallelism into the loop.

The loop invariant, particularly that a and b will be empty upon termination of $f[1]$, cannot be proven directly because the assertion logic is not strong enough to reason about how the channels are used between threads. However, it suffices to prove the loop invariant for the (sequential) loop body of Fig. 2.18 because we have already proved it equivalent to $f[1]$. The last property is the combining transformation, which follows from T-SeqPar and then T-Split $_{\diamond}$.

2.4.4 Phase 4: Maximize concurrency

At this point, we have applied DOACROSS such that the loop is parallelized and access to the counter is correctly synchronized. However, the program in Fig. 2.23 will be slower than the original program because of the added synchronization operations and the fact that all computation is still being performed in sequence. Specifically, all of the computation is performed in critical sections – generally between `recv` and `send`. To make DOACROSS worthwhile, we need to move as much of the computation as possible outside of such critical sections.

```

while i>0 do
  i:= i - 1;
  x:= rand[0..1];
  y:= rand[0..1];
  z:= x*x + y*y;

  if z<1 then
    m:= m + 1;
  send b m;
  if i>0 then
    m:= recv a;
    i:= i - 1

```

(a) The left loop

```

while i>0 do
  x:= rand[0..1];
  y:= rand[0..1];
  z:= x*x + y*y;

  i:= i - 1;
  if z<1 then
    m:= m + 1;
  send b m;
  if i>0 then
    m:= recv a;
    i:= i - 1

```

(b) Commute $i:=i-1$

```

while i>0 do
  x:= rand[0..1];
  y:= rand[0..1];
  z:= x*x + y*y;
  if i>0 then
    i:= i - 1;
    if z<1 then
      m:= m + 1;
    send b m;
    if i>0 then
      m:= recv a;
      i:= i - 1

```

(c) Intro if-statement

Figure 2.24: Moving the left loop computation outside of its critical section (1/2).

We start by focusing on the loop in the left thread that performs the odd iterations, shown in Fig. 2.24a. The computation of x , y , and z is effectively in a critical section because the `recv` statement at the end of the loop blocks it from going to the next iteration to perform the next computation. We can resolve this by moving the computation, which does not depend on m , above the preceding `recv` statement.

Thus our immediate goal is to reorder the loop via a *loop rotation* so that the first computation is performed before the loop and the `recv` statement appears before the remaining computations inside the loop body. We use the following rule in reverse:

$$\frac{\text{if } e \text{ then } t_1; \text{ while } e \text{ do } (t_2; \text{ if } e \text{ then } t_1)}{\text{while } e \text{ do } (t_1; \text{ if } e \text{ then } t_2)} \quad \text{T-Rotate} \quad \clubsuit$$

In T-Rotate, the computation will be represented by t_1 and the rest of the loop by t_2 . To prepare for rotation, we isolate the computation to the top of the loop body. We first commute the decrementing of i to below the computation (Fig. 2.24b), and then introduce an if-statement to surround the rest of the loop (Fig. 2.24c).

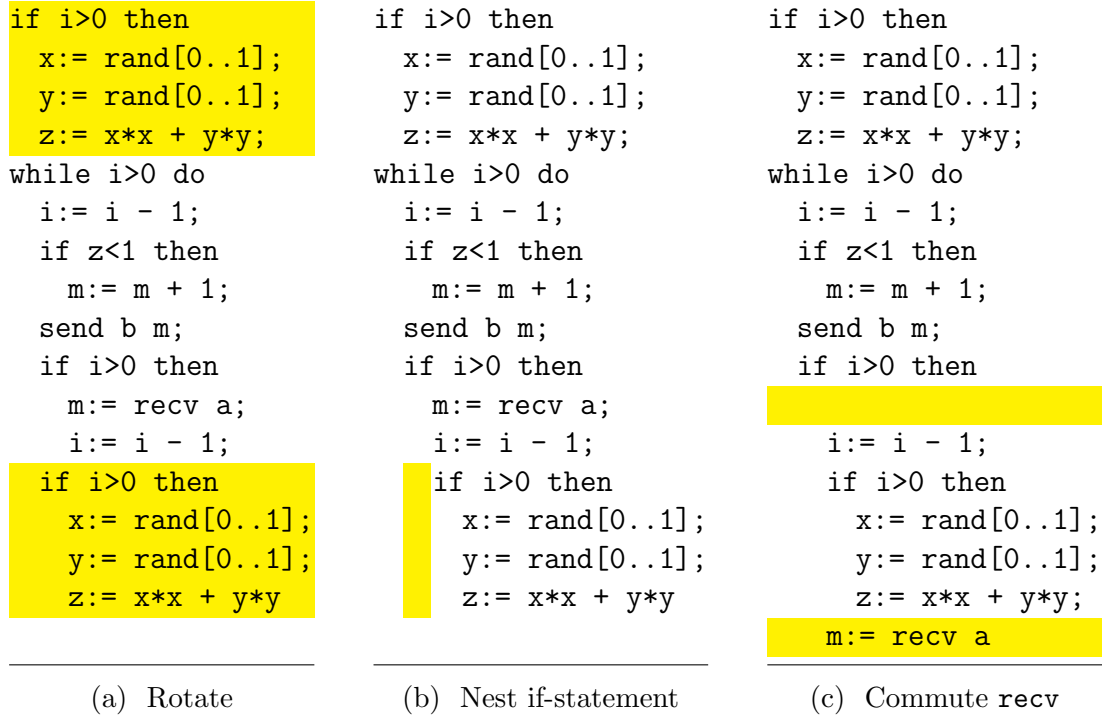


Figure 2.25: Moving the left loop computation outside of its critical section (2/2).

After applying T-Rotate, we get Fig. 2.25a. Then we nest the last if-statement inside of the preceding if-statement via T-IfBack and T-FalseIf in Fig. 2.25b. Finally, we commute `m:= recv a` after the computation. Thus Fig. 2.25c has moved most of its computation outside of its critical section.

Now we focus on the loop in the right thread, which performs the even iterations and is shown in Fig. 2.26a. Conveniently, we will not need to perform loop rotation because the computation already appears syntactically after a `recv` statement. The first step, shown in Fig. 2.26b, is to move the computation above the if-statement; this is to separate it from computing `m`?. Finally, we move `recv` below the computation, as shown in Fig. 2.26c.

After reorganizing the left and right loops to maximize concurrency, the final result of our DOACROSS parallelization is the program listed in Fig. 2.15.

<pre> while i'>0 do m':= recv b; i':= i' - 1; if i'>0 then i':= i' - 1; x':= rand[0..1]; y':= rand[0..1]; z':= x'*x' + y'*y'; if z'<1 then m':= m' + 1; send a m' </pre>	<pre> while i'>0 do m':= recv b; i':= i' - 1; if i'>0 then x':= rand[0..1]; y':= rand[0..1]; z':= x'*x' + y'*y'; if i'>0 then i':= i' - 1; if z'<1 then m':= m' + 1; send a m' </pre>	<pre> while i'>0 do i':= i' - 1; if i'>0 then x':= rand[0..1]; y':= rand[0..1]; z':= x'*x' + y'*y'; m':= recv b; if i'>0 then i':= i' - 1; if z'<1 then m':= m' + 1; send a m' </pre>
(a) The right loop	(b) Commute xyz	(c) Commute recv

Figure 2.26: Moving the right-loop computation outside of its critical section.

2.5 Conclusion & related work

2.5.1 Remarks on loop folding

We considered three variations of loop folding before settling on our current rule.

$$\frac{\{P\}; f[1] \equiv \{P\}; g[1] \quad (f, e, P) \quad (g, e, P)}{\{P\}; f[\infty] \equiv \{P\}; g[\infty]} \quad (2.1)$$

In eq. (2.1), using two loop schemas allows the user to perform more direct loop transformations. For example, we could have transformed Fig. 2.11b into Fig. 2.13b without first parallelizing the loop body (Fig. 2.12). Rather, parallelizing the loop body would have been done by proving $\{P\}; f[1] \equiv \{P\}; g[1]$.

However, this is equivalent to first transforming each loop schema into a **while** loop, as we did in T-FoldLoop. To transform a loop schema into a **while** loop, however, it is necessary to strengthen the termination property so that $\{\neg e \wedge P\}; f[u] \equiv$

$\{\neg e \wedge P\}$ for any u (or precisely, for $u \in \{1, \infty\}$). But we expect that, in practice, $\neg e$ will always cause any loop schema to terminate regardless of how many iterations it is given, so we think this is worth the simplicity of only dealing with one loop schema. (We are not certain whether this more strict termination property is actually equivalent; nondeterminism makes it less clear.)

$$\begin{array}{c}
x > 0 \\
\{P\}; f[x] \equiv \{P\}; f[x]; \{P\} \\
\{\neg e \wedge P\}; f[\infty] \equiv \{\neg e \wedge P\} \\
\{P\}; f[\infty] \equiv \{P\}; f[\infty]; \{\neg e \wedge P\} \\
\forall n \geq 1, u. \{P\}; f[n \square 0]; f[u] \equiv \{P\}; f[n \square u] \\
\hline
\{P\}; \text{while } e \text{ do } f[x] \equiv \{P\}; f[\infty]
\end{array} \tag{2.2}$$

In eq. (2.2), parameter x allows the user to prove loop invariants by considering a number of iterations that may be more convenient; for example, the user could choose a P that holds only every 5 iterations. It also implies loop unrolling. However, we instead prove loop unrolling separately (see T-Unroll in table 2.2) because we can then derive eq. (2.2) from T-FoldLoop.

$$\begin{array}{c}
y \geq 0 \\
\{P\}; f[1] \equiv \{P\}; f[1]; \{P\} \\
\{\neg e \wedge P\}; f[\infty] \equiv \{\neg e \wedge P\} \\
\{P\}; f[\infty] \equiv \{P\}; f[\infty]; \{\neg e \wedge P\} \\
\forall n \geq 1, u. \{P\}; f[n \square 0]; f[u] \equiv \{P\}; f[n \square y + u] \\
\hline
\{P\}; \text{while } e \text{ do } f[1] \equiv \{P\}; f[\infty]
\end{array} \tag{2.3}$$

Parameter y in eq. (2.3) performs loop rotation, but like loop unrolling, we instead prove rotation separately (T-Rotate) so that we may derive eq. (2.3) from T-FoldLoop.

2.5.2 Remarks on parallelization

Although we believe T-SeqPar is very general, there are various uses where it is too strong. In particular, we cannot generally parallelize mutexes (implemented via

`send/recv`) because it does not allow the threads to mutate the same variables (there is a syntactic restriction) and because it does not allow the threads to receive from the same channel (which would be required to implement the mutex). We have not yet looked at adding more kinds of parallelizing transformations.

In the future, we would like to investigate a “cross parallelization” rule that is supported by smaller parallelization transformations and that abstracts further away from how channels and variables are used. What we have in mind might look like:

$$\frac{\{P_1 \star P_4\}; t_1; t_4 \equiv [P_1] t_1 \parallel [P_4] t_4 \quad \{P_2 \star P_3\}; t_2; t_3 \equiv [P_2] t_2 \parallel [P_3] t_3 \quad (\dots)}{([P_1] t_1 \parallel [P_2] t_2); ([P_3] t_3 \parallel [P_4] t_4) \equiv [P_1] (t_1; t_3) \parallel [P_2] (t_2; t_4)}$$

2.5.3 Strengthening the assertion logic

We present a simple assertion logic in this thesis that, as we mentioned in Section 2.3.4, is too weak to reason about how channels are used between threads. Thus we resorted to re-sequentializing the program in order to prove that certain channels remain empty as a postcondition. However, there are several ways to extend this logic to enable a direct proof that a channel is empty upon the join of multiple communicating threads. One is to combine concurrent separation logic with session types [14]. Another is to record the local history of channel usage for each thread, and then combine the histories when the threads join [4, 43]. These approaches effectively record a trace of channel usage that can be used to infer the state of a channel, such as the size of its buffer and possibly its contents, once all ownership is accounted for.

2.5.4 Finite communication channels

We chose to use unbounded communication channels for simplicity. In practice (i.e. in real compilers), these channels will likely have a finite capacity so that `send` will

block if its channel is full. Because our examples, rewrite rules, and proofs only assume that the capacity of each channel is nonzero, making such a change should have little impact on the results of this thesis. However, a general rule to commute `send` around `recv` for the same channel would need to ensure that neither instruction blocks. We would use the stronger assertion logic (above) to prove that the current capacity of the channel is not exceeded when `send` appears first and that the channel is not empty when `recv` appears first.

2.5.5 Existing proof of DSWP

Otoni gave an informal proof of DSWP, although his primary goal was to demonstrate that DSWP is an effective parallelizing optimization. We denote, by “DSWP”, the entire process of finding and applying pipelined thread-level parallelism to loops. However, he makes a distinction between partitioning a loop into tasks, which he calls DSWP, and using the partitioning to generate a parallelized program, which he calls multi-threaded code generation (MTCG).

In his implementation, a sequential source program is first converted into a program dependency graph (PDG) [12]; a graph that denotes the semantics of a program, where edges represent data and control flow dependencies and nodes generally represent blocks of sequential code. DSWP is the process of analyzing the PDG to search for potential pipelined thread-level parallelism in arbitrary loops; it attempts to find the most optimal parallelization strategy and proceeds to mark each node of the PDG with a thread ID accordingly. The next step is for MTCG to take the annotated PDG and, for each dependency between different thread IDs, inserts a pair of send and receive instructions into the connected nodes in order to communicate it between them (using a fresh channel for each edge). The modified PDG is then linearized – the nodes are transcribed into a straight-line program and instructions

are added to create channels and to manage the threads (create, fork, and join) that each node is associated with.

The argument for its soundness is that MTCG preserves all dependency edges. In other words, converting each register, memory, or control dependency to send/receive instructions preserves the edge. Thus the PDGs before and after MTCG are isomorphic. Finally, the Equivalence Theorem, as proved by Horwitz et al. [15], states that any two programs that have isomorphic PDGs are “strongly equivalent.”

They define “strong equivalence” between two programs to mean that, for any input, both will either terminate in the same state or neither will terminate (they may diverge or fail). The theorem assumes that the PDG nodes contain only single assignment statements (rather than blocks of instructions, as used by Ottoni). Sarkar proves a variations of the Equivalence Theorem [37] that supports nodes with blocks of sequential instructions, but requires the PDGs to be equal instead of isomorphic.

Although Ottoni provides a good intuition for why DSWP/MTCG should be sound, it would take a significant amount of work to complete his proof. The first challenge would be to formalize the semantics of a PDG that has concurrency and synchronization between threads. But the main challenge would be to prove the equivalence between a straight-line program and an “optimal” PDG, which we loosely define as a PDG that has at least enough edges removed so as to justify the kinds of optimizations that we should be able to perform on the program. In other words, given a program, it is trivial to find an equivalent PDG: construct a complete graph (i.e., where every pair of vertices is connected by an edge) that has a node for every instruction in the program. But for every edge removed from this graph, we must prove that its behavior does not change.

In our thesis, we have chosen to pursue the foundational questions behind parallelization, and not automation. Thus we do not use the PDG because its primary

purpose is to support automation by recording the results of static analyses so that it may be efficiently searched for the best optimization strategies.

2.5.6 Existing verified parallelizing optimizations

Hurlin proved partial correctness of an automated implementation of DOALL [16], which accepts a program with loops that are annotated with Hoare triples (backed by a separation logic) and transforms them into *annotated* parallel loops where possible. Separation logic is also used as the basis for the dependency analysis. The soundness criteria is that the source and parallelized programs both satisfy the same Hoare triple specification. In other words, if both programs terminate, then the resulting state will satisfy the same postcondition.

Botinčan et al. extended Hurlin’s proof-directed approach to support automatic DOACROSS optimizations by injecting barriers [7]. They prove a termination-sensitive trace equivalence. Scandolo et al. also proved a loop-parallelizing transformation, by injecting barriers, with respect to a semantic equivalence [38].

Kundu et al. [18] developed program equivalence checking (PEC), a translation validation framework to prove the correctness of several *sequential* loop transformations. Tatlock et al. [41] extended CompCert with this framework and proved soundness with respect to weak bisimulation.

Thus both parallelism, loop transformations, and the combination of the two have been explored in prior work. However, there are several differences that set our work apart. First, we target more general dependency patterns; DOALL and DOACROSS respectively assume that interiteration dependencies either are symmetric or do not exist; they use barriers for synchronization. We handle the asymmetric dependencies that DSWP targets, which allows us to support DOALL and DOACROSS optimizations as well; we use shared queues for synchronization (without value passing, these are equivalent to semaphores).

With respect to loop transformations, T-FoldLoop targets an orthogonal set of optimizations than what the related work (above) or even standard loop optimizations perform, such as fission/distribution, permutation, or polyhedra methods. A key difference is that T-FoldLoop is intended to work with potentially infinite loops – the user does not need to prove termination of the loop. Most other loop optimizations only make sense for loops that have fixed number of iterations, or that will at least terminate eventually. Furthermore, T-FoldLoop is *compositional*. For example, we can use it to parallelize a loop by combining it with other transformations, even though it does not know about parallelization by itself.

Finally, we prove T-SeqPar, T-FoldLoop, and most of the other transformations presented in this chapter, sound with respect to a termination sensitive bisimulation. This is a much stronger soundness criterion that typically used (in the work cited above, only Tatlock et al. prove a bisimulation). In contrast, most prior work prove either semantic equivalence (i.e., same output upon termination) or trace equivalence (i.e., same histories of actions). Unlike bisimulation, these two definitions do not distinguish between programs whose behaviors differ due to nondeterminism.

2.5.7 Automation

A clear difference between the above prior work and our own is that we do not have an automated implementation. A key challenge lies in the static analysis and heuristics that were implicit in Sections 2.3.2 and 2.4.1. But DOALL, DOACROSS, and DSWP already have automated algorithms [27] and we expect that similar techniques will apply. We used rewrite rules to prove many small and simple transformations, but translation validation [32] techniques used by Kundu, Tatlock, et al. [18, 41] could be more practical. In our examples, the loop-parallelization itself was relatively simple (aside from proving termination and cotermination), so we expect them to be straightforward to automate. However, any parallelization requires a comparable ter-

mination analysis on the original sequential program. Thus we need only show that introducing channel synchronization does not disrupt the results of the termination analysis that would have been used to plan the optimization in the first place.

2.5.8 Conclusion

We have presented an overview of our strategy for proving soundness of loop parallelization applied to instances of DOALL, DSWP, and DOACROSS. The key result of this chapter is our development of two general transformations for loop-folding and parallelization, and how they compose together to achieve loop parallelization. Another result is the framework of rewrite rules itself; we have shown how to combine small transformations to form complex optimizations.

In the next chapter, we will explore the foundations of parallelization and introduce the formal reasoning that we will use for proving soundness of the whole framework. In chapter 4, we will apply the theory of chapter 3 to the imperative language presented here, defining an operational semantics and finally proving T-FoldLoop.

2.6 Summary of framework rewrite rules

Tables 2.1 to 2.3 on the following pages present the transformation rules of our framework. In the course of doing this research, we have developed three different variations of our system in the Coq Proof Assistant. Various subsets of rules have been verified in different systems and with respect to slightly different equivalence properties. In the tables, we label the context in which each rule has been proved sound:

- * A variation of the Calculus of Communicating Systems (CCS-Seq; chapter 3)
- † An imperative language that was the precursor to the language presented in this chapter.
- ‡ The imperative language presented in this chapter (and formalized in chapter 4)

Note: T-FoldLoop was proved sound with respect to both bisimulation and con-
trasimulation in \ddagger ; but only bisimulation in \dagger .

The main differences between \dagger and \ddagger are the imperative language, observation model, and formulation of the correctness criteria. In general, \ddagger is a simpler and cleaned up development based on lessons learned from \dagger and $*$. It removes language features that we found to be orthogonal to proving the correctness of parallelization: a heap and first-class shared channels. The observation model in \dagger was to use an explicit observer thread that would interact with the program, and this notion was baked into our definitions of program equivalence. However, bisimulations are typically framed with respect to a labeled transition system, where each transition between states is labeled with an action that denotes an observation. We switched to using labeled transition systems in $*$ to study the program equivalences that we had developed because it resulted in a cleaner theory. Although not formally proven, we believe that the program equivalences used in \dagger and \ddagger are essentially equivalent, and we have yet to encounter a rewrite rule that is provable in \dagger or \ddagger but not the other. In other words, all of the following rules should be provable in both \dagger and \ddagger (although with more effort required for the former development).

Congruence

$$\begin{array}{c}
\frac{}{t \equiv t} \text{T-Ref} \quad *{\dagger\dagger} \\
\frac{t_1 \equiv t_2}{t_2 \equiv t_1} \text{T-Sym} \quad *{\dagger\dagger} \\
\frac{t_1 \equiv t_2 \quad t_2 \equiv t_3}{t_1 \equiv t_3} \text{T-Trans} \quad *{\dagger\dagger} \\
\frac{t_1 \equiv t_2}{t_1 \blacktriangleright \equiv t_2 \blacktriangleright} \text{T-Resume} \quad \dagger\dagger \\
\frac{t_1 \equiv t'_1 \quad t_2 \equiv t'_2}{t_1; t_2 \equiv t'_1; t'_2} \text{T-Seq} \quad *{\dagger\dagger} \\
\frac{P_1 \dashv\vdash P'_1 \quad t_1 \equiv t'_1 \quad P_2 \dashv\vdash P'_2 \quad t_2 \equiv t'_2}{[P_1] t_1 \parallel [P_2] t_2 \equiv [P'_1] t'_1 \parallel [P'_2] t'_2} \text{T-Par} \quad *{\dagger} \\
\frac{P \dashv\vdash P'}{\{P\} \equiv \{P'\}} \text{T-Assert} \quad \dagger\dagger \\
\frac{e \dashv\vdash e' \quad t_1 \equiv t'_1 \quad t_2 \equiv t'_2}{e \dashv\vdash e' \quad t \equiv t'} \text{T-If} \quad \dagger\dagger \\
\frac{}{\text{if } e \text{ then } t_1 \text{ else } t_2 \equiv \text{if } e' \text{ then } t'_1 \text{ else } t'_2} \text{T-If} \quad \dagger\dagger \\
\frac{}{\text{while } e \text{ max } z \text{ do } t \equiv \text{while } e' \text{ max } z \text{ do } t'} \text{T-While} \quad \dagger\dagger
\end{array}$$

Analysis

$$\begin{array}{c}
\vdash \{P\} t \{Q\} \\
\frac{}{\{P\}; t \equiv \{P\}; t; \{Q\}} \text{T-Hoare} \quad \dagger\dagger \\
\frac{}{\{P_1\}; \{P_2\} \equiv \{(P_1 * \text{true}) \wedge (P_2 * \text{true})\}} \text{T-AssertSeq} \quad \dagger\dagger \\
\frac{}{x := e \equiv x := e; \{x = e\}} \text{T-AssignAssert} \\
\frac{}{\{P\}; \text{if } e \text{ then } t_1 \text{ else } t_2 \equiv \text{if } e \text{ then } (\{e \wedge P\}; t_1) \text{ else } (\{\neg e \wedge P\}; t_2)} \text{T-IfAssert} \quad \dagger\dagger \\
\frac{}{\{P\}; \text{while } e \text{ max } z \text{ do } t \equiv \text{while } e \text{ max } z \text{ do } (\{e \wedge P\}; t); \{P\}} \text{T-WhileAssert} \quad \dagger \\
\frac{}{\text{while } e \text{ do } t \equiv \text{while } e \text{ do } t; \{\neg e\}} \text{T-WhileFalse} \quad \dagger\dagger
\end{array}$$

Table 2.1

Assignments

$$\frac{\{x = e\}; x := e \equiv \{x = e\}}{\text{T-AssertAssign}} \quad \frac{\{a = []\}; x := e \equiv \{a = []\}; \text{send } a \ e; x := \text{recv } a}{\text{T-AssignSendRecv}}$$

$$\frac{\text{writes } t \cap (\{x\} \cup \text{freevars } e) = \emptyset}{x := e; t \equiv t[e/x]; x := e} \quad \frac{x \notin \text{readst} \cup \text{freevars } e_1 \cup \text{freevars } e_2}{\text{while } e_1 \text{ do } t; x := e_2 \equiv \text{while } e_1 \text{ do } (t; x := e_2); x := e_2} \quad \text{T-WhileAssign}$$

While loops

$$\frac{\text{while } e \text{ max } 0 \text{ do } t \equiv \text{skip}}{\text{T-WhileZero}} \quad \frac{\text{while } e \text{ max } 1 \text{ do } t \equiv \text{if } e \text{ then } t}{\text{T-WhileIf}} \quad \frac{\text{while } e \text{ do } t \equiv \text{while } e \text{ do } (t; \text{if } e \text{ then } t)}{\text{T-Unroll}}$$

$$\frac{\{-e\}; \text{while } e \text{ max } z \text{ do } t \equiv \{-e\}}{\text{T-FalseWhile}} \quad \frac{\text{if } e \text{ then } t_1; \text{while } e \text{ do } (t_2; \text{if } e \text{ then } t_1) \equiv \text{while } e \text{ do } (t_1; \text{if } e \text{ then } t_2)}{\text{T-Rotate}}$$

$$\frac{\text{while } e \text{ max } (z \boxplus u) \text{ do } t \equiv (\text{while } e \text{ max } z \text{ do } t); (\text{while } e \text{ max } u \text{ do } t)}{\text{T-Split}_{\boxplus}} \quad \frac{\text{while } e \text{ max } (n \boxplus z) \text{ do } t \equiv (\text{while } e \text{ max } n \text{ do } t); (\text{while } e \text{ max } z \text{ do } t)}{\text{T-Split}_{\boxplus}}$$

$$\frac{\{P\}; f[1] \equiv \{P\}; f[1]; \{P\} \quad \forall n \geq 1, u. \{P\}; f[n \square 0]; f[u] \equiv \{P\}; f[n \square u]}{\forall u. \{-e \wedge P\}; f[u] \equiv \{-e \wedge P\} \quad \{P\}; f[\infty] \equiv \{P\}; f[\infty]; \{-e \wedge P\}} \quad \frac{\{P\}; \text{while } e \text{ do } f[1] \equiv \{P\}; f[\infty]}{\text{T-FoldLoop}}$$

Table 2.2

If statements

$$\begin{array}{c}
\frac{\{e\}; \text{if } e \text{ then } t_1 \text{ else } t_2 \equiv \{e\}; t_1}{\text{T-TrueIf} \quad \dagger\dagger} \quad \frac{\{\neg e\}; \text{if } e \text{ then } t_1 \text{ else } t_2 \equiv \{\neg e\}; t_2}{\text{T-FalseIf} \quad \dagger\dagger} \\
\frac{\text{if } e \text{ then skip} \equiv \{e \vee \neg e\}}{\text{T-IfSkip} \quad \dagger\dagger} \quad \frac{\text{if } e \text{ then } (t_1; t_2) \text{ else } (t_1; t_3) \equiv \{e \vee \neg e\}; t_1; \text{if } e \text{ then } t_2 \text{ else } t_3}{\text{T-IfFront} \quad \dagger} \\
\frac{\text{if } e \text{ then } (t_1; t_3) \text{ else } (t_2; t_3) \equiv \text{if } e \text{ then } t_1 \text{ else } t_2; t_3}{\text{T-IfBack}}
\end{array}$$

Sequential composition

$$\frac{\text{skip}; t \equiv t}{\text{T-SkipL} \quad \dagger\dagger} \quad \frac{t; \text{skip} \equiv t}{\text{T-SkipR} \quad \dagger\dagger} \quad \frac{(t_1; t_2); t_3 \equiv t_1; (t_2; t_3)}{\text{T-SeqAssoc} \quad \dagger\dagger}$$

Parallel composition

$$\frac{[P_1] t_1 \parallel [P_2] t_2 \equiv [P_2] t_2 \parallel [P_1] t_1}{\text{T-ParCom} \quad * \dagger\dagger} \quad \frac{[P_1] t_1 \parallel [P_2] t_2 \parallel [P_3] t_3}{\text{T-ParAssoc} \quad +} \\
\frac{[P_1] t_1 \parallel [P_2] t_2 \parallel [P_3] t_3 \equiv [P_1] t_1 \parallel [P_2] t_2 \parallel [P_3] t_3}{\text{T-ParAssoc} \quad +} \\
\frac{[P_1] t_1 \uparrow\uparrow [P_2] t_2 \quad \text{precise } P_3 \quad \text{precise } P_4 \quad P_1 * P_2 \vdash \text{owns}(\text{freechans}(t_2; t_4)) * \text{true} \quad \text{recvs}(t_1; t_3) \cap \text{recvs}(t_2; t_4) = \emptyset \quad \text{writes}(t_1; t_3) \cap \text{freevars}(t_2; t_4) = \emptyset \quad \text{freevars}(t_1; t_3) \cap \text{writes}(t_2; t_4) = \emptyset}}{(\text{T-SeqPar} \quad * \dagger)} \\
\frac{([P_1] t_1 \parallel [P_2] t_2); ([P_3] t_3 \parallel [P_4] t_4) \equiv [P_1] (t_1; t_3) \parallel [P_2] (t_2; t_4)}{\text{T-SeqPar} \quad * \dagger}$$

Table 2.3

Chapter 3

Foundations of Parallelization

We present a simple semantic model for parallelizing programs such that termination and behavioral properties are preserved. In this setting, we investigate several foundational questions of parallelization:

- how to state program equivalence (i.e., soundness) in a generic and composable manner,
- what is the strongest such equivalence we can state,
- how to account for nondeterminism, and
- what a minimal language for modeling parallelization might look like.

Additionally, this chapter serves as a primer on the formalisms that we will use in latter chapters to reason about programs, such as labeled transition systems, observability, termination sensitivity, and bisimulation relations. In keeping with existing efforts to prove compiler correctness, we use a bisimulation relation. Such relations are among the strongest in use by, for example, CompCert [20].

When comparing program behaviors, threading primitives – fork, join, and synchronization – and the scheduling of threads are often treated as observable actions. This is partially because it is easier to assume that they are implemented as external system calls rather than directly defining a language with concurrency semantics.

The other reason is that in some situations, how a program uses multithreading is an important attribute that must be preserved. However, admitting parallelization must presume that such threading primitives and the scheduler are *not* directly observable.

Combining parallelization and internal choice – the choices a source program makes that are not directly visible to the observer – raises an interesting challenge because parallelization may cause the nondeterminism to interleave with observable actions. Even when benign, a weak simulation (henceforth referred to as just “simulation”) is not preserved by this behavior. A potential solution is to first refine all internal choice, after which parallelization will preserve a weak bisimulation.

Internal choice can be intentional, however, and refining it may either be incorrect (depending on the language specification), cause the program to run slower (e.g. removing concurrency), or otherwise limit the ability of the compiler to most effectively choose how to refine internal choice. Thus we prove parallelization sound with respect to a new type of simulation relation, which we call *eventual simulation*, that allows the compiler to preserve internal choice.

Because eventual simulation is not symmetric, we use *contrasimulation* [13] to support the algebraic equivalence used in chapter 2. It is implied by eventual simulation and is a congruence for the imperative language used in chapter 2. When source programs do not contain internal choice, contrasimulation reduces to [weak] bisimulation and thus our proof of soundness for parallelization is still applicable in settings such as CompCert.

The contributions we present in this chapter are

- a proof of soundness for a general parallelizing transformation (corresponding to rule T-SeqPar, first described in Section 2.3.4);
- a simple model language, which we call CCS-Seq, that extends the Calculus of Communicating Systems with a sequential operator and semaphores in order to study parallelization;

- a new simulation relation, called eventual simulation, that preserves all forms of nondeterminism throughout parallelization;
- identifying a little-known simulation relation, called contrasimulation, that contains eventual simulation and is a congruence;
- a proof that contrasimulation (and thus eventual simulation) is equivalent to [weak] bisimulation for programs without internal choice; and
- mechanized proofs of this chapter in the Coq Proof Assistant.

These results are based on our conference paper, *Certifiably Sound Parallelizing Transformations* [3].

3.1 Introductory example

We begin by showing how a simple program might be parallelized. This example introduces a few of the basic concepts that we will be using throughout this thesis – transition diagrams, labeled transition systems, observable versus unobservable actions, internal choice, and comparing program behaviors using simulations and bisimulations. Finally, we define eventual simulation and prove that it holds for this example.

The following (sequential) program nondeterministically assigns either 1 or 2 to x , outputs 0, and then outputs x .

$$x := \text{either } 1 \text{ or } 2; \text{ print } 0; \text{ print } x \tag{3.1}$$

A potential result of parallelizing the program is:

$$(x := \text{either } 1 \text{ or } 2 \parallel \text{ print } 0); \text{ print } x, \tag{3.2}$$

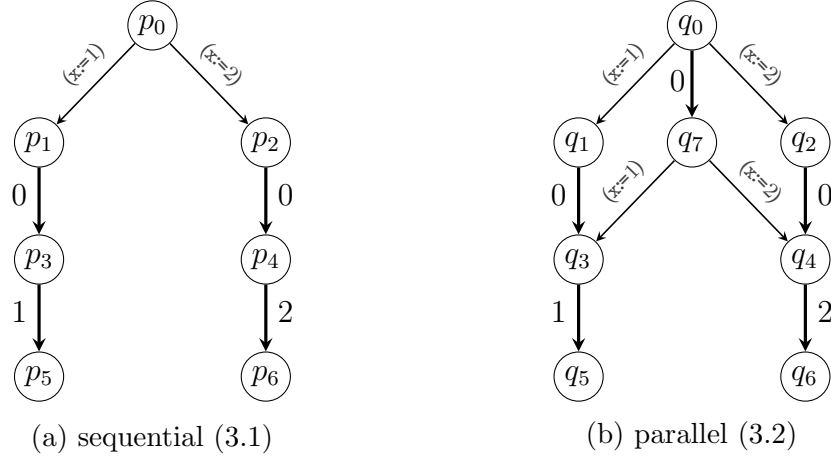


Figure 3.1: Semantics of the programs as transition diagrams.

which may also output 0 before choosing a value for x . The `either-or` statement is an instance of internal choice because it cannot be observed directly. (It is equivalent to $x := \text{rand } [1, 2]$ from chapter 2.) Although the parallelized program is intuitively “correct”, the original program does not simulate it because the parallel program can reorder nondeterministic choice with console output (an observable action).

Figure 3.1 presents the semantics of each program as transition diagrams where nodes represent program states and directed edges represent the possible transitions between states. The initial states are represented by the root nodes; p_0 and q_0 correspond to (3.1) and (3.2) respectively. Unobservable (silent) steps that correspond with a line of source code are labeled in parentheses. For example, choosing to assign either 1 or 2 to x is labeled $(x:=1)$ or $(x:=2)$, respectively. In later examples, silent steps may not be labeled at all. Observable actions (console output) have bold edges and are labeled with the corresponding observation.

Labeled transition systems. A labeled transition system (LTS) is defined by a triple, (S, L, δ) , where S is the set of states, L is the set of observable actions, and $\delta \subseteq S \times (L \cup \{\tau\}) \times S$ is the transition relation. A step from state p to p' that performs

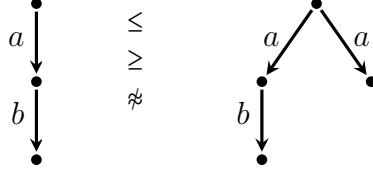


Figure 3.2: Bisimilarity is stronger than simulation equivalence.

action $\alpha \in L \cup \{\tau\}$ is defined if $(p, \alpha, p') \in \delta$ and is denoted by $p \xrightarrow{\alpha} p'$. τ is reserved for internal (silent) transitions; we write $p \rightarrow p'$ instead of $p \xrightarrow{\tau} p'$.

The transitive reflexive closure of \rightarrow is denoted by \Rightarrow . We write $p \xRightarrow{\alpha} p'$ for $\alpha \in L \cup \{\tau\}$ when p performs action α by taking zero or more steps to p' ; i.e. $p \xRightarrow{\alpha} p'$ for $\alpha \in L$ iff $p \Rightarrow \cdot \xrightarrow{\alpha} \cdot \Rightarrow p'$ and $\xrightarrow{\tau}$ is equivalent to \Rightarrow . A weak transition from p to p' that performs actions $\vec{a} = [a_0, \dots, a_n] \in L^*$ is defined as $p \Rightarrow \cdot \xRightarrow{a_0} \dots \xRightarrow{a_n} p'$ and is denoted by $p \xRightarrow{\vec{a}} p'$. Weak transitions over an empty list of actions may take multiple silent steps rather than just zero steps.

Two programs can be compared by showing that one mimics the other indefinitely, which is defined with respect to a LTS.

Definition 1 ($p \leq q$). $\mathcal{R} \subseteq S \times S$ is a *simulation* when for any $(p, q) \in \mathcal{R}$,

- if $p \xrightarrow{\alpha} p'$, then $q \xRightarrow{\alpha} q'$ and $(p', q') \in \mathcal{R}$ for some q' .

State q *simulates* p , written $p \leq q$ (or equivalently $q \geq p$), iff there exists a simulation \mathcal{R} such that $(p, q) \in \mathcal{R}$.

Many verified compilers are founded on bisimulation. It holds between two programs when each mimics (simulates) the other indefinitely, such that all pairs of transitional states continue to be bisimilar. \mathcal{R}^{-1} is the inverse of \mathcal{R} : $(q, p) \in \mathcal{R}^{-1}$ iff $(p, q) \in \mathcal{R}$ for any p and q .


Definition 2 ($p \approx q$). \mathcal{R} is a *bisimulation* when \mathcal{R} and \mathcal{R}^{-1} are simulations. p and q are *bisimilar*, written $p \approx q$, iff $(p, q) \in \mathcal{R}$ for some bisimulation \mathcal{R} .

If $p \geq q$ and $p \leq q$ for some p and q , then we say that p and q are *simulation equivalent*. At first glance, it may appear that simulation equivalence and bisimilarity

coincide, but bisimilarity is strictly stronger because it distinguishes between the *liveness* properties of each program. For example, the two programs in Fig. 3.2 are simulation equivalent and yet not bisimilar. It is possible for the right program to emit action a while following its right branch, only to become stuck (and no longer “live”). The left program simulates it by performing the same action, but then bisimulation fails between the resulting states because the left program is “live” – it may continue to emit b – while the right program has become stuck.

It is easy to prove that state q_0 simulates p_0 (Fig. 3.1) by showing that $\{(p_i, q_i) \mid 0 \leq i \leq 6\}$ is a simulation. However, simulation does not hold in the other direction:

Lemma 3.1. *Program (3.1) does not simulate (3.2): $p_0 \not\geq q_0$.*

Proof. By contradiction: assume $p_0 \geq q_0$. We take step $q_0 \xrightarrow{0} q_7$ without committing to print either 1 or 2. By assumption, p_0 must be able to mimic this action, thus $p_0 \xrightarrow{0} p'$ and $p' \geq q_7$ for some p' . Fig. 3.1a shows that p' is either p_3 or p_4 . In either case, q_7 may perform an action that p' is not capable of, thus $p' \not\geq q_7$. 

Because simulation fails in this direction, we must find a weaker relation for parallelization. Ideally, one that preserves as many of the strong properties of bisimulation as possible, such as a co-inductive proof method and the fact that related programs continue to mimic each other during execution (liveness). Observe that when the simulation fails in lemma 3.1, it is possible for the parallel program to eventually step to a state where simulation can be reestablished: from state q_7 to either q_3 or q_4 . It turns out that this “eventuality” holds for parallelization in general, and thus we formalize this idea as *eventual simulation*.


Definition 3 ($p \lesssim q$). \mathcal{R} is an *eventual simulation* when for any $(p, q) \in \mathcal{R}$,

- if $p \xrightarrow{\vec{a}} p'$, then $p' \Rightarrow p''$, $q \xrightarrow{\vec{a}} q''$, and $(p'', q'') \in \mathcal{R}$ for some p'' and q'' .


State q *eventually simulates* p , written $p \lesssim q$ (or $q \gtrsim p$), if there exists a simulation \mathcal{R} whose inverse is an eventual simulation and $(p, q) \in \mathcal{R}$.

Eventual similarity is like bisimilarity – and unlike plain similarity – in that both programs mimic each other indefinitely, but we still call it a “similarity” because it is asymmetric. It differs from them in that multiple actions *must* be considered for every point in the relationship. (For simulation and bisimulation, handling multiple actions at each point is equivalent to considering just one step) Bisimilarity implies eventual similarity, eventual similarity is reflexive and transitive, and crucially, it holds for a large class of parallelizing transformations in addition to our simple example. Further properties are explored in Section 3.3.


Lemma 3.2. *If \mathcal{R} is a simulation, then it is an eventual simulation.*

Proof. By definition 1 and induction on $p \xrightarrow{\vec{\alpha}} p'$, we can prove that if $p \xrightarrow{\vec{\alpha}} p'$ for any p' and α , then there exists a q' such that $q \xrightarrow{\vec{\alpha}} q'$ and $(p', q') \in \mathcal{R}$. We use this to prove that \mathcal{R} is an eventual simulation by always choosing $p' \Rightarrow p'$ and $q \xrightarrow{\vec{\alpha}} q'$ such that $(p', q') \in \mathcal{R}$. 

Lemma 3.3. *If $p \approx q$, then $p \lesssim q$ (and $p \gtrsim q$ because \approx is symmetric).*

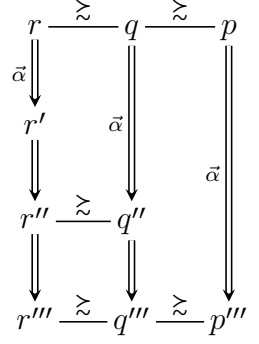
Proof. By \approx , there exists a simulation \mathcal{R} such that \mathcal{R}^{-1} is a simulation and $(p, q) \in \mathcal{R}$. To prove \lesssim , we pick the same \mathcal{R} because its inverse is an eventual simulation by lemma 3.2. 

Lemma 3.4. *\lesssim is reflexive.*

Proof. Follows by proving that \mathcal{I} (the identity relation) is an eventual simulation. 

Lemma 3.5. *\lesssim is transitive.*

Proof. We choose $\mathcal{R} = \{(p, r) \mid \exists q. p \lesssim q \wedge q \lesssim r\}$. Proving that \mathcal{R} is a simulation is straightforward. To prove that \mathcal{R}^{-1} is an eventual simulation, we will need to use the fact that simulation holds in the other direction. Assume that $(p, r) \in \mathcal{R}$ and $r \xRightarrow{\vec{\alpha}} r'$ for some p and r . By $q \lesssim r$, there is an r'' and q'' such that $r' \Rightarrow r''$, $q \xRightarrow{\vec{\alpha}} q''$, and $q'' \lesssim r''$. By $p \lesssim q$, there is a p''' and q''' such that $p \xRightarrow{\vec{\alpha}} p'''$, $q'' \Rightarrow q'''$, and $p''' \lesssim q'''$. By $q'' \lesssim r''$ (in the simulation direction), there is an r''' such that $r'' \Rightarrow r'''$, $p''' \lesssim q''' \lesssim r'''$, and thus $(p''', r''') \in \mathcal{R}$. \square



Lemma 3.6. *Figure 3.1b eventually simulates Fig. 3.1a: $p_0 \lesssim q_0$.*

Proof. We select relation $\mathcal{R} = \{(p_i, q_i) \mid 0 \leq i \leq 6\}$. Trivially, $(p_0, q_0) \in \mathcal{R}$ and \mathcal{R} is a simulation. Finally, we prove that \mathcal{R}^{-1} is an eventual simulation. The interesting case is for $(p_0, q_0) \in \mathcal{R}$, when $q_0 \xRightarrow{0} q_7$. In response, we have p_0 follow by $p_0 \xRightarrow{0} p_3$ and $q_7 \Rightarrow q_3$. (We may have instead chosen to step to p_4 and q_4 .) \square

3.2 CCS-Seq

We now investigate parallelization for an extension of the Calculus of Communicating Systems (CCS) [22]. CCS is widely used as a model for analyzing bisimulation relations and the behavior of programs and systems with multiple concurrent agents acting in concert via message passing and synchronization. However, we must extend CCS with a sequential composition operator in order to model parallelizing transformations. Furthermore, implementing asynchronous communication on top of CCS-style synchronous channels is tedious and not modular (requiring auxiliary threads for buffering), so we replace its channels with semaphores. We refer to this language as CCS-Seq.

$$\begin{array}{c}
\frac{P \xrightarrow{\alpha} P'}{P; Q \xrightarrow{\alpha} P'; Q} \quad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \quad \frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'} \quad \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \quad \frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'} \\
\frac{P \xrightarrow{\alpha} P' \quad \alpha \notin \{a, \bar{a}\}}{va:n.P \xrightarrow{\alpha} va:n.P'} \quad \frac{P \xrightarrow{\bar{a}} P'}{va:n.P \rightarrow va:(n+1).P'} \quad \frac{P \xrightarrow{a} P'}{va:(n+1).P \rightarrow va:n.P'} \\
\frac{}{\alpha.P \xrightarrow{\alpha} P} \quad \frac{P|!P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'} \quad \frac{}{\mathbf{0}; P \rightarrow P} \quad \frac{}{va:n.\mathbf{0} \rightarrow \mathbf{0}} \quad \frac{}{\mathbf{0}|\mathbf{0} \rightarrow \mathbf{0}} \quad \frac{}{\mathbf{0}+\mathbf{0} \rightarrow \mathbf{0}}
\end{array}$$

Figure 3.3: Operational semantics for CCS-Seq.

$$\begin{array}{l}
\alpha ::= \tau \mid a \mid \bar{a} \\
P ::= \mathbf{0} \mid P+P \mid P|P \mid \alpha.P \mid !P \mid P;P \mid va:n.P
\end{array}$$

Metavariables P , Q , M , N , and R refer to processes; a is the name of a semaphore; α is an action (τ is internal), and n is a natural number. Figure 3.3 lists the operational semantics. Action prefixing, $\alpha.P$, emits action α and resolves to P . $\mathbf{0}$ is a terminated process (we abbreviate $\alpha.\mathbf{0}$ as α), $P|Q$ is parallel composition, $P;Q$ is sequential composition, and $P+Q$ represents a choice between executing either P or Q . A process may create infinite, parallel copies of itself by replication: $!P$. Although we do not use replication directly in this thesis, its presence gives the language “teeth” – so that proving termination properties is not trivial (for this purpose, a termination rule for replication is unnecessary).

Restriction, $va:n.P$, declares that a is a semaphore, local to P , with state n . It is a way of introducing a fresh semaphore name that is hidden from any observer or process outside of P . When P emits action \bar{a} or a , the semaphore is incremented or decremented, respectively, and the observed action is τ . If the semaphore count is zero, then P cannot emit a to decrement the semaphore until the count becomes

nonzero. We will reason about programs with an arbitrary number of semaphores, so we define a vectorized form of restriction.

Definition 4. $va_1:n_1 \dots va_k:n_k.P$ is abbreviated as $\Upsilon \vec{a}:\vec{n}.P$.

We define a LTS for CCS-Seq in the usual way. Processes are synonymous with states, an action is either a or \bar{a} for any semaphore a , and the set of single steps defined in Fig. 3.3 is the transition relation.

Bisimulation alone is not a congruence for sequential composition. For example, even though $\mathbf{0} \approx !\tau$, it is the case that $\mathbf{0};a \not\approx !\tau;a$. This is easy to fix by augmenting any simulation relation with *termination sensitivity*.

Definition 5. A relation \mathcal{R} is *one-way termination sensitive* when for any $(p, q) \in \mathcal{R}$, if p is *halted* (for CCS-Seq, if $p = \mathbf{0}$), then $q \Rightarrow p$. \mathcal{R} is *termination sensitive* if \mathcal{R} and \mathcal{R}^{-1} are one-way termination sensitive.


Definition 6 ($p \approx_{\downarrow} q$). States p and q are termination sensitive bisimilar, written $p \approx_{\downarrow} q$, if there exists a termination sensitive bisimulation \mathcal{R} such that $(p, q) \in \mathcal{R}$.

Definition 7 ($p \lesssim_{\downarrow} q$). State q termination sensitive eventually simulates p , written $p \lesssim_{\downarrow} q$ (or $q \gtrsim_{\downarrow} p$), if there exists a termination sensitive simulation \mathcal{R} , whose inverse is an eventual simulation, such that $(p, q) \in \mathcal{R}$.

Lemma 3.7 (Compositional properties of \approx , \approx_{\downarrow} , \lesssim , and \lesssim_{\downarrow}). *Where \equiv ranges over $\{\approx, \approx_{\downarrow}, \lesssim, \lesssim_{\downarrow}\}$; if $P \equiv Q$ then: $P|R \equiv Q|R$, $\alpha.P \equiv \alpha.Q$, $!P \equiv !Q$, $va:n.P \equiv va:n.Q$, $R;P \equiv R;Q$, and $\tau.P + R \equiv \tau.Q + R$. If $P \approx_{\downarrow} Q$, then $P;R \approx_{\downarrow} Q;R$. If $P \lesssim_{\downarrow} Q$, then $P;R \lesssim_{\downarrow} Q;R$.* ✎

Before presenting a general parallelization transformation for sequential composition, we warm up with a simpler form of parallelization in the following lemma. By targeting the sequentialism found in action prefixing, the lemma suggests that eventual simulation may have some uses in plain CCS as well.

Lemma 3.8. $\tau.(P|Q) + \tau.(P|R) \lesssim_{\downarrow} P|(\tau.Q + \tau.R)$.

Proof. We choose $\mathcal{R} = \cup_P(\tau.(P|Q) + \tau.(P|R), P|(\tau.Q + \tau.R)) \cup \mathcal{I}$, where \mathcal{I} is the identity relation. Showing that \mathcal{R} is a simulation and that \mathcal{I} is an eventual simulation and simulation is trivial. We show here that the first part of \mathcal{R} is an eventual simulation. The right program may either 1) choose between Q or R , or 2) avoid choosing and only run P via $P|(\tau.Q + \tau.R) \xrightarrow{\vec{\alpha}} P'|(\tau.Q + \tau.R)$. *Case 1:* the left program can converge to the same state. *Case 2:* we arbitrarily pick Q such that $P'|(\tau.Q + \tau.R) \Rightarrow P'|Q$; the left program can then converge to the same state. 

If we choose $P = 0.0$, $Q = 1.0$, and $R = 2.0$, then sequential program (3.1) roughly corresponds to $\tau.(0|1) + \tau.(0|2)$ and parallel program (3.2) roughly corresponds to $0|(\tau.1 + \tau.2)$. (The “rough” difference is that action 0 is allowed to interleave with actions 1 and 2 in more ways than in Fig. 3.1.)

3.2.1 The parallelization transformation

The key idea of lemma 3.8 is that the more-parallel program may be able to perform some action (by executing P) without making an internal choice (between Q and R). However, the more-sequential program will not be able to simulate this (by running P) before first committing itself to one of those choices. Eventual simulation holds because the more-parallel program can take extra steps to resolve the same choices so that both programs converge to the same state.

This same idea applies to our key result: a general parallelizing transformation between sequential and parallel programs. An obvious schema (despite the subtle premises) for a parallelizing transformation converts two programs in sequence into two programs in parallel, which we describe here. (Section 3.5 goes into further detail and provides proof sketches.)

Proposition 3.1. *If P may always silently terminate (modulo \vec{a}), P and Q do not both decrement any of the same semaphores, and either P or Q never performs an observable action (modulo \vec{a}), then $\Upsilon\vec{a}:\vec{n}.(P;Q) \lesssim_{\downarrow} \Upsilon\vec{a}:\vec{n}.(P|Q)$.*

We specify a list of actions, \vec{a} , to facilitate unobservable communication between P and Q ; when we state that an execution is “silent”, we mean that only hidden actions (i.e. those named by \vec{a}) may be performed. If P “may always silently terminate”, then no matter how P executes (even performing observable actions), we can always ask it to then silently transition to a terminated state. This allows the sequential program, in response to the parallel program executing Q before P terminates, to “catch up” by forcing both to terminate P (possibly making some arbitrary internal choices in doing so) and converging to the same state. (Although this premise is complex, it is more general than simply not allowing P to diverge at all and requiring P to be completely silent.)

However, it is not enough for P to terminate in isolation because Q will interleave with P . The second premise ensures that Q cannot block P by stealing a semaphore and causing it to deadlock.¹ The last premise, where either P or Q must be silent, prevents parallelization from resulting in new interleavings of actions; such a difference would be trivial for an observer to detect.

We also prove a transformation that combines two parallel programs (corresponding to T-SeqParC from chapter 2). Two processes, P_1 and P_2 , *coterminate* (modulo \vec{a}) when (1) $P_1 | P_2 \xrightarrow{\vec{a}} \mathbf{0} | P'_2$ implies P'_2 may always silently terminate (modulo \vec{a}); and when (2) $P_1 | P_2 \xrightarrow{\vec{a}} P'_1 | \mathbf{0}$ implies P'_1 may always silently terminate (modulo \vec{a}).

Proposition 3.2. *If P_1 and P_2 coterminate (modulo \vec{a}), P_1 and P_3 do not decrement any of the same semaphores as P_2 and P_4 , and P_2 and P_4 never perform an observable action (modulo \vec{a}), then $\Upsilon\vec{a}:\vec{n}((P_1 | P_2); (P_3 | P_4)) \lesssim_{\downarrow} \Upsilon\vec{a}:\vec{n}((P_1; P_3) | (P_2; P_4))$.*

¹If we were to extend the language with shared queues of values, then this condition would also prevent Q from interfering by stealing values intended for P , even when it does not cause a deadlock.


Proposition 3.2 is strictly more general than proposition 3.1 because it allows P_1 and P_2 to coordinate termination (or not terminate at all). It results in the parallelization of P_3 with P_2 and P_4 with P_1 . The above definitions, including proofs, are made concrete in Section 3.5.

Now we show that proposition 3.1 is sufficient to parallelize a CCS-Seq implementation of program (3.1) into (3.2). Below, channels c and d are used to make a silent internal choice between 1 and 2, respectively. Once 0 is printed, the internal choice is forwarded via channels e and f to the last statement, which finally prints 1 or 2. (The absence of local variables necessitates the ungainly communication between sequentially composed programs.) Notice that channels c , d , e , and f have different scope restrictions. This is because proposition 3.1 does not allow both threads to be public. It is for this reason that e and f are used in addition to c and d : to forward the choice to the last statement.

Lemma 3.9. *Given*

$$\begin{aligned} M &= \Upsilon[e, f]:[0, 0]. (\Upsilon[c, d]:[0, 0]. (\tau.\bar{c} + \tau.\bar{d}; \bar{0}; c.\bar{e} + d.\bar{f}); e.\bar{1} + f.\bar{2}) \\ N &= \Upsilon[e, f]:[0, 0]. (\Upsilon[c, d]:[0, 0]. ((\tau.\bar{c} + \tau.\bar{d}) | (\bar{0}; c.\bar{e} + d.\bar{f})); e.\bar{1} + f.\bar{2}), \end{aligned}$$

where M and N correspond to (3.1) and (3.2), respectively: $M \lesssim_{\downarrow} N$.

Proof. By proposition 3.1 and lemma 3.7. Because actions \bar{c} and \bar{d} are hidden by the semaphore restriction, $\tau.\bar{c} + \tau.\bar{d}$ silently terminates. Finally, $\tau.\bar{c} + \tau.\bar{d}$ does not decrement any semaphores and thus the process does not interfere with $\bar{0}; c.\bar{e} + d.\bar{f}$. 

3.3 Contrasimulation

Eventual similarity is reflexive, transitive, and compositional, but it is not a congruence because it lacks symmetry. Symmetry is a useful property for the rewriting

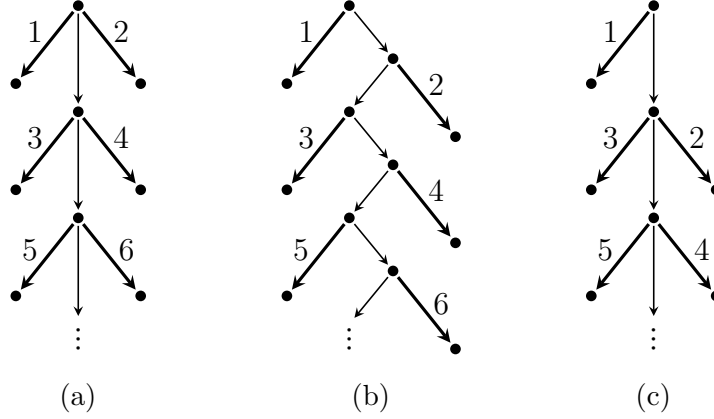


Figure 3.4: Counterexample of transitivity for \approx . Each transition diagram depicts an infinite series of states where the actions continue to increase. Although $a \approx b$ and $b \approx c$, it is *not* the case that $a \approx c$.

framework presented in chapter 2, and enables some useful optimization strategies. For example, commuting two blocks of instructions by first parallelizing them, swapping the threads, and then applying parallelization in reverse (via symmetry).

Although symmetry is not always necessary (e.g. when refining unspecified behavior), there is no clear benefit to the asymmetry in eventual similarity. In fact, it is even asymmetric in the wrong direction – it allows more interleavings to be *added*, not refined. To obtain symmetry, we might attempt to define a relation where eventual simulation holds in both directions.

Definition 8. Programs p and q are eventually bisimilar, written $p \approx q$, if there exists an \mathcal{R} such that \mathcal{R} and \mathcal{R}^{-1} are eventual simulations and $(p, q) \in \mathcal{R}$.

Lemma 3.10. *If $p \preceq q$ or $p \succeq q$, then $p \approx q$.*

Proof. Simulation implies eventual simulation by lemma 3.2. ▣

However, \approx is not transitive for *divergent* LTSs, which have infinite sequences of silent transitions. Figures 3.4a and 3.4b, and Figs. 3.4b and 3.4c are eventually bisimilar. But Figs. 3.4a and 3.4c are not because once Fig. 3.4a takes a silent step, there does not exist an eventual state where Fig. 3.4a and Fig. 3.4c will have the

same observable actions available to them. This limits its use to languages without general loops or recursion. (It is transitive for LTSs that do not diverge.)

Parrow and Sjödin worked on a problem similar to parallelization that also needed a coarser view of internal choice than bisimulation afforded [30]. They developed *coupled simulation* to relate the behavior of multiway distributed internal choice to a reference implementation that resolves all choices in one synchronous step. Coupled simulation is finer than eventual simulation and is also not transitive for divergent LTSs. (Like bisimulation, coupled simulation does not hold between Figs. 3.1a and 3.1b.) Should they need transitivity, they suggested use of *contrasimulation* [13], which contains coupled simulation.

Definition 9 ($p \approx_c q$). \mathcal{R} is a *contrasimulation* when for any $(p, q) \in \mathcal{R}$,


- if $p \xrightarrow{\vec{\alpha}} p'$, then $q \xrightarrow{\vec{\alpha}} q'$ and $(p', q') \in \mathcal{R}^{-1}$ for some q' .

Note the reversal of \mathcal{R} . State q *partially contrasimulates* p , written $p \leq_c q$, iff there exists a contrasimulation \mathcal{R} such that $(p, q) \in \mathcal{R}$. States p and q are *contrasimilar*, written $p \approx_c q$, iff there exists a contrasimulation \mathcal{R} such that $(p, q) \in \mathcal{R} \cap \mathcal{R}^{-1}$.

Lemma 3.11. *If $p \leq_c q$ and $p \geq_c q$, then $p \approx_c q$.*

Proof. Follows directly from definition 9. 


Lemma 3.12. *If $p \approx q$, then $p \approx_c q$.*

Proof. We prove that $\mathcal{R} = \{(p, q) \mid \exists q'. q \Rightarrow q' \wedge p \approx q'\}$ is a contrasimulation and (trivially) show that $p \approx q$ implies $(p, q) \in \mathcal{R}$. To prove contrasimulation, assume $(p, q) \in \mathcal{R}$ for some p and q , and thus $q \Rightarrow q'$ and $p \approx q'$ for some q' . If $p \xrightarrow{\vec{\alpha}} p'$, then by \approx , there exists a p'' and q'' such that $p' \Rightarrow p''$, $q' \xrightarrow{\vec{\alpha}} q''$, and $p'' \approx q''$. Thus $q \xrightarrow{\vec{\alpha}} q''$ and $(p', q'') \in \mathcal{R}^{-1}$. 

When two programs are contrasimilar, they *take turns* simulating each other indefinitely, starting with either program. Unlike bisimilarity, the relation between two

programs only needs to be symmetric for the initial states. Crucially, contrasimulation is an equivalence.


Lemma 3.13. *Contrasimulation is reflexive, symmetric, and transitive.*

Proof. Reflexivity is proved by \mathcal{I} . We prove transitivity via lemma 3.11 by first proving it for \leq_c using $\{(p, r) \mid \exists q. p \leq_c q \wedge q \leq_c r\}$. Symmetry follows directly from the definition of \approx_c . 

As a sanity check, contrasimulation is stronger than *trace equivalence*:

Definition 10 ($p \approx_{tr} q$). p and q are trace equivalent, written $p \approx_{tr} q$,


- if $p \xrightarrow{\bar{\alpha}} p'$, then there exists a q' such that $q \xrightarrow{\bar{\alpha}} q'$; and
- if $q \xrightarrow{\bar{\alpha}} q'$, then there exists a p' such that $p \xrightarrow{\bar{\alpha}} p'$.

Lemma 3.14. *If $p \approx_c q$, then $p \approx_{tr} q$.* 

Trace equivalence can be a sufficient soundness criterion in some situations, but it is still useful to prove a bisimulation relation because trace equivalence is not a congruence for parallel composition and the co-inductive proof method can be easier to work with. Of course, when the LTS is deterministic, these relationships are all equivalent to bisimulation. But we can prove that contrasimulation is equivalent to bisimulation when all silent transitions are deterministic. This suggests that it is the finest behavioral equivalence that contains parallelization.


Theorem 3.1. *If $p \rightarrow p'$ implies $p \approx p'$ for any p and p' , then \approx_c is equivalent to \approx .*

Proof. Lemmas 3.3, 3.10 and 3.12 prove $\approx \subseteq \approx_c$. In the other direction, we show that $\mathcal{R} = \approx_c$ itself is a bisimulation. Because it is symmetric, we need only prove that \mathcal{R} is a simulation. Assume $p \approx_c q$ and $p \xrightarrow{\alpha} p'$; there must exist a q' such that $q \xrightarrow{\alpha} q'$ and $q' \leq_c p'$. The trick is to flip the direction in which partial contrasimulation holds between p' and q' , which will imply $p' \approx_c q'$. By $q' \Rightarrow q'$, there exists a p'' such that

$p' \Rightarrow p''$ and $p'' \leq_c q'$. By the premise, $p'' \approx p'$, and by lemmas 3.3, 3.10, 3.12 and 3.13, $p' \leq_c q'$. Thus $p' \approx_c q'$. 

As a corollary, \lesssim also collapses to \approx when there is no internal choice. We define termination sensitive contrasimulation and then show that contrasimilarity has the same compositional properties as bisimilarity and eventual similarity.


Definition 11 ($p \approx_{\downarrow c} q$). States p and q are termination sensitive contrasimilar, written $p \approx_{\downarrow c} q$, iff there exists a one-way termination sensitive contrasimulation \mathcal{R} such that $(p, q) \in \mathcal{R} \cap \mathcal{R}^{-1}$.

Lemma 3.15 (Compositional properties of \approx_c and $\approx_{\downarrow c}$). *Where \equiv ranges over $\{\approx_c, \approx_{\downarrow c}\}$; if $P \equiv Q$ then: $P|R \equiv Q|R$, $\alpha.P \equiv \alpha.Q$, $!P \equiv !Q$, $va:n.P \equiv va:n.Q$, $R;P \equiv R;Q$, and $\tau.P + R \equiv \tau.Q + R$. If $P \approx_{\downarrow c} Q$, then $P;R \approx_{\downarrow c} Q;R$.* 

Like coupled similarity, contrasimilarity is congruent for $+$ when the processes are equally *stable*.

Definition 12 (stable p). p is stable if there does not exist a p' such that $p \rightarrow p'$.

Lemma 3.16. *If $P \approx_c Q$ and (stable P iff stable Q), then $P + R \approx_c Q + R$.*

(And likewise for $\approx_{\downarrow c}$). 

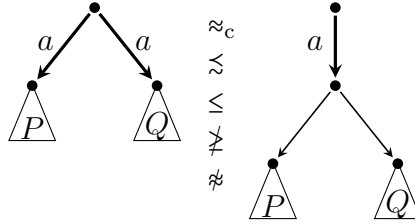
Compiled languages do not usually allow “mixed choice”, where one option is stable and the other is not, so both bisimulation and contrasimulation are often full congruences in practice. Thus we can build correct, *modular* optimizations based on contrasimulation (or bisimulation) using the above congruence results. Because bisimulation is finer than contrasimulation, all of its algebraic properties for CCS (and CCS-Seq) hold for contrasimulation.

Voorhoeve and Mauw investigate further properties of contrasimulation and describe an axiomatization for CCS [44]. Their axiomatization splits a stable internal choice into the action followed by a silent internal choice.

Lemma 3.17. $a.P + a.Q \approx_c a.(\tau.P + \tau.Q)$. ✎

Interestingly, this holds for \lesssim as well.

Lemma 3.18. $a.P + a.Q \lesssim a.(\tau.P + \tau.Q)$. ✎



Combined with a few algebraic properties of bisimilarity, like $\tau.P + P \approx \tau.P$, lemma 3.17 proves equivalence between programs (3.1) and (3.2).

A drawback in the definition of contrasimulation (definition 9) is that we must consider any number of steps of p for every point in the relation instead of just a single step of p at a time (like most other bisimulation-style relations). This is inconvenient because it requires that we perform induction on the number of steps taken, or alternately, to find a multistep inversion principle for the basic structures of our language.

To prove congruence for **while** loops (from chapter 2), we found it difficult to even *state* the multistep inversion principle because more than one iteration could execute. Then we realized that although it is necessary to match at least one step of p , we do not need to match all of them. Thus we came up with an equivalent definition of contrasimulation that affords us some flexibility to choose how many steps to match. (For **while** loops, we match no more than one iteration at a time.) We state the next lemma with respect to *counted executions*: when $p \xrightarrow{\vec{a}} p'$ takes no more than n steps, we write $p \xrightarrow{\vec{a}}_n p'$.

Lemma 3.19. \mathcal{R} is a contrasimulation iff for any $(p, q) \in \mathcal{R}$, if $p \xrightarrow{\vec{a}}_n p''$, then either

- $q \xrightarrow{\vec{a}} q''$ and $(p'', q'') \in \mathcal{R}^{-1}$ for some q'' , or

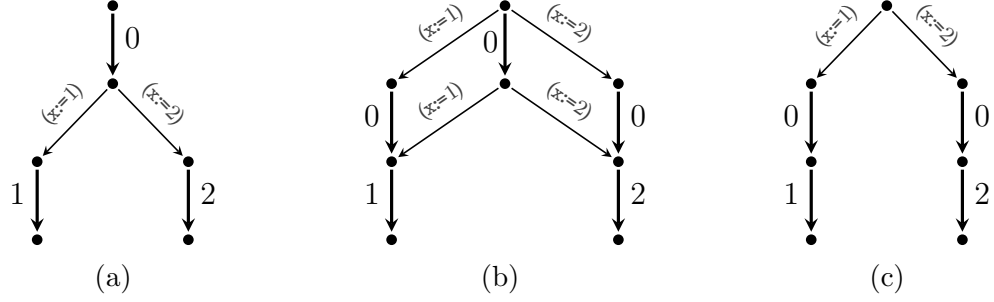


Figure 3.5: Subfigure a is a transition diagram for program (3.3). For comparison, we reiterate the semantics of (3.2) and (3.1) in subfigures b and c. Contrasimilarity holds between b and c, but does not hold between a and b (or c).

- $p \xrightarrow{\vec{a}_1}_{n_1} p' \xrightarrow{\vec{a}_2}_{n_2} p'', q \xrightarrow{\vec{a}_1} q', n_1 > 0, n = n_1 + n_2, \vec{a} = \vec{a}_1 \cdot \vec{a}_2,$ and $(p', q') \in \mathcal{R}$ for some $n_1, n_2, a_1, a_2, p',$ and q' .

Note that the direction of \mathcal{R} only reverses in the first case (normal contrasimulation).

Proof. Contrasimulation trivially implies the above. The other direction follows by induction on n . ◻

3.4 Delayed observations

Contrasimulation effectively allows a program to delay an internal choice until after an observable action. In other words, a program that makes a choice before an observable action is equivalent to a program that can do them in any order. But this does not allow the observation to be fully commuted.

Consider a sequential program that prints 0 *before* choosing a value for x ,

$$\text{print } 0; x := \text{either } 1 \text{ or } 2; \text{print } x, \tag{3.3}$$

the semantics of which are presented in Fig. 3.5a. A parallelization of it,

$$(\text{print } 0 \parallel x := \text{either } 1 \text{ or } 2); \text{print } x,$$

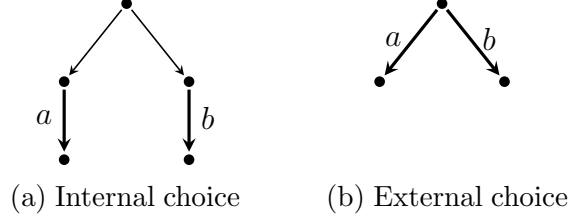


Figure 3.6

is equivalent to (3.2); we reiterate its semantics in Fig. 3.5b. For comparison, we also repeat the semantics of (3.1) in Fig. 3.5c.

All three programs are intuitively (albeit, arguably) equivalent. But Fig. 3.5a is clearly not contrasimilar to Fig. 3.5b or (3.5c): if the latter programs choose to follow the $x := 1$ branch, then Fig. 3.5a will be unable to commit to the same choice without first observing 0.

We can relate such programs by modifying definition 3 so that extra observable actions may be performed when “eventually” simulating each other:

Definition 13. \mathcal{R} is an *amortized eventual simulation* when for any $(p, q) \in \mathcal{R}$,

- if $p \xrightarrow{\vec{a}} p'$, then $p' \xrightarrow{\vec{b}} p''$, $q \xrightarrow{\vec{a}\vec{b}} q''$, and $(p'', q'') \in \mathcal{R}$ for some p'' , q'' , and \vec{b} .

If we combine this with a simulation relation a la eventual similarity, the resulting relation is not symmetric. Worse, it would not hold between Figs. 3.5a and 3.5c – commuting choice around observation would not be admissible. We can obtain symmetry a la eventual bisimilarity (definition 8). We can also modify contrasimulation to keep track of a “deficit” of actions between programs:

Definition 14. \mathcal{R} is an *amortized contrasimulation* when for any $(p, \vec{a}, q) \in \mathcal{R}$,

- if $p \xrightarrow{\vec{b}} p'$, then $q \xrightarrow{\vec{a}\vec{b}\vec{c}} q'$ and $(p', \vec{c}, q') \in \mathcal{R}^{-1}$ for some q' and \vec{c} .

Programs p and q are *amortized contrasimilar* if there exists an amortized contrasimulation \mathcal{R} such that $(p, [], q) \in \mathcal{R} \cap \mathcal{R}^{-1}$.

Although these three relations trivially imply trace equivalence, they cannot discriminate between internal and external choice because they hold between Figs. 3.6a

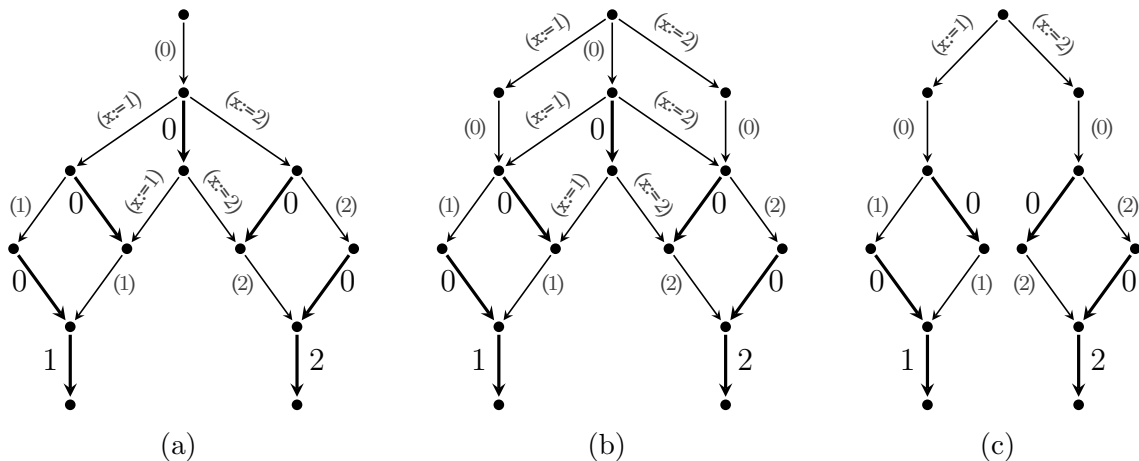


Figure 3.7: Transition diagrams for programs (3.3), (3.2), and (3.1) for a semantics with delayed observations. Figures b and c are contrasimilar, while Figs. a and b are bisimilar.

and 3.6b. Thus they are not appropriate for studying nondeterministic reactive systems. (They are akin to saying that two vending machines are equivalent when one lets you choose between coffee and tea, while the other machine arbitrarily chooses for you.) We have yet to find a satisfactory equivalence that holds for Fig. 3.5a.

However, if (3.3), (3.1), and (3.2) were C programs, their semantics would be subtly different. Many programmers often first realize this only after attempting find the source of a bug using `printf`: most IO operations are buffered. In other words, C allows observable actions to be delayed. For example, if `f(); printf("0"); g()` crashes but 0 is not printed to the console, it is still possible that `printf("0")` was executed and that `g()` caused the crash before the buffer `printf` could be flushed to the console.

We say that a LTS has *delayed observations* when output is queued before appearing on the screen at a nondeterministic point in the future. Figure 3.7 gives the semantics of programs (3.3), (3.2), and (3.1) using delayed observations. Figures 3.7b and 3.7c are still contrasimilar. Moreover, Figs. 3.7a and 3.7b are now *bisimilar*.

Although contrasimulation cannot directly allow observations to be delayed until after internal choice, we can side-step the issue by choosing a semantics with delayed

observations. In such a setting, propositions 3.1 and 3.2 can be used to parallelize programs such as (3.3). They also become somewhat easier to use: by delaying all observations until after termination, proving termination is enough to prove *silent* termination. However, a limitation remains: not all observations may be delayed. For example, C's `fflush` forces immediate observation, thus commuting it with internal choice would not maintain a contrasimulation.

3.5 Proof of parallelization

We first define convergence, termination entailment, cotermination, free variables, and some helper lemmas before describing the proofs of propositions 3.1 and 3.2.

3.5.1 Preliminary definitions

Definition 15. $\vec{\alpha} - \vec{a}$ is the trace of labels $\vec{\alpha}$ with the semaphores in \vec{a} removed. It represents the result of multiple semaphore restrictions on a trace.

Definition 16 (Well-formed traces). A trace of labels $\vec{\alpha}$ is well-formed with respect to semaphore a with count n if there exists a final count n' for the semaphore such that the trace does not decrement the semaphore below a count of 0. We denote this as $n \rightsquigarrow_{\vec{\alpha}} a n'$ and define it recursively on the structure of $\vec{\alpha}$.

$$\begin{array}{ll}
 n \overset{\square}{\rightsquigarrow}_a n' & \text{if } n' = n \\
 n \overset{\alpha': \vec{\alpha}}{\rightsquigarrow}_a n' & \text{if } \alpha' \notin \{a, \bar{a}\} \text{ and } n \rightsquigarrow_{\vec{\alpha}} a n' \\
 n + 1 \overset{a: \vec{\alpha}}{\rightsquigarrow}_a n' & \text{if } n \rightsquigarrow_{\vec{\alpha}} a n' \quad (\text{decrements } a) \\
 n \overset{\bar{a}: \vec{\alpha}}{\rightsquigarrow}_a n' & \text{if } n + 1 \rightsquigarrow_{\vec{\alpha}} a n'. \quad (\text{increments } a)
 \end{array}$$

We then define a well-formed trace with respect to a list of semaphores, $\vec{n} \rightsquigarrow_{\vec{a}} \vec{n}'$, recursively on the structure of \vec{a} .

$$(n :: \vec{m}) \xrightarrow[\vec{a}::\vec{b}]{\vec{\alpha}} (n' :: \vec{m}') \quad \text{always} \\ \text{if } n \xrightarrow[\vec{a}]{\vec{\alpha}-\vec{b}} n' \text{ and } \vec{m} \xrightarrow[\vec{b}]{\vec{\alpha}} \vec{m}'.$$

Definition 16 appears only in the next definition. However, it is used extensively by helper lemmas in our Coq proof development to separate the details of how a particular process runs from how its semaphores are used. For example, to state that a sequential and parallelized program use their semaphores in the same way despite their syntactic difference.

Silent termination and cotermination were introduced in Section 3.2.1 for use in propositions 3.1 and 3.2. We now give concrete definitions; recall the notation for vectorized semaphore restriction from definition 4.

Definition 17 (Silent termination). P silently terminates, written $P \Downarrow^{\vec{a}:\vec{n}}$, if for any P' and $\vec{\alpha}$, $P \xrightarrow{\vec{\alpha}} P'$ implies $\Upsilon \vec{a}:\vec{n}.P' \Rightarrow \mathbf{0}$ and $\vec{n} \xrightarrow[\vec{a}]{\vec{\alpha}} \vec{n}'$ for some \vec{n}' .

Definition 18 (Termination entailment & cotermination). P_1 entails the termination of P_2 , written $P_1 \Downarrow^{\vec{a}:\vec{n}} P_2$, if $\Upsilon \vec{a}:\vec{n}.(P_1 | P_2) \xrightarrow{\vec{\alpha}} \Upsilon \vec{a}':\vec{n}'.(\mathbf{0} | P_2)$ implies $P_2' \Downarrow^{\vec{a}':\vec{n}'}$. P_1 and P_2 coterminate, written $P_1 \Updownarrow^{\vec{a}:\vec{n}} P_2$, iff $P_1 \Downarrow^{\vec{a}:\vec{n}} P_2$ and $P_2 \Downarrow^{\vec{a}:\vec{n}} P_1$.

In order to state noninterference properties between processes, we define functions to find the sets of free variables used to increment semaphores, decrement semaphores, and the union of each within a process.

Definition 19 (Free observable actions).

$$\begin{aligned} \text{fa}(\bar{a}.P) &= \{\bar{a}\} \cup \text{fa}(P) \\ \text{fa}(a.P) &= \{a\} \cup \text{fa}(P) \\ \text{fa}(\tau.P) &= \text{fa}(P) \\ \text{fa}(\mathbf{0}) &= \{\} \\ \text{fa}(!P) &= \text{fa}(P) \\ \text{fa}(va:n.P) &= \text{fa}(P) \setminus a \setminus \bar{a} \end{aligned}$$

$$\text{fa}(P_1 + P_2) = \text{fa}(P_1) \cup \text{fa}(P_2)$$

$$\text{fa}(P_1 | P_2) = \text{fa}(P_1) \cup \text{fa}(P_2)$$

$$\text{fa}(P_1; P_2) = \text{fa}(P_1) \cup \text{fa}(P_2)$$

Definition 20 (Free variables: increment, decrement, and both).

$$\text{fv}_V(P) = \{a \mid \bar{a} \in \text{fa}(P)\}$$

$$\text{fv}_P(P) = \{a \mid a \in \text{fa}(P)\}$$

$$\text{fv}(P) = \text{fv}_V(P) \cup \text{fv}_P(P)$$


The semaphores that a process, P , can increment and decrement are respectively limited by $\text{fv}_V(P)$ and $\text{fv}_P(P)$. $\text{fv}(P)$ is the set of semaphores that P can decrement or increment. We use this, for example, to show that processes P and Q cannot decrement the same semaphores by ensuring $\text{fv}_P(P) \cap \text{fv}_P(Q) = \emptyset$.

3.5.2 Proof

This first lemma performs case analysis on a single step of a “sequential” program in order to show that the parallelized program can perform the same action.

Lemma 3.20. *If $\Upsilon \vec{a} : \vec{n}.((P_1 | P_2); (P_3 | P_4)) \xrightarrow{\alpha} p'$, then either*

- *there exists \vec{n}' , P'_1 , and P'_2 such that*
 - $p' = \Upsilon \vec{a} : \vec{n}.((P'_1 | P'_2); (P_3 | P_4))$ and
 - $\Upsilon \vec{a} : \vec{n}.(P_1 | P_2) \xrightarrow{\alpha} \Upsilon \vec{a} : \vec{n}'.(P'_1 | P'_2)$
 - (and thus $\Upsilon \vec{a} : \vec{n}.((P_1; P_3) | (P_2; P_4)) \xrightarrow{\alpha} \Upsilon \vec{a} : \vec{n}'.((P'_1; P_3) | (P'_2; P_4))$),
- *or $P_1 = P_2 = \mathbf{0}$ and $p' = \Upsilon \vec{a} : \vec{n}.\mathbf{0}; (P_3 | P_4)$.*

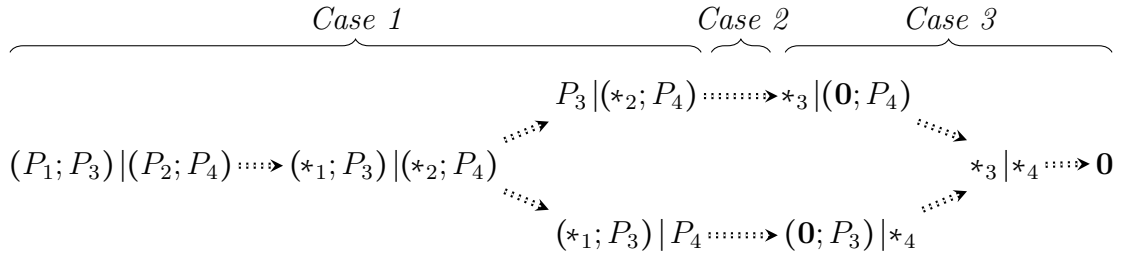
Proof. By case analysis on $\Upsilon \vec{a} : \vec{n}.((P_1 | P_2); (P_3 | P_4)) \xrightarrow{\alpha} p'$. 

In the following lemma, we look at an execution of the parallelized program over multiple steps and show that the sequential program can either simulate it directly, or that there exists a future state where they can converge.

Lemma 3.21. *If $\Upsilon \vec{a} : \vec{n}.((P_1; P_3)|(P_2; P_4)) \xrightarrow{\vec{\alpha}} p'$, $P_1 \updownarrow^{\vec{a}; \vec{n}} P_2$, $\text{fv}(P_2; P_4) \subseteq \vec{a}$, and $\text{fv}_{\mathbb{P}}(P_1; P_3) \cap \text{fv}_{\mathbb{P}}(P_2; P_4) = \emptyset$, then either*

- *there exists \vec{n}' , P'_1 , and P'_2 such that*
 - $p' = \Upsilon \vec{a} : \vec{n}'.((P'_1; P_3)|(P'_2; P_4))$,
 - $\Upsilon \vec{a} : \vec{n}.P_1 | P_2 \xrightarrow{\vec{\alpha}} \Upsilon \vec{a} : \vec{n}.(P'_1 | P'_2)$
 - (and thus $\Upsilon \vec{a} : \vec{n}.((P_1 | P_2); (P_3 | P_4)) \xrightarrow{\vec{\alpha}} \Upsilon \vec{a} : \vec{n}.((P'_1 | P'_2); (P_3 | P_4))$);
- *or there exists p'' such that*
 - $p' \Rightarrow p''$ and
 - $\Upsilon \vec{a} : \vec{n}.((P_1 | P_2); (P_3 | P_4)) \xrightarrow{\vec{\alpha}} p''$.


Proof. We consider the three outcomes of running $\Upsilon \vec{a} : \vec{n}.((P_1; P_3)|(P_2; P_4)) \xrightarrow{\vec{\alpha}} p'$. In the illustration below, $*_n$ represents the final state that process P_n reaches (if it takes at least one step but does not terminate). We focus on the second case as the other two are relatively easy.



Case 1. $P_1 | P_2 \xrightarrow{\vec{\alpha}_1} p_1 | p_2$ and $\vec{\alpha} = \vec{\alpha}_1 - \vec{a}$ for some p_1 and p_2 . (Neither P_3 nor P_4 execute.) The sequential program runs $P_1 | P_2$ to match the actions without converging to the same state.

Case 2. The parallel program has the form of either $p_3 | (p_2; P_4)$ or $(p_1; P_3) | p_4$. We consider only the first (top) form; the second is similar. Because P_1 terminated, p_2 can silently terminate. This yields $\Upsilon \vec{a} : \vec{n}.(p_3 | (p_2; P_4)) \Rightarrow \Upsilon \vec{a} : \vec{n}'.(p_3 | P_4)$ for some

\vec{n}' . However, we need P_2 to terminate *before* P_3 even runs in order for the sequential program to mimic the behavior. This is possible if P_2 did not emit observable actions (a premise of this lemma) and P_3 did not influence P_2 as they interleaved. The last could only have happened if P_3 incremented a semaphore on which P_2 would otherwise deadlock. Because P_2 was capable of terminating by the time P_3 ran, such deadlocking was impossible. Thus we know we can run $\Upsilon\vec{a}:\vec{n}.(P_1|P_2) \xrightarrow{\vec{\alpha}_{12}} \Upsilon\vec{a}:\vec{n}'^0.(\mathbf{0}|\mathbf{0})$, followed by $\Upsilon\vec{a}:\vec{n}'^0.(P_3|P_4) \xrightarrow{\vec{\alpha}_3} \Upsilon\vec{a}:\vec{n}'.(p_3|P_4)$, for some \vec{n}'^0 and such that $\vec{\alpha}$ is equal to some $\vec{\alpha}_{12}$ appended with $\vec{\alpha}_3$. Both sequential and parallel programs can converge to state $p'' = \Upsilon\vec{a}:\vec{n}'.(p_3|P_4)$.

Case 3. $P_1|P_2 \xrightarrow{\vec{\alpha}_1} \mathbf{0}|\mathbf{0}$, $P_3|P_4 \xrightarrow{\vec{\alpha}_2} p_3|p_4$, and $\vec{\alpha} = (\vec{\alpha}_1 \cdot \vec{\alpha}_2) - \vec{a}$ for some $p_3, p_4, \vec{\alpha}_1$, and $\vec{\alpha}_2$. The sequential program runs $P_1|P_2$ to termination and then runs $P_3|P_4$ to converge to the same state as the parallel program. 

Theorem 3.2 (Proof of proposition 3.2). *If*

- P_1 and P_2 coterminate: $P_1 \updownarrow^{\vec{a}:\vec{n}} P_2$;
- the processes do not interfere: $\text{fv}_P(P_1; P_3) \cap \text{fv}_P(P_2; P_4) = \emptyset$; and
- P_2 and P_4 cannot be observed: $\text{fv}(P_2) \cup \text{fv}(P_4) \subseteq \vec{a}$,

then

$$\Upsilon\vec{a}:\vec{n}.\left(\begin{array}{c} (P_1|P_2); \\ (P_3|P_4) \end{array}\right) \lesssim_{\downarrow} \Upsilon\vec{a}:\vec{n}.\left(\left(\begin{array}{c} P_1; \\ P_3 \end{array}\right) \middle| \left(\begin{array}{c} P_2; \\ P_4 \end{array}\right)\right).$$


Proof. We choose

$$\begin{aligned} \mathcal{R} = & \{(p, q) \mid p \approx_{\downarrow} q\} \cup \{(p, q) \mid \exists \vec{a}, \vec{n}, P_1, P_2. \\ & \wedge p = \Upsilon\vec{a}:\vec{n}.\left(\left(\begin{array}{c} P_1; \\ P_3 \end{array}\right) \middle| \left(\begin{array}{c} P_2; \\ P_4 \end{array}\right)\right) \\ & \wedge q = \Upsilon\vec{a}:\vec{n}.\left(\left(\begin{array}{c} P_1|P_2; \\ P_3|P_4 \end{array}\right)\right) \\ & \wedge P_1 \updownarrow^{\vec{a}:\vec{n}} P_2 \end{aligned}$$

$$\begin{aligned} & \wedge \text{fv}(P_2) \cup \text{fv}(P_4) \subseteq \vec{a} \\ & \wedge \text{fv}_P(P_1; P_3) \cap \text{fv}_P(P_2; P_4) = \emptyset \}, \end{aligned}$$

and show that \mathcal{R} is a termination sensitive simulation, \mathcal{R}^{-1} is an eventual simulation, and that, trivially, $(\Upsilon \vec{a} : \vec{n}.((P_1 | P_2); (P_3 | P_4)), \Upsilon \vec{a} : \vec{n}.((P_1; P_3) | (P_2; P_4))) \in \mathcal{R}$. The remaining step is to consider all pairs $(p, q) \in \mathcal{R}$ and show that they behave accordingly for termination sensitivity, simulation and eventual simulation. We look at both cases of $(p, q) \in \mathcal{R}$:


Case 1: $p \approx_{\downarrow} q$. Termination sensitivity holds because \approx_{\downarrow} is termination sensitive. Simulation and (inverse) eventual simulation hold because \approx_{\downarrow} implies both.

Case 2. There exists \vec{a} , \vec{n} , P_1 , and P_2 such that $p = \Upsilon \vec{a} : \vec{n}.((P_1; P_3) | (P_2; P_4))$, $q = \Upsilon \vec{a} : \vec{n}.((P_1 | P_2); (P_3 | P_4))$, etc. Termination sensitivity holds because neither p nor q are halted. To satisfy simulation, we assume that $p \xrightarrow{\alpha} p'$ and must show that there exists a matching q' such that $q \xRightarrow{\alpha} q'$ and $(p', q') \in \mathcal{R}$. This follows directly from lemma 3.20. Finally, we must satisfy eventual simulation. Assuming $q \xRightarrow{\vec{\alpha}} q'$, we show that there exists a p'' and q'' such that $p \xRightarrow{\alpha} p''$, $q' \Rightarrow q''$, and $(p'', q'') \in \mathcal{R}$. (Notice that this flips the direction of eventual simulation because we started with it holding for \mathcal{R}^{-1} .) This follows from lemma 3.21. 

Corollary 3.1 (Proof of proposition 3.1). *If*

- P_1 silently converges: $P_1 \Downarrow^{\vec{a}; \vec{n}}$;
- the processes do not interfere: $\text{fv}_P(P_1) \cap \text{fv}_P(P_2) = \emptyset$; and
- either P_1 or P_2 cannot be observed: $\text{fv}(P_1) \subseteq \vec{a}$ or $\text{fv}(P_2) \subseteq \vec{a}$,

then $\Upsilon \vec{a} : \vec{n}.(P_1; P_2) \lesssim_{\downarrow} \Upsilon \vec{a} : \vec{n}.(P_1 | P_2)$.

Proof. This reduces to proving either $\Upsilon \vec{a} : \vec{n}.((P_1 | \mathbf{0}); (\mathbf{0} | P_2)) \lesssim_{\downarrow} \Upsilon \vec{a} : \vec{n}.((P_1; \mathbf{0}) | (\mathbf{0}; P_2))$ or $\Upsilon \vec{a} : \vec{n}.((\mathbf{0} | P_1); (P_2 | \mathbf{0})) \lesssim_{\downarrow} \Upsilon \vec{a} : \vec{n}.((\mathbf{0}; P_1) | (P_2; \mathbf{0}))$ by theorem 3.2, depending on whether P_2 or P_1 is unobservable. 

3.6 Conclusion

We have proved the soundness of a very general parallelizing transformation for CCS-Seq with respect to a new type of simulation relation, called eventual similarity, that allows internal choice to be preserved. Additionally, we identify contrasimilarity as a congruence that contains eventual similarity when symmetry is needed. In the event that an input program does not have internal choice, both eventual similarity and contrasimilarity reduce to bisimulation.

Furthermore, we have identified a potential limitation of using a model based on CCS to study parallelization, where it is standard for external actions to be immediately observable. In such a setting, we were unable to find a bisimulation-style congruence that allows an observable action followed by internal choice to be parallelized or commuted. Our solution is to augment our labeled transition systems with delayed observations (buffered IO) so that eventual similarity and contrasimilarity hold. Buffered IO is primarily used to increase the performance of programs, and it is often ignored when modeling concurrency. A surprising result was that its benefits extend to justifying the correctness of program transformations by allowing us to side-step the issue altogether.

3.6.1 Coq proof development

Everything in this chapter has been mechanically checked by the Coq Proof Assistant. We use the resulting proof development as the basis of bisimulation and contrasimulation in the next chapter, where we apply these results to a simple imperative language and prove the soundness of loop folding and much of the the framework from chapter 2.

Chapter 4

A Semantic Framework

In chapter 2, we introduced the imperative language and framework of rewrite rules that we use to prove the soundness of loop-parallelizing optimizations in this thesis. Then we formalized a small model language in order to study and prove the correctness of parallelization in chapter 3. Our final step is to pin down the semantics of the imperative language and framework presented in chapter 2 using the formal techniques and results of chapter 3, and then prove loop folding in this new setting.

Parallelization. The lessons from chapter 3 that we will use in this chapter:

- **Contrasimulation** – at the top level, we will define \equiv as a variation of contrasimulation in order to support the framework of congruent rewrite rules that support parallelization.
- **Delayed observations** – allow more flexibility in the types of parallelizing and commuting optimizations that contrasimulation admits.

Loop Folding. We first introduced our loop folding transformation, T-FoldLoop, in Section 2.3.4. Viewing a `while` loop as a stream of iterations, T-FoldLoop uses a user-supplied combining transformation that we use to co-inductively fold all of the iterations together. A combination of iterations that have been folded together

is described by a loop schema, which may take the form of a single `while` loop or more complicated structures, such as multiple loops composed in parallel. The loop schema is parameterized by a bound on the maximum number of iterations that it may perform, which may be finite, infinite, or an iteration barrier.

A loop schema is defined by (f, e, P) , where e is the loop condition, P is the loop invariant, and f is a program with holes that are filled by a maximum bound on the number of iterations it may execute. A combination of z iterations is written as $f[z]$ (where the holes in f are filled by z). The end result of T-FoldLoop is to show $\{P\}; \text{while } e \text{ do } f[1] \equiv \{P\}; f[\infty]$.

Overview. We first present the syntax of program terms, program state, and the operational semantics for the imperative language. Then we give a semantic definition for algebraic congruence used in chapter 2, \equiv , based on bisimilarity and contrasimilarity. We state the soundness theorem for several basic algebraic and congruence rewrite rules in Section 4.3. In Section 4.4, we motivate our use of iteration barriers and prove the soundness of T-FoldLoop, and then in Section 4.5, we state the theorem of soundness for T-SeqParC. At the end of this chapter, we also show how Hoare triples relate to program equivalence and can be used to prove that assertions may be inserted into the program while preserving equivalence.

4.1 Technical development

4.1.1 Machine state

The machine state is a tuple composed of a global state and the set of threads and their respective local states. As described in Fig. 4.1, C denotes the global state, a partial mapping from channels to a list of values; it represents the current queue of values that have been sent into each channel and not yet received. A thread is

t	<i>program term</i>
$V = \{\text{true}, \text{false}\} \cup \mathbb{Z} \cup \dots$	<i>values include booleans, integers, etc.</i>
$v \in V$	<i>value</i>
$a \in A$	<i>channel name</i>
$x \in X$	<i>variable name</i>
$\sigma \in \text{list } V$	<i>list of values</i>
$\Pi = \{\perp, \top\}$	<i>permissions</i>
$\pi \in \Pi$	<i>a permission</i>
$C \subseteq A \rightarrow \text{list } V$	<i>channels state</i>
$S = A \rightarrow \Pi$	<i>sending-endpoint permissions</i>
$R = A \rightarrow \Pi$	<i>receiving-endpoint permissions</i>
$E = X \rightarrow (\Pi \times V)$	<i>environment</i>
$w \in S \times R \times E$	<i>local state</i>
$w.S \in S$	<i>send endpoint of w</i>
$w.R \in R$	<i>receive endpoint of w</i>
$w.E \in E$	<i>environment of w</i>

Figure 4.1: Machine state metavariables

$$b ::= w : t \mid b; t \mid b_1 \parallel b_2$$

Figure 4.2: Syntax of the state of threads

a pairing of a local state, w , with a program term, t . The state of threads, b , may include one or multiple threads in parallel or composed in sequence with program terms; its syntax is given in Fig. 4.2.

When a value is sent or received from a channel, its queue must be updated. We write $C[a \mapsto \sigma]$ to map channel a to queue σ in global state C .

Definition 21 (Channel update).

$$C[a \mapsto \sigma] = \lambda a'. \begin{cases} \sigma & a' = a \\ C(a') & \text{otherwise.} \end{cases}$$

The local state of a thread contains an environment that maps variable names to values. In addition, it maps channel and variable names to *permissions*, which are used to track the resources to which each thread claims access: the ability to

send values over a channel, to receive from a channel, and to read/write a variable. A permission of \perp denotes no access to a resource; a permission of \top denotes full access.¹ Given a local state, w , the permissions to send to channels is denoted by $w.S$; the permissions to receive from channels is denoted by $w.R$; and the mapping from variable names to permissions and values is denoted by $w.E$.

The contents of a channel is only visible to threads that fully own both producing and consuming endpoints of the channel. To prevent a thread with less than full ownership of a channel from referring to its contents, we apply a filter on channels before testing whether a predicate holds.

Definition 22 (Channel filter).

$$C\#w = (C', w), \quad \text{where} \quad C' = \lambda a. \begin{cases} C(a) & w.S(a) = w.R(a) = \top \\ \text{undefined} & \text{otherwise} \end{cases}$$

To map variable x to value v in world w , we write $w[x \mapsto v]$.

Definition 23 (Updating a local variable).

$$w[x \mapsto v] = (w.S, w.R, E'), \quad \text{where} \quad E' = \lambda x'. \begin{cases} (\pi, v) & x' = x \text{ and } w.E(x) = (\pi, -) \\ w.E(x') & \text{otherwise.} \end{cases}$$

To remove variable x from world w , we write $w[x \mapsto \cdot]$.

Definition 24 (Unmapping a local variable).

$$w[x \mapsto \cdot] = (w.S, w.R, w.E - x).$$

We use a separation logic (SL) [35] to enable local reasoning for program transformations; in particular, so that we may state disjoint usage of resources between

¹It is straightforward to generalize this to fractional permissions [5], which we have done in our Coq development a la [10].

$$\begin{array}{c}
\overline{\pi \oplus \perp = \pi} \quad \overline{\perp \oplus \pi = \pi} \quad \overline{(\pi, v_1) \oplus (\perp, v_2) = (\pi, v_1)} \quad \overline{(\perp, v_1) \oplus (\pi, v_2) = (\pi, v_2)} \\
\forall a. w_1.s(a) \oplus w_2.s(a) = w.s(a) \\
\forall a. w_1.R(a) \oplus w_2.R(a) = w.R(a) \\
\forall x. w_1.E(x) \oplus w_2.E(x) = w.E(x) \\
\hline
w_1 \oplus w_2 = w
\end{array}$$

Figure 4.3: Separation algebra; disjoint resources

threads. To provide a semantic model for the separation logic, we use a *separation algebra* (SA) [8] over permissions and local states.

We define a SA using a partial binary *join* operator, \oplus , which forms a cancellative, partial commutative monoid. The join operator forms a SA over permissions to send and receive endpoints, and permissions to access local variables (in the variables-as-resources style [6]), so that two local state join only if their send/receive endpoints join and they do not both claim ownership on the same variable.

Given an x and y , the existence of a z such that $x \oplus y = z$ is not guaranteed. For example, $(\top, 6) \oplus (\top, 7)$ is undefined because two threads cannot both claim access to the same resource.

Figure 4.3 depicts the SA for permissions and local states. The SA for permissions is specified by two rules such that $\perp \oplus \pi = \pi$ and $\pi \oplus \perp = \pi$ are defined for any permission π . Then we lift this to form a SA on pairs of permissions and values; the join is defined when the permissions join and when the result takes on the value associated with a permission of \top . In other words, if a pair has no permission (\perp), then the value associated with it is ultimately ignored. Finally, a SA on local states is defined when, for each channel, the permissions to send and receive join, and for each variable, the permissions to access the variable join.

$$\begin{array}{ll}
\llbracket v \rrbracket_w = v & \\
\llbracket x \rrbracket_w = v & \text{if } w.\mathbb{E}(x) = (\pi, v) \text{ and } \pi \neq \perp \\
\llbracket f(e) \rrbracket_w = f(v) & \text{if } \llbracket e \rrbracket_w = v \\
\llbracket e_1 \odot e_2 \rrbracket_w = v_1 \odot v_2 & \text{if } \llbracket e_1 \rrbracket_w = v_1 \text{ and } \llbracket e_2 \rrbracket_w = v_2
\end{array}$$

Figure 4.4: Expression semantics; \odot ranges over operators such as $+$, $-$, $<$, $=$, etc.

4.1.2 Expressions

Expressions are partial functions over a local environment. An expression may be composed of constant values, variables names, simple functions, and operators:

$$e ::= v \mid x \mid e_1 + e_2 \mid e_1 = e_2 \mid e_1 < e_2 \mid \dots,$$

but may not refer to channels. The evaluation of expression e under local state w to value v is denoted as $\llbracket e \rrbracket_w = v$ and is defined in Fig. 4.4. A constant value evaluates to itself, a variable name evaluates to the value that the variable maps to in the local environment, and a function/operator over expressions evaluates to the function/operator over the value of the expressions. Evaluation of an expression may be undefined. For example, if the local state does not have permission to access variable x , then the evaluation of $x + 1$ is undefined; evaluation of $5/0$ is undefined regardless of the local state.

4.1.3 Predicates

Predicates are a first-order SL used by programs to assert conditions on the state of the program and to specify how to divide resources when a program forks into two threads. SL has been used as the basis of Hoare triple program annotations and to encode the results of dependency/shape analysis, and thus we find it to be a convenient way to specify the side conditions for program transformations using assertions. Chapter 2 introduced several predicate forms:

$C, w \models \text{true}$	<i>always</i>
$C, w \models \text{false}$	<i>never</i>
$C, w \models \neg P$	$C, w \models P$ does not hold
$C, w \models P_1 \wedge P_2$	$C, w \models P_1$ and $C, w \models P_2$
$C, w \models P_1 \vee P_2$	$C, w \models P_1$ or $C, w \models P_2$
$C, w \models P_1 \star P_2$	$w_1 \oplus w_2 = w$, $C, w_1 \models P_1$, and $C, w_2 \models P_2$ for some w_1 and w_2
$C, w \models \forall v. Pv$	$C, w \models Pv$ for all v
$C, w \models \exists v. Pv$	$C, w \models Pv$ for some v
$C, w \models \text{emp}$	$w.S(a) = w.R(a) = \perp$ and $w.E(x) = (\perp, -)$ for all a and x
$C, w \models \text{cansend } a$	$w.S(a) \neq \perp$
$C, w \models \text{canrecv } a$	$w.R(a) \neq \perp$
$C, w \models \text{owns } x$	$w.E(x) = (\top, -)$
$C, w \models \text{owns } a$	$w.S(a) = w.R(a) = \top$
$C, w \models \text{has } x$	$w.E(x) = (\pi, -)$ and $\pi \neq \perp$
$C, w \models e \Downarrow v$	$\llbracket e \rrbracket_w = v$
$C, w \models e$	$C, w \models e \Downarrow \text{true}$
$C, w \models a = [e_1, \dots, e_n]$	$C(a) = [\llbracket e_1 \rrbracket_w, \dots, \llbracket e_n \rrbracket_w]$ and $C, w \models \text{owns } a$

Figure 4.5: Predicate semantics

$$\begin{aligned}
P ::= & \neg P \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \star P_2 \mid \text{owns } x \\
& \mid \text{owns } a \mid \text{cansend } a \mid \text{canrecv } a \mid a = [e_1, \dots, e_n] \mid \dots
\end{aligned}$$

Predicate P is satisfied by global channels C and local state w when $C, w \models P$, which is defined in Fig. 4.5.

Definition 25. P_1 entails P_2 , written $P_1 \vdash P_2$, if $C, w \models P_1$ implies $C, w \models P_2$ for all C and w .

Definition 26. P_1 is equivalent to P_2 , written $P_1 \dashv\vdash P_2$, if $P_1 \vdash P_2$ and $P_2 \vdash P_1$.

Definition 27. precise P iff $P \star (A \wedge B) \dashv\vdash (P \star A) \wedge (P \star B)$ for every A and B .

Many of the logical operators are standard. If $P_1 \star P_2$ holds on a local state, then the state can be divided into two substates with disjoint channel permissions and disjoint variable permissions, which respectively satisfy P_1 and P_2 .

$u ::=$	∞	unbounded
	$ $	n
		finite bound ($n \in \{0, 1, \dots\}$)
$z ::=$	u	unbounded or finite bound
	$ $	$n \square u$
		iteration barrier

Figure 4.6: Iteration bounds: the maximum number of iterations a loop may execute.

We use `owns x` to specify full ownership of variable x , and `has x` to specify *some* (not \perp) access to the variable. For the SAs defined in this thesis, the two predicates are equivalent. However, because our Coq development uses fractional permissions (for which the two predicates are not equivalent), we make a distinction between them when writing the theorems and definitions in this thesis.

Several predicates are used for channels: `owns a` , `cansend a` , `canrecv a` , and `$a = [e_1, \dots, e_n]$` . The first specifies full-ownership of channel a , so that the owner is the only one able to send or receive values from a . Predicates `cansend a` and `canrecv a` correspond to `has`, specifying permission to send and receive values to/from channel a , respectively. Finally, `$a = [e_1, \dots, e_n]$` implies `owns a` and states that the current contents of channel a are e_1, \dots, e_n , where e_1 is the most recent value sent into the channel and e_n is the value that will be consumed by the next receive operation.

4.1.4 Iteration bounds

In Section 2.2.4, we introduced loops that are bounded by a maximum number of iterations. The full syntax of these iteration bounds are given by Fig. 4.6. A normal loop only terminates if and when its loop condition becomes false, and thus is unbounded (i.e., bounded by ∞). When bounded by a natural number, n , a loop may execute at most n iterations (or fewer if its loop condition becomes false), at which point it is forced to terminate. The last form, $n \square u$, is an iteration barrier that bounds a loop by $n + u$ iterations; however, it “pauses” (gets stuck) after only n iterations unless it is “resumed.”

$$\begin{array}{ll}
n \boxplus \infty = \infty & \infty \boxplus u = \infty \\
n_1 \boxplus n_2 = n_1 + n_2 & n \boxplus u = n \boxplus u \\
n_1 \boxplus (n_2 \square u) = (n_1 + n_2) \square u & (n \square u_1) \boxplus u_2 = n \square (u_1 \boxplus u_2)
\end{array}$$

Figure 4.7: Adding a finite number to the beginning (\boxplus) of an iteration bound and adding a finite or infinite value to the end (\boxplus) of an iteration bound.

$$\frac{}{\infty < \infty} \quad \frac{n_1 < n_2}{n_1 < n_2} \quad \frac{n_1 < n_2}{n_1 \square u_1 < n_2 \square u_2}$$

Figure 4.8: Comparing iteration bounds

Figure 4.7 defines \boxplus and \boxplus , which add a finite or infinite number to an iteration bound. The first increases the number of iterations an iteration barrier will allow before pausing. The second increases the bound after an iteration barrier resumes. Given a loop that is bounded by z_1 iterations, we may execute an iteration only if its loop condition is true and we can find some z_2 such that $z_2 < z_1$, as defined in Fig. 4.8.

We now define iteration substitution over terms, threads, and machine states:

Definition 28 (Substitution of iteration barriers on terms).

$$\begin{aligned}
(\text{while } e \text{ max } n \square u \text{ do } t)[z] &= \text{while } e \text{ max } n \boxplus z \boxplus u \text{ do } t[z] \\
(\text{while } e \text{ max } u \text{ do } t)[z] &= \text{while } e \text{ max } u \text{ do } t[z] \\
(\text{if } e \text{ then } t_1 \text{ else } t_2)[z] &= \text{if } e \text{ then } t_1[z] \text{ else } t_2[z] \\
(t_1; t_2)[z] &= t_1[z]; t_2[z] \\
([P_1] t_1 \parallel [P_2] t_2)[z] &= [P_1] t_1[z] \parallel [P_2] t_2[z] \\
(x := e)[z] &= x := e \\
(x := \text{rand } V)[z] &= x := \text{rand } V \\
(\text{skip})[z] &= \text{skip} \\
(\{P\})[z] &= \{P\} \\
(\text{send } a \ e)[z] &= \text{send } a \ e \\
(x := \text{recv } a)[z] &= x := \text{recv } a
\end{aligned}$$

Definition 29 (Substitution of iteration barriers on threads).

$$\begin{aligned}
(w : t)[z] &= w : t[z] \\
(b; t)[z] &= b[z]; t[z] \\
(b_1 \parallel b_2)[z] &= b_1[z] \parallel b_2[z]
\end{aligned}$$

$t ::=$	$x := e$	assign the value of e to x
	$x := \text{rand } V$	assign a random value in V to x
	skip	do nothing
	if e then t_1 else t_2	branch; if e then execute t_1 , otherwise t_2
	while e max z do t	loop; iterate t while e is true for at most z iterations
	$\{P\}$	assertion; is stuck iff P does not hold on the state
	$t_1; t_2$	instruction sequencing
	$[P_1] t_1 \parallel [P_2] t_2$	run t_1 in parallel with t_2
	send a e	enqueue the value of e in channel a
	$x := \text{recv } a$	dequeue a value from channel a into variable x

Figure 4.9: Program syntax

Definition 30 (Substitution of iteration barriers on machine states).

$$(C, b)[z] = (C, b[z])$$

Substituting iteration barriers with z , written $t[z]$, $b[z]$, and $(C, b)[z]$, causes all occurrences of bound $n \square u$ in a program to be replaced by $n \boxplus z \boxplus u$. Resuming barriers, written $t \blacktriangleright$, is defined as $t[0]$ (and likewise for threads and machine states).

4.1.5 Program syntax

The program syntax is listed in Fig. 4.9. Assigning the value of expression e to local variable x is denoted by $x := e$. We also include a special assignment statement, $x := \text{rand } V$, which nondeterministically picks a value from set V and assigns it to local variable x (we do not actually model randomness). We use **skip** to denote a terminated program. If-statements are written as **if** e **then** t_1 **else** t_2 , where t_1 is executed if e evaluates to **true**, and t_2 is executed if e evaluates to **false** (otherwise, the statement is stuck). We abbreviate **if** e **then** t **else** **skip** as **if** e **then** t .

$\{P\}$ is an assertion, which is equivalent to **skip** only if P holds on the current state; otherwise it is stuck. Instruction sequencing is written $t_1; t_2$, which runs t_1 until it becomes **skip**, at which point t_2 is run. Fork/join parallelism is written $[P_1] t_1 \parallel [P_2] t_2$, which forks the current thread and divides the state into two sub-

states so that the first is satisfied by P_1 and the second by P_2 , and then interleaves the execution of t_1 and t_2 . If the state cannot be divided as such, then the fork/join statement is stuck. When both t_1 and t_2 terminate (i.e. each becomes `skip`), then their substates will be joined and the program will step to `skip`.

A traditional while-loop is written as `while e max ∞ do t`, which we abbreviate as `while e do t`, because it has no maximum bound on the number of iterations that it may execute. We may instead bound a loop by a finite number of iterations. A loop bounded by 0, `while e max 0 do t`, must immediately terminate and thus is equivalent to `skip`. Bounded by 1, `while e max 1 do t` is equivalent to `if e then t`.

The behavior of a loop bounded by an iteration barrier that immediately pauses and then terminates upon resuming, `while e max 0 \square 0 do t`, depends on the context. In our proofs, it is stuck unless e evaluates to `false` or until we tell it to resume. However, it is *not* equivalent to `while e max 0 \square 0 do t; {false}` because our notion of “equivalence” must continue to hold after resuming iteration barriers. For any finite count n , bound z , and finite/infinite u , `while e max $n \boxplus z$ do t` is equivalent to `while e max n do t; while e max z do t` and `while e max $z \boxplus u$ do t` is equivalent to `while e max z do t; while e max u do t`. (Recall that \boxplus adds a finite number of iterations *before* a barrier, whereas \boxplus adds finite or infinite iterations *after* a barrier; if z is not a barrier, then \boxplus and \boxplus are equivalent.)

Channel communication is done using `send` and `recv` statements. Sending the value of expression e into channel a is performed by `send a e`; this will never block because channel queues are unbounded. Receiving a value from channel a and storing it into variable x is performed by `x := recv a`; it will block until a value becomes available to consume if the channel queue is empty.

Definition 31 (worlds-of). The join of all local states in a program:

$$\frac{}{\text{worlds-of}(w : t) = w} \quad \frac{\text{worlds-of}(b_1) = w_1 \quad \text{worlds-of}(b_2) = w_2 \quad w_1 \boxplus w_2 = w}{\text{worlds-of}(b_1 \parallel b_2) = w}$$

$\alpha ::=$	τ	internal (silent) action
	$ $	$w_O : a!v$ the observer sends v into channel a using permissions w_O
	$ $	$w_O : a?v$ the observer receives v from channel a using permissions w_O

Figure 4.10: Label syntax

4.1.6 Labels

Figure 4.10 presents the syntax of labels, α , which have three forms. The first, τ , is an internal action that is not directly observable. Because we use delayed observations (see Section 3.4), the remaining two forms are somewhat atypical for a LTS. Rather than labeling the actions of a program executing `send` or `recv`, we label the actions of the observer as it communicates with the program using channels.

In other words, the observer behaves like a thread that is composed in parallel with a program and may communicate with the program using public channels. For example, if action $w_O : a!v$ is emitted, then it is as if an external thread with local state w_O (which must claim permission to access channel a) has sent value v into channel a . We restrict the observer to accessing only public channels by requiring that w_O must join with the local state of the program – if the observer were to access a private channel, a , then w_O must claim the relevant permission to a and would thus not join with the local state of the program.

To state that an observation joins with the local state of a program, we define `worlds-of` for labels to extract the local state claimed by the observer when it makes an observation.

Definition 32 (Extracting the local state of a label).

$$\begin{aligned} \text{worlds-of}(\tau) &= (\lambda a.\perp, \lambda a.\perp, \lambda x.(\perp, 0)) \\ \text{worlds-of}(w_O : a!v) &= w_O \\ \text{worlds-of}(w_O : a?v) &= w_O \end{aligned}$$

4.1.7 Operational semantics

Figure 4.11 lists the operational semantics, which contains several unusual rules. First, S-WhileZero and S-WhileTrue work together to ensure that only loops with a nonzero bound may iterate. Upon iteration, the bound reduces from z to z' such that $z' < z$; once the bound reaches zero, S-WhileZero terminates the loop.

Second, we implement delayed observations (Section 3.4): S-Send and S-Recv are not observable, but they indirectly interact with the observer via S-SendObs and S-RecvObs by modifying the global state. Observations may only be made by communicating with a program using public channels, which S-SendObs and S-RecvObs do. These “observation” rules result in a label that contains some local state, w_O , which must be joinable with the local state of the running program, $\mathbf{worlds-of}(b)$.

When we combine two programs together in parallel, some channels that were once public may become private. Given observation α , S-ParL and S-ParR make sure that the permissions of the observer, $\mathbf{worlds-of}(\alpha)$, is disjoint from *both* threads.

We define a LTS for this language in the usual manner (see Section 3.1): program states have form (C, b) and are often denoted by p or q , labels are defined by $\alpha \in \{a, \tau\}$, $\xrightarrow{\alpha}$ is the transition relation, and a is an observation. We abbreviate $p \xrightarrow{\tau} p'$ as $p \rightarrow p'$. The transitive reflexive closure of \rightarrow is denoted by \Rightarrow . We define $p \xRightarrow{a} p'$ to be $p \Rightarrow \cdot \xrightarrow{a} \cdot \Rightarrow p'$. A weak transition from p to p' that performs actions $\vec{a} = [a_0, \dots, a_n]$ is denoted by $p \Rightarrow \cdot \xRightarrow{a_0} \dots \xRightarrow{a_n} p'$. A silent transition from p to p' that takes at least one step, $p \rightarrow \cdot \Rightarrow p'$, is denoted by $p \Rightarrow^+ p'$. A transition from p to p' , performing actions \vec{a} , that takes at least one step is denoted by $p \xRightarrow{\vec{a}}^+ p'$; it is equivalent to $p \Rightarrow^+ p'$ when $\vec{a} = []$ and $p \xRightarrow{\vec{a}} p'$ when $a \neq []$. A counted execution, where $p \xRightarrow{\vec{a}} q$ takes at most n steps, is denoted by $p \xRightarrow{\vec{a}}_n q$.

$$\begin{array}{c}
\frac{C \# w \models P \star \text{true}}{(C, w : \{P\}) \rightarrow (C, w : \text{skip})} \text{S-Assert} \\
\frac{(C, b_1) \xrightarrow{\alpha} (C', b'_1)}{(C, b_1; t_2) \xrightarrow{\alpha} (C', b'_1; t_2)} \text{S-Seq} \\
\frac{}{(C, w : \text{skip}; t_2) \rightarrow (C, w : t_2)} \text{S-SeqSkip} \\
\frac{[e]_w = v \quad w.\text{E}(x) = (\top, -)}{(C, w : x := e) \rightarrow (C, w[x \mapsto v] : \text{skip})} \text{S-Assign} \\
\frac{v \in V \quad w.\text{E}(x) = (\top, -)}{(C, w : x := \text{rand } V) \rightarrow (C, w[x \mapsto v] : \text{skip})} \text{S-Rand} \\
\frac{C(a) = \sigma \quad w.\text{s}(a) \neq \perp \quad [e]_w = v}{(C, w : \text{send } a \ e) \rightarrow (C[a \mapsto v :: \sigma], w : \text{skip})} \text{S-Send} \\
\frac{C(a) = \sigma :: v \quad w.\text{R}(a) \neq \perp \quad w.\text{E}(x) = (\top, -)}{(C, w : x := \text{recv } a) \rightarrow (C[a \mapsto \sigma], w[x \mapsto v], \text{skip})} \text{S-Recv} \\
\frac{w_1 \oplus w_2 = w \quad C \# w_1 \models P_1 \quad C \# w_2 \models P_2}{(C, w : [P_1] t_1 \parallel [P_2] t_2) \rightarrow (C, w_1 : t_1 \parallel w_2 : t_2)} \text{S-ParFork} \\
\frac{w_1 \oplus w_2 = w}{(C, w_1 : \text{skip} \parallel w_2 : \text{skip}) \rightarrow (C, w : \text{skip})} \text{S-ParJoin} \\
\frac{[e]_w = \text{true} \quad z' < z}{(C, w : \text{while } e \ \text{max } z \ \text{do } t) \rightarrow (C, w : t; \text{while } e \ \text{max } z' \ \text{do } t)} \text{S-WhileTrue} \\
\frac{[e]_w = \text{false} \quad 0 < z}{(C, w : \text{while } e \ \text{max } z \ \text{do } t) \rightarrow (C, w : \text{skip})} \text{S-WhileFalse} \\
\frac{}{(C, w : \text{while } e \ \text{max } 0 \ \text{do } t) \rightarrow (C, w : \text{skip})} \text{S-WhileZero} \\
\frac{[e]_w = \text{true}}{(C, w : \text{if } e \ \text{then } t_1 \ \text{else } t_2) \rightarrow (C, w : t_1)} \text{S-IfTrue} \\
\frac{[e]_w = \text{false}}{(C, w : \text{if } e \ \text{then } t_1 \ \text{else } t_2) \rightarrow (C, w : t_2)} \text{S-IfFalse} \\
\frac{C(a) = \sigma \quad w_O.\text{s}(a) \neq \perp \quad \text{worlds-of}(b) \oplus w_O = w}{(C, b) \xrightarrow{w_O : a \mapsto v} (C[a \mapsto v :: \sigma], b)} \text{S-SendObs} \\
\frac{C(a) = \sigma :: v \quad w_O.\text{R}(a) \neq \perp \quad \text{worlds-of}(b) \oplus w_O = w}{(C, b) \xrightarrow{w_O : a \mapsto v} (C[a \mapsto \sigma], b)} \text{S-RecvObs} \\
\frac{(C, b_1) \xrightarrow{\alpha} (C', b'_1) \quad \text{worlds-of}(b_1 \parallel b_2) \oplus \text{worlds-of}(\alpha) = w}{(C, b_1 \parallel b_2) \xrightarrow{\alpha} (C', b'_1 \parallel b_2)} \text{S-ParL} \\
\frac{(C, b_2) \xrightarrow{\alpha} (C', b'_2) \quad \text{worlds-of}(b_1 \parallel b_2) \oplus \text{worlds-of}(\alpha) = w}{(C, b_1 \parallel b_2) \xrightarrow{\alpha} (C', b_1 \parallel b'_2)} \text{S-ParR}
\end{array}$$

Figure 4.11: Operational semantics

4.1.8 Free variables

Definition 33 (Free variables of an expression).

$$\text{freevars } e = \{x \mid \forall w. \llbracket e \rrbracket_w = v \implies \llbracket e \rrbracket_{w[x \mapsto \cdot]} \text{ is undefined}\}.$$

Definition 34 (Free variables of a predicate).

$$\text{freevars } P = \{x \mid \forall C, w. C, w \models P \implies C, w[x \mapsto \cdot] \not\models P\}.$$

Definition 35 (Variables read by a term).

$$\begin{aligned} \text{reads}(\text{skip}) &= \emptyset \\ \text{reads}(\{P\}) &= \text{freevars } P \\ \text{reads}(x := \text{rand } V) &= \emptyset \\ \text{reads}(x := \text{recv } a) &= \emptyset \\ \text{reads}(x := e) &= \text{freevars } e \\ \text{reads}(\text{send } a \ e) &= \text{freevars } e \\ \text{reads}(\text{if } e \text{ then } t_1 \text{ else } t_2) &= \text{freevars } e \cup \text{reads } t_1 \cup \text{reads } t_2 \\ \text{reads}(\text{while } e \text{ max } z \text{ do } t) &= \text{freevars } e \cup \text{reads } t \\ \text{reads}(t_1; t_2) &= \text{reads } t_1 \cup \text{reads } t_2 \\ \text{reads}([P_1] t_1 \parallel [P_2] t_2) &= \text{freevars } P_1 \cup \text{reads } t_1 \cup \text{freevars } P_2 \cup \text{reads } t_2 \end{aligned}$$

Definition 36 (Variables written by a term).

$$\begin{aligned} \text{writes}(\text{skip}) &= \emptyset \\ \text{writes}(\{P\}) &= \emptyset \\ \text{writes}(x := \text{rand } V) &= \{x\} \\ \text{writes}(x := \text{recv } a) &= \{x\} \\ \text{writes}(x := e) &= \{x\} \\ \text{writes}(\text{send } a \ e) &= \emptyset \\ \text{writes}(\text{if } e \text{ then } t_1 \text{ else } t_2) &= \text{writes } t_1 \cup \text{writes } t_2 \\ \text{writes}(\text{while } e \text{ max } z \text{ do } t) &= \text{writes } t \\ \text{writes}(t_1; t_2) &= \text{writes } t_1 \cup \text{writes } t_2 \\ \text{writes}([P_1] t_1 \parallel [P_2] t_2) &= \text{writes } t_1 \cup \text{writes } t_2 \end{aligned}$$

Definition 37 (Sending channels of a term).

$$\begin{aligned}
\text{sends}(\text{skip}) &= \emptyset \\
\text{sends}(\{P\}) &= \emptyset \\
\text{sends}(x := \text{rand } V) &= \emptyset \\
\text{sends}(x := \text{recv } a) &= \emptyset \\
\text{sends}(x := e) &= \emptyset \\
\text{sends}(\text{send } a \ e) &= \{a\} \\
\text{sends}(\text{if } e \ \text{then } t_1 \ \text{else } t_2) &= \text{sends } t_1 \cup \text{sends } t_2 \\
\text{sends}(\text{while } e \ \text{max } z \ \text{do } t) &= \text{sends } t \\
\text{sends}(t_1; t_2) &= \text{sends } t_1 \cup \text{sends } t_2 \\
\text{sends}([P_1] t_1 \parallel [P_2] t_2) &= \text{sends } t_1 \cup \text{sends } t_2
\end{aligned}$$

Definition 38 (Receiving channels of a term).

$$\begin{aligned}
\text{recvs}(\text{skip}) &= \emptyset \\
\text{recvs}(\{P\}) &= \emptyset \\
\text{recvs}(x := \text{rand } V) &= \emptyset \\
\text{recvs}(x := \text{recv } a) &= \{a\} \\
\text{recvs}(x := e) &= \emptyset \\
\text{recvs}(\text{send } a \ e) &= \emptyset \\
\text{recvs}(\text{if } e \ \text{then } t_1 \ \text{else } t_2) &= \text{recvs } t_1 \cup \text{recvs } t_2 \\
\text{recvs}(\text{while } e \ \text{max } z \ \text{do } t) &= \text{recvs } t \\
\text{recvs}(t_1; t_2) &= \text{recvs } t_1 \cup \text{recvs } t_2 \\
\text{recvs}([P_1] t_1 \parallel [P_2] t_2) &= \text{recvs } t_1 \cup \text{recvs } t_2
\end{aligned}$$

Definition 39 (Variables read by a thread).

$$\begin{aligned}
\text{reads}(w : t) &= \text{reads } t \\
\text{reads}(b; t) &= \text{reads } b \cup \text{reads } t \\
\text{reads}(b_1 \parallel b_2) &= \text{reads } b_1 \cup \text{reads } b_2
\end{aligned}$$

Definition 40 (Variables written by a thread).

$$\text{writes}(w : t) = \text{writes } t$$

$$\begin{aligned}\text{writes}(b; t) &= \text{writes } b \cup \text{writes } t \\ \text{writes}(b_1 \parallel b_2) &= \text{writes } b_1 \cup \text{writes } b_2\end{aligned}$$

Definition 41 (Sending channels of a thread).

$$\begin{aligned}\text{sends}(w : t) &= \text{sends } t \\ \text{sends}(b; t) &= \text{sends } b \cup \text{sends } t \\ \text{sends}(b_1 \parallel b_2) &= \text{sends } b_1 \cup \text{sends } b_2\end{aligned}$$

Definition 42 (Receiving channels of a thread).

$$\begin{aligned}\text{recvs}(w : t) &= \text{recvs } t \\ \text{recvs}(b; t) &= \text{recvs } b \cup \text{recvs } t \\ \text{recvs}(b_1 \parallel b_2) &= \text{recvs } b_1 \cup \text{recvs } b_2\end{aligned}$$

4.2 Program equivalence

Program equivalence between terms is just a wrapper around a congruence between machine states.

Definition 43 (Program equivalence). Given relation \approx between machine states, $t_1 \equiv_{\approx} t_2$ iff $(C, w : t_1) \approx (C, w : t_2)$ for any global state C and local state w .

Definition 44 (Compatible with \blacktriangleright). A relation between machine states, \mathcal{R} , is compatible with \blacktriangleright when for any $(p, q) \in \mathcal{R}$,

- $(p_{\blacktriangleright}, q_{\blacktriangleright}) \in \mathcal{R}$.

4.2.1 Bisimilarity and contrasimilarity

We define \equiv with respect to bisimilarity and contrasimilarity relations that are iteration, and termination sensitive.

Definition 45 (Bisimilarity). Machine states p and q are bisimilar, written $p \approx_{\downarrow^*} q$, iff there exists an \mathcal{R} such that $(p, q) \in \mathcal{R}$ and where \mathcal{R} is:

- a bisimulation,
- compatible with \blacktriangleright , and
- termination sensitive.


Definition 46 (Partial contrasimilarity). Machine state q partially contrasimulates p , written $p \leq_{\downarrow^*c} q$, iff there exists an \mathcal{R} such that $(p, q) \in \mathcal{R}$ and where \mathcal{R} is:

- a contrasimulation,
- compatible with \blacktriangleright ,
- one-way termination sensitive, and


Definition 47 (Contrasimilarity). Machine states p and q are contrasimilar, written $p \approx_{\downarrow^*c} q$, iff there exists an \mathcal{R} such that $(p, q) \in \mathcal{R} \cap \mathcal{R}^{-1}$ and where \mathcal{R} is:

- a contrasimulation,
- compatible with \blacktriangleright ,
- one-way termination sensitive, and

Lemma 4.1. *If $p \leq_{\downarrow^*c} q$ and $q \leq_{\downarrow^*c} p$, then $p \approx_{\downarrow^*c} q$.*

Proof. Follows directly from definitions 46 and 47. 

We often find ourselves proving \approx_{\downarrow^*} for a symmetric \mathcal{R} , in which case it is not necessary to show simulation and termination sensitivity in both directions. The following lemma helps us simplify our proofs of bisimulation for symmetric relations.

Lemma 4.2. *If \mathcal{R} is symmetric, then \mathcal{R} is a bisimulation if it is a simulation, compatible with \blacktriangleright , and one-way termination sensitive.* 

Lemma 4.3. *Given relations \approx_1 and \approx_2 , if $\approx_1 \subseteq \approx_2$, then $\equiv_{\approx_1} \subseteq \equiv_{\approx_2}$.*

Corollary 4.1. *If $t_1 \equiv_{\approx_{\downarrow^*}} t_2$, then $t_1 \equiv_{\approx_{\downarrow^*c}} t_2$.*

We will prove the soundness of the transformation framework presented in chapter 2 by proving $\equiv_{\approx_{\downarrow^*c}}$ for each \equiv rule.


4.3 Congruence

In this section, we state the soundness theorems for several of the basic rewrite rules in our framework. To support the main theorems, we will list a variety of supporting lemmas in Section 4.3.1 that will be used in the rest of this section and chapter. Then, we will state the soundness theorems for some basic algebraic rules in Section 4.3.2 and the compositional rules in Section 4.3.3. For most theorems in this section, we will not go into much detail because the results are not new. At the end of this section, we will prove that *bisimulation* is congruent for **while** loops because the proof will be an informative exercise in preparation for proving loop folding.

Although we are only interested in soundness with respect to contrasimulation, we prove a bisimulation wherever possible because we can apply corollary 4.1 to obtain a contrasimilarity. However, the compositional rules must be proved with respect to contrasimulation.

4.3.1 Supporting lemmas

Lemma 4.4 (Resuming barriers). *If $p \approx_{\downarrow^*c} q$, then $p_{\blacktriangleright} \approx_{\downarrow^*c} q_{\blacktriangleright}$.*

Proof. By $p \approx_{\downarrow^*c} q$, there must exist an \mathcal{R} that is compatible with \blacktriangleright and such that $(p, q) \in \mathcal{R}$. Definition 44 implies $(p_{\blacktriangleright}, q_{\blacktriangleright}) \in \mathcal{R}$, thus $p_{\blacktriangleright} \approx_{\downarrow^*c} q_{\blacktriangleright}$. 


Lemma 4.5. *If $(C, w : \mathbf{skip}) \xrightarrow{\bar{a}} (C', w : \mathbf{skip})$, then $C \# w \vDash P$ iff $C' \# w \vDash P$.*

Proof. The intuition is that assertions are invariant as observations are made. This holds because C and C' can only differ by channel states for which w does not claim full ownership, so $C \# w = C' \# w$. □

Lemma 4.6. *If $(C, w : \mathbf{skip}) \approx q$, then $q \Rightarrow (C, w : \mathbf{skip})$, where $\approx \in \{\approx_{\downarrow^*}, \leq_{\downarrow^*c}, \approx_{\downarrow^*c}\}$.*

Proof. Follows by termination sensitivity. 


Lemma 4.7. *If $p \leq q$ and $p \Rightarrow (C', w' : \mathbf{skip})$, then $q \Rightarrow (C', w' : \mathbf{skip})$, where $\leq \in \{\approx_{\downarrow^*}, \leq_{\downarrow^*c}\}$.*

Proof. There exists a q' such that $q \Rightarrow q'$ and $(C', w' : \mathbf{skip}) \geq q'$. By $q' \Rightarrow q'$, there exists a p' such that $(C', w' : \mathbf{skip}) \Rightarrow p'$ and $p' \leq q'$. However, \mathbf{skip} cannot step, so $p' = (C', w' : \mathbf{skip})$. Thus $q' \Rightarrow (C', w' : \mathbf{skip})$ by lemma 4.6. 


Lemma 4.8 (Multistep inversion principle for sequential composition).

If $(C, b; t) \xRightarrow{\vec{a}} p'$, then either:


- $(C, b) \xRightarrow{\vec{a}} (C', b')$ and $p' = (C', b'; t)$ for some C' and b' , or
- $(C, b) \xRightarrow{\vec{a}_1} (C', w' : \mathbf{skip})$, $(C', w' : t) \xRightarrow{\vec{a}_2} p'$, and $\vec{a} = \vec{a}_1 \cdot \vec{a}_2$ for some \vec{a}_1 , \vec{a}_2 , C' and w' .

Proof. By induction on $\xRightarrow{\vec{a}}$. 

Lemma 4.9. *If $(C, b) \approx (C, b_2; \{P\})$ and $(C, b) \xRightarrow{\vec{a}} (C', b')$, then $(C', b') \approx (C', b'; \{P\})$, where \approx is a termination sensitive relation such as \approx_{\downarrow^*} or \approx_{\downarrow^*c} .*


Proof. By induction on $\xRightarrow{\vec{a}}$. 

Lemma 4.10 (Resuming a loop schema). $f[u]_{\blacktriangleright} = f[u]$.


Proof. The term, $f[u]$ has already had its iteration holes filled by u (which is either a finite number or infinity), so there are no iteration barriers to resume. 

4.3.2 Algebraic

Lemma 4.11 (T-Refl). $t \equiv_{\approx_{\downarrow^*c}} t$.


Proof. Follows by the reflexivity of \approx_{\downarrow^*c} . 


Lemma 4.12 (T-Sym). *If $t_1 \equiv_{\approx_{\downarrow}^* c} t_2$, then $t_2 \equiv_{\approx_{\downarrow}^* c} t_1$.*


Proof. Follows by the symmetry of $\approx_{\downarrow}^* c$. 


Lemma 4.13 (T-Trans). *If $t_1 \equiv_{\approx_{\downarrow}^* c} t_2$ and $t_2 \equiv_{\approx_{\downarrow}^* c} t_3$, then $t_1 \equiv_{\approx_{\downarrow}^* c} t_3$.*


Proof. Follows by the transitivity of $\approx_{\downarrow}^* c$. 

Lemma 4.14 (T-WhileIf). *$\text{while } e \text{ max } 1 \text{ do } t \equiv_{\approx_{\downarrow}^*} \text{if } e \text{ then } t$.* 

Lemma 4.15 (T-FalseWhile). *$\{-e\}; \text{while } e \text{ max } z \text{ do } t \equiv_{\approx_{\downarrow}^*} \{-e\}$.* 

Lemma 4.16 (T-WhileFalse). *$\text{while } e \text{ do } t \equiv_{\approx_{\downarrow}^*} \text{while } e \text{ do } t; \{-e\}$.* 

Lemma 4.17 (T-Split $_{\diamond}$). *$\text{while } e \text{ max } z \diamond u \text{ do } t \equiv_{\approx_{\downarrow}^* c} (\text{while } e \text{ max } z \text{ do } t);$
 $(\text{while } e \text{ max } u \text{ do } t)$.* 


Lemma 4.18 (T-Split $_{\boxplus}$). *$\text{while } e \text{ max } n \boxplus z \text{ do } t \equiv_{\approx_{\downarrow}^* c} (\text{while } e \text{ max } n \text{ do } t);$
 $(\text{while } e \text{ max } z \text{ do } t)$.* 

4.3.3 Compositional

Lemma 4.19 (T-Resume). *If $t_1 \equiv_{\approx_{\downarrow}^* c} t_2$, then $t_{1\blacktriangleright} \equiv_{\approx_{\downarrow}^*} t_{2\blacktriangleright}$.*

Proof. This follows directly from lemma 4.4. 

Lemma 4.20 (T-Seq). *If $t_1 \equiv_{\approx_{\downarrow}^* c} t'_1$ and $t_2 \equiv_{\approx_{\downarrow}^* c} t'_2$, then $t_1; t_2 \equiv_{\approx_{\downarrow}^* c} t'_1; t'_2$.*

Proof. We use lemma 4.1 and, because our goal and premises are symmetric, need only prove partial contrasimilarity in one direction. We break the remaining proof into two parts. The first is to show that we can substitute t_1 , for which the proof is straightforward using $\mathcal{I} \cup \{((C_1, b_1; t), (C_2, b_2; t)) \mid (C_1, b_1) \approx_{\downarrow}^* c(C_2, b_2)\}$. The the second step is to substitute t_2 , for which we use $\mathcal{I} \cup \{((C, b; t_1), (C, b; t_2))\}$. 

Lemma 4.21 (T-Par). *If $P_1 \dashv\vdash P'_1$, $t_1 \equiv_{\approx_{\downarrow^*c}} t'_1$, $P_2 \dashv\vdash P'_2$, $t_2 \equiv_{\approx_{\downarrow^*c}} t'_2$, then $[P_1] t_1 \parallel [P_2] t_2 \equiv_{\approx_{\downarrow^*c}} [P'_1] t'_1 \parallel [P'_2] t'_2$.*

Lemma 4.22 (T-If). *If $e \dashv\vdash e'$, $t_1 \equiv_{\approx_{\downarrow^*c}} t'_1$, and $t_2 \equiv_{\approx_{\downarrow^*c}} t'_2$, then **if** e **then** t_1 **else** $t_2 \equiv_{\approx_{\downarrow^*c}}$ **if** e' **then** t'_1 **else** t'_2 .*

Proof. Using lemma 4.1, we only need to prove partial contrasimilarity in one direction, for which we prove that $\{((C, w : \mathbf{if} \ e \ \mathbf{then} \ t_1 \ \mathbf{else} \ t_2), (C, w : \mathbf{if} \ e' \ \mathbf{then} \ t'_1 \ \mathbf{else} \ t'_2))\} \cup \leq_{\downarrow^*c}$ is a contrasimulation using lemma 3.19, termination sensitive, and compatible with \blacktriangleright . Lemma 3.19 allows us to only consider the first step of the **if** statement to see if e is true or false, and then let $t_1 \equiv_{\approx_{\downarrow^*c}} t'_1$ or $t_2 \equiv_{\approx_{\downarrow^*c}} t'_2$ take care of the rest. \blacksquare

Before we prove congruence for **while** with respect to contrasimilarity, we look at the case of bisimilarity to get a clearer understanding of how it works.

Lemma 4.23 (T-While w.r.t. bisimulation). *If $t_1 \equiv_{\approx_{\downarrow^*}} t_2$, then **while** e **do** $t_1 \equiv_{\approx_{\downarrow^*}}$ **while** e **do** t_2 .*

Proof. By definition 43, we assume that $(C, w : t_1) \approx_{\downarrow^*} (C, w : t_2)$ for all C and w , and given some C and w , we must prove $(C, w : \mathbf{while} \ e \ \mathbf{do} \ t_1) \approx_{\downarrow^*} (C, w : \mathbf{while} \ e \ \mathbf{do} \ t_2)$. To do so, we propose a relation that holds between the machine states of the two loops as they execute,

$$\begin{aligned} \mathcal{R} = & \{(p, q) \mid \exists C, w. p = (C, w : \mathbf{while} \ e \ \mathbf{do} \ t_1) \wedge q = (C, w : \mathbf{while} \ e \ \mathbf{do} \ t_2)\} \\ & \cup \{(p, q) \mid \exists C_1, C_2, b_1, b_2. (C_1, b_1) \approx_{\downarrow^*} (C_2, b_2) \\ & \quad \wedge p = (C_1, b_1; \mathbf{while} \ e \ \mathbf{do} \ t_1) \wedge q = (C_2, b_2; \mathbf{while} \ e \ \mathbf{do} \ t_2)\} \\ & \cup \mathcal{I}. \end{aligned} \tag{4.1}$$

The intuition behind eq. (4.1) is that the loops continue to unfold iterations as long as e is true, and the states derived from those iterations will be equivalent:

$(C_1, b_1) \approx_{\downarrow*} (C_2, b_2)$. After each iteration is completed, lemma 4.6 (termination sensitivity) ensures that the loops begin again with the same channel and local state. If e becomes false, then both loops will step to $(C', w' : \mathbf{skip})$ for some C' and w' , and will thus be in the identity relation, \mathcal{I} . The initial loops are related by the first subset that defines \mathcal{R} .

Finally, we prove that \mathcal{R} is a bisimulation. Because \mathcal{R} is symmetric, we need only show that it is a simulation, compatible with \blacktriangleright , and one-way termination sensitive.

Simulation: We are given $(p, q) \in \mathcal{R}$ and $p \xrightarrow{\alpha} p'$, and must show that $q \xRightarrow{\alpha} q'$ and $(p', q') \in \mathcal{R}$ for some q' . Corresponding to the three sets that define \mathcal{R} , there are three cases for $(p, q) \in \mathcal{R}$:

Case 1. Assume $p = (C, w : \mathbf{while} \ e \ \mathbf{do} \ t_1)$ and $q = (C, w : \mathbf{while} \ e \ \mathbf{do} \ t_2)$. There are three ways that $(C, w : \mathbf{while} \ e \ \mathbf{do} \ t_1) \xrightarrow{\alpha} p'$ may step:

1. α is observed (via S-SendObs or S-RecvObs). The same observation can be made of q .
2. e is true and $p' = (C, w : t_1; \mathbf{while} \ e \ \mathbf{do} \ t_1)$. There exists $q' = (C, w : t_2; \mathbf{while} \ e \ \mathbf{do} \ t_2)$ such that $(C, w : \mathbf{while} \ e \ \mathbf{do} \ t_2) \xrightarrow{\alpha} (C, w : t_2; \mathbf{while} \ e \ \mathbf{do} \ t_2)$ and $(p', q') \in \mathcal{R}$.
3. e is false and $p' = w : \mathbf{skip}$. There exists $q' = (C, w : \mathbf{skip})$ such that $(C, w : \mathbf{while} \ e \ \mathbf{do} \ t_2) \xrightarrow{\alpha} (C, w : \mathbf{skip})$ and $(p', q') \in \mathcal{I} \subset \mathcal{R}$.

Case 2. Assume $p = (C_1, b_1; \mathbf{while} \ e \ \mathbf{do} \ t_1)$, $q = (C_2, b_2; \mathbf{while} \ e \ \mathbf{do} \ t_2)$, and $(C_1, b_1) \approx_{\downarrow*} (C_2, b_2)$. There are three ways that $(C_1, b_1; \mathbf{while} \ e \ \mathbf{do} \ t_1) \xrightarrow{\alpha} p'$ may step:

1. α is observed. The same observation can be made of q .
2. $b_1 = w_1 : \mathbf{skip}$, $\alpha = \tau$, and $p' = (C_1, w_1 : \mathbf{while} \ e \ \mathbf{do} \ t_1)$. The termination sensitivity of $(C_1, b_1) \approx_{\downarrow*} (C_2, b_2)$ implies $(C_2, b_2) \Rightarrow (C_1, w_1 : \mathbf{skip})$ because b_1 is halted (lemma 4.6). Thus there exists $q' = (C_1, w_1 : \mathbf{while} \ e \ \mathbf{do} \ t_2)$ such that $(C_2, b_2; \mathbf{while} \ e \ \mathbf{do} \ t_2) \Rightarrow q'$ and $(p', q') \in \mathcal{R}$.

3. $(C_1, b_1) \xrightarrow{\alpha} (C'_1, b'_1)$ and $p' = (C'_1, b'_1; \text{while } e \text{ do } t_1)$. There must exist some (C'_2, b'_2) such that $(C_2, b_2) \xrightarrow{\alpha} (C'_2, b'_2)$ and $(C'_1, b'_1) \approx_{\downarrow^*} (C'_2, b'_2)$ because $(C_1, b_1) \approx_{\downarrow^*} (C_2, b_2)$. Thus there exists $q' = (C'_2, b'_2; \text{while } e \text{ do } t_2)$ such that $(C_2, b_2; \text{while } e \text{ do } t_2) \xrightarrow{\alpha} q'$ and $(p', q') \in \mathcal{R}$.

Case 3. Assume $p = q$. Trivial.

Compatibility with \blacktriangleright : *We are given $(p, q) \in \mathcal{R}$, and must show $(p_{\blacktriangleright}, q_{\blacktriangleright}) \in \mathcal{R}$. For each case of $(p, q) \in \mathcal{R}$, $(p_{\blacktriangleright}, q_{\blacktriangleright}) \in \mathcal{R}$ simplifies to a form that holds by lemma 4.4.*

One-way termination sensitivity: *We are given $((C, w : \text{skip}), q) \in \mathcal{R}$, and must show that $q \Rightarrow (C, w : \text{skip})$. It must be the case that the programs are related by \mathcal{I} , so $q = (C, w : \text{skip})$ (taking 0 steps).*



Proving congruence with respect to contrasimilarity works much the same. However, we must consider multiple steps at once. The inversion principle for **while** is complicated because multiple iterations may execute, but by counted contrasimulation (lemma 3.19), we need only consider at most one iteration.

Lemma 4.24 (T-While). *If $e_1 \dashv\vdash e_2$ and $t_1 \equiv_{\approx_{\downarrow^*c}} t_2$, then $\text{while } e_1 \text{ max } z \text{ do } t_1 \equiv_{\approx_{\downarrow^*c}} \text{while } e_2 \text{ max } z \text{ do } t_2$.*

Proof. We use lemma 4.1 and, because our goal and premises are symmetric, need only prove partial contrasimilarity in one direction. The relation we use is:

$$\begin{aligned} & \{(p, q) \mid \exists C, w, z. p = (C, w : \text{while } e_1 \text{ max } z \text{ do } t_1) \\ & \quad \wedge q = (C, w : \text{while } e_2 \text{ max } z \text{ do } t_2)\} \\ \cup & \{(p, q) \mid \exists C_1, C_2, b_1, b_2, z. (C_1, b_1) \leq_{\downarrow^*c} (C_2, b_2) \\ & \quad \wedge p = (C_1, b_1; \text{while } e_1 \text{ max } z \text{ do } t_1) \\ & \quad \wedge q = (C_2, b_2; \text{while } e_2 \text{ max } z \text{ do } t_2)\} \\ \cup & \mathcal{I}. \end{aligned}$$

We prove that this relation has all of the properties of partial contrasimilarity, but use lemma 3.19 to prove contrasimulation. The proof is very similar to that of lemma 4.23.



4.4 Loop folding

We now begin to fully explain loop folding and the motivation behind iteration barriers. Because loop folding shows that a loop schema behaves like a single `while` loop, we use a strategy similar to our proof of T-While in lemma 4.23. The take-away from the proof is that a `while` loop has a recurring structure – the loop itself, prefixed by an executing iteration – that we can use to compare its behavior to other loops. This same approach can be applied to comparing a folded loop ($f[\infty]$) to another loop because the folded loop should be equivalent to a `while` loop.

But to use this strategy, we must show that loop schemas have a *loop decomposition* property, which is akin to an inversion principle for the execution of a `while` loop. Namely, that any state derived from a loop is equivalent to the sequential composition of a state derived from a finite number of iterations and the original loop.

Recall that a loop schema is a program, loop condition, and invariant such that a combining transformation holds and termination implies and is implied-by the loop condition becoming false.

Definition 48 (Loop schema). Where \approx is a relation between machine states, $(f, e, P)_{\approx}$ defines a loop schema such that:

- $\{P\}; f[1] \equiv_{\approx} \{P\}; f[1]; \{P\}$, *(P is a loop invariant)*
- $\forall u. \{\neg e \wedge P\}; f[u] \equiv_{\approx} \{\neg e \wedge P\}$, *(¬e implies termination)*
- $\{P\}; f[\infty] \equiv_{\approx} \{P\}; f[\infty]; \{\neg e \wedge P\}$, and *(¬e is implied by termination)*
- $\forall n \geq 1, u. \{P\}; f[n \square 0]; f[u] \equiv_{\approx} \{P\}; f[n \square u]$. *(combining transformation)*

It is a simple exercise to show that `while` loops are loop schemas.

Lemma 4.25 (Simple loop schema). *If $\{P\}; t \equiv_{\approx} \{P\}; t; \{P\}$ and $\approx \in \{\approx_{\downarrow*}, \approx_{\downarrow*c}\}$, then $(\{P\}; \text{while } e \text{ max } \square \text{ do } t, e, P)_{\approx}$.*

Proof. Follows by lemmas 4.14 to 4.17. 

Up to this point, we have not explained why the combining transformation uses iteration barriers. As we introduce the concepts behind loop folding, we will initially assume that iteration barriers are not necessary – that the following property suffices as the combining transformation:

$$\forall n \geq 1, u. \{P\}; f[n]; f[u] \equiv \{P\}; f[n \boxplus u]. \quad (4.2)$$

However, as we attempt to prove the loop decomposition property, it will become clear that we need a stronger definition for loop schemas; by either restricting the syntax of f or making the combining transformation stronger. We choose the latter by adding iteration barriers to the combining transformation, which has the added benefit of enabling extensional reasoning about the looping behavior of loop schemas.

4.4.1 Loop decomposition

We now explain loop decomposition. The purpose of this section is exposition; for this reason we will introduce the main ideas using bisimilarity ($\approx_{\downarrow*}$) rather than con-
trasimilarity. Although we will reach a dead end in our first attempt to prove loop decomposition, we will ultimately succeed in Section 4.4.2 with just a minor adjustment (adding iteration barriers) to our approach.

If we were to use the relation in eq. (4.1), verbatim, to prove loop folding for some $(f, e, P)_{\approx}$, we would quickly fail: if $(C, w : f[\infty]) \xrightarrow{\alpha} p'$, it is not necessarily (or even likely) the case that $p' = (C', b'; f[\infty])$ for some C' and b' . But since an execution of $f[\infty]$ can use at most a finite number of iterations, we should be able to run $f[n]$, for some n , to achieve the same results. More generally:

Lemma 4.26 (Finite iterations). *If $p[\infty] \xrightarrow{\bar{a}} p'$, then there exists an $n > 0$ and p'' such that $p[n \square \infty] \xrightarrow{\bar{a}} p''$ and $p' = p''$. \blacktriangleright*

By the combining transformation, $\{P\}; f[n]; f[\infty] \equiv_{\approx_{\downarrow}^*} \{P\}; f[\infty]$, so perhaps p' is equivalent to $(C', b'; f[\infty])$ for some b' derived from $f[n]$.

Proposition 4.1 (Loop decomposition). *Given $(f, e, P)_{\approx_{\downarrow}^*}$, if $C \# w \models P \star \text{true}$ and $(C, w : f[\infty]) \xrightarrow{\bar{a}} p'$, then $(C, w : f[n]) \xrightarrow{\bar{a}} (C', b')$ and $p' \approx_{\downarrow}^*(C', b'; f[\infty])$ for some n , C' , and b' .*

With proposition 4.1, we can use the following relation to prove loop folding:

$$\{(p, q) \mid \exists C, b. p \approx_{\downarrow}^*(C, b; \{P\}; \text{while } e \text{ do } f[1]) \wedge q \approx_{\downarrow}^*(C, b; \{P\}; f[\infty])\} \quad (4.3)$$

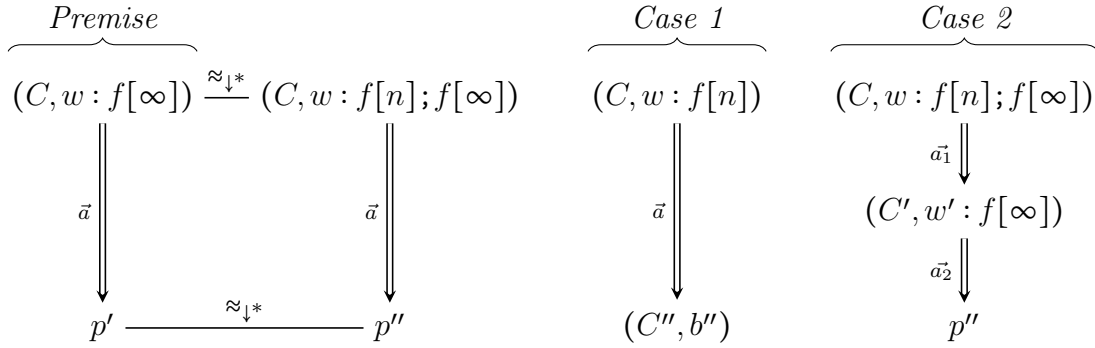
Notice that eq. (4.3) is simpler than eq. (4.1), which is a union of three subsets that handle the three forms a loop may take during its execution. Using a bisimulation relation, rather than equality, allows us to collapse these cases into one. For example, if $p = (C, w : \text{while } e \text{ do } f[1])$, then P holds and $p \approx_{\downarrow}^*(C, w : \text{skip}; \{P\}; \text{while } e \text{ do } f[1])$. If $p = (C, w : \text{skip})$, then the loop must have terminated and $C \# w \models \neg e \wedge P \star \text{true}$, so $p \approx_{\downarrow}^*(C, w : \text{skip}; \{P\}; \text{while } e \text{ do } f[1])$.

However, proving that eq. (4.3) is a bisimulation by showing that it is a simulation (among other properties) in both directions is redundant. By generalizing the relation to compare two loop schemas (see lemma 4.25) so that it is symmetric, we can use lemma 4.2 to nearly halve our proof obligations.

$$\begin{aligned} \{(p, q) \mid \exists f, g, C, b. (f, e, P)_{\approx_{\downarrow}^*} \wedge (g, e, P)_{\approx_{\downarrow}^*} \wedge \{P\}; f[1] \equiv_{\approx_{\downarrow}^*} \{P\}; g[1] \\ \wedge p \approx_{\downarrow}^*(C, b; \{P\}; f[\infty]) \wedge (C, b; \{P\}; g[\infty]) \approx_{\downarrow}^* q\}. \end{aligned} \quad (4.4)$$

Proposition 4.1 and eq. (4.4) allow us to prove the soundness of loop folding by using the same strategy that we used to prove lemma 4.23. We now turn to proving proposition 4.1.

Assume we are given $(f, e, P)_{\approx_{\downarrow}^*}$, $C \# w \models P \star \text{true}$, and $(C, w : f[\infty]) \xrightarrow{\vec{a}} p'$, and we know that at most n iterations were used in the execution. By the combining transformation, we can derive:



There are two cases for p'' : either 1) $(C, w : f[n]) \xrightarrow{\vec{a}} (C'', b'')$ and $p'' = (C'', b''; f[\infty])$, or 2) $(C, w : f[n]) \xrightarrow{\vec{a}_1} (C', w' : \text{skip})$, $(C', w' : f[\infty]) \xrightarrow{\vec{a}_2} p''$, and $\vec{a} = \vec{a}_1 \cdot \vec{a}_2$. In the first case, loop decomposition trivially holds. But the second case brings us back to square one – attempting to decompose the behavior of $f[\infty]$. We might attempt to keep increasing n to see if a big enough value will cause $f[n]$ to not terminate, but there is no guarantee that the result will be any different. Consider loop schema $(f_1, \text{true}, \text{true})_{\approx_{\downarrow}^*}$, where

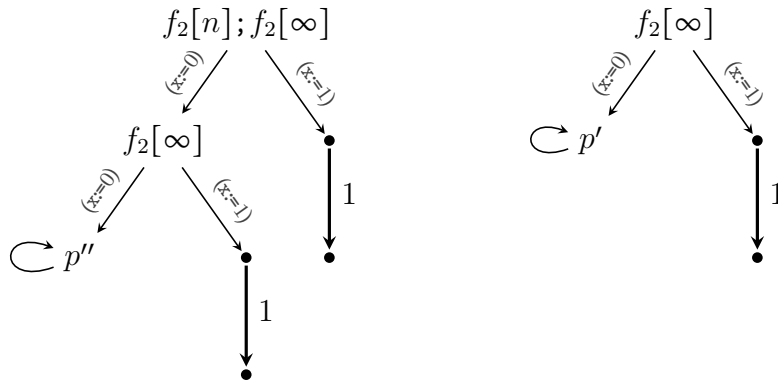
$$\begin{aligned}
 f_1 = & \text{while true max } \square \text{ do} & (4.5) \\
 & (x := 0 \parallel x := 1); \\
 & \text{if } x=0 \text{ then} \\
 & \quad \text{skip} \\
 & \text{else} \\
 & \quad y := y+1
 \end{aligned}$$

Because the combining transformation is user-supplied, we must assume the worst case – that for every n , $f_1[n]$ will always choose $x := 0$ and terminate when $f_1[n]; f_1[\infty]$ simulates $f_1[\infty]$. We say that $f_1[n]$ can “fall through” for every n when this happens.

On the other hand, it *is* possible for $f_1[n]; f_1[\infty]$ to simulate $f_1[\infty]$ without falling through. To prove loop decomposition, perhaps a viable strategy is to inspect the structure of the loop schema to reconstruct a “good” execution that does not fall through. The next example shows that such “good” executions do not always exist:

$$\begin{aligned}
 f_2 &= (x := 0 \parallel x := 1); & (4.6) \\
 &\text{while true max } \square \text{ do} \\
 &\quad \text{if } x=1 \text{ then} \\
 &\quad \quad \text{print 1;} \\
 &\quad \quad \{\text{false}\}
 \end{aligned}$$

Unlike eq. (4.5), the ability of $f_2[n]$ to fall through is the only reason that eq. (4.2) holds for eq. (4.6). To illustrate why, consider the following transition diagrams that are equivalent to $f_2[n]; f_2[\infty]$ and $f_2[\infty]$, respectively:



The left diagram shows that $f_2[n]; f_2[\infty]$ can choose between diverging or printing 1 *twice*. The right diagram shows that $f_2[\infty]$ may only make this choice *once*. Thus in response to $f_2[\infty]$ transitioning to state p' , a bisimilar state may only be reached by $f_2[n]; f_2[\infty]$ if $f_2[n]$ falls through and the subsequent $f_2[\infty]$ transitions to p'' .

Furthermore, f_2 is a degenerate schema because it does not hold for loop folding. This is because the folded loop, $f_2[\infty]$, can initially choose $x := 0$ and commit to never printing 1, while the original program, `while true do $f_2[1]$` , cannot because its choice is repeated upon every iteration.

There are several ways we can resolve these issues. We could disallow internal choice. (But we have already stated that supporting internal choice is a key goal.) Or we could restrict the syntax of loop schemas; e.g. by not allowing code to appear outside of a bounded `while` loop. We have instead found a third option.

Iteration barriers prevent degenerate loop schemas. For the above example, $f_2[1 \square 0]; f_2[\infty] \not\approx_{\sim_1^*} f_2[1 \square \infty]$. If $f_2[1 \square \infty]$ picks $x := 0$, then $f_2[1 \square 0]; f_2[\infty]$ must make the same choice. But there is no way for $f_2[1 \square 0]$ to fall through because of the iteration barrier. By T-Resume, they must remain bisimilar after the barriers are resumed, but having not fallen through, the program derived from $f_2[1 \square 0]; f_2[\infty]$ has a second chance to choose $x := 1$ and print 1 while the program derived from $f_2[1 \square \infty]$ does not. Iteration barriers also let us prove proposition 4.1 for programs such as eq. (4.5) via extensional reasoning.

4.4.2 The proof of loop folding

With the combining transformation that uses iteration barriers (definition 48), we are now ready to prove the loop decomposition property and, finally, loop folding. Unlike Section 4.4.1, we focus on proving soundness with respect to contrasimilarity because it is the underlying equivalence of our framework. We start off with two helper lemmas, which are used by the loop decomposition theorem. The first lemma allows us to re-run an execution without iteration barriers. The second lemma reasons that if a program with an iteration barrier terminates, then the iteration barrier must not have been reached and thus we can substitute it with any iteration bound.

Lemma 4.27 (Resuming executions). *If $p \xrightarrow{\bar{a}} p'$ then $p \blacktriangleright \xrightarrow{\bar{a}} p' \blacktriangleright$.*

Proof. Iteration barriers only cause programs to become stuck; their removal cannot reduce the behaviors/transitions available to a program. \blacksquare

Lemma 4.28 (Termination of iteration barriers). *If $p[n \square u] \xrightarrow{\vec{a}} (C', w' : \mathbf{skip})$ then $p[n \boxplus z] \xrightarrow{\vec{a}} (C', w' : \mathbf{skip})$ for any z .*

Proof. If a program with an iteration barrier terminates, then all loops bounded by the barrier must have naturally terminated (i.e. their loop conditions became false) before pausing at a barrier because $\xrightarrow{\vec{a}}$ does not permit resuming barriers. Thus we can substitute the barriers with any bound z without changing the execution. \blacksquare

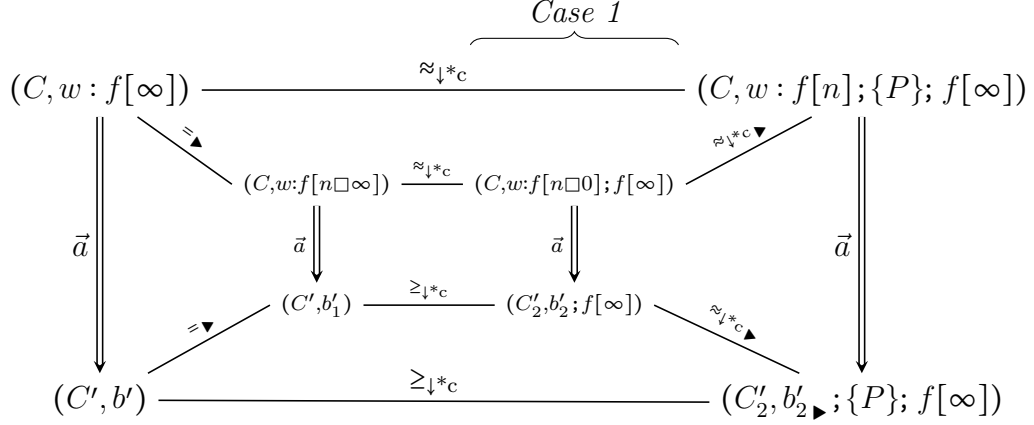
Theorem 4.1 (Loop decomposition; proof of proposition 4.1). *Given loop schema $(f, e, Q)_{\approx_{\downarrow^*c}}$, if $(C, w : \{P\}; f[\infty]) \xrightarrow{\vec{a}} (C', b')$, then $(C, w : \{P\}; f[n]) \xrightarrow{\vec{a}} (C'', b'')$ and $(C', b') \geq_{\downarrow^*c} (C'', b''; \{P\}; f[\infty])$ for some $n > 0$, C'' , and b'' .*

Proof. We assume that at least one step is taken (otherwise, this theorem trivially holds), so that the assertion holds: $C \# w \models P \star \mathbf{true}$. In our first attempt to prove loop decomposition, we let n be number of iterations that $f[\infty]$ executed and ran $f[n]$. This time, we make a slight adjustment by running $f[n \square 0]$ instead. First, we find n : by lemma 4.26, there exists $n > 0$ and b'_1 such that $(C, w : f[n \square \infty]) \xrightarrow{\vec{a}} (C', b'_1)$ and $b' = b'_1 \blacktriangleright$.

The combining transformation allows us to split off the ∞ iterations so that $(C, w : f[n \square \infty]) \approx_{\downarrow^*c} (C, w : f[n \square 0]; f[\infty])$; by the multistep inversion principle for sequential composition (lemma 4.8), there must exist a p' such that $(C, w : f[n \square 0]; f[\infty]) \xrightarrow{\vec{a}} p'$ and $(C', b'_1) \geq_{\downarrow^*c} p'$. By lemma 4.8, we can break the execution of $f[n \square 0]; f[\infty]$ into two cases:

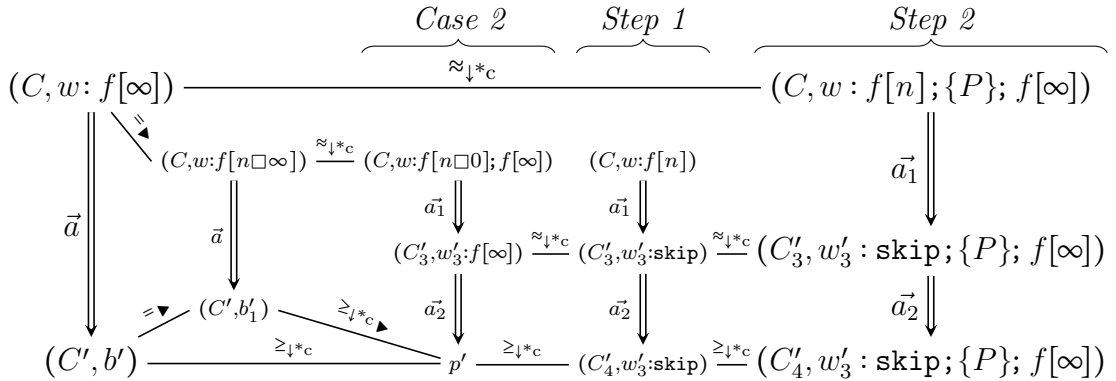
Case 1. Assume $(C, w : f[n \square 0]) \xrightarrow{\vec{a}} (C'_2, b'_2)$ and $p' = (C'_2, b'_2; f[\infty])$ for some C'_2 and b'_2 . We can perform the same execution after resuming the barriers such that $(C, w : f[n]) \xrightarrow{\vec{a}} (C'_2, b'_2 \blacktriangleright)$ (by lemma 4.27) and $(C', b') \geq_{\downarrow^*c} (C'_2, b'_2 \blacktriangleright; f[\infty])$ (by lemma 4.4).

By the loop invariant of f and lemma 4.9, we insert an assertion before $f[\infty]$ so that $(C'_2, b'_{2\blacktriangleright}; f[\infty]) \approx_{\downarrow^*c} (C'_2, b'_{2\blacktriangleright}; \{P\}; f[\infty])$. This theorem holds for n , C'_2 , and $b'_{2\blacktriangleright}$.




Case 2. Assume $(C, w : f[n \square 0]) \xrightarrow{\bar{a}_1} (C'_3, w'_3 : \mathbf{skip})$, $(C'_3, w'_3 : f[\infty]) \xrightarrow{\bar{a}_2} p'$, and $\bar{a} = \bar{a}_1 \cdot \bar{a}_2$ for some $\bar{a}_1, \bar{a}_2, C'_3, w'_3$.

Like before, the tricky case of falling through is not prevented by iteration barriers. The key difference we see by using iteration barriers, however, is what fall through now entails: if $f[n \square 0]$ terminates, then $f[\infty]$ must also terminate in the same way.



Step 1: By lemma 4.28, $(C, w : f[\infty]) \xrightarrow{\vec{a}_1} (C'_3, w'_3 : \text{skip})$. By the termination of $f[\infty]$, it must be the case that the loop condition has become false: $C'_3 \# w'_3 \models \neg e \wedge P \star \text{true}$. And because the loop condition is false, the remaining $f[\infty]$ is also (effectively) terminated: $(C'_3, w'_3 : f[\infty]) \approx_{\downarrow \star c} (C'_3, w'_3 : \text{skip})$. We cause $(C'_3, w'_3 : \text{skip})$ to match the behavior of $(C'_3, w'_3 : f[\infty])$ so that $(C'_3, w'_3 : \text{skip}) \Rightarrow (C'_4, w'_3 : \text{skip})$ and $p' \geq_{\downarrow \star c} (C'_4, w'_3 : \text{skip})$. (It is not necessarily the case that $C'_4 = C'_3$ and $\vec{a}_2 = []$ because observations may also be made on terminated processes via S-SendObs and S-RecvObs.) After removing the loop barrier, by lemma 4.27, we obtain an execution from $f[n]$ to $(C'_4, w'_3 : \text{skip})$.

Step 2: By sequentially composing the execution with $\{P\}; f[\infty]$, we get $(C, w : f[n]; \{P\}; f[\infty]) \xrightarrow{\vec{a}_1} (C'_3, w'_3 : \text{skip}; \{P\}; f[\infty]) \xrightarrow{\vec{a}_2} (C'_4, w'_3 : \text{skip}; f[\infty])$. The last step is to show $(C'_4, w'_3 : \text{skip}) \geq_{\downarrow \star c} (C'_4, w'_3 : \text{skip}; f[\infty])$, which follows by lemma 4.5 and $C'_3 \# w'_3 \models P \star \text{true}$.

Thus our theorem holds for n , C'_4 , and $w'_3 : \text{skip}$. 

Theorem 4.2 (Loop decomposition for bisimulation). *Given loop schema $(f, e, Q)_{\approx_{\downarrow \star}}$, if $(C, w : \{P\}; f[\infty]) \xrightarrow{\vec{a}} (C', b')$, then $(C, w : \{P\}; f[n]) \xrightarrow{\vec{a}} (C'', b'')$ and $(C', b') \approx_{\downarrow \star} (C'', b''; \{P\}; f[\infty])$ for some $n > 0$, C'' , and b'' .*

Proof. The proof is nearly identical to theorem 4.1. 

Lemma 4.29. *If $(f, e, P)_{\approx}$, $(g, e, P)_{\approx}$, and $\{P\}; f[1] \equiv_{\approx} \{P\}; g[1]$, then $\{P\}; f[n] \equiv_{\approx} \{P\}; g[n]$ for any $n > 0$ and $\approx \in \{\approx_{\downarrow \star}, \approx_{\downarrow \star c}\}$.*

Proof. By induction on n . The base case, $n = 0$, is a contradiction. The inductive case follows:

$$\begin{array}{ll}
\{P\}; f[1+n] \equiv_{\approx} \{P\}; g[1+n] & \text{assumption} \\
\{P\}; f[1]; f[n] \equiv_{\approx} \{P\}; g[1]; g[n] & \text{combining transformation} \\
\{P\}; f[1]; \{P\}; f[n] \equiv_{\approx} \{P\}; g[1]; \{P\}; g[n] & \text{loop invariant} \\
\{P\}; f[1]; \{P\}; f[n] \equiv_{\approx} \{P\}; f[1]; \{P\}; g[n] & \text{assumption} \\
\{P\}; f[1]; \{P\}; f[n] \equiv_{\approx} \{P\}; f[1]; \{P\}; f[n] & \text{inductive hypothesis} \quad \text{QED}
\end{array}$$

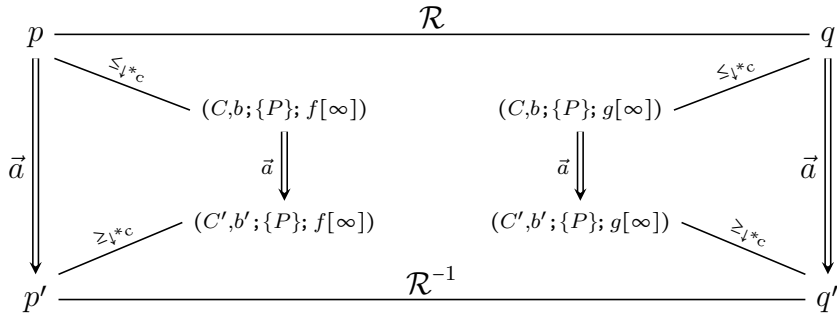
Theorem 4.3 (Loop folding). *If $(f, e, P)_{\approx_{\downarrow^*c}}$, $(g, e, P)_{\approx_{\downarrow^*c}}$, $\{P\}; f[1] \equiv_{\approx_{\downarrow^*c}} \{P\}; g[1]$, and $C \# w \models P \star \text{true}$, then $(C, w : f[\infty]) \leq_{\downarrow^*c} (C, w : g[\infty])$.*

Proof. We adapt the relation from eq. (4.4) by replacing bisimilarity with contrasimilarity and partial contrasimilarity:

$$\mathcal{R} = \{(p, q) \mid \exists f, g, C, b. (f, e, P)_{\approx_{\downarrow^*c}} \wedge (g, e, P)_{\approx_{\downarrow^*c}} \wedge \{P\}; f[1] \equiv_{\approx_{\downarrow^*c}} \{P\}; g[1] \wedge p \leq_{\downarrow^*c} (C, b; \{P\}; f[\infty]) \wedge (C, b; \{P\}; g[\infty]) \leq_{\downarrow^*c} q\}.$$

Contrasimulation: *Given $(p, q) \in \mathcal{R}$ and $p \xrightarrow{\vec{a}} p'$, show that there exists some q' such that $q \xrightarrow{\vec{a}} q'$ and $(p', q') \in \mathcal{R}^{-1}$. By the definition of \mathcal{R} , $p \leq_{\downarrow^*c} (C, b; \{P\}; f[\infty])$ and $q \leq_{\downarrow^*c} (C, b; \{P\}; g[\infty])$. We cause $b; \{P\}; f[\infty]$ to mimic p ; there are two cases.*

Case 1. Only b runs, so $b; \{P\}; g[\infty]$ (and thus q) trivially mimics p . Specifically, $(C, b) \xrightarrow{\vec{a}} (C', b')$ and $p' \geq_{\downarrow^*c} (C', b'; \{P\}; f[\infty])$. Thus $(C, b; \{P\}; g[\infty]) \xrightarrow{\vec{a}} (C', b'; \{P\}; g[\infty])$, $q \xrightarrow{\vec{a}} q'$, $(C', b'; \{P\}; g[\infty]) \geq_{\downarrow^*c} q'$, and $(p', q') \in \mathcal{R}^{-1}$ for some q' .

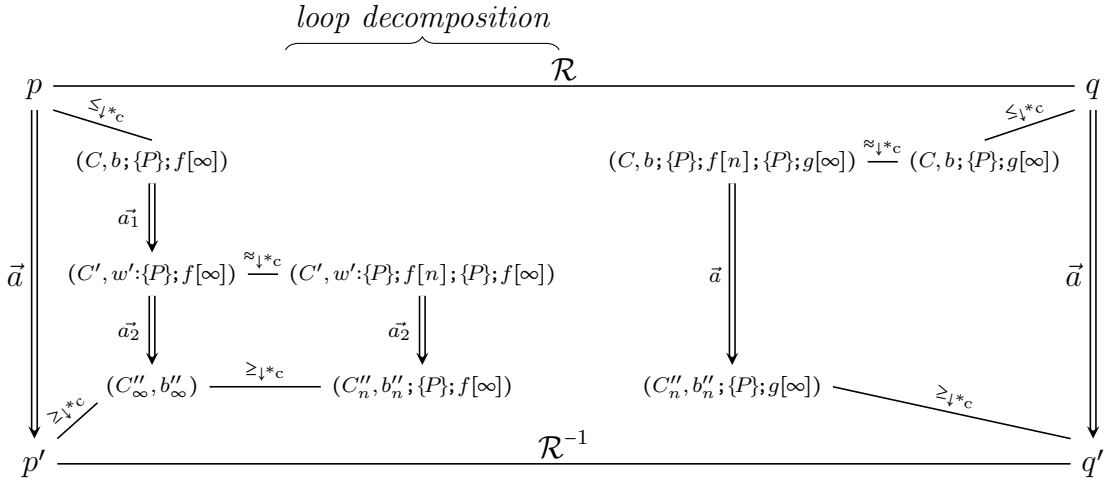


Case 2. b runs to termination, and then $\{P\}; f[\infty]$ runs. Specifically, $(C, b) \xrightarrow{\vec{a}_1} (C', w' : \text{skip})$, and then $(C', w' : \{P\}; f[\infty]) \xrightarrow{\vec{a}_2} (C''_n, b''_n)$, where $p' \geq_{\downarrow^*c} (C''_n, b''_n)$ and $\vec{a} = \vec{a}_1 \cdot \vec{a}_2$. We cannot directly compare the execution of $f[\infty]$ to $g[\infty]$, so we compare a finite number of executions by using loop decomposition (theorem 4.1) to find an $n > 0$ such that $(C', w' : \{P\}; f[n]) \xrightarrow{\vec{a}_2} (C''_n, b''_n)$ and $(C''_n, b''_n) \geq_{\downarrow^*c} (C''_n, b''_n; \{P\}; f[\infty])$.

Now we construct an execution of q that mimics p . We append $\{P\}; g[\infty]$ to the execution of b and $f[n]$ to obtain $(C, b; \{P\}; f[n]; \{P\}; g[\infty]) \xrightarrow{\vec{a}} (C''_n, b''_n; \{P\}; g[\infty])$. By lemma 4.29, the loop invariant of g , and the combining transformation of g , state q partially contrasimulates $(C, b; \{P\}; f[n]; \{P\}; g[\infty])$:

$$\begin{aligned}
(C, b; \{P\}; f[n]; \{P\}; g[\infty]) &\leq_{\downarrow *c} (C, b; \{P\}; g[n]; \{P\}; g[\infty]) \\
&\leq_{\downarrow *c} (C, b; \{P\}; g[n]; g[\infty]) \\
&\leq_{\downarrow *c} (C, b; \{P\}; g[\infty]) \\
&\leq_{\downarrow *c} q.
\end{aligned}$$

Thus there exists a q' such that $q \xrightarrow{\vec{a}} q'$, $(C''_n, b''_n; \{P\}; g[\infty]) \geq_{\downarrow *c} q'$, and $(p', q') \in \mathcal{R}^{-1}$.

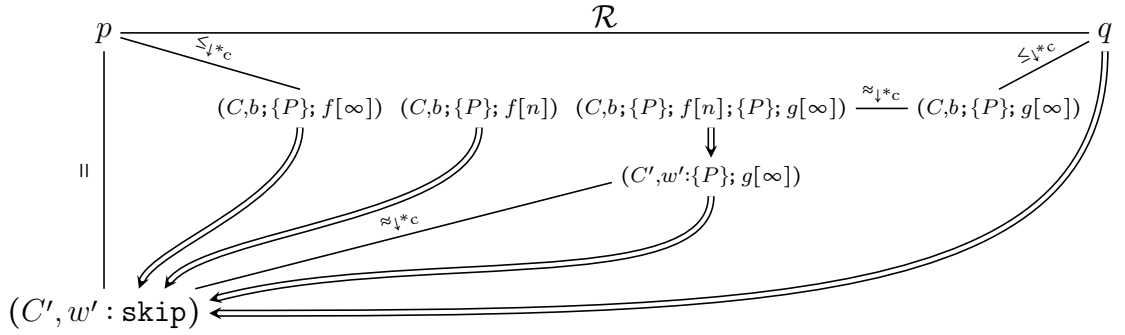


Compatibility with \blacktriangleright : Follows directly by lemmas 4.4 and 4.10.

One-way termination sensitive: Given $(p, q) \in \mathcal{R}$ and $p = (C', w' : \text{skip})$, show that $q \Rightarrow (C', w' : \text{skip})$. From the definition of \mathcal{R} , $p \leq_{\downarrow *c} (C, b; \{P\}; f[\infty])$ and $(C, b; \{P\}; g[\infty]) \leq_{\downarrow *c} q$.

By lemma 4.6 (termination sensitivity), $(C, b; \{P\}; f[\infty])$ also terminates in state $(C', w' : \text{skip})$. Because $f[\infty]$ terminates, there must exist an n for which $f[n]$ will terminate in the same state (lemmas 4.26 and 4.28): $(C, b; \{P\}; f[n]) \Rightarrow (C', w' : \text{skip})$. Furthermore, e must have become false and, by the loop invariant: $C \# w \models \neg e \wedge P \star \text{true}$.

We append $g[\infty]$ to the execution of b and $f[n]$ so that $(C, b; \{P\}; f[n]; g[\infty]) \Rightarrow (C', w' : \{P\}; g[\infty])$. Because e is false under w' , $(C', w' : \{P\}; g[\infty])$ is equivalent to $(C', w' : \text{skip})$ by the termination property of g ; thus $(C', w' : \{P\}; g[\infty])$ also converges to state $(C', w' : \text{skip})$ (by lemma 4.6). Finally, q will also converge to the same halted state by lemma 4.7. \blacksquare



Corollary 4.2 (Contrasimulation of loop folding). *If $(f, e, P)_{\approx_{\downarrow}^* c}$, $(g, e, P)_{\approx_{\downarrow}^* c}$, $\{P\}; f[1] \equiv_{\approx_{\downarrow}^* c} \{P\}; g[1]$, and $C \# w \models P \star \text{true}$, then $(C, w : f[\infty])_{\approx_{\downarrow}^* c} (C, w : g[\infty])$.*

Proof. Follows by theorem 4.3 and lemma 4.1. \blacksquare

Theorem 4.4 (Bisimulation of loop folding). *If $(f, e, P)_{\approx_{\downarrow}^*}$, $(g, e, P)_{\approx_{\downarrow}^*}$, $\{P\}; f[1] \equiv_{\approx_{\downarrow}^*} \{P\}; g[1]$, and $C \# w \models P \star \text{true}$, then $(C, w : f[\infty])_{\approx_{\downarrow}^*} (C, w : g[\infty])$.*

Proof. The proof is nearly identical to theorem 4.3 (but uses theorem 4.2 for loop decomposition). \blacksquare

Corollary 4.3 (Soundness of T-FoldLoop). *If*

- $\{P\}; f[1] \equiv_{\approx_{\downarrow *c}} \{P\}; f[1]; \{P\};$
- $\forall u. \{\neg e \wedge P\}; f[u] \equiv_{\approx_{\downarrow *c}} \{\neg e \wedge P\};$
- $\{P\}; f[\infty] \equiv_{\approx_{\downarrow *c}} \{P\}; f[\infty]; \{\neg e \wedge P\};$ and
- $\forall n \geq 1, u. \{P\}; f[n \square 0]; f[u] \equiv_{\approx_{\downarrow *c}} \{P\}; f[n \square u],$


then $\{P\}; \mathbf{while} \ e \ \mathbf{do} \ f[1] \equiv_{\approx_{\downarrow *c}} \{P\}; f[\infty].$

Proof. By definition 43, our goal is to prove $(C, w : \{P\}; \mathbf{while} \ e \ \mathbf{do} \ f[1]) \approx_{\downarrow *c} (C, w : \{P\}; f[\infty]).$ We create a second loop schema, $(g, e, P)_{\approx_{\downarrow *c}}$ where

$$g = \{P\}; \mathbf{while} \ e \ \mathbf{max} \ \square \ \mathbf{do} \ f[1]$$

by lemma 4.25. The rest follows by theorem 4.3. 

Corollary 4.4. *We can prove a slightly weaker version of T-While (for unbounded loops) by loop folding. If $e_1 \dashv\vdash e_2$ and $t_1 \equiv_{\approx_{\downarrow *c}} t_2$, then $\mathbf{while} \ e_1 \ \mathbf{do} \ t_1 \equiv_{\approx_{\downarrow *c}} \mathbf{while} \ e_2 \ \mathbf{do} \ t_2.$*

Proof. Follows by theorem 4.3 and lemma 4.25 for $f = \{\mathbf{true}\}; \mathbf{while} \ e_1 \ \mathbf{max} \ \square \ \mathbf{do} \ t_1$ and $g = \{\mathbf{true}\}; \mathbf{while} \ e_2 \ \mathbf{max} \ \square \ \mathbf{do} \ t_2.$ 

4.5 Parallelization

We will not go into much detail explaining why the parallelization transformation, T-SeqPar, is sound for the imperative language we present in this chapter. The interesting and challenging concepts behind the proof have already been explored in chapter 3, and we also refer the reader to our Coq proofs² for even more detail.

Between the proof in Section 3.5 and here, the main differences are due to the model of observation. In particular, we use resources (i.e. permissions on channel

²<http://www.cs.princeton.edu/~cbell/par>

endpoints) to delineate between what can and cannot be observed rather than a syntactic mechanism to restrict the scope of a channel. However, when checking if an observation can be made on a channel, there is little meaningful difference between doing so at the site of channel scope restriction (a la CCS-Seq) or parallel composition (via S-ParL and S-ParR). Other differences between CCS-Seq and our imperative language do not have much impact on the proofs.

Definition 49 (Commitment to termination). We write $p \Downarrow$ when for any \vec{a} and p' , $p \xrightarrow{\vec{a}} p'$ implies $p' \Rightarrow (C'', w'' : \mathbf{skip})$ for some C'' and w'' . In other words, any state derived from p can terminate; p cannot become stuck, but this does not preclude divergence.

Definition 50 (Termination entailment). We write $[P_1] t_1 \Downarrow [P_2] t_2$ when for any \vec{a} , C' , w'_1 , and b'_2 , if the left thread terminates via $(C, w : [P_1] t_1 \parallel [P_2] t_2) \xrightarrow{\vec{a}} (C', w'_1 : \mathbf{skip} \parallel b'_2)$, then the right thread can also terminate: $(C', w'_1 : \mathbf{skip} \parallel b'_2) \Downarrow$. In other words, as t_1 and t_2 run in parallel (while possibly communicating), if one terminates then so may the other.

Definition 51 (Termination entailment of threads). We write $b_1 \Downarrow^C b_2$ when for any \vec{a} , C' , w'_1 , and b'_2 such that $\mathbf{worlds-of}(b_1) \oplus \mathbf{worlds-of}(b_2)$ is defined, if the left thread terminates via $(C, w : b_1 \parallel b_2) \xrightarrow{\vec{a}} (C', w'_1 : \mathbf{skip} \parallel b'_2)$, then the right thread can also terminate: $(C', w'_1 : \mathbf{skip} \parallel b'_2) \Downarrow$.

Definition 52 (Cotermination). We write $[P_1] t_1 \Downarrow\Downarrow [P_2] t_2$ when $[P_1] t_1 \Downarrow [P_2] t_2$ and $[P_2] t_2 \Downarrow [P_1] t_1$.

Definition 53. State p ensures that P holds upon termination, written $p \Downarrow P$, if for any $p \xrightarrow{\vec{a}} (C', w' : \mathbf{skip})$, it is the case that $C' \# w' \models P \star \mathbf{true}$. We write $b \Downarrow P$ when $(C, b) \Downarrow P$ for all C .

Lemma 4.30. *If $(C_1, b_1) \leq_{\downarrow \star c} (C_2, b_2; \{P\})$, then $(C_1, b_1) \Downarrow P$.*

Proof. If (C_1, b_1) terminates, then by termination-sensitivity, $(C_2, b_2; \{P\})$ must terminate in the same state; this ensures that P will hold in the final state. \blacksquare

Lemma 4.31. *If $(C, b) \Downarrow P$, then $(C, b) \approx_{\downarrow^*} (C, b; \{P\})$.*

Proof. We show that $\{(p, q) \mid \exists C, b. p = (C, b) \wedge q = (C, b; \{P\}) \wedge (C, b) \Downarrow P\} \cup \approx_{\downarrow^*}$ is a termination sensitive bisimulation that is compatible with \blacktriangleright . The interesting case is when $p = (C, w : \text{skip})$ and $q = (C, w : \text{skip}; \{P\})$, where $(C, w : \text{skip}) \Downarrow P$ ensures $C \# w \vDash P \star \text{true}$ and thus $(C, w : \text{skip}) \approx_{\downarrow^*} (C, w : \{P\})$. \blacksquare

Theorem 4.5. *If $(C, (b_1 \parallel b_2); [P_3] t_3 \parallel [P_4] t_4) \xrightarrow{a} (C', b')$, then either*

- *one of the threads takes a step: there exists b'_1 and b'_2 such that $b' = (b'_1 \parallel b'_2); [P_3] t_3 \parallel [P_4] t_4$ and $(C, b_1 \parallel b_2) \xrightarrow{a} (C', b'_1 \parallel b'_2)$; or*
- *the threads join: there exists w_1, w_2 , and w such that $b_1 = w_1 : \text{skip}$, $b_2 = w_2 : \text{skip}$, $w_1 \oplus w_2 = w$, $C' = C$, and $b' = w : \text{skip}; [P_3] t_3 \parallel [P_4] t_4$.*

Theorem 4.6. *If $(C, (b_1; t_3) \parallel (b_2; t_4)) \xrightarrow{\bar{a}} (C', b')$, and $b_1 \Downarrow P_3$; $b_2 \Downarrow P_4$; $b_1 \Downarrow^C b_2$; $b_2 \Downarrow^C b_1$; $\text{recvs}(b_1; t_3) \cap \text{recvs}(b_2; t_4) = \emptyset$; $\text{writes}(b_1; t_3) \cap \text{freevars}(b_2; t_4) = \emptyset$; and $\text{freevars}(b_1; t_3) \cap \text{writes}(b_2; t_4) = \emptyset$, then either*

- *$(C, b_1 \parallel b_2) \xrightarrow{\bar{a}} (C', b'_1 \parallel b'_2)$ and $b' = (b'_1; t_3) \parallel (b'_2; t_4)$ for some b'_1 and b'_2 ; or*
- *$(C', b') \Rightarrow p''$ and $(C', (b_1 \parallel b_2); [P_3] t_3 \parallel [P_4] t_4) \xrightarrow{\bar{a}} p''$ for some p'' .*

Theorem 4.7 (Soundness of T-SeqPar). *If*

- *t_1 and t_2 terminate together:*
 $[P_1] t_1 \Updownarrow [P_2] t_2$;
- *the right threads can only communicate with the left:*
 $P_1 \star P_2 \vdash \text{owns}(\text{freechans}(t_2; t_4)) \star \text{true}$;
- *the left and right threads do not attempt to receive from the same channel:*
 $\text{recvs}(t_1; t_3) \cap \text{recvs}(t_2; t_4) = \emptyset$;
- *only read-only variables are shared:*
 $\text{writes}(t_1; t_3) \cap \text{freevars}(t_2; t_4) = \emptyset$ and $\text{freevars}(t_1; t_3) \cap \text{writes}(t_2; t_4) = \emptyset$; and

- the termination of t_1 and t_2 respectively satisfies P_3 and P_4 :

$$\{P_1\}; t_1 \equiv_{\approx_{\downarrow}^*} \{P_1\}; t_1; \{P_3\} \text{ and } \{P_2\}; t_2 \equiv_{\approx_{\downarrow}^*} \{P_2\}; t_2; \{P_4\},$$

then $([P_1] t_1 \parallel [P_2] t_2); ([P_3] t_3 \parallel [P_4] t_4) \equiv_{\approx_{\downarrow}^*} [P_1] (t_1; t_3) \parallel [P_2] (t_2; t_4)$.

4.6 Hoare triples

Many of the rewrite rules in our framework are guarded by assertions that help ensure the transformation preserves the program behavior. Although not a focus of this thesis, making effective use of our framework to construct correct optimizations requires a significant amount of program analysis.

It is possible to perform this static analysis at the level of rewrite rules; table 2.1 lists a selection of rules that would support this. Our framework also supports a Hoare logic analysis, which is advantageous because it is well known and is used by many existing static analyses tools.

The following is how a Hoare triple (that respects the frame rule of separation logic) could be defined in our system.


Definition 54 (Hoare triple). $\vdash\{P\} t \{Q\}$ iff for all C , w , and F , if $C\#w \vDash P \star F$ and $(C, w : t) \xRightarrow{\vec{a}} (C', b')$, then either:

- $(C', b') \rightarrow (C'', b'')$ for some C'' and b'' , *(safety)*
- or $b' = w' : \text{skip}$ and $C'\#w' \vDash Q \star F$. *(the postcondition holds)*

Lemma 4.32. *If $\vdash\{P\} t \{Q\}$, then $\{P \star F\}; t \equiv_{\approx_{\downarrow}^*} [P] t \parallel [F] \text{skip}$.*

Lemma 4.33. *If $\vdash\{P\} t \{Q\}$ and $C\#w \vDash P \star \text{true}$, then $(C, w : t) \Downarrow Q$.*

Lemma 4.34 (Soundness of T-Hoare). *If $\vdash\{P\} t \{Q\}$, then $\{P\}; t \equiv_{\approx_{\downarrow}^*} \{P\}; t; \{Q\}$.*

Proof. Definition 53 is directly implied by $\vdash\{P\} t \{Q\}$ (lemma 4.33). This lemma thus holds by lemmas 4.3 and 4.31. 

It is clear to see that definition 54 implies T-Hoare. In fact, we could substantially weaken definition 54 by not requiring safety, and T-Hoare would still be sound.

4.7 Coq Proof Development

Our initial attempts to prove loop parallelization targeted a richer imperative language than we present in this thesis; it included a shared heap and first class channels (with a concurrent separation logic for passing resources across channels). In this initial proof development, we mechanically proved parallelization and loop folding, in addition to many rewrite rules, in the Coq Proof Assistant. (But loop folding was only proved with respect to bisimulation in this setting.)

However, the shared heap and first class channels complicated many of our proofs, while having little interesting impact on the correctness of loop parallelization. Furthermore, we had not developed a strong theory behind parallelization at this point, and thus our definition of program equivalence was ad-hoc and growing more difficult to work with. Thus our next project was to focus on just parallelization in order to understand why bisimulation did not hold and whether the definition we had come up with was a suitably strong notion of equivalence. Our results from this effort were presented in chapter 3.

The language and other definitions presented in this chapter are the result of refocusing our previous work in order to more narrowly concentrate on loop parallelization, which has resulted in a new (and much cleaner) Coq proof development. The main differences are the removal of a shared heap and first class channels, and a simplification of the definition of program equivalence, which uses the proof development from chapter 3. We also reformulated the operational semantics as a labeled transition system in order to more cleanly align with the bisimulation theory.

In this new setting, we have mechanically proved the soundness of loop folding with respect to both bisimulation and contrasimulation, in addition to numerous other rewrite rules. However, we have not had the time to port the proof of parallelization to this setting, so the results Section 4.5 are backed only by our original Coq proof development and by chapter 3. We do not expect any significant challenges in porting the proof.

Chapter 5

Conclusion & Future Work

5.1 Concluding remarks

In this thesis, we have explored the theory of loop parallelization as performed by optimizations such as DOALL, DOACROSS, and DSWP. Our two key contributions are the development and proof of soundness for a loop folding transformation and parallelization, which work together to enable loop parallelization. To demonstrate the effectiveness of these results, we created a framework of rewrite rules and used them to implement instances of DOALL, DSWP, and DOACROSS. Most of this work has been formalized and proved in the Coq Proof Assistant.

5.1.1 Loop folding

Our loop folding transformation converts a program with arbitrary structure, but which *behaves* like a single loop, into a `while` loop. Unlike traditional loop transformations, loop folding

- is not restricted to `while` loops,
- does not assume that the loop is counted (like a “for” loop),
- does not assume termination (it may be infinite or nondeterministic),

- uses extensional reasoning (e.g., it does not require the program to have a particular syntax), and
- is compositional – the side conditions are written as smaller transformations.

To describe a program that “behaves like a loop”, the user provides a program that is parameterized by a bound on the maximum number of iterations that it may execute, a loop condition, and a loop invariant. Then the user proves four properties, which are stated as transformations:

- if the loop condition is false, then the program does nothing and terminates;
- if the unbounded program terminates, then the loop condition will be false;
- if the loop invariant holds, then it will continue to hold after any number of iterations are run; and
- if the program appears in sequence with itself, each bounded by different iterations, then the programs can be combined into one that is bounded by the sum of the two iterations.

We have mechanically proved the loop folding transformation sound with respect to both (weak) bisimilarity and contrasimilarity. The main challenge was that we needed to reason about the behavior of a program that is parameterized by a finite bound of iterations, but without allowing the equivalence relation to assume that the bound is finite. In response, we developed iteration barriers, which are a special bound on the number of iterations that a loop may execute before becoming stuck. Then, we required our equivalence relation to be compatible with \blacktriangleright , so that every equivalence must continue to hold after removing the iteration barrier.

5.1.2 Parallelization

Our second contribution is the development and proof of soundness of a parallelizing transformation. The statement of the transformation is unique in that it targets a

sequential composition of two parallel programs, then combines them together – this allows it to be used with loop folding.

While developing the parallelization transformation, we discovered that it does not hold for any of the bisimulation relations that have been used with verified compilers, nor any other well known variation of bisimulation in the presence of internal choice. In response, we created a model semantics based on CSS to study the issue and develop an equivalence relation that does work. Our first result was eventual simulation, which holds for parallelization but cannot be made both transitive and symmetric in a straightforward manner. Then we discovered an obscure relation, called *contrasimulation*, that is implied by eventual simulation, is a congruence for the imperative language we are studying, and yet maintains the strong properties that make bisimulation relations desirable to use. Without internal choice, *contrasimulation* is equivalent to bisimulation.

5.1.3 Loop parallelization

Our loop folding and parallelization transformations compose together to create a loop-parallelizing transformation. To demonstrate its effectiveness, we wrote down a set of high-level rewrite rules to serve as an “API” for transforming programs, with the goal in mind of using these transformations to implement various [parallelizing] optimizations, such as DOALL, DOACROSS, and DSWP.

By proving each rule in this framework sound with respect to *contrasimulation* (which we did for many of the rules, particularly loop folding and parallelization; see Section 2.6 for a summary), any optimization implemented using the framework will be correct by construction. Using rewrite rules on high level program syntax in order to implement optimizations has been done before, but it is still an unusual approach: existing implementations of automatic DOALL, DOACROSS, and DSWP optimizations work on control flow graphs instead.

Working at such a high level allowed us to focus on developing the theory and proofs. We believe the results of this thesis can be applied to a lower level, unstructured, language (like an assembly language, LLVM bytecode, or control flow graphs). On the other hand, targeting a high level language makes the framework more accessible to users, raising the possibility of allowing programmers to interact with the compiler to correctly optimize their programs by hand.

5.1.4 Coq proofs

Most of our proofs have been mechanically check by the Coq Proof Assistant. Sections 2.6, 3.6.1 and 4.7 discuss these results in further detail.

5.2 Future work

5.2.1 Divergence sensitivity

We would like to further strengthen our results by incorporating Dockins’ and van Glabbeek’s formulation of divergence sensitivity [11, 42]. This would be particularly important for loop folding because it deals with potentially diverging loops; we need to be careful not to introduce or remove such behavior. Adapting their definition from branching bisimulations to weak bisimulations by replacing \rightarrow with \Rightarrow^+ , the definition of divergence sensitivity follows:

Definition 55 (Divergence). \mathcal{D} is a *divergence* when for any $p \in \mathcal{D}$,

- $p \Rightarrow^+ p'$ and $p' \in \mathcal{D}$ for some p' .

Program p *diverges* if there exists a divergence \mathcal{D} such that $p \in \mathcal{D}$.

Definition 56 (Divergence sensitivity). \mathcal{R} is one-way divergence sensitive iff for any $(p, q) \in \mathcal{R}$,

- if $p \in \mathcal{D}_p$ and \mathcal{D}_p is a divergence, then there exists a divergence \mathcal{D}_q such that $q \in \mathcal{D}_q$ and $\forall q' \in \mathcal{D}_q. \exists p' \in \mathcal{D}_p. (p', q') \in \mathcal{R}$.

Our definition of program equivalence with respect to contrasimilarity (definitions 43, 46 and 47) would then be adjusted to require that \mathcal{R} and \mathcal{R}^{-1} also be one-way divergence sensitive.

We have already made some progress in this direction: many of the simpler rewrite rules have been proved sound with respect to divergence sensitivity. We have also informally proved divergence sensitivity (by hand) for loop folding. However, it is a subtle proof that warrants mechanizing, which we have not yet completed.

Furthermore, it is not yet clear whether definition 56 holds for parallelization, or whether it is fully compatible with contrasimulation. We may need to adjust it so that, like contrasimulation, the relation (\mathcal{R}) is flipped.

5.2.2 Incorporation into an existing verified compiler

The first step to incorporating the results of this thesis into any compiler will be to implement an automatic loop parallelizing optimization that uses our framework. Automation is discussed further in Section 2.5.7.

Our choice to use a high-level structured language in this thesis could make combining our results with an existing verified compiler quicker and easier by allowing us to target the front end of the compiler. After applying all parallelizing optimizations, we would then translate the language into a concurrent language that is already supported by a verified compiler. Two candidates are Concurrent C Minor (VST) [1, 40] and CompCertTSO [40], which have verified compilers based on CompCert. The last step would be to verify the translation with respect to the verified compiler's definition of correctness, and then compose this definition with contrasimulation. However, composing the two correctness criteria would likely be nontrivial.

Bibliography

- [1] Andrew W. Appel. Verified software toolchain. In *Proceedings of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software*, ESOP'11/ETAPS'11, pages 1–17, Berlin, Heidelberg, 2011. Springer-Verlag.
- [2] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A. Padua. Automatic program parallelization, 1993.
- [3] Christian J. Bell. Certifiably sound parallelizing transformations. In *Certified Programs and Proofs*, volume 8307 of *Lecture Notes in Computer Science*, pages 227–242. Springer, 2013.
- [4] Christian J. Bell, Andrew W. Appel, and David Walker. Concurrent separation logic for pipelined parallelization. In *Proceedings of the 17th International Conference on Static Analysis*, SAS'10, pages 151–166, Berlin, Heidelberg, 2010. Springer-Verlag.
- [5] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 259–270, New York, NY, USA, 2005. ACM.
- [6] Richard Bornat, Cristiano Calcagno, and Hongseok Yang. Variables as resource in separation logic. *Electron. Notes Theor. Comput. Sci.*, 155:247–276, May 2006.
- [7] Matko Botinčan, Mike Dodds, and Suresh Jagannathan. Proof-directed parallelization synthesis by separation logic. *TOPLAS*, 2013. To appear in TOPLAS.
- [8] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science*, LICS '07, pages 366–378, Washington, DC, USA, 2007. IEEE Computer Society.
- [9] Adam Chlipala. The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, ICFP '13, pages 391–402, New York, NY, USA, 2013. ACM.

- [10] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pages 161–177, Berlin, Heidelberg, 2009. Springer-Verlag.
- [11] Robert W. Dockins. *Operational Refinement for Compiler Correctness*. PhD thesis, Princeton University, 35 Olden Street, Princeton NJ 08540-5233, September 2012.
- [12] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.
- [13] Rob J. van Glabbeek. The linear time - branching time spectrum II. In *Proceedings of the 4th International Conference on Concurrency Theory*, CONCUR '93, pages 66–81, London, UK, UK, 1993. Springer-Verlag.
- [14] Tony Hoare and Peter O’Hearn. Separation logic semantics for communicating processes. *Electron. Notes Theor. Comput. Sci.*, 212:3–25, April 2008.
- [15] Susan Horwitz, Jan Prins, and Thomas Reps. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 146–157, New York, NY, USA, 1988. ACM.
- [16] Clément Hurlin. Automatic parallelization and optimization of programs by proof rewriting. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 52–68, Berlin, Heidelberg, 2009. Springer-Verlag.
- [17] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
- [18] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 327–337, New York, NY, USA, 2009. ACM.
- [19] Kim G. Larsen and Arne Skou. Bisimulation through probabilistic testing. In *in Conference Record of the 16th ACM Symposium on Principles of Programming Languages (POPL)*, pages 344–352, 1989.
- [20] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
- [21] The Coq development team. The Coq Proof Assistant. <http://coq.inria.fr/>, 2013.

- [22] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.
- [23] J Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5:461–492, 1989.
- [24] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM.
- [25] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 83–94, New York, NY, USA, 2000. ACM.
- [26] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 268–280, New York, NY, USA, 2004. ACM.
- [27] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture*, November 2005.
- [28] Guilherme Ottoni, Ram Rangan, Neil Vachharajani, and David I. August. Decoupled software pipelining: A promising technique to exploit thread level parallelism. In *Proceedings of the 4th Workshop on Explicitly Parallel Instruction Computing Techniques*, March 2005.
- [29] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.
- [30] Joachim Parrow and Peter Sjödin. The complete axiomatization of cs-congruence. In *Proceedings of STACS 94*, number 775 in Lecture notes in computer science, pages 557–568. Springer-Verlag, 1994.
- [31] Larry Paulson, Nipkow Tobias, and Makarius Wenzel. Isabelle. <http://isabelle.in.tum.de/>, 2013.
- [32] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, pages 151–166, London, UK, UK, 1998. Springer-Verlag.
- [33] Prakash Prabhu, Soumyadeep Ghosh, Yun Zhang, Nick P. Johnson, and David I. August. Commutative set: A language extension for implicit parallel programming. *SIGPLAN Not.*, 46(6):1–11, June 2011.

- [34] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 114–123, New York, NY, USA, April 2008. ACM.
- [35] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.
- [36] Davide Sangiorgi. *An introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.
- [37] Vivek Sarkar. A concurrent execution semantics for parallel program graphs and program dependence graphs. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 16–30, London, UK, UK, 1993. Springer-Verlag.
- [38] Leonardo Scandolo, César Kunz, and Manuel Hermenegildo. Program parallelization using synchronized pipelining. In *Proceedings of the 19th international conference on Logic-Based Program Synthesis and Transformation*, LOPSTR'09, pages 173–187, Berlin, Heidelberg, 2010. Springer-Verlag.
- [39] Roberto Segala and Nancy Lynch. Probabilistic simulations for probabilistic processes. *Nordic J. of Computing*, 2(2):250–273, June 1995.
- [40] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Relaxed-memory concurrency and verified compilation. *SIGPLAN Not.*, 46(1):43–54, January 2011.
- [41] Zachary Tatlock and Sorin Lerner. Bringing extensibility to verified compilers. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, volume 45 of *PLDI '10*, pages 111–121, New York, NY, USA, June 2010. ACM.
- [42] Rob J. van Glabbeek, Bas Luttik, and Nikola Trčka. Branching bisimilarity with explicit divergence. *Fundam. Inform.*, 93(4):371–392, 2009.
- [43] Jules Villard, Étienne Lozes, and Cristiano Calcagno. Proving copyless message passing. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pages 194–209, Berlin, Heidelberg, 2009. Springer-Verlag.
- [44] Marc Voorhoeve and Sjouke Mauw. Impossible futures and determinism. *Inf. Process. Lett.*, 80(1):51–58, October 2001.
- [45] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

- [46] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. *SIGPLAN Not.*, 47(1):427–440, January 2012.