FORMAL VERIFICATION OF PROGRAMS AND THEIR TRANSFORMATIONS

by

Guodong Li

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

December 2010

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Guodong Li**

has been approved by the following supervisory committee members:

Ganesh Gopalakrishnan	, Chair	**Sept. 9, 2010** Date Approved
Konrad Slind	, Member	**Aug. 22, 2010** Date Approved
Mary Hall	, Member	**Sept. 10, 2010** Date Approved
Matthew Flatt	, Member	**Sept. 13, 2010** Date Approved
John Regehr	, Member	**Sept. 13, 2010** Date Approved

and by **Martin Berzins**, Chair of the Department of **School of Computing**

and by Charles A. Wight, Dean of The Graduate School.

ABSTRACT

Formal verification is an act of using formal methods to check the correctness of intended programs. The verification is done by providing a formal proof on an abstract mathematical model of the program, with respect to a certain formal specification or property.

We present three case studies on using formal methods to verify programs and their transformations: (1) we use term rewriting and theorem proving to construct and validate a compiler from logic specifications to ARM assembly code; the equivalence of a source specification and the generated assembly code is proven mechanically with respect to the formal semantics; (2) we model, in an "executable" declarative language TLA+, the Message Passing Interface (MPI) 2.0 library as well as C programs using MPI calls for parallel computations; and use explicit model checking to check the specifications and programs; and (3) we model CUDA kernel programs as symbolic logical formulas, and use constraint solving to automatically reason about these Graphics Processing Unit (GPU) kernels.

We have built a couple of unique verification tools to check intrinsic properties (*e.g.* race freedom for concurrent programs and translation correctness for compilers) and user-defined properties (*e.g.* functional correctness). Specifically, the presented compiler is the first trusted compiler translating logic specifications embedded in a theorem prover to low-level code; the MPI specification is the first attempt to provide executable semantics for a comprehensive set of message passing Application Programming Interfaces (APIs); and the CUDA verifier is the only existing formal symbolic checker for GPU kernel programs.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Professor Ganesh Gopalakrishnan. Without his guidance and continued support, my dissertation would not have been possible. He led me to the area of formal verification and provided continuous support for my research. His enthusiasm on a variety of exciting and challenging topics kept me moving ahead. He recommended many good materials to me and kept track of my progress. He helped me establish connections to the formal methods community. With his strong recommendation, I was able to work with researchers in some world-renowned research labs as a summer intern, including Fujitsu Labs of America and NEC Labs of America.

I am deeply indebted to my previous advisor, Dr. Konrad Slind, for his contribution in the development of my research. He taught me a lot about theorem proving, especially how to master the HOL theorem prover. He also introduced me to an exciting project about constructing and verifying a compiler for logic specifications in HOL. This project constitutes one of the main parts of this thesis.

I also want to thank my supervisory committee members: Professors John Regehr, Matthew Flatt, and Mary Hall. They helped me to improve my research using their expertise in embedded systems, functional languages, parallel programming, and compilers. Their insightful comments helped me to find the right research directions and enabled me to have a broad vision of the studied problems.

Special thanks to Sreeranga P. Rajan, Indradeep Ghosh, Oksana Tkachuk, Aarti Gupta, Nishant Sinha, and Chao Wang. My collaboration with them during my terrific internships with them strengthened my background on model checking, symbolic analysis, and system software development.

Many thanks to my labmates: Robert Palmer, Subodh Sharma, Alan Humphrey, and other members in the Formal Verification group. Many thanks to my classmates and friends: Junxing Zhang, Lu Zhao, Jianjun Duan, and Yu Yang. I really enjoyed and benefited from the discussions with them. I really appreciate Lu Zhao's effort on proof reading this thesis. I am deeply grateful to Karen Feinauer.

She patiently and meticulously handled all my paperwork. It is a blessing to have her as the graduate coordinator during my Ph.D. career.

CHAPTER 1

INTRODUCTION

Formal verification is an act of using formal methods to check the correctness of intended programs. The verification is done by providing a formal proof on an abstract mathematical model of the program, with respect to a certain formal specification or property. After the formal model of a program is built, we can validate a variety of properties over the model.

Users often define certain properties to express the correctness of their programs. For example, the user can specify the functional correctness of a sorting program to be "the elements in the output array are sorted." For a cryptographic program performing encryption and decryption, the user may want to make sure that the decryption of the ciphertext will return the original plaintext.

Besides, some correctness properties are *intrinsic* to the programs. For example, data races are usually considered errors for concurrent programs running on shared memory multiprocessors; barrier mismatches will lead to deadlocks in Single Program Multiple Data (SPMD) programs.

We present three formal verification tools which check both intrinsic and user-defined properties for sequential and concurrent programs. In particular, we are interested in checking the correctness of *program translation and transformation*. In *program translation*, a program written in one language is translated into another language. For example, a high-level language program is translated into a low-level format ready for execution by a compiler. During this process, the program may be simplified to a canonical form which is easier to analyze and optimize. In *program transformation*, a program is converted to another in the same language. A typical example is a source-to-source transformation, and a typical reason for transformation is optimization.

Formally verifying a program semantically equivalent to another is a very difficult task in general, because a program may be processed by a number of translations and transformations, resulting in a significantly different format. For

example, the resultant program may not have the same variables and structures as the original program, or the resultant program uses many threads and can run in parallel while the original program is sequential.

We want to ensure that the resultant program is *provably* equivalent to the original program in its semantics, and this thesis presents some techniques to attack this problem in different settings ranging from developing a compiler that compiles a simple yet expressive programming language into ARM assembly code with theorems justifying the correctness of compilation to verifying the correctness of practical C parallel programs.

1.1 Compiling Sequential Functional Programs

In imperative languages such as C and Java, the meaning of language constructs depends heavily on the state of computation. Since the state is only exactly known at runtime, it is nearly impossible to apply equational reasoning to parts of programs. In contrast, (pure) functional programs are regarded as mathematical functions whose meaning is independent of runtime states. Therefore, it is possible to apply equational rewriting and reasoning to them. We aim at compiling a language embedded in the HOL theorem prover [44]—the specification language of HOL, which is a ML-style pure functional language, into ARM assembly code. Programs written in HOL are also logic specifications which can be reasoned about directly in the prover, since their semantics is given by the prover. We define the syntax and semantics of the target language using datatypes and evaluation rules. We construct an end-to-end compiler that carries out a series of translations, each of which is accomplished by term rewriting, and we prove their correctness by directly utilizing the prover's reasoning facilities.

1.2 Analyzing Parallel Programs with Message Passing Communications

Application Programming Interfaces (API) (also known as libraries) are an indispensable part of modern concurrent programming. Message Passing Interferace (MPI) [111] has become a *de facto* standard in High Performance Computing (HPC). MPI calls are traditionally described in natural languages, and such descriptions are prone to misinterpretation. We define a formal syntax and operational semantics for a majority of MPI 2.0 functions in the TLA+ language

[112], and we devise a compiler to translate C programs containing MPI calls to their TLA+ models, which can be examined in the TLC model checker [112]. The model checker requires concrete values on the inputs and explores all interleavings and computation paths in a search for possible property violations. This way we can formally analyze C MPI programs and their properties. For example, we are able to check whether a program using broadcasting is equivalent to one using nonblocking point-to-point operations. Moreover, we may transform a program in runtime, as in just-in-time (JIT) compilers, by modifying its concurrent behaviors with respect to the semantics. Particularly, we verify the correctness of dynamic partial order reduction (POR) algorithms.

1.3 Verifying Parallel Programs with Shared Memory Communications

Interest in Graphical Processing Units (GPUs) is skyrocketing due to their potential to yield spectacular performance on many important computing applications. Unfortunately, writing efficient GPU kernels requires painstaking manual optimization effort. For example, there exist heroic acts of programming: (i) keep all the fine-grained GPU threads busy; (ii) ensure coalesced [53] data movements from the *global memory* (accessed commonly by CPUs and GPUs) to the *shared memory* (accessed commonly by the GPU threads); and (iii) minimize bank conflicts when the GPU threads step through the shared memory. These manual manipulations are very error prone, and data races and incorrect barrier placements are frequently introduced during CUDA programming. Furthermore, since a lot of optimizations may be performed on a kernel, the equivalence of the optimized program with the original one is not obvious. We present the first comprehensive symbolic verifier for kernels written in CUDA [23] C. In this verifier, we model program semantics with symbolic relational constraints, and the constraint formulea capture all possible (concurrent) behaviors of a kernel running in multiple threads. We then dump the conjunction of the model and the negation of properties to a Satisfiability Modulo Theories (SMT) solver [107] for satisfiability check. We are able to check races, barrier mismatches, and functional correctness. Notice that it does not check a transformation directly; instead, it uses the translation validation [91] technique to prove the equivalence by comparing the symbolic models of the source and transformed kernels.

1.4 Discussions

We summarize the methods and their settings in Table 1.1. Programs written in HOL are logic specifications which can be reasoned about directly in the prover. The syntax and semantics of the target language, *e.g.* ARM, are defined using datatypes and evaluation rules. We construct an end-to-end compiler by applying a series of transformations, each of which is accomplished by term rewriting. The correctness proof utilizes the prover's reasoning facilities, *e.g.* we apply syntax directed tactics to compose reasoning rules.

The syntax and the operational semantics of MPI calls are specified in the TLA+ language. We devise a compiler to translate C programs containing MPI calls to their TLA+ models, which can be examined in the model checker TLC [112]. This explicit model checker requires concrete values on the inputs. It explores all the interleavings and computation paths in search for possible property violations. Although this framework can be used to analyze source-to-source transformations on MPI programs, we are mainly interested in the dynamic behaviors of an MPI program, *e.g.* the soundness of dynamic partial order reduction (DPOR) algorithms.

A CUDA kernel is modeled by a symbolic formula depicting its operational semantics. This formula captures all possible (concurrent) behaviors of this kernel running in multiple threads. To detect property violations, we dump the conjunction of the model and the negation of the property to the constraint solver (*e.g.* a SMT solver) for satisfiability check. It is able to check races, barrier mismatches, and functional correctness. Note that it does not check a transformation directly; instead, it uses the translation validation [91] technique to prove the equivalence by comparing the symbolic models of the source and transformed kernels.

Our contributions include:

- We show how to use pure term rewriting to mechanically construct a compiler from an advanced functional language (HOL functions) to assembly code.

Table 1.1: Summary of methods presented in the thesis.

Program	Program Semantics	Properties	Methodology
HOL	logic specifications	translation correctness	theorem proving
MPI	TLA+ specifications	dynamic reduction	model checking
CUDA	symbolic formulas	races, deadlocks, *etc.*	constraint solving

- We show how to build a reasoning mechanism (a program logic) for imperative languages modeled formally in HOL by "decompiling" an imperative program into a HOL function with correctness proof.

- We use the HOL logic to model various intermediate representations of a compiler. This technique dramatically reduces the burden of constructing and validating nontrivial compilers.

- We present the first formal specification for a comprehensive set of message passing APIs. The semantics is not only formal (described in a mathematical language) but also "executable" (running in a model checker).

- We present the first formal symbolic checker for GPGPU kernel programs. Novel techniques include encoding all interleavings in a formula, handling loops through loop normalization and customized over-approximation, and so on.

- We propose a novel method to perform parameterized verification of GPGPU kernel programs. Only one parameterized thread is investigated regardless of the number of threads.

- We demonstrate that it is elegant and practical to use formal methods to check programs with various computation and communication models.

These practices represent substantial advances in the relevant areas; more details can be found in the respective chapters. The organization of this thesis is as follows. We first describe the construction and validation of the compiler from HOL specifications to ARM code (Chapters 2, 3, and 4). This compiler consists of a front-end (Chapter 3), a middle-end (Chapter 3), and two back-ends (Chapter 4). Then, we show how to formalize and analyze MPI programs using TLA+/TLC (Chapter 5). Next, we depict a symbolic checker for CUDA kernel programs (Chapter 6), with emphasis on giving formal semantics and checking essential properties such as races, barrier mismatches (deadlocks), and functional correctness. Then, we illustrate a parameterized method to check a kernel's functional correctness as well as the equivalence of a CUDA kernel and its optimized versions (Chapter 7). Finally, we conclude and discuss the future work (Chapter 8).

CHAPTER 2

COMPILING LOGIC SPECIFICATIONS: OVERVIEW

Guaranteeing the correctness of a system implementation is not straightforward. A high level of assurance may be achieved by formally verifying these implementations. However, it is impractical at present to verify implementations written in industry standard high-level languages because these languages are too complex to have tractable formal definitions (*i.e.* definitions which can be reasoned about easily), making it impossible to rigorously prove properties about designs expressed in them. What is worse, the properties proven on a high-level program may not hold on the binary form anymore since compilers may introduce bugs, and users often make over-simplifying assumptions on the machine model.

A possible solution is to reason directly about low-level programs, like JVM or MC68020 [97] instructions, which can be given an accurate formal semantics via a formal model of the machine on which they run. This approach gives the highest assurance, since what is verified is what is executed. However, programming in machine code is expensive and slow. We prefer a high-level programming language with a tractable semantics (*i.e.* a semantics that supports nontrivial reasoning) together with an implementation method guaranteeing that the semantics corresponds exactly to the actual behaviors when programs are run. Unfortunately, existing high-level languages have neither a tractable semantics nor formally verified implementations.

Giving realistic programming languages like C a correct semantics is difficult. It is even more so to make such semantics tractable so that we can reason about nontrivial programs in a formal setting. Some widely used functional languages have been given a formal semantics, *e.g.* Scheme has a denotational semantics [51] and ML has a formal operational semantics [74]. However, these semantics do not as yet provide a practical basis for formal reasoning about programs, although they are extremely valuable as reference documents and for proving meta-theorems

(like type preservation).

A grand challenge would be to design a high-level programming language that realistically competes with C++, C# and Java for implementing industrial strength systems, and which has (i) a tractable semantics, (ii) a highly assured implementation guaranteeing that the semantics is correct, and (iii) a value proposition that ensures it gets used in the real world. This challenge is too grand — it is probably impossible.

As a partial solution to this challenge, we can program some practically useful systems directly in logic, and then compile these logical specifications to realistic platforms for execution. This method allows formal reasoning to the maximum extent since applications are modeled directly in logic. Particularly, we can specify both the algorithms and the mathematics needed for their verification in higher-order logic, and then compile the verified algorithms to low-level platforms modeled in the same logic.

Furthermore, we may translate programs written in (a subset of) a realistic high-level language such as ML or C to equivalent logic specifications, then prove properties about them, and then reuse the compiler for logic to obtain trusted machine code.

2.1 A Trusted Compiler for HOL Specifications

In this thesis, we present a software compiler [63,66,67] which produces assembly code for a subset of the specification language of the HOL theorem prover — Total Functional Language (TFL) [106] — a pure, total functional programming layer built on top of higher-order logic and implemented in both the HOL-4 [44] and Isabelle [83] systems. TFL enables abstract algorithms to be specified in a mixture of mathematics and programming idioms and then reasoned about using theorem proving. Roughly speaking, this language comprises ML-style pure terminating functional programs, *i.e.* those (computable) functions that can be expressed by well-founded recursion in higher-order logic. Features like type inference, polymorphism, higher-order functions and pattern matching make it a comfortable setting in which to program. It can express a very wide range of algorithms. The trade-off is that the compilation of logic specifications written in this language is fairly complicated. As far as we know, this compiler is the first validated compiler that compiles logic specifications coded in such advanced

functional languages as TFL.

The front-end of our compiler [67] translates a source program into a simpler intermediate format LF1 (for *Logic Form* 1) by compiling away many advanced features, *e.g.* it performs monomorphisation and defunctionalization to eliminate polymorphism and higher-order functions.

The mid-end [66] of the compiler further simplifies LF1 programs into LF2 forms which are close to assembly code. Specifically, since there is still a big gap between the front-end output LF1 and the back-end input LF2, the mid-end bridges this gap by adopting plenty of transformations and optimizations such as code motion and register allocation.

The back-end of the compiler (Chapter 4) generates from a LF2 form either (1) an equivalence *imperative* program, *e.g.* a Heap and Stack Level (HSL) program, which will be translated to other imperative IRs and finally to the machine code; or (2) an abstract structured assembly (SAL) code, which can be used to generate machine code for various architectures such as ARM.

Figure 2.1 shows the relation between the main components of the compiler. All the transformations in the front-end and mid-end are implemented as rewrite rules. The correctness of each transformation is proven on the fly: after a program is translated, a theorem is given as by-product that states the equivalence of the produced code and this program. Although standard compilation techniques developed for functional programming may be applied here, new challenges are posed due to the fact that (i) the source language is not visible in the logic — it is the logic itself that is taken as the source language; (ii) TFL programs have a set-theoretic semantics rather than an operational or denotational semantics; and (iii) all transformations must be formalized and verified in the logic that is compiled. Since TFL programs are not given in terms of datatypes and do not have an evaluation semantics, widely-used techniques [10,20,21,58] that are based on structural induction over syntax datatypes and rule-induction over evaluation relations cannot be applied here.

Our compiler demonstrates the first attempt to unite most of the phases of an optimizing compiler together at the logic level: both the source language and most of the IRs (*e.g.* LF1 and LF2) are represented directly in the logic; all transformations and optimizations are performed and verified directly in the logic.

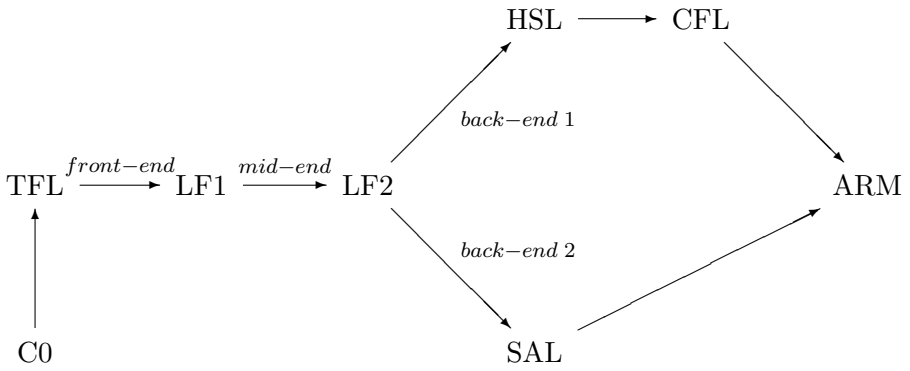


Figure 2.1: Main phases of the HOL compiler.

2.2 Motivating Example

Consider a program shown in Figure 2.2 which first calculates the factorial of x then switches on whether this factorial is greater than y. We show below its ML (or TFL) version and C version.

A compiler may generate the following ARM-style assembly code cs_{ex}. This code first sets register r_1's value to 1. It then checks whether r_0 is 0; if yes, then it jumps to label l_{+5} by increasing the program pointer by 4. In this case, the instructions at $l_{+5} - l_{+9}$ are executed, producing the output in r_1. Otherwise, r_1 is multiplied by r_0 and r_0 is decreased by 1. Then, the code jumps back to label l_{+1} by decreasing the pc by 3. Obviously, the instructions from l_{+1} to l_{+4} constitutes a loop.

ML$_{ex}$

```
f_ex (x, y) ≐
  let f_fact (x, a) =
    (if x = 0 then (x, a)
     else f_fact (x − 1, x ∗ a))
  in
  let (v0, v1) = f_fact (x, 1) in
  if v1 ≥ y then v1 − y
  else v1 + 2 ∗ y
```

C$_{ex}$

```
f_ex (int x, int y) {
  int a = 1;
  while x ≠ 0
    {a = x ∗ a; x = x − 1;}
  if a ≥ y
    return a − y;
  else
    return a + 2 ∗ y;
}
```

Figure 2.2: A motivating example f_{ex}.

$$
\begin{array}{ll}
l: & \text{mov } r_1\ 1 \\
l_{+1}: & \text{beq } r_0\ 0\ (+4) \\
l_{+2}: & \text{mul } r_1\ r_1\ r_0 \\
l_{+3}: & \text{sub } r_0\ r_0\ 1 \\
l_{+4}: & \text{b } (-3)
\end{array}
\qquad
\begin{array}{ll}
l_{+5}: & \text{blt } r_1\ r_2\ (+3) \\
l_{+6}: & \text{sub } r_1\ r_1\ r_2 \\
l_{+7}: & \text{b } (+3) \\
l_{+8}: & \text{mul } r_2\ 2\ r_2 \\
l_{+9}: & \text{add } r_1\ r_1\ r_2 \\
l_{+10}: &
\end{array}
$$

Our goal is to not only produce the assembly code, but also prove the compilation correct. Specifically, the relation between the input in (r_0, r_2) before the execution and the output in r_1 after the execution shall be represented by f_{ex}. Let $\sigma[v]$ denote reading v's value from state σ, and $\mathsf{run}\ cs\ \sigma$ denote the execution of cs starting from state σ until cs terminates. The correctness theorem claims that $\forall\sigma.\ (\mathsf{run}\ cs_{ex}\ \sigma)[r_1] = f_{ex}\ (\sigma[(r_0, r_2)])$.

Our front-end is able to translate the C version C_{ex} to the ML version ML_{ex}, which is written in the term language of the HOL logic. The mid-end transforms f_{ex} into simpler formats, as shown in Figure 2.3. The one on the left is in an ANF [31]+SSA format. It can be transformed to the one on the right by converting the tail recursive function into a loop and performing register allocation. The definition of "while" is given by $\mathsf{while}\ c\ f \doteq \lambda x.\ \mathsf{if}\ \neg\ c\ x\ \mathsf{then}\ x\ \mathsf{else}\ \mathsf{while}\ c\ f\ (f\ x)$.

It is not very difficult to produce code cs_{ex} from function f_{ex_2}. The main challenge, however, is to prove the equivalence of their semantics. We might take another look at the relation between the code and the function. Suppose we are given the code cs_{ex}, can we "decompile" it to a function like f_{ex_2}? If yes, then

```
f_ex1 (v0, v1) ≐
 let f(v0, v1) =
   if v0 = 0 then (v0, v1)
   else
    let v2 = v1 * v0 in
    let v3 = v0 − 1 in
    f (v3, v2)
 in
 let v2 = 1 in
 let (v3, v4) = f(v0, v2) in
 if v4 < v1 then
   let v5 = 2 * v1 in
   let v6 = v1 + v5 in v6
 else
   let v5 = v4 − v1 in v5
```

```
f_ex2 (r0, r2) ≐
let r2 = 1 in
let (r0, r1) =
 while (λ(r0, r1). r0 ≠ 0)
       (λ(r0, r1).
         let r1 = r1 * r0 in
         let r0 = r0 − 1 in
         (r0, r1))
 (r0, r1)
in
if r1 < r2 then
  let r2 = 2 * r2 in
  let r1 = r1 + r2 in r1
else
  let r1 = r1 − r2 in r1
```

Figure 2.3: Two intermediate forms of the f_{ex} example.

we obtain a logic model for cs_{ex}. Essentially, the function derived from the code "recovers" the code's control flow structure, and interprets the code in a more functional manner by abstracting away the imperative features of the code. This method brings another advantage: we may decompile a piece of code cs'_{ex} produced by a third-party code generator, and prove that this code also implements the source program correctly by comparing the derived function and f_{ex_2}.

However, the decompilation from assembly code often requires discovering the control flow of a program, while we already have this information at the immediate format level! Thus, it may be more convenient to record the control flow structure and other information (*e.g.* heap and stack information) of a piece of code using imperative intermediate formats. We introduce the intermediate languages HSL and Control Flow Level (CFL) to achieve this goal. In order to prove a program in these formats is equivalent to a corresponding LF2 function, we decompile them to obtain equivalent LF2 functions.

Another approach is to directly model the heap and stacks in the logic level, and devise a way to connect logic expressions with low-level code. We introduce an abstract structured assembly language (SAL) and define its semantics according to LF2. This provides another elegant way to manage the translation between LF2 and machine code.

In sum, our logic-based compiler is able to translate (or model) both the source program and the target code (or the IR close to target code) into logic, and perform all the transformations and optimizations at the logic level. This facilitates greatly the construction and validation of nontrivial compilers because it makes full use of the prover HOL. For example, the prover guarantees the preservation of scoping, and it automates many frequently-occurring tasks, including scoping, substitution, and rewriting strategies. And, in most cases, the correctness of the compilation depends solely on a small set of rewrite rules that are written in the language of formal mathematics, whose correctness proof is performed directly in the logic.

2.3 Related Work

The notion of compiler verification dates back to McCarthy and Painter [72]. Their mathematical paradigm for verified compilation, such as abstract syntax of languages, abstract mathematical definitions of compilation, and correctness proof based on structural induction, has been adopted by most subsequent researchers.

Over more than 40 years, there have been a lot of attempts to develop trusted compilers for various source and target languages. Dave's bibliography [25] gives a list of publications on compiler verifications.

2.3.1 What is a Verified Compiler?

In this section, we introduce Leroy's classification of trusted compilers [58] and characterize our compiler in terms of his categories.

The correctness of a compiler is often specified by a correctness property $Prop(S, C)$ between a source program S and its compiled code C. Examples of such properties include: (1) S and C are observationally equivalent; (2) if S is well-defined (does not go "wrong"), then S and C are observationally equivalent; (3) if S is well-defined and $P(S)$ holds, then $P(C)$ holds; (4) if S is type- and memory-safe, then so is C. Property (1) requires C to go wrong whenever S does; this restriction is released in property (2). Property (2) implies (3) if P depends only on the observable behavior of the program. Property (4) is typical of a type-preserving compiler. Our compiler falls into the third category.

A compiler $Comp$ is a total function from source programs to either compiled code (written $Comp(S) = Some(C)$) or an error (written $Comp(S) = None$). This requires that an error will occur when a compiler fails to produce code. Leroy specifies three kinds of verification:

- Certified compilers. A certified compiler is any compiler $Comp$ accompanied with a formal proof of a theorem: $\forall S\, C.\, (Comp(S) = Some(C)) \Rightarrow Prop(S, C)$. In other words, either the compiler reports an error or produces code that satisfies the desired correctness property. It is warranted that the compiler never silently produces incorrect code.

- Translation validation. In this approach [80, 91], the standard compiler $Comp$ is complemented by a verifier $Verif(S, C)$ such that $\forall S\, C.\, Verif(S, C) \Rightarrow Prop(S, C)$. The advantage is no formal verification of the compiler itself is needed.

- Proof-carrying code. In Proof-carrying code [4, 81] and credible compilation [96], the compiler either fails or returns both a compiled code C and a proof π of the property $Prop(S, C)$ (*i.e.* $Comp(S) = SOME(c, \pi)$). The proof π

can be checked independently by the code user. It is sufficient to generate enough hints so that such a full proof can be reconstructed cheaply by a specialized prover.

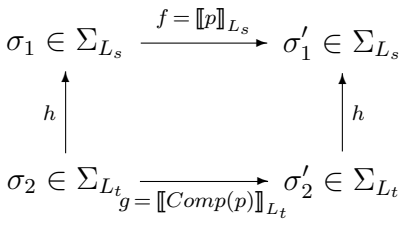
Roughly, our work falls into the translation validation category. In particular, we perform per-run correctness checks by comparing S and C only; thus, no verification of the compiler itself is needed. However, since the verifier *Verif* is the proof system of HOL, it turns out that our compiler behaves like certified compilers. That is, we guarantee that $\forall S\, C.\, (Comp(S) = Some(C)) \Rightarrow Prop(S, C)$. We have to point out that so far, there has been no consensus on the terminology; thus, we use general terms "ad hoc certified compilation" and "incremental translation validation" to characterize our compiler.

2.3.2 Correctness Statement

The correctness of the compilation *Comp* from source language L_s to target language L_t is specified by relating a source program $p \in L_s$ to its target program $Comp(p) \in L_t$ with respect to their semantics $[\![p]\!]_{L_s}$ and $[\![Comp(p)]\!]_{L_t}$. Typically, the correctness of *Comp* is expressed by the following commutative diagram.

$$\begin{array}{ccc} p \in L_s & \xrightarrow{\quad semantics \quad} & [\![p]\!]_{L_s} \\ \Big\downarrow Comp & & \Big\downarrow ? \\ Comp(p) \in L_t & \xrightarrow[semantics]{} & [\![Comp(p)]\!]_{L_t} \end{array}$$

$[\![p]\!]_{L_s}$ and $[\![Comp(p)]\!]_{L_t}$ can be compared in terms of their denotational semantics which use mathematical objects to denote the meaning of programs. For instance, the semantics of these two programs can be functions that map input into output. In contrast, the operational semantics of a language is typically modeled by a function mapping program states (which map variables to values) $\sigma \in \Sigma$ to program states. For example, in the following diagram, p's semantics $[\![p]\!]_{L_s}$ is captured by the function f mapping from state σ_1 to σ_1'. An abstraction function h "constructs" program states from machine states by extracting values from concrete machine representations. The strongest correctness statement, which establishes strong semantical equivalence of source and target program, is specified as $h \circ g = f \circ h$.



Weaker commutativity of the diagram is possible [35]: (1) $h \circ g \subseteq f \circ h$. The target program terminates; (2) $f \circ h \subseteq h \circ g$. The source program is well-defined; (3) $(f \circ h)\ \sigma = (f \circ h)\ \sigma$ for every $\sigma \in \Sigma_{L_s}$ if both sides are well-defined. Our approach belongs to the strongest case because the compiler accepts only well-defined programs and always generates terminating target code.

2.3.3 Literature Review : Compiler Verification

2.3.3.1 Certified Compilers

In the late 1980s, Computational Logic, Inc (CLI) built and verified the "CLI stack" using the Boyer-Moore theorem prover [77]. This stack formalizes a register transfer level (RTL) design for a microprocessor, a machine code instruction set architecture, the stack-based assembly language Piton [76], two simple high-level languages (derived from Gypsy and Lisp, respectively), and a simple operating system. Compilers from high-level languages to assembly language and then to RTL were constructed and verified. Later on, this group investigated the JVM runtime system and compilers for JVM byte code. This project is close to ours except that it targets different source and target languages; in fact, they do not compile logic specifications.

In the CLI stack, a compiler for a simple subset of Gypsy (Micro-Gypsy) to the Piton assembly language was constructed by Young [125]. Gypsy is a combined programming and specification language descended from Pascal. In his work, those features difficult to represent within the Boyer-Moore logic, such as unbounded quantification, dynamic data types, type abstraction, and so on, are discarded. The semantics of Micro-Gypsy and the target language are defined explicitly as interpreter functions, which are executable in the Boyer-Moore logic. The correctness proof goes by relating the Micro-Gypsy interpreter and the target language interpreter. This approach defines operational semantics for all languages, thus is different from our declarative approach where no semantics is needed to be defined for the logic specification language. In this aspect, the related work mentioned

below is akin to Young's.

The ProCos approach [35] embedded source languages in a refinement algebra and expresses the semantics of target languages by interpreters written in the source languages. In the PVS prover [85], a specialized bootstrapping and double checking technique was used to verify the compilation from COMLISP to hardware platforms. This group also verified in PVS a compiler translating a basic block oriented intermediate language with expressions to target language based on the DEC-Alpha processor family [26]. They used abstract state machines (ASM) to formalize the operational semantics of the languages and proved the correctness of single transformations by symbolic execution of program pieces. In addition, they decomposed the construction of compiler back-ends into single term-rewrite rules, and reduced the correctness of the back-ends to the local correctness of these rules [128].

Hannan and Pfenning [40] constructed a verified compiler in LF for the untyped λ-calculus. The target machine is a variant of the CAM runtime and differs greatly from real machines. In their work, programs are associated with operational semantics, and both compiler transformation and verifications are modeled as deductive systems.

Chlipala [20] further considered compiling a simply-typed λ-calculus to assembly language. He proved semantics preservation based on denotational semantics assigned to the intermediate languages. Type preservation for each compiler pass was also verified. The source language in these works is the bare lambda calculus and is thus much simpler than TFL; thus, their compilers only begin to deal with the high-level issues we discuss in this thesis. Chlipala [21] further considered translating a simple impure functional language to an idealized assembly language. One of the main points is to avoid binder manipulation by using a parametric higher-order abstract syntax to represent programs; while in our case, this is automatically taken care of by the prover. Its representative optimization, common subexpression elimination, is accomplished in our compiler by a single rewrite rule.

Broy *et al.* [18] proved in the Isabelle theorem prover the partial and total correctness of an interpreter for a small first-order functional language with denotational semantics. The interpreter, which generates normal forms, is given as a recursively defined function. The correctness of the interpreter, which specifies

that a program terminates with the same value as the denotational semantics of this program, is proved using structural induction.

A purely operational-semantics-based development is that of Klein and Nipkow [54] which gives a thorough formalization of a Java-like language. A compiler from this language to a subset of the Java Virtual Machine is verified using Isabelle/HOL. Strecker [109] verified a compiler from a subset of Java source language to Java bytecode in Isabelle/HOL. In this work, the behavior of Java programs is defined by a big-step (natural) operational semantics in the form of an evaluation relation; and the correctness proof is by mutual induction on the evaluation relation.

The Isabelle/HOL theorem prover is also used to verify the compilation from a type-safe subset of C to DLX assembly code [57], where a big-step semantics and a small-step semantics for this language are defined. Meyer and Wolff [73] derived in Isabelle/HOL a verified compilation of a lazy language (called MiniHaskell) to a strict language (called MiniML) based on the denotational semantics of these languages.

Leroy [15, 58] verified a compiler from a subset of C, *i.e.* Clight, to PowerPC assembly code in the Coq system. The semantics of Clight is completely deterministic and specified as big-step operational semantics. The proof of semantics preservation for the translation proceeds by induction over the Clight evaluation derivation, while our proofs proceed by verifying the rewriting steps. As demonstrated in [117], his compiler needs extensive manual effort to verify new optimizations, while our rewriting-based approach is very flexible and easy to accommodate nontrivial optimizations. In fact, our modeling of IRs directly in the logic is intended to mitigate the burden of manual proof.

Benton and Zarfaty [11] interprets types as binary relations. They proved a semantic type soundness for a compiler from a simple imperative language with heap-allocated data into an idealized assembly language. Similar interpretation [10] is used to connect the denotational semantics of a simply typed functional language and the operational behavior of low-level programs in a SECD machine. This allows, as we did, the modeling of low-level code using a mathematical, domain-theoretic functions, as well as the proof of a simple compiler. But we do not need to define the semantics in terms of tricky and customized interpretations.

2.3.3.2 Translation Validation

The notion of translation validation was first introduced by Pnueli *et al.* in 1998 [91]. They validated a translation from the synchronous multiclock data-flow language SIGNAL to asynchronous (sequential) C code. Rather than proving in advance that the compiler always produces target code correctly, translation validation warrants that the code generated by each individual run of the compiler correctly implements the submitted source program. They reduce the correctness of the translation to verification conditions that can be solved by model checkers.

Necula [80] built a translation validation infrastructure for many intra-procedural optimizations performed in the GNU C compiler. During the compilation the infrastructure compares the intermediate form of the program before and after each compiler pass and verifies that the executions of these two forms lead to the same sequence of function calls and returns (which is an approximation to semantic equivalence). Symbolic evaluation is used to compute the effect of basic blocks, and an inference algorithm is devised to collect the constraints representing the correctness of translation. However, Necula's method does not utilize any rigorous prover to guarantee that the infrastructure itself is sound.

Tristan and Leroy [117] developed translation validators and Coq proofs of correctness for two instruction scheduling optimizations: list scheduling and trace scheduling. The validation algorithm is based on symbolic execution of the original and transformed codes at the level of basic blocks; and the correctness of the validators is proved against an operational semantics in a combination of small-step and big-step styles.

2.3.3.3 Proof-carrying Code

In the practice of proof-carrying code (PCC) [4, 81], the compiler produces annotations about the correctness of the translations so that a full proof can be reconstructed on these annotations using specialized provers. Specifically, the code producer creates a proof, then the code consumer generates verification conditions and checks whether the proof proves the verification conditions. In other words, some proof validators are used to check whether such conditions are valid and hence the code is safe to execute. Thus, the crucial part of the correctness proof, namely, the generation and validation of verification conditions, is not performed automatically by the compiler.

2.3.4 Rewriting-based Compilation

Term rewriting is commonly used in compiler construction for the specification of transformations, especially in back-end generators. The basic idea is that an (intermediate) form can be viewed as a term, and this term can be reduced to simpler form in a reduction step. Many transformations in our compiler are implemented as term rewriting rules [66, 67].

There are some systems that use rewrite rules to specify program transformations. For instance, in the ASF+SDF environment [119], transformations and evaluation can be specified as rewrite rules. There is also some work that uses logical frameworks to simplify the construction of compilers. For instance, Liang [68] implemented a compiler for a simple imperative language using a higher-order abstract syntax representation in λ-Prolog. Boyle, Resler, and Winter [48] proposed using rewrites that model code transformation to build trusted compilers. They also introduced a transformation grammar to guide the application of rewrites [123]. Similarly, Sampaio [101] used term rewriting to convert source programs to their normal forms, which represent object code. However, these works do not address the issue of validation of the rewrite rules.

Hickey and Nogin [43] worked in the MetaPRL logical framework to construct a compiler from a full higher-order, untyped, functional language to Intel x86 code, based on higher-order rewrite rules. A set of rewriting rules are used to convert a high-level program to a low-level program. They use higher-order abstract syntax to represent programs and do not define the semantics of these programs. Thus, no formal verification of the rewriting rules is done.

2.3.5 Related Work in Cambridge

The back-ends presented in this thesis have the same purpose as [79] does, but uses a different reasoning method. That method relies on a Hoare Logic built for ARM or X86, and composes reasoning rules in a bottom-up style to obtain LF2 functions. Note that it uses a more detailed and more faithful ARM model. It can be regarded as an extension of our back-end for more target architectures. An interesting point is to reuse our front-end and mid-end to generate LF2 programs as the inputs of that (back-end) compiler.

2.4 Source Language TFL

TFL is a subset of the higher-order logic built in HOL; thus, their syntax and the semantics have already been defined in the logic. So do the IRs in the front-end and the mid-end. That is, programs written in TFL or IRs are simply mathematical functions defined in the HOL logic. It is this feature that enables us to use standard mathematics to reason about these languages. This supports much flexibility and allows the meaning of a program to be transparent. In particular, two programs are equivalent when the mathematical functions represented by them are equal.

Many front end tasks are already provided by the HOL-4 system: lexical analysis, parsing, type inference, overloading resolution, function definition, and termination proof (needed to admit recursive functions, since HOL is a logic of total functions). TFL is a polymorphic, higher-order and terminating functional language supporting algebraic datatypes and pattern matching. Its syntax is shown in Figure 2.4, where $[term]_{separator}$ denotes a sequence of $term$'s separated by the *separator*.

We make an extension to TFL by having it handle (functional) arrays. An array is declared using the new operator. It is modeled as a function mapping

τ	$::= T \mid t$	primitive type, type variable
	$\mid \tau\ D$	algebraic type
	$\mid \tau \# \tau \mid \tau \to \tau$	tuple and arrow (function) type
at_c	$::= id \mid id$ of $[\tau]_{\Rightarrow}$	algebraic datatype clause
at	$::=$ datatype $id = [at_c]_{\mid}$	algebraic datatype
	$\mid [at]_{;}$	mutually recursive datatype
pt	$::= i \mid v \mid c\ \overrightarrow{pt}$	pattern
e	$::= i : T \mid v : \tau$	constant and variable
	$\mid \overrightarrow{e} \mid (e, e)$	tuple
	$\mid p\ e$	primitive application
	$\mid c\ e$	constructor application
	$\mid$ new (τ, i)	array declaration
	$\mid v[e]$	array access
	$\mid f_{id}$	function identifier
	$\mid e\ e$	composite application
	$\mid$ if e then e else e	conditional
	$\mid$ case e of $[(c\ e) \rightsquigarrow e]_{\mid}$	case splitting
	$\mid$ let $v = e$ in e	let binding
	$\mid [\lambda\, v]_{.}\ e$	anonymous function
f_{decl}	$::= f_{id}\ ([pt],) = e$	pattern matching clause
	$\mid [f_{decl}]_{\wedge}$	function declaration
	$\mid v = e$	top level variable declaration

Figure 2.4: The syntax of the total function language TFL.

natural numbers to values; initially, it maps each index to an arbitrary number ϵ. Notation $a[i]$ is used to access the i^{th} element in array a. For example, the semantics of `let` $a =$ `new` $(\texttt{int}, 2)$ `in let` $a[i] = 10$ `in` $a[j]$ is `let` $a = \lambda i.\, \epsilon$ `in let` $a = (\lambda k.\, \mathsf{if}\ k = i\ \mathsf{then}\ 10\ \mathsf{else}\ \epsilon)$ `in` $a\ j$. This extension supports writing elegant and efficient programs, while TFL is still pure since an array is nothing but a function with array-like syntactic sugars.

$$\begin{array}{lcl} \texttt{new}\ (\tau, n) & \doteq & \lambda i.\, \epsilon \\ a[i] = e & \doteq & a = \lambda j.\, \mathsf{if}\ j = i\ \mathsf{then}\ e\ \mathsf{else}\ a\ j \end{array}$$

2.4.1 Notational Conventions

We use $f\ x_1\ \ldots\ x_n$ to represent the application of f to $x_1, \ldots, x_n$. From another perspective, f is a term containing free variables $x_1, \ \ldots\ , x_n$. For example, $x + y$ is the same as $(\lambda v.\, x + v)\ y$.

A rewrite rule is of format [name] redex $\longleftrightarrow$ contractum $\Leftarrow$ cond. It specifies an expression that matches the redex can be replaced with the contractum provided that the side condition cond over the redex holds. The side condition is encoded at the meta-level such that we need not formalize it in HOL.

2.5 From Imperative Language C0 to TFL

Importing terminating ML programs into TFL is easy due to the high similarity in their syntaxes and semantics. One of the main issues — termination proof — is handled by proving that the generated TFL function is total. Moreover, the imported programs will be type checked by the prover.

It is also possible to import programs written in an imperative language such as a small subset of C. As a demonstration, we have developed a method to for such a subset C0. The syntax of C0's control flow structures is shown below. Note that variable v may be a tuple or an array.

e		C style expressions
s	$::= v := e$	assignment
	$\mid$ `return` v	return
	$\mid s; s$	sequential statement
	$\mid$ `IF` e `THEN` s `ELSE` s	conditional jump
	$\mid$ `WHILE` $e\ s$	loop
	$\mid v := p_{id}\ s$	procedure call
p	$::= [p_{id}\ v = s]_;$	programs

We first define an operational semantics (omitted here due to lack of space) for C0 and then derive an axiomatic semantics from it. Each axiomatic semantics rule is specified as a Hoare triple $\{precondition\}\ structure\ \{postcondition\}$:

$$\frac{\{P\}\ S_1\ \{Q\} \quad \{R\}\ S_2\ \{T\} \quad Q \Rightarrow R}{\{P\}\ (S_1\ ;\ S_2)\ \{T\}}$$

$$\frac{\{P\}\ S\ \{P\}}{\{P\}\ (\texttt{WHILE}\ C\ S)\ \{P \wedge \neg C\}}$$

$$\frac{\{P \wedge C\}\ S_t\ \{Q\} \quad \{P \wedge \neg C\}\ S_f\ \{Q\}}{\{P\}\ (\texttt{IF}\ C\ \texttt{THEN}\ S_t\ \texttt{ELSE}\ S_f)\ \{Q\}}$$

$$\frac{\{P\}\ S_t\ \{Q\} \quad \{P\}\ S_f\ \{R\}}{\{P\}\ (\texttt{IF}\ C\ \texttt{THEN}\ S_t\ \texttt{ELSE}\ S_f)\ \{\texttt{if}\ C\ \texttt{then}\ Q\ \texttt{else}\ R\}}$$

In order to connect the semantics of a C0 structure S to a TFL function f, we introduce the following judgment to characterize S's axiomatic semantics as a predicate, where $\sigma[x]$ returns the value of variable x in state σ; and $\mathsf{eval}\ S\ \sigma$ returns the new state after S's execution. Notation (i, f, o) specifies that: if the initial value of input i is v, then in the state after the execution of S, the value left in output o is equal to applying the function f to the initial value v. Basically, a judgment can be obtained by instantiating the P and Q in a Hoare triple $\{P\}\ s\ \{Q\}$ to $\lambda\sigma.\,\sigma[i] = v$ and $\lambda\sigma.\,\sigma[o] = f\ v$, respectively. If a judgment synthesizes f with respect to the input i and output o, then we claim that S correctly implements function f.

$$S \vdash (i, f, o) \ \doteq\ \forall\sigma\forall v.\,(\sigma[i] = v) \Rightarrow ((\mathsf{eval}\ S\ \sigma)[o] = f\ v)$$

We derive a couple of rules (see Figure 2.5) to synthesize a function by composing the judgments. A judgment may contain an extra field ex, which will be explained later. Notation $\hat{v}$ generates a TFL variable for a C0 variable v, and $\hat{e}$ returns the TFL expression corresponding to a C0 expression e. Notation `fv` returns the free variables in an expression. We use $\doteq$ to introduce abbreviations.

Rule assgn builds a judgment for a C0 assignment $v := e$. The input consists of all the free variables in e and the output is v; the synthesized function calculates the expression e. For example, the judgment $x := y{+}z{+}1 \vdash \langle (y, z),\ (\lambda(y, z).\,y{+}z{+}1),\ x\rangle$ synthesizes a function $\lambda(y, z).\,y + z + 1$ for input (y, z) and output x. Rule return synthesizes an identity function for the same input and output v.

$$\frac{i \doteq \texttt{fv}\ e}{v\ :=\ e\ \vdash\ \langle i,\ \lambda\hat{i}.\,\hat{e},\ v\rangle}\ \text{assgn}$$

$$\frac{}{\texttt{return}\ v\ \vdash\ \langle v,\ \lambda\hat{v}.\,\hat{v},\ v\rangle}\ \text{return}$$

$$\frac{S_1\ \vdash\ \langle i_1,\ f_1,\ o_1\rangle \downarrow ex_1 \quad S_2\ \vdash\ \langle o_1,\ f_2,\ o_2\rangle \downarrow ex_2}{S_1;\ S_2\ \vdash\ \langle i_1,\ f_2\ \texttt{o}\ f_1,\ o_2\rangle \downarrow (ex_1 \cup ex_2)}\ \text{seq}$$

$$\frac{S_1\ \vdash\ \langle i,\ f_1,\ o\rangle \downarrow ex_1 \quad S_2\ \vdash\ \langle i,\ f_2,\ o\rangle \downarrow ex_2}{\begin{array}{l}\texttt{IF}\ e\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2\ \vdash \\ \langle i,\ (\lambda\hat{i}.\,\texttt{if}\ \hat{e}\ \texttt{then}\ f_1\ \hat{i}\ \texttt{else}\ f_2\ \hat{i}),\ o\rangle \downarrow\ (ex_1 \cup ex_2)\end{array}}\ \text{cond}$$

$$\frac{S\ \vdash\ \langle i,\ f,\ i\rangle \downarrow ex}{\texttt{WHILE}\ e\ S\ \vdash\ \langle i,\ \texttt{while}\ (\lambda\hat{i}.\,\hat{e})\ f,\ i\rangle \downarrow ex}\ \text{while}$$

$$\frac{p_{id}\ i := S \quad S\ \vdash\ \langle i,\ f,\ o\rangle \downarrow ex}{w := p_{id}\ v\ \vdash\ \langle v,\ f,\ w\rangle \downarrow \{v \in ex \mid v \text{ is global}\}}\ \text{call}$$

$$\frac{S\ \vdash\ \langle i,\ f,\ o\rangle \downarrow ex}{S\ \vdash\ \langle i,\ \lambda\hat{i}.\,\texttt{let}\ (\hat{o}_1, \hat{o}_2) = f\ \hat{i}\ \texttt{in}\ \hat{o}_1,\ o_1\rangle \downarrow\ (ex \cup \{o_2\})}\ \text{shrink}$$

$$\frac{S\ \vdash\ \langle i,\ f,\ o\rangle \downarrow ex \quad v \notin ex \quad v \notin o}{S\ \vdash\ \langle (i, v),\ (\lambda(\hat{i}, \hat{v}).\,(f\ \hat{i}, \hat{v})),\ (o, v)\rangle \downarrow ex}\ \text{frame}$$

Figure 2.5: Compositional rules for converting C0 to TFL.

Rules seq, cond, while, and call are used to synthesize functions for sequential structures, conditional structures, loops, and procedure calls, respectively. Most of them are self-explanatory. Recall that the "while" in rule while is defined by while $c\ f \doteq \lambda x.$ if $\neg\ c\ x$ then x else while $c\ f\ (f\ x)$.

An important rule, frame, is used to match the inputs and outputs of different judgments. For instance, suppose we want to use the seq rule to compose judgments $S_1 \vdash \langle i_1, f_1, o_1\rangle$ and $S_2 \vdash \langle i_2, f_2, o_2\rangle$. If $o_1 \neq i_2$, we must adjust the judgments to make $o_1 = i_2$. This is accomplished by the frame rule which allows adding extra variables into the input and output.

Since all the variables updated in a structure will appear in the output, we might safely assume that those not in the output are unchanged. As in separation logic [95], we can add these unchanged variables into the input/output using the frame rule if needed. On the other hand, as in the shrink rule, we may remove from the output those variables which will not be referenced anymore. Since these variables may be updated by the execution, we record them in an *exception* set ex so that the application of frame will rule them out. When the exception set is empty we do not present it.

The application of the composition rules is syntax directed, and proceeds in a bottom-up manner. For illustration, consider the C version of the running example. The judgments for the two statements within the loop are as follows.

$$a := x * a \ \vdash \ \langle (x, a),\ \lambda(x, a).\, x * a,\ a \rangle$$
$$x := x - 1 \ \vdash \ \langle x,\ \lambda x.\, x - 1,\ x \rangle$$

Since the output of the first judgment is not the same as the input of the second judgment, we apply the frame rule to adjust them, then these two judgment can be composed.

$$a := x * a \ \vdash \ \langle (x, a),\ \lambda(x, a).\, (x, x * a),\ (x, a) \rangle$$
$$x := x - 1 \ \vdash \ \langle (x, a),\ \lambda(x, a).\, (x - 1, a),\ (x, a) \rangle$$
$$a := x * a;\ x := x - 1 \ \vdash$$
$$\langle (x, a),\ \lambda(x, a).\, (x - 1, x * a),\ (x, a) \rangle$$

Let g_1 be an abbreviation of $\lambda(x, a).\, (x-1, x*a)$. Next, we apply the while rule to get a judgment for the loop. The composition of this judgment and the one for a := 1 yields a new judgment, where $g_2 \doteq \mathsf{while}\ (\lambda(x, a).x \neq 0)\ g)\ \mathsf{o}\ (\lambda x.(x, 1))$.

$$\text{WHILE } (x \neq 0)\ \{a := x * a;\ x := x - 1;\} \ \vdash$$
$$\quad \langle (x, a),\ \mathsf{while}\ (\lambda(x, a).x \neq 0)\ g,\ (x, a) \rangle$$
$$a := 1;\ \text{WHILE } (x \neq 0)\ \{a := x * a;\ x := x - 1;\} \ \vdash \ \langle x, g_2, (x, a) \rangle$$

Similarly we obtain the judgment for the conditional statement.

$$\text{if } (a \geq y) \text{ return } a - y;\ \text{else } a + 2 * y; \ \vdash$$
$$\langle (a, y),\ \lambda(a, y).\, \mathsf{if}\ a \geq y\ \mathsf{then}\ a - y\ \mathsf{else}\ a + 2 * y,\ (a, y) \rangle$$

Since variable x is not used in this judgment, we can eliminate it (through rule shrink) from the judgment for the loop. Then we add the y into the input and output through the frame rule.

$$a := 1;\ \text{WHILE } (x \neq 0)\ \{a := x * a;\ x := x - 1;\}$$
$$\vdash \ \langle x,\ \lambda(x, a).\, \mathtt{let}\ (x, a) = g_2\ (x, a)\ \mathtt{in}\ a,\ a \rangle \downarrow \{x\}$$
$$\vdash \ \langle (x, y),\ (\lambda((x, a), y).\, (\mathtt{let}\ (x, a) = g_2\ (x, a)\ \mathtt{in}\ a,\ y),\ (a, y)) \rangle \downarrow \{x\}$$

Finally we synthesize a function for the entire program

$$(\lambda(a, y).\, \mathsf{if}\ a \geq y\ \mathsf{then}\ a - y\ \mathsf{else}\ a + 2 * y)\ \mathsf{o}$$
$$(\lambda((x, a), y).\, (\mathtt{let}\ (x, a) = g_2\ (x, a)\ \mathtt{in}\ a,\ y))$$

which can be rewritten to $\texttt{ML}_{ex}$ by using the definition of "while" and some rewrite rules about "let" expressions such as:

$$(\lambda y.\, f_1\ y)\ \texttt{o}\ (\lambda x.\, f_2\ x) \longleftrightarrow \texttt{let}\ y = f_2\ x\ \texttt{in}\ f_1\ y$$
$$\texttt{let}\ (x, y) = (e, y)\ \texttt{in}\ f\ x\ y \longleftrightarrow \texttt{let}\ x = e\ \texttt{in}\ f\ x\ y\ .$$

After a C0 program is converted to an equivalent TFL program, we can reason about it directly in the prover and reuse the TFL compiler to produce assembly code. Note that the C0 language is still very simple; we leave its extension to larger subsets of C (*e.g.* supporting structures and pointers) to further work.

CHAPTER 3

COMPILING LOGIC SPECIFICATIONS : FRONT-END AND MID-END

In this chapter, we present the main conversions in the front-end [67] and the mid-end [66].

3.1 Main Conversions in the Front-end

Given a TFL program, the front-end performs transformations that are familiar from existing functional language compilers [115] except that it does so by proof. TFL's high-level features such as polymorphism, higher-order functions, pattern matching, and composite expressions need to be expressed by more lower-level structures:

- The translator removes polymorphism from TFL programs by making duplications of polymorphic datatype declarations and functions for each distinct combination of instantiating types.
- The translator applies defunctionalization to remove higher-order functions by creating algebraic datatypes to represent function closures and type-based dispatch functions to direct the control to top-level function definitions.
- The translator converts pattern matching first into nested case expressions, then into explicit conditional expressions.

All intermediate forms are still mathematical functions. The correctness proof of a transformation on a source program p proceeds, in a *translation validation* [91] style, by showing the generated program q computes the same mathematical function as p.

3.1.1 Monomorphisation

This transformation eliminates polymorphism and produces a simply-typed intermediate form that enables good data representations. The basic idea is to

duplicate a datatype declaration at each type used and a function declaration at each type used, resulting in multiple monomorphic clones of this datatype and function. This step paves the way for subsequent conversions such as the type-based defunctionalization. Although this seems to lead to code explosion in theory, it is manageable in practice (MLton [75], a fancy ML compiler, uses similar techniques and reports maximum increase of 30% in code size).

The first step is to build an instantiation map that enumerates for each datatype and function declaration the full set of instantiations for each polymorphic type. A TFL program will be type checked by the HOL system and be annotated with polymorphic type identifiers such as $'a, 'b, \ldots$ when it is defined. In particular, type inference has been done for (mutually) recursive functions. The remaining task is to instantiate the generic types of a function with the actual types of arguments at its call sites.

The notation used in this section is as follows. A substitution rule $R = (t \hookrightarrow \{T\})$ maps an abstract type t to a set of its type instantiations; an instantiation set $S = \{R\}$ is a set of substitution rules; and an instantiation map $M = \{z \hookrightarrow S\}$ maps a datatype or a function z to its instantiation set S. We write $M.y$ for the value at field y in the map M; if $y \notin Dom\ M$ then $M.y$ returns an empty set. The union of two substitution sets $S_1 \cup_s S_2$ is $\{t \hookrightarrow S_1.t \cup S_2.t \mid t \in Dom\ S_1 \cup Dom\ S_2\}$. We write $\bigcup_s \{S\}$ for the combined union of a set of substitution rules. The union of two instantiation maps $M_1 \bigcup_m M_2$ is defined similarly. The composition of two instantiation sets S_1 and S_2, denoted as $S_1\ o_r\ S_2$, is $\{t \hookrightarrow \bigcup \{S_2.t \mid t \in Dom\ S_1\} \mid z \in Dom\ S_1\}$. And, the composition of an instantiation map M and a set S is defined as $M\ o_m\ S = \{z \hookrightarrow M.z\ o_r\ S \mid z \in Dom\ M\}$.

The instantiation information of each occurrence of a polymorphic function and datatype is coerced into an instantiation map during a syntax-directed bottom-up traversal. The main conversion rules Γ and Δ shown in Figure 3.1 build the instantiation map by investigating types and expressions, respectively. The rule for a single variable/function declaration is trivial and omitted here: we just need to walk over the right hand side of its definition. If a top-level function f is called in the body of another function top level g, then g must be visited first to generate an instantiation map M_g, and then f is visited to generate M_f; finally these two maps are combined to a new one, *i.e.* $((M_f \circ M_g.f) \cup_m M_g)$. The clauses in mutually recursive functions can be visited in an arbitrary order.

$$
\begin{array}{lcl}
\Gamma[[\tau]] & = & \{\}, \quad \text{for } \tau \in \{T, t\} \\
\Gamma[[\tau\ D]] & = & \{D \hookrightarrow \mathsf{match\_tp}\ (\mathsf{at\_tp}\ D)\ \tau\} \\
\Gamma[[\tau_1\ op_t\ \tau_2]] & = & \Gamma[[\tau_1]] \cup_m \Gamma[[\tau_2]], \quad \text{for } op_t \in \{\#, \rightarrow\} \\
\Delta[[i]] & = & \{\} \\
\Delta[[v : \tau]] & = & \Gamma[[\tau]] \\
\Delta[[[e],]] & = & \bigcup_m \{\Gamma[[e]]\} \\
\Delta[[p\ e]] & = & \Delta[[e]] \\
\Delta[[(c : \tau)\ e]] & = & \{\mathsf{con2tp}\ c \hookrightarrow \mathsf{match\_tp}\ (\mathsf{con2tp}\ c)\ \tau\} \\
 & & \cup_m \Gamma[[\tau]] \cup_m \Delta[[e]] \\
\Delta[[(f : \tau)\ e]] & = & \{f_{id} \hookrightarrow \mathsf{match\_tp}\ f_{id}\ \tau\} \cup_m \Gamma[[\tau]] \cup_m \Delta[[e]] \\
\Delta[[\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]] & = & \Delta[[e_1]] \cup_m \Delta[[e_2]] \cup_m \Delta[[e_3]] \\
\Delta[[\mathsf{case}\ e_1\ \mathsf{of}\ [((c : \tau)\ e_2) \rightsquigarrow e_3]_{|}]] & = & \Delta[[e_1]] \cup_m \bigcup_m \{\{\mathsf{con2tp}\ c \hookrightarrow \mathsf{match\_tp}\ (\mathsf{con2tp}\ c)\ \tau\} \\
 & & \cup_m \Delta[[e_2]] \cup_m \Delta[[e_3]]\} \\
\Delta[[\mathsf{let}\ v\ =\ e_1\ \mathsf{in}\ e_2]] & = & (\Delta[[e_1]]\ o_m\ \Delta[[e_2]].v) \cup_m \Delta[[e_2]] \\
\Delta[[[\lambda\, v.]^* e]] & = & \Delta[[e]]
\end{array}
$$

Figure 3.1: Build instantiation maps for polymorphic components.

This algorithm makes use of a couple of auxiliary functions provided by the HOL system. Function $\mathsf{con2tp}\ c$ maps a constructor c to the datatype to which it belongs; $\mathsf{at\_tp}\ \mathsf{D}$ returns σ if there is a datatype definition $\mathsf{datatype}\ \sigma = D\ \mathsf{of}\ \ldots$; when x is either a function name or a constructor, $\mathsf{match\_tp}\ x\ \tau$ matches the original type of x (*i.e.* the type when x is defined) with τ and returns a substitution set.

After the final instantiation map is obtained, we duplicate a polymorphic datatype / function for all combinations of its type instantiations, and replace each call of the polymorphic function with the call to its monomorphic clone with respect to the type. The automatic correctness proof for the transformation is trivial: each duplication of a polymorphic function computes the same function on the arguments of the instantiating types.

Now, we give a simple example to illustrate the transformation.

$$
\begin{array}{l}
\mathsf{datatype}\ \sigma = C\ \mathsf{of}\ 'a\,\#\,'b \qquad\qquad f\ (x :' a) = x \\
g\ (x :' c, y :' d) = \mathsf{let}\ (h :' d \rightarrow ('c\,\#\,'d)\ \sigma) = \lambda z :' d. \\
\qquad (C : ('c\,\#\,'d) \rightarrow ('c\,\#\,'d)\ \sigma)\ ((f :' c \rightarrow' c)\ x, (f :' d \rightarrow' d)\ z)\ \mathsf{in}\ h\ y \\
j = (g\ (1 : num, \bot : bool),\ g\ (\bot : bool, \top : bool))
\end{array}
$$

The algorithm builds the instantiation maps shown in Figure 3.2. Then, for datatype σ, function f, and function g, a monomorphic clone is created for each combination of instantiating types. Calls to the original functions are replaced with the appropriate copies of the right type. For example, function j is converted to $j = (g_{num\#bool}\ (1, \bot),\ g_{bool\#bool}\ (\bot, \top))$, where $g_{num\#bool}$ and $g_{bool\#bool}$ are

Investigate j : $M_j = \{g \hookrightarrow \{'c \hookrightarrow \{bool, num\}, 'd \hookrightarrow \{bool\}\}\}$
Investigate g : $M_g = \{f \hookrightarrow \{'a \hookrightarrow \{'c, 'd\}\}, \sigma \hookrightarrow \{'a \hookrightarrow \{'c\}, 'b \hookrightarrow \{'d\}\}\}$
Compose M_g and M_j: $M_{g \circ j} = M_g \circ M_j.g =$
$\{\ f \hookrightarrow \{'a \hookrightarrow \{bool, num\}\},\ \sigma \hookrightarrow \{'a \hookrightarrow \{bool, num\},\ 'b \hookrightarrow \{bool\}\}\ \}$
Union M_g and $M_{g \circ j}$: $M_{\{g,j\}} = M_g \cup_m M_j =$
$\{\ f \hookrightarrow \{'a \hookrightarrow \{bool, num\}\},\ g \hookrightarrow \{'c \hookrightarrow \{bool, num\},\ 'd \hookrightarrow \{bool\}\},$
$\sigma \hookrightarrow \{'a \hookrightarrow \{bool, num\},\ 'b \hookrightarrow \{bool\}\}\ \}$
Investigate f : no changes, $M_{\{f,g,j\}} = M_{\{g,j\}}$

Figure 3.2: An example set of instantiation maps.

the two clones of g. The correctness of j's conversion is proved based on the theorems showing that g's copies compute the same function as g with respect to the instantiating types: $\Vdash_{thm} g_{num\#bool} = g \ \wedge \ g_{bool\#bool} = g$.

3.1.2 Defunctionalization

In this section, we convert higher-order functions into equivalent first-order functions and hoist nested functions to the top level through a type-based closure conversion. After the conversion, no nested functions exist, and function call is made by dispatching on the closure tag followed by a top-level call.

Function closures are represented as algebraic data types in a way that, for each function definition, a constructor taking the free variables of this function is created. For each arrow type, we create a dispatch function, which converts the definition of a function of this arrow type into a closure constructor application. A nested function is hoisted to the top level with its free variables to be passed as extra arguments. After that, the calling to the original function is replaced by a calling to the relevant dispatch function passing a closure containing the values of this function's free variables. The dispatch function examines the closure tag and passes control to the appropriate hoisted function. Thus, higher-order operations on functions are replaced by equivalent operations on first-order closure values.

As an optimization, we first run a pass to identify all 'targeted' functions which appear in the arguments or outputs of other functions and record them in a side effect variable `Targeted`. Nontargeted functions need not be closure converted, and calls to them are made as usual. During this pass, we also find out the functions to be defined at the top level and record them in `Hoisted`. Finally, `Hoisted` contains all top-level functions and nested functions to be hoisted.

The conversion works on simple typed functions obtained by monomorphisation. We create a closure datatype and a dispatch function for each of the arrow types that targeted functions may have. A function definition is replaced by a binding to an application of the corresponding closure constructor to this function's free variables. Suppose the set of targeted functions of type τ is $\{f_i\ x_i = e_i \mid i = 1, 2, \dots\}$, then the following algebraic datatype and dispatch function are created, where tp_of and fv return the type and free variables of a term, respectively (and the type builder Γ will be described below):

$$
\begin{array}{lcll}
clos_\tau & = & cons^\tau_{f_1}\ \mathsf{of}\ \Gamma[[\mathsf{tp\_of}\ (\mathsf{fv}\ f_1)]] \mid cons^\tau_{f_2}\ \mathsf{of}\ \Gamma[[\mathsf{tp\_of}\ (\mathsf{fv}\ f_2)]] \mid \dots & \\
(dispatch_\tau(cons^\tau_{f_1}, x_1, y_1) & = & (f_1 : \Gamma[[\tau]])\ (x_1, y_1)) & \wedge \\
(dispatch_\tau(cons^\tau_{f_2}, x_2, y_2) & = & (f_2 : \Gamma[[\tau]])\ (x_2, y_2)) & \wedge \\
 & & \dots &
\end{array}
$$

As shown in Figure 3.3, the main translation algorithm inspects the references and applications of targeted functions and replaces them with the corresponding closures and dispatch functions. Function Γ returns the new types of variables. When walking over expressions, Δ replaces calls to unknown functions (*i.e.* those not presented in `Hoisted`) with calls to the appropriate dispatch function, and calls to known functions with calls to hoisted functions. In this case, the values of free variables are passed as extra arguments. Function references are also replaced with appropriate closures. Finally, `Redefn` contains all converted functions, which will be renamed and redefined in HOL at the top level.

Now, we show the technique to prove the equivalence of a source function f to its converted form f'. We say that a variable $v' : \tau'$ corresponds $v : \tau$ iff: (1) $v = v'$ if both τ and τ' are closure type or neither of them is. (2) $\forall x \forall x'.\, dispatch_{\tau'}(v', x') = v\ x$ if v' is a closure type and v is an arrow type, and x' corresponds to x; or vice versa. Then f' is equivalent to f iff they correspond to each other. The proof process is simple, as it suffices to simply rewrite with the old and new definitions of the functions.

As an example, the following higher-order program

$$
\begin{array}{l}
f\ (x : num) = x * 2 < x + 10 \\
g\ (s : num \to bool, x : num) = \\
\quad \mathsf{let}\ h_1 = \lambda y.\, y + x\ \mathsf{in}\ \mathsf{if}\ s\ x\ \mathsf{then}\ h_1\ \mathsf{else}\ \mathsf{let}\ h_2 = \lambda y.\, h_1\ y * x\ \mathsf{in}\ h_2 \\
k\ (x : num) = \mathsf{if}\ x = 0\ \mathsf{then}\ 1\ \mathsf{else}\ g\ (f, x)\ (k\ (x - 1))
\end{array}
$$

$$
\begin{array}{lcl}
\Gamma[[v : T]] & = & T \\
\Gamma[[v : \tau_1 \to \tau_2]] & = & \text{if } v \in \underline{\texttt{Targeted}} \text{ then } clos_{\tau_1 \to \tau_2} \text{ else } \tau_1 \to \tau_2 \\
\Gamma[[v : \tau\ D]] & = & \Gamma[[\tau]]\ D \\
\Gamma[[[v],]] & = & [\Gamma[[v]]], \\
\Delta[[v : \tau]] & = & \text{if } v \in \underline{\texttt{Targeted}} \text{ then } cons^{\tau}_{v} \text{ else } v : clos_{\tau} \\
\Delta[[[e],]] & = & [\Delta[[e]]], \\
\Delta[[p\ e]] & = & p\ (\Delta[[e]]) \\
\Delta[[c\ e]] & = & c\ (\Delta[[e]]) \\
\Delta[[(f : \tau)\ e]] & = & \text{if } f \in \underline{\texttt{Hoisted}} \text{ then } (\mathsf{new\_name\_of}\ f)\ (\Delta[[e]], \mathsf{fv}\ f) \\
 & & \text{else } dispatch_{\tau}\ (f : clos_{\tau}, \Delta[[e]]) \\
\Delta[[\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]] & = & \mathsf{if}\ \Delta[[e_1]]\ \mathsf{then}\ \Delta[[e_2]]\ \mathsf{else}\ \Delta[[e_3]] \\
\Delta[[\mathsf{case}\ e_1\ \mathsf{of}\ [c\ e_2 \rightsquigarrow e_3]_{|}]] & = & \mathsf{case}\ \Delta[[e_1]]\ \mathsf{of}\ [(\Delta[[c\ e_2]]) \rightsquigarrow \Delta[[e_3]]]_{|} \\
\Delta[[\mathsf{let}\ f = \lambda \overrightarrow{v}.\ e_1\ \mathsf{in}\ e_2]] & = & (\Phi[[f\ \overrightarrow{v} = e_1]]\ ;\ \Delta[[e_2]]) \\
\Delta[[\mathsf{let}\ v = e_1\ \mathsf{in}\ e_2]] & = & \mathsf{let}\ v = \Delta[[e_1]]\ \mathsf{in}\ \Delta[[e_2]] \quad \text{when } e_1 \text{ is not a } \lambda \text{ expression} \\
\Phi[[f_{id}\ (\overrightarrow{v} : \tau) = e]] = & & \\
\quad \text{let } e' = \Delta[[e]] \text{ in} & & \\
\quad \underline{\texttt{Redefn}} := \underline{\texttt{Redefn}} + (f_{id} \hookrightarrow \underline{\texttt{Redefn}}.f_{id} \cup \{(f_{id} : \tau \to \Gamma[[\mathsf{tp\_of}\ e']])\ \overrightarrow{v} = e'\} & & \\
\Phi[[[f_{decl}]_{\wedge}]] = [\Phi[[f_{decl}]]]; & &
\end{array}
$$

Figure 3.3: Remove higher-order functions through closure conversion.

is closure converted to

$$
\begin{array}{l}
\mathsf{datatype}\ clos_{\tau_1} = cons^{\tau_1}_{f} \\
\mathsf{datatype}\ clos_{\tau_2} = cons^{\tau_2}_{h_1}\ \mathsf{of}\ num \mid cons^{\tau_2}_{h_2}\ \mathsf{of}\ num \\
dispatch_{\tau_1}\ (cons^{\tau_1}_{f} : clos_{\tau_1}, x : num) = f'\ x \ \ \wedge \ \ f'\ x = x * 2 < x + 10 \\
dispatch_{\tau_2}\ (cons^{\tau_2}_{h_1}\ y : clos_{\tau_2},\ x : num) = h'_1\ (y, x)) \ \wedge \\
dispatch_{\tau_2}\ (cons^{\tau_2}_{h_2}\ y : clos_{\tau_2},\ x : num) = h'_2\ (y, x)) \ \wedge \\
h'_1\ (y, x) = y + x \ \ \wedge \ \ h'_2\ (y, x) = h'_1(y, x) * x \\
g'\ (s : clos_{\tau_1}, x : num) = \mathsf{if}\ dispatch_{\tau_1}(s, x)\ \mathsf{then}\ cons^{\tau_2}_{h_1}\ x\ \mathsf{else}\ cons^{\tau_2}_{h_2}\ x \\
k'\ (x : num) = \mathsf{if}\ x = 0\ \mathsf{then}\ 1\ \mathsf{else}\ g\ (cons^{\tau_1}_{f}, x), (k'\ (x - 1)) \\
\text{where } \tau_1 \text{ and } \tau_2 \text{ stand for arrow types } num \to bool \text{ and } num \to num, \text{ respectively}
\end{array}
$$

And the following theorems (which are proved automatically) justify the correctness of this conversion:

$$
\begin{array}{l}
\Vdash_{thm}\ \ f = f' \qquad \Vdash_{thm}\ \ k' = k \\
\Vdash_{thm}\ \ (\forall x.\, dispatch_{\tau_1}\ (s', x) = s\ x) \Rightarrow \forall x \forall y.\, dispatch_{\tau_2}\ (g'\ (s', x), y) = (g\ (s, x))\ y
\end{array}
$$

3.1.3 Lightweight Closure Conversion

In many cases, we found that the defunctionalization was more complicated and inefficient than necessary; thus, we develop a lightweight closure conversion.

This conversion captures the free variables for nested functions in an environment as passed to the function as an extra argument. The function body is modified so that references to free variables are now references to the environment.

When a function is referenced, the function is paired with the environment as a closure.

The clos_init rule creates a closure for closing the first free variable v in the body of function f. Administrative term `clos` is used to record the transformed function and the environment. By definition $\forall c.\, \texttt{clos}\ (f, c) = f$. We uses tactics (at the meta-level) to control the application of this rule such that it will not be applied to functions without free variables. Rule clos_one handles extra free variables and builds the environment as a tuple. It is applied repeatedly until no free variable remains in the function body.

$$
\begin{array}{ll}
[\mathsf{clos\_init}] & \texttt{let}\ f = g\ v\ \texttt{in}\ e\ f \longleftrightarrow \\
 & \texttt{let}\ f = \texttt{clos}\ (g,\ v)\ \texttt{in}\ e\ (f\ v) \\
[\mathsf{clos\_one}] & \texttt{let}\ f = \texttt{clos}\ ((\lambda c.\, g\ v\ c),\ c)\ \texttt{in}\ e\ (f\ c) \longleftrightarrow \\
 & \texttt{let}\ f = \texttt{clos}\ ((\lambda (c, v).\, g\ v\ c), (c, v))\ \texttt{in}\ e\ (f\ (c, v))
\end{array}
$$

We show below a simple example, where f' is an abbreviation of $\lambda x.\, x + y + z$. We explicitly write out the g and e for clarity. The final step performs explicit tuple allocation, where #1 and #2 take the first and second components of a tuple, respectively.

$$
\begin{array}{l}
\texttt{let}\ f = \lambda x.\, x + y + z\ \texttt{in}\ f\ 1 \quad = \\
\texttt{let}\ f = (\lambda y.\, f')\ y\ \texttt{in}\ (\lambda f.\, f\ 1)\ f \longleftrightarrow \\
\texttt{let}\ f = \texttt{clos}\ ((\lambda y.\, f'),\ y)\ \texttt{in}\ (\lambda f.\, f\ 1)\ (f\ y) \quad = \\
\texttt{let}\ f = \texttt{clos}\ ((\lambda y.\, (\lambda z.\, \lambda y.\, f')\ z\ y),\ y)\ \texttt{in}\ (\lambda f.\, f\ 1)\ (f\ y) \longleftrightarrow \\
\texttt{let}\ f = \texttt{clos}\ ((\lambda (y, z).\, f'),\ (y, z))\ \texttt{in}\ (\lambda f.\, f\ 1)\ (f\ (y, z)) \ = \\
\texttt{let}\ f = \lambda (y, z).\, \lambda x.\, x + y + z\ \texttt{in}\ f\ (y, z)\ 1 \quad = \\
\texttt{let}\ f\ c\ x = \texttt{let}\ y = \#1\ c\ \texttt{in}\ \texttt{let}\ z = \#2\ c\ \texttt{in}\ x + y + z\ \texttt{in} \\
f\ (y, z)\ 1
\end{array}
$$

3.1.4 Pattern Matching

This conversion to nested case expressions is based on Augustsson's original work [7], which was adapted by Slind [106] for function description in HOL. A preprocessing pass is first performed to deal with incomplete and overlapping patterns: incomplete patterns are made complete by adding rows for all missing constructors, and overlapping patterns are handled by replacing a value with possible constructors. Note that this approach may make the pattern exponentially larger because no heuristics are used to choose the "best" order in which subterms

of any term are to be examined.

The translation rule Δ shown below converts patterns $[pat_i \rightsquigarrow rhs_i]_|$ into a nested case expression. It takes two arguments: a stack of variables that are yet to be matched, and a matrix whose rows correspond to the clauses in the pattern. All rows are of equal length, and the elements in a column should have the same type.

Conversion Δ proceeds from left to right, column by column. At each step, the first column is examined. If each element in this column is a variable, then the head variable z in the stack is substituted for the corresponding v_i for the right hand side of each clause. If each element in the column is the application of a constructor for type τ, and τ contains constructor $C_1, \ldots, C_n$, then the rows are partitioned into n groups of size $k_1, \ldots, k_n$ according to the constructors. After partitioning, a row $(C(\bar{p}) :: pats;\ rhs)$ has its lead constructor discarded, resulting in a row expression $(\bar{p} \mathbin{@} pats; rhs)$. Here :: is the list constructor, and @ appends the second list to the first one. If constructor C_i has type $\tau_1 \to \cdots \to \tau_j \to \tau$, then a set ν_i of new variables $v_1, \ldots, v_j$ are pushed onto the stack. Finally, the results for all groups are combined into a case expression on the head of the stack.

$$\Delta \left(\begin{array}{l} z :: stack \\ \hline v_1 :: pats_1 \rightsquigarrow rhs_1, \\ \ldots \\ v_n :: pats_n \rightsquigarrow rhs_n \end{array} \right) = \Delta \left(\begin{array}{l} stack \\ \hline pats_1 \rightsquigarrow rhs_1[z \leftarrow v_1], \\ \ldots \\ pats_n \rightsquigarrow rhs_n[z \leftarrow v_2] \end{array} \right), \text{ and}$$

$$\Delta \left(\begin{array}{l} z :: stack \\ \hline C_1\ \overline{p_{11}} :: pats_{11} \rightsquigarrow rhs_{11}, \\ \ldots \\ C_n\ \overline{p_{1k_1}} :: pats_{1k_1} \rightsquigarrow rhs_{1k_1} \\ C_n\ \overline{p_{n1}} :: pats_{n1} \rightsquigarrow rhs_{11}, \\ \ldots \\ C_n\ \overline{p_{1k_n}} :: pats_{nk_n} \rightsquigarrow rhs_{nk_n} \end{array} \right) = \mathsf{tp\_case}\ (\lambda\nu_1.M_1)\ \ldots\ (\lambda\nu_n.M_n)\ z$$

$$\text{where } M_i = \Delta \left(\begin{array}{l} \nu_1 :: stack \\ \hline \overline{p_{i1}} \mathbin{@} pats_{k1} \rightsquigarrow rhs_{k1}, \\ \ldots \\ \overline{p_{ik_i}} \mathbin{@} pats_{ik_i} \rightsquigarrow rhs_{ik_i} \end{array} \right) \text{ for } i = 0, \ldots, n$$

When a datatype tp with n constructors is declared, a case expression theorem $\forall x.\, tp\_\mathsf{case}\ f_1\ \ldots\ f_n\ (C_i\ x) \equiv f_i\ x$ for $i = 1, \ldots, n$ is stored in HOL. For example, the case expression for natural number is $(\mathsf{num\_case}\ b\ f\ 0 = b) \wedge (\mathsf{num\_case}\ b\ f\ (\mathsf{Suc}\ n) = f\ n)$.

For example, this step translates the Greatest Common Divisor function *gcd* to a form taking only one argument:

$$
\begin{array}{l}
gcd\ (0, y) = y \qquad gcd\ (\mathsf{Suc}\ x, 0) = \mathsf{Suc}\ x \\
gcd\ (\mathsf{Suc}\ x, \mathsf{Suc}\ y) = \mathsf{if}\ y \leq x\ \mathsf{then}\ gcd\ (x - y, \mathsf{Suc}\ y)\ \mathsf{else}\ gcd\ (\mathsf{Suc}\ x, y - x) \\
\qquad \Rightarrow \\
gcd\ z = \mathsf{pair\_case}\ (\lambda v\ v_1.\ \mathsf{num\_case}\ v_1 (\lambda v_2.\ \mathsf{num\_case}\ (\mathsf{Suc}\ v_2) \\
\quad (\lambda v_3.\ \mathsf{if}\ v_3 \leq v_2\ \mathsf{then}\ gcd\ (v_2 - v_3, \mathsf{Suc}\ v_3)\ \mathsf{else}\ gcd\ (\mathsf{Suc}\ v_2, v_3 - v_2))\ v_1)\ v)\ z
\end{array}
$$

In the next step, case expressions are interpreted as conditional expressions based on the following theorem:

$$
\begin{array}{l}
\mathsf{tp\_case}\ (\lambda x.f_1\ x)\ (\lambda x.f_2\ x)\ \ldots\ z = \\
\mathsf{if}\ \mathsf{is\_}C_1\ z\ \mathsf{then}\ f_1\ (\mathsf{destruct}_{C_1}\ z)\ \mathsf{else}\ \mathsf{if}\ \mathsf{is\_}C_2\ z\ \mathsf{then}\ f_2\ (\mathsf{destruct}_{C_2}\ z)\ \mathsf{else}\ldots
\end{array}
$$

Here, operator $\mathsf{is\_}C_i$ tells whether a variable matches the i^{th} constructor C_i, *i.e.* $\mathsf{is\_}C_i\ (C_j\ x) = \top$ iff $i = j$; and operator $\mathsf{destruct}_{C_i}$ is the destructor function for constructor C_i. For example, $\mathsf{destruct}_{\mathsf{Suc}}\ (\mathsf{Suc}\ x) = x$. These operators will be implemented as datatype access operations in later compilation phases. In addition, an optimization is performed to tuple variables: if an argument x has type $\tau_1 \# \ldots \# \tau_n$, then it is replaced by a tuple of new variables $(x_1, \ldots, x_n)$. Superfluous branches and 'let' bindings are also removed. In this manner, the *gcd* function is converted to

$$
\begin{array}{l}
gcd\ (z_1, z_2) = \mathsf{if}\ z_1 = 0\ \mathsf{then}\ z_2 \\
\quad \mathsf{else\ let}\ v_2 = \mathsf{destruct}_{Suc}\ z_1\ \mathsf{in} \\
\quad \mathsf{if}\ z_2 = 0\ \mathsf{then\ Suc}\ v_2\ \mathsf{else\ let}\ v_3 = \mathsf{destruct}_{Suc}\ z_2\ \mathsf{in} \\
\quad\quad \mathsf{if}\ v_3 \leq v_2\ \mathsf{then}\ gcd\ (v_2 - v_3, \mathsf{Suc}\ v_3)\ \mathsf{else}\ gcd\ (\mathsf{Suc}\ v_2, v_3 - v_2)
\end{array}
$$

3.2 Main Conversions in the Mid-end

The mid-end converts LF1 functions to LF2 functions; it contains most of the optimizations of the entire compiler. As an illustration, we show some intermediate forms of a simple program f_1 in Figure 3.4.

3.2.1 Normalization

We may normalize a program to a certain format to make subsequent transformations easier. For example, we simplify conditional expressions whose conditions can be evaluated to $\top$ (true) or $\bot$ (false), and apply De Morgan laws to move

```
fact i =def
  if i = 0 then 1
  else i * fact (i − 1)
f1 (k0, k1, k2) =def
  let y = k2 + 100 in
  let g (x, y) = y − (x ∗ k0) in
  let z =
    if fact 3 < 10 ∧ y + 2 ∗ k1 > k0
    then g (k1, k2) else y
  in z ∗ y
```

(a)

```
fact =
  λv1.
  if v1 = 0 then 1
  else
    let v2 = v1 − 1 in
    let v3 = fact v2 in
    let v4 = v1 ∗ v3 in v4
```

(b)

```
f1 =
  letrec v4 = (
    λv11λ(v12, v13).
    let v14 = v11 ∗ v12 in
    let v15 = v13 − v14
    in v15)
  in
  λ(v1, v2, v3).
  let v5 = v3 + 100 in
  let v6 = 2 ∗ v2 in
  let v7 = v5 + v6 in
  let v8 = (
    if v7 ≤ v1 then v5
    else
      let v10 = v4 v1 (v2, v3)
      in v10)
  in
  let v9 = v5 ∗ v8 in v9
```

(c)

Figure 3.4: (a) Source programs fact and f_1; (b) fact's intermediate form before register allocation; (c) f_1's intermediate form after closure conversion.

negations in over the conjunction, disjunction, and conditional expressions. Note that the default implication set includes rewrites rules for Boolean operations (*e.g.* $e \wedge \bot = \bot$) and Presburger arithmetic (*e.g.* $b + a - b = a$) such that relevant optimizations are performed automatically. To facilitate rewriting, we introduce administrative term atom to mark a constant or single variable. By definition $\mathsf{atom} = \lambda x.\, x$.

[if_and]	if $c_1 \wedge c_2$ then e_1 else $e_2 \longleftrightarrow$
	let $x = e_2$ in (if c_1 then (if c_2 then e_1 else x) else x)
[if_or]	if $c_1 \vee c_2$ then e_1 else $e_2 \longleftrightarrow$
	let $x = e_1$ in (if c_1 then x else (if c_2 then x else e_2))
[if_gt]	if $a > b$ then e_1 else $e_2 \longleftrightarrow$ if $a \leq b$ then e_2 else e_1
[if_ge]	if $a \geq b$ then e_1 else $e_2 \longleftrightarrow$ if $b \leq a$ then e_1 else e_2
[if_true]	if $\top$ then e_1 else $e_2 \longleftrightarrow e_1$
[if_false]	if $\bot$ then e_1 else $e_2 \longleftrightarrow e_2$

In a high-level program, the value of a compound expression is computed by a sequence of low-level instructions. By defining every intermediate result of computation as a variable, we can convert such compound expressions into sequences of let-expressions corresponding to assembly instructions. This leads to one IR that is a combination of K-normal forms [12] and A-normal forms [31], where intermediate computations and their results are made explicit. The core

of the transformation is to remove compound expressions so that every target of basic operations such as arithmetic operations and function applications is now a variable.

The first step is to perform a continuation-passing style (CPS) transformation. It repeatedly rewrites with the following theorems in a syntax-directed manner to transform a program into its continuation form. Here, $\mathsf{C}\ e\ k$ denotes the application of the continuation k to an expression e, and its value is equal to $k\ e$.

$$
\begin{array}{ll}
[\mathsf{C\_intro}] & e \longleftrightarrow \mathsf{C}\ e\ (\lambda x.x) \\
[\mathsf{C\_binop}] & \mathsf{C}\ (e_1\ \mathsf{op}_b\ e_2)\ k \longleftrightarrow \\
 & \mathsf{C}\ e_1\ (\lambda x.\ \mathsf{C}\ e_2\ (\lambda y.\ \mathsf{C}\ (x\ \mathsf{op}_b\ y)\ k)) \\
[\mathsf{C\_pair}] & \mathsf{C}\ (e_1, e_2)\ k \longleftrightarrow \\
 & \mathsf{C}\ e_1\ (\lambda x.\ \mathsf{C}\ e_2\ (\lambda y.\ \mathsf{C}\ (x, y)\ k)) \\
[\mathsf{C\_array}] & \mathsf{C}\ (e_1[e_2])\ k \longleftrightarrow \mathsf{C}\ e_2\ (\lambda x.\ \mathsf{C}\ (e_1[x])\ k) \\
[\mathsf{C\_let\_ANormal}] & \mathsf{C}\ (\mathsf{let}\ v = e\ \mathsf{in}\ f\ v)\ k \longleftrightarrow \\
 & \mathsf{C}\ e\ (\lambda x.\ \mathsf{C}\ (f\ x)\ (\lambda y.\ k\ y)) \\
[\mathsf{C\_let\_KNormal}] & \mathsf{C}\ (\mathsf{let}\ v = e\ \mathsf{in}\ f\ v)\ k \longleftrightarrow \\
 & \mathsf{C}\ e\ (\lambda x.\ \mathsf{C}\ x\ (\lambda y.\ \mathsf{C}\ (f\ y)\ (\lambda z.\ k\ z))) \\
[\mathsf{C\_abs}] & \mathsf{C}\ (\lambda v.\ f\ v)\ k \longleftrightarrow \mathsf{C}\ (\lambda v.\ (\mathsf{C}\ (f\ v)\ (\lambda x.\ x)))\ k \\
[\mathsf{C\_app}] & \mathsf{C}\ (f\ e)\ k \longleftrightarrow \\
 & \mathsf{C}\ f\ (\lambda g.\ \mathsf{C}\ e\ (\lambda x.\ \mathsf{C}\ (g\ x)\ (\lambda y.\ k\ y))) \\
[\mathsf{C\_cond}] & \mathsf{C}\ (\mathsf{if}\ (c_1\ \mathsf{op}_r\ c_2)\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ k \longleftrightarrow \\
 & \quad \mathsf{C}\ c_1\ (\lambda p.\ \mathsf{C}\ c_2\ (\lambda q.\ \mathsf{C}\ (\mathsf{if}\ (p\ \mathsf{op}_r\ q)\ \mathsf{then} \\
 & \quad\quad \mathsf{C}\ e_1\ (\lambda x.\ x)\ \mathsf{else}\ \mathsf{C}\ e_2\ (\lambda y.\ y))\ (\lambda z.\ k\ z)))
\end{array}
$$

The next step converts the continuation form into a readable, 'let'-based normal form using the following rewrite rules. Since the logical framework takes care of program scoping and substitution implicitly, during the rewriting, fresh variables are generated and bound to the results of intermediate computations automatically.

$$
\begin{array}{ll}
[\mathsf{C\_atom}] & \mathsf{C}\ (\mathsf{atom}\ v)\ k \longleftrightarrow \mathsf{atom}\ v \\
[\mathsf{C\_to\_let}] & \mathsf{C}\ e\ k \longleftrightarrow \mathsf{let}\ x = e\ \mathsf{in}\ k\ x
\end{array}
$$

The following example illustrates this transformation, where c_1 and c_2 are the two constructors of a datatype.

Original: $f\ (x, y, z) = \mathsf{case}\ x - y - z\ \mathsf{of}\ c_1\ a \Rightarrow f(x-1, a, y) \mid c_2\ b \Rightarrow b + y$
Converted: $f\ (x, y, z) =$
$\quad \mathsf{let}\ v_1 = x - y - z\ \mathsf{in}$
$\quad\quad \mathsf{case}\ v_1\ \mathsf{of}\ c_1\ a \Rightarrow \mathsf{let}\ v_2 = x - 1\ \mathsf{in}\ f(v_2, a, y) \mid c_2\ b \Rightarrow b + y$

3.2.2 Optimizations

3.2.2.1 Dead Code Elimination

When a variable will not be referenced anymore, its assignment can be removed. Rule elim_let eliminates unused variable/function definitions. It requires that x does not appear free in e_2 (note that x's appearing in e_2 is denoted by $e_2\ x$).

$$[\text{elim\_let}] \quad \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \longleftrightarrow e_2$$

3.2.2.2 β Conversion and Constant Folding

It is often useful to reduce $\mathsf{let}\ v_1 = v_2\ \mathsf{in}\ e\ v_1$ to $e\ v_2$ when v_2 is a single variable or a constant. Note that, after some optimization, an expression may include only constant values, thus creating an opportunity for constant folding. This is accomplished by the built-in decision procedure for integer and real number arithmetic.

$$[\beta\text{\_reduct}] \qquad \mathsf{let}\ x = \mathsf{atom}\ v\ \mathsf{in}\ e\ x \longleftrightarrow e\ v$$

3.2.2.3 Common Subexpression Elimination

This optimization avoids redundant evaluation of the same expression by reusing the result of the first evaluation.

$$[\text{cse}] \quad \texttt{let}\ x = e\ \texttt{in}\ f\ e \longleftrightarrow \texttt{let}\ x = e\ \texttt{in}\ f\ x\ x$$

3.2.2.4 Tail Recursion

As discussed before, we convert tail recursive functions to while loops, which may create opportunities for loop optimizations.

3.2.2.5 Code Motion

Instruction scheduling may be used to maximize the throughput of a processor pipeline by reordering unrelated statements. Loop hoisting is another example of code motion which lifts code out of a loop. Before applying the loop_hoist rule, we will apply instruction scheduling on the loop body to move those "let" expressions not referring to y to the beginning of the body. For example, $\mathsf{while}\ c\ (\lambda y.\ \texttt{let}\ z = y + 2\ \texttt{in let}\ x = e\ \texttt{in}\ x + z)$ is converted to $\texttt{let}\ x = e\ \texttt{in}\ \mathsf{while}\ c\ (\lambda y.\ \texttt{let}\ z = y + 2\ \texttt{in}\ x + z)$.

[inst_schd]
$$\texttt{let}\ x = e_1\ \texttt{in}\ \texttt{let}\ y = e_2\ \texttt{in}\ f\ x\ y\ \longleftrightarrow$$
$$\texttt{let}\ y = e_2\ \texttt{in}\ \texttt{let}\ x = e_1\ \texttt{in}\ f\ x\ y$$

[loop_hoist]
$$\mathsf{while}\ c\ (\lambda y.\ \texttt{let}\ x = e\ \texttt{in}\ f\ x\ y)\ \longleftrightarrow$$
$$\texttt{let}\ x = e\ \texttt{in}\ \mathsf{while}\ c\ (\lambda y.\ f\ x\ y)$$

3.2.2.6 Inline Expansion

This transformation replaces calls to small functions with their bodies. If the size of the body e in a function f is less than a specific threshold t, f will be expanded. The prover will take care of variable name conflicts introduced by the inlining by capturing program scopes and renaming variables automatically. For a recursive function, we avoid code explosion by expanding its body for only a certain number of times. The resulting expression is further simplified by other transformations like constant folding until no more simplications can be made. Here, fun is an administrative term marking a small function.

[mark_fun] $\quad \mathsf{let}\ f = \lambda x.e_1\ x\ \mathsf{in}\ e_2\ f\ \longleftrightarrow$
$$\mathsf{let}\ f = \mathsf{fun}\ (\lambda x.e_1\ x)\ \mathsf{in}\ e_2\ f \quad \Leftarrow \quad \text{size}\ e_1 < t$$

[inline_fun] $\quad \mathsf{let}\ f = \mathsf{fun}\ e_1\ \mathsf{in}\ e_2\ f\ \longleftrightarrow\ e_2\ e_1$

3.2.2.7 SSA (Static Single-Assignment) Form

In the SSA format, each variable has only one definition in the program text. This format paves the way for our register allocation algorithm. The core is to rename all bound variables of a program to fresh names. Initially, all free variables in a function are replaced with fresh variables beginning with a "v". Then any variable in the left-hand side of a let-expression is substituted by a fresh new variable. As a result, an α-equivalent expression is returned. Note that HOL regards α-equivalent expressions to be the same expressions.

3.2.3 Register Allocation

One of the most sophisticated processes in the compiler is register allocation. Our rewriting-based algorithm is a simple greedy algorithm with backtracking for early spilling.

The basic policy of register allocation is to avoid registers already assigned to live variables. Variables live at the same time should not be allocated to the same

register. We adopt a naming convention: variables yet to be allocated begin with v, variables spilled begin with t (temporary in the stack), and those in registers begin with r (register variable). Notation _ matches a variable of any of these kinds. Notations $\hat{v}$, $\hat{r}$, and $\hat{t}$ stand for a fresh variable, a unused register, and a new stack location, respectively. avail e returns the set of available registers after allocating e. Administrative terms app, save, and restore are all defined as $\lambda x.x$. app is used to mark function applications. Finally, to $(v, l) = l$ indicates that variable v is allocated to resource l.

When variable v in expression let $v = e_1$ in e_2 v is to be assigned a register, the live variables to be considered are just the free variables in e_2 excluding v. If live variables do not use up all the machine registers, then we pick an available register and assign v to it by applying rule assgn_reg. Otherwise, we spill to the memory a variable consuming a register, and assign this register to v. In some cases, we prefer to spill a variable as early as possible: in the early_spill rule, variable w's value is spilled from r for future use; but r may not be allocated to v in the subsequent allocation. When encountering a memory variable in later phases, we need to generate code that will restore its value from the memory to a register (the $\hat{v}$ in rule restore will be assigned a register by the subsequent application of rule assgn_reg).

$$
\begin{array}{ll}
[\text{assgn\_reg}] & \text{let } v = e_1 \text{ in } e_2\ v \longleftrightarrow \\
 & \text{let } \hat{r} = e_1 \text{ in } e_2 \text{ to}(v, \hat{r}) \quad \Leftarrow \quad \text{avail } e_2 \neq \{\} \\
[\text{spill}] & \text{let } v = e_1 \text{ in } e_2\ v \text{ to}(w, r) \longleftrightarrow \\
 & \text{let } \hat{t} = \text{save } r \text{ in let } r = e_1 \text{ in} \\
 & e_2 \text{ to}(v, r) \text{ to}(w, \hat{t}) \quad \Leftarrow \quad \text{avail } e_2 = \{\} \\
[\text{early\_spill}] & \text{let } v = e_1 \text{ in } e_2\ v \text{ to}(w, r) \longleftrightarrow \\
 & \text{let } \hat{t} = \text{save } r \text{ in let } v = e_1 \text{ in } e_2\ v \text{ to}(w, \hat{t}) \\
 & \quad \Leftarrow \quad \text{avail } e_2 = \{\} \\
[\text{restore}] & e \text{ to}(v, t) \longleftrightarrow \text{let } \hat{v} = \text{restore } t \text{ in } e\ \hat{v} \\
[\text{caller\_save}] & \text{let } \_ = \text{app } f \text{ in } e\ \_ \text{ to}(w, r) \longleftrightarrow \\
 & \text{let } \hat{t} = \text{save } r \text{ in let } \_ = \text{app } f \text{ in } e\ \_ \text{ to}(w, \hat{t}) \\
[\text{spill\_if}] & \text{let } \_ = \text{if } e_1 \text{ then } e_2 \text{ to}(w, r_1) \\
 & \qquad \text{else } e_3 \text{ to}(w, r_2) \text{ in } e_4 \text{ to}(w, r_0) \longleftrightarrow \\
 & \text{let } \hat{t} = \text{save } r_0 \text{ in} \\
 & \text{let } \_ = \text{if } e_1 \text{ then } e_2 \text{ to}(w, \hat{t}) \text{ else } e_3 \text{ to}(w, \hat{t}) \\
 & \text{in } e_4 \text{ to}(w, \hat{t}) \quad \Leftarrow \quad \neg(r_0 = r_1 = r_2)
\end{array}
$$

Saving is necessary not only when registers are spilled, but also when functions are called. Our compiler adopts the *caller-save* convention, so every function call

is assumed to destroy the values of all registers. Therefore, we need to save the values of all registers that are live at that point. In addition, as we allocate the two branches of a conditional expression separately, a variable may be assigned different registers by the branches. This will contradict the convention that a variable should be assigned only one register. In this case, we will early spill it through the spill_if rule.

In the final step, all save, store, and to in an expression are eliminated, resulting in an equivalent expression containing only register variables and stack variables. In practice, in order to improve the performance, we do not have to perform equivalence check for every rewrite step. Instead, after all the rewrites are done, we apply the following rules to obtain an expression that is α-equivalent to the original expression. We call such methods, which performs the proof a posteriori, *offline* methods.

$$\begin{array}{ll} [\mathsf{elim\_save}] & \mathsf{let}\ t = \mathsf{save}\ r\ \mathsf{in}\ e\ t \longleftrightarrow e\ r \\ [\mathsf{elim\_store}] & \mathsf{let}\ r = \mathsf{store}\ t\ \mathsf{in}\ e\ r \longleftrightarrow e\ t \end{array}$$

3.2.3.1 Offline Register Allocation

We may use an offline (third-party) algorithm, *e.g.* a graph coloring allocation algorithm, to produce an allocation scheme. Interestingly, the algorithm itself does not have to be verified; instead, the computed coloring can be taken and used to build a term incorporating the required spilling, and this term can be shown to be α-equivalent to the one before allocation. This nice trick was first noticed by Hickey and Nogin [43] and is also used by Leroy [58]. In [63] we also used a graph coloring allocation algorithm without having to verify its correctness.

3.2.3.2 Example

In order to see the effect of spilling and restoring, we specify the number of available registers to be 3 when running the allocator for the example function f_1. The resulting intermediate form is shown at the left of Figure 3.5.

3.2.4 Exposing Heap and Stack

This phase places heap objects and stack objects in the memory. To model the memory, we introduce a function m mapping addresses to values. Heap variables and stack variables are indexed indirectly through the heap register *hp* and frame

```
f1 =
  λ(r0, r1, r2).
  let m1 = r2 in let m2 = r0 in
  let r0 = m1 in let r0 = r0 + 100 in
  let m3 = r0 in let r0 = 2 * r1 in
  let r2 = m3 in let r0 = r2 + r0 in
  let r2 = m2 in
  let r0 = (
    if r0 ≤ r2 then let r0 = m3 in r0
    else
      let r0 = r2 * r1 let r1 = m1 in
      let r0 = r1 − r0 in r0)
  in
  let r1 = m3 in let r0 = r1 * r0
  in r0
```

```
program: f1
input: (r0, r1, r2)
output: r0
 (l1   {m1 := r2} ⊎ {m2 := r0} ⊎
       {r0 := m1} ⊎ {r0 := r0 + 100} ⊎
       {m3 := r0} ⊎ {r0 := 2 * r1} ⊎
       {r2 := m3} ⊎ {r0 := r2 + r0} ⊎
       {r2 := m2}
  l2)  ⊎
 (l2   ifgoto (r0 ≤ r2) l3 l4) ⊎
 (l4   {r0 := r2 * r1} ⊎ {r1 := m1} ⊎
       {r0 := r1 − r0}
  l5)  ⊎
 (l3   {r0 := m3} l5) ⊎
 (l5   {r1 := m3} ⊎ {r0 := r1 * r0} l6)
```

Figure 3.5: f_1's FIL (left) and SAL (right)

register fp, respectively. Stack register sp is also used to keep track of the current stack pointer. They are global variables whose initial values satisfying $fp = hp{-}1 \wedge sp = fp - 1$, *e.g.* the heap space (with increasing addresses) and stack space (with decreasing addresses) are adjacent. As the first slot in the frame is reserved for the return address, the stack space of a function starts from $fp - 1$. A stack variable t_i is represented by $m[fp - i - 1]$; and a heap variable $a[r_i]$ is by $m[hp + \hat{a} + r_i]$ where $\hat{a}$ is the starting address of heap object a. Here, we assume variable values are integers for simplicity.

Consider the following (unoptimized) example. We introduce an administrative term $\texttt{let}_m$ whose semantics is the same as let to mark the "let" expressions involving memory access.

$$
\begin{array}{l}
f_1\ (r_0, r_1) = \\
\texttt{let}_\texttt{m}\ t_0 = r_0\ \texttt{in let}\ r_1 = r_0 * r_0\ \texttt{in} \\
\texttt{let}_\texttt{m}\ a_1 = \texttt{new}\ (\texttt{int}, 3)\ \texttt{in let}_\texttt{m}\ a_2 = \texttt{new}\ (\texttt{int}, 2)\ \texttt{in} \\
\texttt{let}\ r_2 = 0\ \texttt{in let}\ r_0 = r_1 + 10\ \texttt{in} \\
\texttt{let}_\texttt{m}\ a_1[r_2] = r_1\ \texttt{in let}_\texttt{m}\ a_2[r_2 + 1] = r_0\ \texttt{in} \\
\texttt{let}_\texttt{m}\ r_1 = f_2\ (r_0, a_2)\ \texttt{in} \\
\texttt{let}\ r_2 = a_1[2]\ \texttt{in let}_\texttt{m}\ r_0 = t_0\ \texttt{in} \\
(r_0, r_1, r_2)
\end{array}
$$

$$
\begin{array}{l}
f_2\ (r_0, a) = \\
\texttt{let}\ r_1 = r_0 * 5\ \texttt{in let}_\texttt{m}\ a[0] = r_0 + r_1\ \texttt{in}\ (r_0, r_1)
\end{array}
$$

We use an offline analysis to allocate the heap for the arrays and calculate the start addresses of the arrays, then generate a term representing the allocation. We can prove that this term equals the above one by eliminating all the "$\texttt{let}_m$"s in

the two terms, *e.g.* $\forall m.\, f_1\ (r_0, r_1) = (\texttt{let}\ (v, m) = f_1'\ (r_0, r_1, m)\ \texttt{in}\ v)$.

$$
\begin{array}{l}
f_1'\ (r_0, r_1, m) = \\
\texttt{let}_{\texttt{m}}\ m[fp - 1] = r_0\ \texttt{in}\ \texttt{let}\ r_1 = r_0 * r_0\ \texttt{in} \\
\texttt{let}_{\texttt{m}}\ a_1 = 0\ \texttt{in}\ \texttt{let}_{\texttt{m}}\ a_2 = 3\ \texttt{in} \\
\texttt{let}\ r_2 = 0\ \texttt{in}\ \texttt{let}\ r_0 = r_1 + 10\ \texttt{in} \\
\texttt{let}_{\texttt{m}}\ m[hp + a_1 + r_2] = r_1\ \texttt{in} \\
\texttt{let}_{\texttt{m}}\ m[hp + a_2 + r_2 + 1] = r_0\ \texttt{in} \\
\texttt{let}_{\texttt{m}}\ (r_1, m) = f_2'\ (r_0, a_2, m)\ \texttt{in} \\
\texttt{let}_{\texttt{m}}\ r_2 = m[hp + a_1 + 2]\ \texttt{in}\ \texttt{let}_{\texttt{m}}\ r_0 = m[fp - 1]\ \texttt{in} \\
(r_0, r_1, r_2, m)
\end{array}
$$

$$
\begin{array}{l}
f_2'\ (r_0, a, m) = \\
\texttt{let}\ r_1 = r_0 * 5\ \texttt{in}\ \texttt{let}_{\texttt{m}}\ m[hp + a] = r_0 + r_1\ \texttt{in} \\
(r_0, r_1, m)
\end{array}
$$

If the offline allocator makes a mistake and produces $\texttt{let}_{\texttt{m}}\ a_0 = 0\ \texttt{in}\ \texttt{let}\ a_1 = 0\ \texttt{in}\ \ldots$, then the correctness proof will fail since the address spaces of the two arrays overlap and a write to the second array may pollute the content of the first one. The rewrite rules for heap allocation include the following, where p_h marks the starting address of the available heap space.

$$
\begin{array}{l}
[\mathsf{heap\_alloc}] \\
\texttt{let}_{\texttt{m}}\ a = \texttt{new}\ (\tau, n)\ \texttt{in}\ e\ (a[i]) \longleftrightarrow \\
\texttt{let}_{\texttt{m}}\ p_h = p_h + n * (\texttt{size}\ \tau)\ \texttt{in} \\
\texttt{let}_{\texttt{m}}\ a = p_h\ \texttt{in}\ e\ (m[hp + a + i * (\texttt{size}\ \tau)])
\end{array}
$$

We may implement a garbage collector in a similar manner. This requires more advanced data structures (*e.g.* a table recording the liveness information of the heap objects) and some collection algorithms (*e.g.* the mark-and-sweep algorithm). The core is to model the collector as an offline algorithm in the logic. We leave its implementation to further work.

Note that this conversion is only needed in back-end II since back-end I uses an imperative language HSL to handle the heap and stacks.

CHAPTER 4

COMPILING LOGIC SPECIFICATIONS : BACK-END

In this chapter, we describe two methods to construct the back-end. The first method [63] introduces an imperative language HSL to model the heap and stack information, and another language CFL to model the control flow. As shown in Figure 4.1, we define the operational semantics of two imperative languages explicitly. The translation from HSL to CFL then to ARM is validated in a *Compiler Verification* manner. The second approach [66] models the heap and stacks directly in the logic level (see Section 3.2.4), and generates structured abstract assembly (SAL) programs. Producing ARM code from SAL programs is trivial.

4.1 Back-end I

We first refine the LF2 form slightly by introducing combinators for sequential composition (sc), conditionals (cj), and tail-recursion (tr). This makes it more succinct to represent the control flow of a LF2 function.

$$\vdash_{def} \mathsf{sc}\ f_1\ f_2 \doteq f_1 \oslash f_2$$
$$\vdash_{def} \mathsf{cj}\ e\ f_1\ f_2 \doteq \lambda x.\ \mathsf{if}\ e\ x\ \mathsf{then}\ f_1\ x\ \mathsf{else}\ f_2\ x$$
$$\vdash_{def} \mathsf{tr}\ f_1\ f_2 \doteq \lambda x.\ \mathsf{if}\ f_1\ x\ \mathsf{then}\ x\ \mathsf{else}\ \mathsf{tr}\ f_1\ f_2\ (f_2\ x)$$
$$\vdash_{thm} (f\ x = \mathsf{if}\ f_1\ x\ \mathsf{then}\ f_2\ x\ \mathsf{else}\ f\ (f_3\ x)) \Leftrightarrow (f = \mathsf{sc}\ (\mathsf{tr}\ f_1\ f_3)\ f_2)$$

4.1.1 Imperative Languages

HSL is a simple imperative language that supports various structured control statements including blocks (BLK), sequential composition (SC), conditionals (CJ), and tail recursion (TR), plus an important structure for function call — FC. Figure 4.1 shows its syntax and semantics. Variables are divided into register variables, heap (global) variables, and stack (local) variables. A BLK structure is just a list of atomic instructions. An FC structure consists of an argument passing pair

$\texttt{op}^\texttt{b}$	::= $add \mid sub \mid mul \mid ror \mid lsr \mid asr \mid$	
	$\mid lsl \mid and \mid orr \mid eor \mid rsb \mid mla \ , \ldots$	arithmetic and bitwise operators
r	::= $r_0 \mid r_1 \mid \ldots \mid r_8$	register variable
v	::= $r \mid sk[.]$	register and stack variable
y	::= $w \mid r$	word constant and register
x	::= $w \mid v$	constant and variable
$inst$	::= $\texttt{op}^\texttt{b}\ r\ y\ y$	arithmetic and bitwise operation
	$\mid ldr\ r\ (hp[i] \mid sk[.]) \mid str\ (hp[i] \mid sk[.])\ r$	access to heap and stack
s	::= BLK $\widetilde{inst}$	basic block containing an instr. list
	$\mid$ CJ $(x, op_r, x)\ s\ s$	conditional jump
	$\mid$ TR $(x, op_r, x)\ s$	tail recursion (loop)
	$\mid$ FC $(\widetilde{x}, \widetilde{v})\ s\ (\widetilde{v}, \widetilde{x})$	function call
p	::= $(\ \overrightarrow{v}, s, \overrightarrow{x}\ )$	programs

r_d	::= HSL.r	data register
r_b	::= $hp \mid fp \mid ip \mid sp \mid lr$	base (pointer) register
r	::= $r_d \mid r_b$	register
m	::= $m[r_b, +i] \mid m[r_b, -i]$	memory location
v, y, x, p	::= similar to v, y, x, p in HSL	
$inst$	::= $\texttt{op}^\texttt{b}\ r\ y\ y \mid ldr\ r\ m \mid str\ m\ r \mid push\ \widetilde{r} \mid pop\ \widetilde{r}$	single instruction
s	::= BLK $\widetilde{inst}$ $\mid$ CJ $(x, op_r, x)\ s\ s$ $\mid$ TR $(x, op_r, x)\ s$	control flow structures

$$\frac{}{\text{BLK } [] \vdash \sigma \rightarrowtail \sigma} \qquad \frac{\texttt{eval\_inst}\ inst\ \sigma = \sigma_1 \quad \text{BLK } instL \vdash \sigma_1 \rightarrowtail \sigma_2}{\text{BLK } (inst{::}instL) \vdash \sigma \rightarrowtail \sigma_2}$$

$$\frac{S_1 \vdash \sigma \rightarrowtail \sigma_1 \quad S_2 \vdash \sigma_1 \rightarrowtail \sigma_2}{\text{SC } S_1\ S_2 \vdash \sigma \rightarrowtail \sigma_2} \qquad \frac{S_1 \vdash \sigma \rightarrowtail \sigma_1 \quad \texttt{is\_true}\ (\texttt{eval\_cond}\ cond\ \sigma)}{\text{CJ } cond\ S_1\ S_2 \vdash \sigma \rightarrowtail \sigma_1}$$

$$\frac{S_2 \vdash \sigma \rightarrowtail \sigma_1 \quad \texttt{is\_false}\ (\texttt{eval\_cond}\ cond\ \sigma)}{\text{CJ } cond\ S_1\ S_2 \vdash \sigma \rightarrowtail \sigma_1} \qquad \frac{\texttt{is\_true}\ (\texttt{eval\_cond}\ cond\ \sigma)}{\text{TR } cond\ S \vdash \sigma \rightarrowtail \sigma}$$

$$\frac{S \vdash \sigma \rightarrowtail \sigma_1 \quad \texttt{is\_false}(\texttt{eval\_cond}\ cond\ \sigma) \quad \text{TR } cond\ S \vdash \sigma_1 \rightarrowtail \sigma_2}{\text{TR } cond\ S \vdash \sigma \rightarrowtail \sigma_2}$$

$$\frac{\texttt{copy}\ (\sigma_\epsilon, \sigma)\ (callee.\vec{i}, caller.\vec{i}) = \sigma_1 \quad S \vdash \sigma_1 \rightarrowtail \sigma_2 \quad \texttt{copy}\ (\sigma, \sigma_2)\ (caller.o, callee.i) = \sigma_3}{\text{FC } (caller.\vec{i}, callee.\vec{i})\ S\ (caller.\vec{o}, callee.\vec{o}) \vdash \sigma \rightarrowtail \sigma_3}$$

Figure 4.1: Syntax for HSL (top) and CFL (middle), and evaluation rules (bottom). (Note: FC structures only appear in HSL)

(the first component is for the caller, the second component for is the callee), a body statement, and a result passing pair. Heap variables are not allowed in parameters or results since their values are not transferred through the stack. A HSL program will never contain any comparison or jump instructions. Variables are divided into register variables, heap variables, and stack variables. Variables in LF2 format have been mapped to either register, heap, or stack variables by register allocation and interprocedural analysis.

CFL explicitly lays out the heap and stacks for function calls. It specifies machine registers and memory locations for the variables in HSL. A function call in HSL is implemented by dividing the processing into three phases: precall processing, function body execution and postcall processing. Pointer registers `hp` (heap pointer), `fp` (frame pointer), `ip` (intra-procedure register pointer), `sp` (stack

pointer), and `lr` (link register) are used to control the layout of the heap and stack frames for functions. CFL works over machine registers and memory; thus, a (one-to-one) mapping from HSL variables to them is required.

The translation from CFL to the object code simply performs the linearization of control-flow structures. The format of an ARM instruction is: $op\{cond\}\ d_1\ d_2$. The *cond* field controls conditional execution of the instruction, it is omitted for unconditional execution; d_1 and d_2 are the destination operand and source operands, respectively. Figure 4.2 gives the syntax and semantics of the machine language.

In our machine model, the data memory is separated from instruction memory (also known as the *instruction buffer*, which is modeled as a function mapping an address to an instruction). At each step, the instruction pointed to by the *pc* is executed. A program is executed until the first position beyond the code area is reached.

4.1.2 From LF2 to HSL

To support reasoning about HSL programs, we use the following Hoare triples:

$$\{P\}\ S\ \{Q\} \doteq \forall \sigma_{\text{hsl}}.P\ \sigma_{\text{hsl}} \Rightarrow Q(\mathsf{run}_{\text{hsl}}\ S\ \sigma_{\text{hsl}})$$

We first derive standard Hoare rules. Then, to bridge the semantic gap between an LF2 function g with inputs $\vec{i}$ and outputs $\vec{o}$, and the HSL structure S built from g's LF2, we specialize the axiomatic semantics to obtain a refined set of Hoare

r	::=	CFL.r | pc	machine register
m, v, y, x	::=	similar to m, v, y, x in CFL	
$inst$	::=	$b\{op_r\}\ +\ k\ \mid\ b\{op_r\}\ -\ k$	branch instruction
		| `cmp` $y\ y$ | `tst` $y\ y$	comparison instruction
		| CFL.$inst$	operation instruction
p	::=	$(\ \overrightarrow{v}, \widetilde{inst}, \overrightarrow{x}\ )$	programs

$$\frac{\texttt{eval\_op}\ (op\ y\ \vec{x})\ \omega = \omega_1}{op\ y\ \vec{x} \vdash (pc,cpsr,\omega) \rightarrowtail (pc{+}1,cpsr,\omega_1)} \qquad \frac{\texttt{update\_cpsr}\ cpsr\ d_1\ d_2 = cpsr_1}{cmp\ d_1\ d_2 \vdash (pc,cpsr,\omega) \rightarrowtail (pc{+}1,cpsr_1,\omega)}$$

$$\frac{\texttt{is\_true}\ (\texttt{eval\_cpsr}\ cpsr\ rop)}{b\{rop\}\ (+/-)\ k \vdash (pc,\ cpsr,\ \omega) \rightarrowtail (pc\ (+/-)\ k,\ cpsr,\ \omega)}$$

$$\frac{\texttt{is\_false}\ (\texttt{eval\_cpsr}\ cpsr\ rop)}{b\{rop\}\ (+/-)\ k \vdash (pc,\ cpsr,\ \omega) \rightarrowtail (pc+1,\ cpsr,\ \omega)}$$

Figure 4.2: Syntax and evaluation rules of the machine language

rules—dubbed the *projective* Hoare rules. A projective Hoare rule says: provided that inputs $\vec{i}$ have initial values $\vec{v}$, and any variable x in the live variable set ξ has value k, then in the state σ' after the execution of S, the values left in outputs $\vec{o}$ are equal to applying the function f to the initial values $\vec{v}$, and x's value is still k:

$$\begin{array}{l} S \vdash \xi \uparrow (\vec{i}, f, \vec{o}) \doteq \\ \quad \forall x \in \xi\, \forall \vec{v} \forall k \forall \sigma_{\text{hsl}}.(\vec{i_f}\ \sigma_{\text{hsl}} = \vec{v}) \wedge (\sigma_{\text{hsl}}[[x]] = k) \Rightarrow \\ \quad \texttt{let}\ \sigma'_{\text{hsl}} = \mathsf{run}_{\text{hsl}}\ S\ \sigma_{\text{hsl}}\ \texttt{in}\ \wedge (\vec{o_f}\ \sigma'_{\text{hsl}} = f\ \vec{v}) \wedge (\sigma'_{\text{hsl}}[[x]] = k) \end{array}$$

where functions i_f and o_f project from a data state the values of vector $\vec{i}$ and $\vec{o}$. If the judgement embodied by a projective Hoare rule holds on the S derived from g, then the synthesized function f should be equivalent to g and, indeed, this is easy to prove automatically since they are quite similar.

The projective Hoare rules utilize the following definitions. Operator `mk_cnd` turns a condition into a condition function. Suppose $\overrightarrow{\xi}$ turns a set ξ into a vector, and $\overleftarrow{v}$ turns a vector $\vec{v}$ into a set, then the product of a vector and a set makes a new vector that comprises $\vec{v_1}$ and all elements in ξ, $\vec{v_1} \times \xi \doteq (\vec{v_1}, \overrightarrow{\xi})$. The dot product of a function and a set gives a new function: $(\lambda \vec{x}.f\ \vec{x}) \odot \xi \doteq \lambda(\vec{x}, \overrightarrow{\xi}).(f\ \vec{x}, \overrightarrow{\xi})$. A vector and a projective function are interchangeable.

$$\frac{s_1 \vdash \xi_1 \uparrow (\vec{i_1}, f_1, \vec{o_1}) \quad s_2 \vdash \xi_2 \uparrow (\vec{o_1}, f_2, \vec{o_2})}{\mathsf{SC}\ s_1\ s_2 \vdash \xi_1 \cap \xi_2 \uparrow (\vec{i_1}, \mathsf{sc}\ f_1\ f_2, \vec{o_2})}\ \text{sc\_rule}$$

$$\frac{s_1 \vdash \xi_1 \uparrow (\vec{i}, f_1, \vec{o}) \quad s_2 \vdash \xi_2 \uparrow (\vec{i}, f_2, \vec{o})}{\mathsf{CJ}\ cnd\ s_1\ s_2 \vdash \xi_1 \cap \xi_2 \uparrow (\vec{i}, (\mathsf{cj}\ (\texttt{mk\_cnd}\ cnd)\ f_1\ f_2), \vec{o})}\ \text{cj\_rule}$$

$$\frac{s \vdash \xi \uparrow (\vec{i}, f, \vec{i})}{\mathsf{TR}\ cnd\ s \vdash \xi \uparrow (\vec{i}, (\mathsf{tr}\ (\texttt{mk\_cnd}\ cnd)\ f), \vec{i})}\ \text{tr\_rule} \qquad \frac{s \vdash \xi \uparrow (\vec{i}, f, \vec{o}) \quad g\ \vec{i'} = f\ \vec{i}}{s \vdash \xi \uparrow (\vec{i'}, g, \vec{o})}\ \text{shuffle\_rule}$$

$$\frac{s \vdash \xi \uparrow (\vec{i}, f, \vec{o}) \quad \xi' \subseteq \xi}{s \vdash \xi \uparrow (\vec{i} \times \xi', f \odot \xi', \vec{o} \times \xi')}\ \text{pick\_rule} \qquad \frac{s \vdash \xi \uparrow (\vec{i}, f, \vec{o}) \quad \xi' \subseteq \xi}{s \vdash \xi' \uparrow (\vec{i}, f, \vec{o})}\ \text{shrink\_rule}$$

$$\frac{s \vdash \xi \uparrow (callee.\vec{i}, f, callee.\vec{o}) \quad \overleftarrow{caller.\vec{o}} \cap \xi' = \phi}{\mathsf{FC}\ (caller.\vec{i}, f, callee.\vec{i})\ s\ (caller.\vec{o}, f, callee.\vec{o}) \vdash \xi' \uparrow (caller.\vec{i}, f, caller.\vec{o})}\ \text{fc\_rule}$$

These rules are used to keep track of how the relation between specific inputs and outputs change during the execution. Rules sc_rule, cj_rule, and tr_rule are control flow rules and their meaning is self-explanatory. The live variable set ξ stores the variables that are still live but not modified by the current statement. In other words, when the value of a live variable is not altered by the current statement, it is stored in ξ for future use. A live variable is either in ξ, or in

the outputs $\vec{o}$. When it becomes not live any more, it should be removed from ξ. Maintaining a ξ helps to reduce the number of variables in the inputs and outputs. Rule pick_rule is for extracting variables from the live variable set, while shrink_rule is used to discard variables not live any more from the set. Rule shuffle_rule is to restructure the input vector. Restructuring the ouput vector is accomplished by appending an empty block and applying the shuffle_rule to it. A basic block is simulated as a whole as it is a macro instruction; thus, there exists no rule for it.

Application of projective rules is controlled by an annotated structure with inputs, outputs, and context information, which guides the symbolic simulation and the application of rules. Control flow rules `sc_rule`, `cj_rule`, and `tr_rule` are applied on structures SC, CJ, and TR, respectively. For instance, when reasoning about a (CJ *cond* S_1 S_2) structure, we first reason about S_1 and S_2 separately, then apply the `cj_rule` rule. The application of data flow rules `pick_rule`, `shrink_rule`, and `shuffle_rule` are guided by the "use" and "def" information of a structure maintained by the compiler.

4.1.3 From HSL to CFL

The main task for this translation is to implement function calls and map heap variables and stack variables to memory (for wider application we handle heap variables here although they are replaced with stack variables during closure conversion). Obviously the mapping function, $\searrow$, shall be a one-to-one function.

The storage for local (stack) variables is allocated on function entry and released on function exit. In particular, local variables are held in a stack frame that will be "destroyed" on function exit, and the storage for its stack can be "collected" and reused for other function calls. The memory is modelled as a finite map with addresses ranging from 0 to $2^{32} - 1$.

We introduce an injection relation $\backsimeq^{\searrow}$ to relate the states occurring during the execution of HSL code and that of the translated CFL code, where $\searrow$ consists of three injective functions $\searrow_{rg}$, $\searrow_{hp}$ and $\searrow_{sk}$ that map logical registers, heap variables, and stack variables to machine registers and memory locations. Of course, all procedures use the same $\searrow_{hp}$ as they share the global heap. The correctness statement amounts to showing that the execution of a HSL statement S_{hsl} has the same effect on a HSL state as the execution of its corresponding CFL statement S_{cfl} (notation D_σ and D_S return the domains of the finite maps in σ and the

variables accessed by the instruction in S).

$$\vdash_{def} \mathsf{one\_one\_inj}\ \sigma_{\text{hsl}}\ \searrow\ \sigma_{\text{cfl}} \doteq \forall v_1, v_2 \in D_{\sigma_{\text{hsl}}}.\ \mathsf{addr}\ \sigma_{\text{cfl}}\ v_1^{\searrow} \neq \mathsf{addr}\ \sigma_{\text{cfl}}\ v_2^{\searrow}$$
$$\vdash_{def} \sigma_{\text{hsl}} \backsimeq^{\searrow} \sigma_{\text{cfl}} \doteq \forall v \in D_{\sigma_{\text{hsl}}}.\ \sigma_{\text{hsl}}[[v]] = \sigma_{\text{cfl}}[[v^{\searrow}]]$$
$$\vdash_{def} (S_{\text{hsl}} \equiv^{\searrow} S_{\text{cfl}}) \doteq \forall \sigma_{\text{hsl}} \forall \sigma_{\text{cfl}}.\ (D_{S_{\text{hsl}}} = D_{\sigma_{\text{hsl}}} \wedge \sigma_{\text{hsl}} \backsimeq^{\searrow} \sigma_{\text{cfl}}) \Rightarrow (\mathsf{run}_{\text{hsl}}\ S_{\text{hsl}}\ \sigma_{\text{hsl}}\ \backsimeq^{\searrow}\ \mathsf{run}_{\text{cfl}}\ S_{\text{cfl}}\ \sigma_{\text{cfl}})$$

The function addr returns the address of a mapped variable. An address is parameterized by a state containing the values of base registers (*e.g.* fp and sp). Given an injection $\searrow$, the translation from HSL to CFL for most structures is simple and we just need to replace HSL variables with their mapped machine registers and memory locations. A FC structure will be converted to the sequential composition of precall processing, callee's body, and postcall processing:

$$r_i^{\searrow} \doteq \searrow_{rg}\ r_i \quad hp[i]^{\searrow} \doteq m[\searrow_{hp}\ i] \quad sk[i]^{\searrow} \doteq m[\searrow_{sk}\ i] \quad S^{\searrow} \doteq \forall v \in D_S.\ S[v \leftarrow v^{\searrow}]$$
$$\Gamma_{\text{hsl}}\ S \doteq S^{\searrow} \quad \text{when } S \text{ is a BLK,SC,CJ or TR structure}$$
$$\Gamma_{\text{hsl}}\ (\mathsf{FC}\ (caller.\vec{i}, callee.\vec{i})\ S\ (caller.\vec{o}, callee.\vec{o})) \doteq \mathsf{SC}\ (\mathsf{SC}\ pre\ (\Gamma_{\text{hsl}}\ S))\ post \quad \text{for valid } pre, post \text{ and } \searrow' \text{ described below}$$

When $\searrow_{sk}$ maps different stack variables to different memory locations, the translation for BLK, SC, CJ, and TR structures guarantees semantics preservation. The translation for FC is more complicated: we require that the precall processing and postcall processing fulfill the parameter passing and result returning task; and the execution of the precall processing, function body and postcall processing should not modify the values of the caller's register and stack variables except for those set to receive results (we name this the *value recovering* property). Assuming that $\searrow$ is an one-to-one injection, we have:

$$\frac{}{(\mathsf{BLK}\ S) \equiv^{\searrow} (\mathsf{BLK}\ S^{\searrow})} \qquad \frac{S_{\text{hsl\_1}} \equiv^{\searrow} S_{\text{cfl\_1}} \quad S_{\text{hsl\_2}} \equiv^{\searrow} S_{\text{cfl\_2}}}{\mathsf{SC}\ S_{\text{hsl\_1}}\ S_{\text{hsl\_2}} \equiv^{\searrow} \mathsf{SC}\ S_{\text{cfl\_1}}\ S_{\text{cfl\_2}}}$$

$$\frac{S_{\text{hsl\_1}} \equiv^{\searrow} S_{\text{cfl\_1}} \quad S_{\text{hsl\_2}} \equiv^{\searrow} S_{\text{cfl\_2}}}{\mathsf{CJ}\ cond\ S_{\text{hsl\_1}}\ S_{\text{hsl\_2}} \equiv^{\searrow} \mathsf{CJ}\ cond^{\searrow}\ S_{\text{cfl\_1}}\ S_{\text{cfl\_2}}} \qquad \frac{S_{\text{hsl}} \equiv^{\searrow} S_{\text{cfl}}}{\mathsf{TR}\ cond\ S_{\text{hsl}} \equiv^{\searrow} \mathsf{TR}\ cond^{\searrow}\ S_{\text{cfl}}}$$

$$\frac{\begin{array}{c}\forall \sigma.\sigma[[caller.\vec{i}^{\searrow}]] = (\mathsf{run}_{\text{cfl}}\ pre\ \sigma)[[callee.\vec{i}^{\searrow'}]] \quad S_{\text{hsl}} \equiv^{\searrow'} S_{\text{cfl}} \\ \forall \sigma.\sigma[[callee.\vec{o}^{\searrow'}]] = (\mathsf{run}_{\text{cfl}}\ post\ \sigma)[[caller.\vec{o}^{\searrow}]] \\ \forall \sigma.\forall v \in (D^{rg,sk}_{S_{caller}} \setminus \overleftarrow{caller.\vec{o}}).\ \sigma[[v^{\searrow}]] = (\mathsf{run}_{\text{cfl}}\ (\mathsf{SC}\ (\mathsf{SC}\ pre\ S_{\text{cfl}})\ post)\ \sigma)[[v^{\searrow}]]\end{array}}{\mathsf{FC}\ (caller.\vec{i}, callee.\vec{i})\ S_{hsl}\ (caller.\vec{o}, callee.\vec{o}) \equiv^{\searrow} \mathsf{SC}\ (\mathsf{SC}\ pre\ S_{\text{cfl}})\ post}$$

There are many ways to guarantee that the value recovering property holds. One of them is to layout the frames of the caller and callee in such a way that their domains do not intersect with each other, and the values of register variables

modified by the callee's execution are recovered on the function entry. This leads to a valid implementation of a frame layout and a function call procedure. The areas in the memory devoted to stack frames (*i.e.* the activation record) are marked by the *ip*, *fp*, and *sp*. When the callee is called, space for results are reserved by growing the stack, then the caller pushes all parameters into the stack, and then the frame for the callee is created. Specifically, when a callee is called, its stack frames shall not be overlapped with the callee's frame.

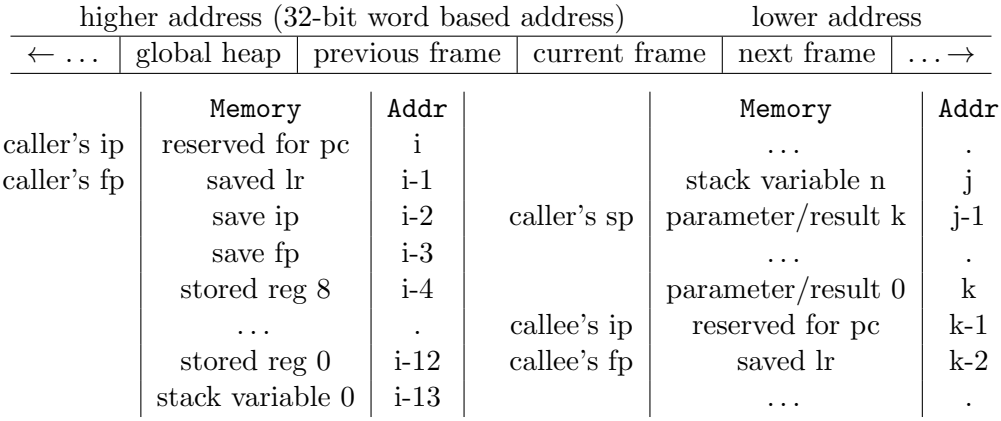
As indicated by the following rule, an implementation is valid if it ensures that: (1) the parameter/result passing and the body execution do not change the values of stack variables in the caller's frame except those for receiving results (*i.e.* $caller.\vec{o}$); (2) all register variables are pushed into memory before parameter passing on function entry and then popped from memory before result passing on function exit. In the following rule, $\sigma\langle v\rangle$ represents reading the value at concrete address v from state σ, and D_r is the abbreviation of $D_{S_{caller}}$.

$$\frac{\begin{array}{c}\sigma_1 = \mathsf{run}_{\mathrm{cfl}}\ pre\ \sigma \quad \sigma_2 = \mathsf{run}_{\mathrm{cfl}}\ S_{\mathrm{cfl}}\ \sigma_1 \quad \sigma_3 = \mathsf{run}_{\mathrm{cfl}}\ post\ \sigma_2 \\ \forall v \in (D_r^{sk})^{\searrow}.\sigma\langle v\rangle = \sigma_1\langle v\rangle \quad \exists x_i.\, \sigma_1\langle x_i\rangle = \sigma[[r_i]] \text{ for } i \in D^{rg}_{S_{callee}} \\ \forall v \in (D_r^{sk})^{\searrow} \cup \{x_i \mid i \in D^{rg}_{S_{callee}}\}.\sigma_2\langle v\rangle = \sigma_1\langle v\rangle \\ \forall v \in (D_r^{sk} \setminus \overleftarrow{caller.\vec{o}})^{\searrow}.\, \sigma_3\langle v\rangle = \sigma_2\langle v\rangle \quad \forall r_i \in (D_r^{rg} \setminus \overleftarrow{caller.\vec{o}}).\, \sigma_3[[r_i]] = \sigma_2\langle x_i\rangle\end{array}}{\forall \sigma. \forall v \in (D_r^{rg,sk} \setminus \overleftarrow{caller.\vec{o}}).\, \sigma[[v^{\searrow}]] = (\mathsf{run}_{\mathrm{cfl}}\ (\mathsf{SC}\ (\mathsf{SC}\ pre\ S_{\mathrm{cfl}})\ post)\ \sigma)[[v^{\searrow}]]}$$

Complying with these requirements, our implementation compiles function calls into a callee-save style calling convention. Specifically, $\searrow_{sk} = \searrow'_{sk} = \lambda i.(fp, -(i+12))$, $\searrow_{rg} = \searrow'_{rg} = \lambda r.r$ and $\searrow_{hp} = \searrow'_{hp} = \lambda i.(hp, -i)$. By carefully moving the pointers *fp*, *ip*, and *sp*, we keep the caller's frame and callee's frame located in separate areas in the memory (see Figure 4.3). All parameters and results are passed through the stack, and the callee saves all data registers (*i.e.* $r_0 - r_8$) in all cases. This solution is suboptimal but easier to verify. In particular, it allows us, while performing colouring register allocation, not to add interferences between caller-save registers and temporaries that are live across a call.

One subtlety appearing in proofs is that the initial values of *hp*, *sp*, *ip*, and *fp* must be greater than specific values so that the memory can accomodate all stack frames and the areas consumed by preprocessing and postprocessing.

Both the heap and the stacks are simply finite maps; thus, we do not formalize and rely on any heap management and stack property. In [16] a block-base memory model between a machine memory and a high-level view is introduced to manage



$$
\begin{aligned}
pre = \quad & \mathsf{BLK}\ [sub\ sp\ sp\ (max(\#caller.\vec{i}, \#caller.\vec{o}) - \#caller.\vec{i});\ push\ caller.\vec{i};\\
& \quad mov\ ip\ sp;\ sub\ fp\ ip\ 1;\ sub\ sp\ sp\ 1;\ push\ \{r_0, \ldots, r_8, fp, ip, lr\};\\
& \quad add\ sp\ sp\ 12;\ pop\ callee.\vec{i};\ sub\ sp\ fp\ (12 + \#stack\_variables)]\\
post = \quad & \mathsf{BLK}\ [add\ sp\ ip\ \#callee.\vec{o};\ push\ callee.\vec{o};\ sub\ sp\ fp\ 12;\\
& \quad pop\ \{r_0, \ldots, r_8, fp, ip, lr\};\ mov\ sp\ ip;\ pop\ caller.\vec{o};\\
& \quad sub\ sp\ fp\ (12 + \#stack\_variables)]
\end{aligned}
$$

Figure 4.3: Memory layout in HSL.

frame stacks. As in our method, separation is enforced between stack blocks belonging to different function activation records.

4.1.4 From CFL to ARM

The translation from CFL to ARM proceeds by linearizing the SC, CJ, and TR structures. The instructions in basic blocks are already in the right format. Our translation always generates flat code satisfying good properties including: (1) any execution of the translated code will not access beyond its own area in the instruction buffer; (2) the data state after an execution is independent of the initial values of *pc* and *cpsr*; (3) all executions terminate.

The translation verification for CJ proceeds by case analysis on the condition, while that for TR by the induction on the number of rounds the body is executed. This linearization scheme turns out to be most succinct in terms of the length of generated code. One optimization is performed at the flat code level for function calls: all occurrences of a callee are moved to the same area in the code so that only one copy is left. Unconditional jumps are inserted appropriately. The correctness proof for this relocation is straightforward because the adjusted code runs in the same way as its old version.

$$
\begin{aligned}
\Gamma_{\text{cfl}}\ (\mathsf{BLK}\ (inst :: instL)) &\doteq\ inst :: \Gamma_{\text{cfl}}(\mathsf{BLK}\ instL)\\
\Gamma_{\text{cfl}}\ (\mathsf{BLK}\ [\,]) &\doteq\ [\,]
\end{aligned}
$$

$$
\begin{array}{rl}
\Gamma_{\text{cfl}}\ (\mathsf{SC}\ s_1\ s_2) \doteq & (\Gamma_{\text{cfl}}\ s_1)\ \uplus\ (\Gamma_{\text{cfl}}\ s_2) \\
\Gamma_{\text{cfl}}\ (\mathsf{CJ}\ (v_1, rop, v_2)\ s_t\ s_f) \doteq & \mathsf{let}\ (\rho_t\ \rho_f) = (\Gamma_{\text{cfl}}\ s_t, \Gamma_{\text{cfl}}\ s_f)\ \mathsf{in} \\
& \quad (\mathsf{cmp}\ v_1\ v_2) :: (\mathsf{b}\{rop\}\ +\ \|\rho_f\| + 2) :: \\
& \quad \rho_f \uplus [\mathsf{bal}\ +\ \|\rho_t\| + 1] \uplus \rho_t \\
\Gamma_{\text{cfl}}\ (\mathsf{TR}\ (v_1, rop, v_2)\ s) \doteq & \mathsf{let}\ \rho = \Gamma_{\text{cfl}}\ s\ \mathsf{in} \\
& (\mathsf{cmp}\ v_1\ v_2) :: (\mathsf{b}\{rop\} + \|\rho\| + 2) :: \rho\ \uplus [\mathsf{bal} - (|\rho| + 2)]
\end{array}
$$

Note that $\|\rho\|$ returns the number of instructions in ρ, and $\rho_1 \uplus \rho_2$ appends ρ_2 to ρ_1.

Now, we consider a simple example. With the following abbreviations,

$$
\begin{array}{l}
body \doteq \mathsf{BLK}\ [msub\ r_3\ r_0\ 1w;\ mmul\ r_2\ r_0\ r_1;\ mmov\ r_0\ r_3;\ mmov\ r_1\ r_2] \\
blk_1 \doteq \mathsf{BLK}\ [mmov\ r_2\ r_1] \quad snd \doteq \lambda(v_0, v_1).v_1 \\
f_1 \doteq \lambda(v_0, v_1).(v_0 - 1w, v_0 + v_1) \quad f_2 \doteq \mathsf{tr}\ (\lambda(v_0, v_1).v_0 = 0w)\rangle)\ f_1
\end{array}
$$

the intermediate forms of the factorial function and the derivation of the specification connecting the $fact_{hsl}$ and $fact_{acf}$ (where $Axiom_1 = blk_1 \vdash \{\} \uparrow ((r_0, r_1), snd, r_2)$) are:

$$
\begin{array}{ll}
\text{TFL:} & fact\ (x, a) \doteq \mathtt{if}\ x = 0w\ \mathtt{then}\ a\ \mathtt{else}\ fact\ (x - 1w, x \times a) \\
\text{LF2:} & fact_{acf} \doteq \mathsf{sc}\ (\mathsf{tr}\ (\lambda(v_0, v_1).v_0 = 0w)\ f_1)\ snd \\
\text{HSL:} & fact_{hsl} \doteq \mathsf{SC}\ (\mathsf{TR}\ (r_0, eq, 0w)\ body)\ blk_1 \\
\text{CFL:} & fact_{cfl} \doteq \Gamma_{hsl}\ fact_{hsl} = fact_{hsl} \\
\text{ARM:} & fact_{arm} \doteq \Gamma_{\text{cfl}}\ fact_{cfl} = [cmp\ r_0\ r_1;\ beq\ +\ 6;\ sub\ r_3\ r_0\ 1w;\ mul\ r_2\ r_0\ r_1; \\
& \quad mov\ r_0\ r_3;\ mov\ r_1,\ r_2;\ bal\ -\ 6;\ mov\ r_2,\ r_1]
\end{array}
$$

$$
\dfrac{\dfrac{body \vdash \{\} \uparrow ((r_0, r_1), f_1, (r_0, r_1))}{\mathsf{TR}\ (r_0, ne, 0w)\ body \vdash \{\} \uparrow ((r_0, r_1), f_2, (r_0, r_1))}\ \text{tr\_rule} \qquad Axiom_1}{\mathsf{SC}\ (\mathsf{TR}\ (r_0, ne, 0w)\ body)\ blk_1 \vdash \{\} \uparrow ((r_0, r_1), fact_{acf}, r_2)}\ \text{sc\_rule}
$$

4.2 Back-end II

This back-end admits only tail recursive programs. For those nontail recursive programs, we rely on a third party translator to turn them into equivalent tail recursion. A preliminary tool linRec has been developed to translate linear recursions to tail recursions [36].

As in the first back-end, we convert a tail recursive program into the sequential composition of its body loop (represented by a tr structure) and its basic base through theorem conv_tr. Theorem tr_ind enables us to reason about tr structures through induction. At the next step, this tail recursive equation is translated to abstract assembly code.

$\Vdash_{def}$ $\mathsf{tr}\ c\ f \doteq \lambda x.\mathsf{if}\ c\ x\ \mathsf{then}\ x\ \mathsf{else}\ \mathsf{tr}\ c\ (f\ x)$ [tr_def]
$\Vdash_{thm}$ $(f\ x = \mathsf{if}\ c\ x\ \mathsf{then}\ f_1\ x\ \mathsf{else}\ f\ (f_2\ x)) \Leftrightarrow (f\ x = \mathsf{let}\ v = \mathsf{tr}\ c\ f_2\ x\ \mathsf{in}\ f_1\ v)$ [conv_tr]
$\Vdash_{thm}$ $\forall P.\,(\forall x.\,(\neg c\ x \Longrightarrow P\ (f\ x)) \Longrightarrow P\ x) \Longrightarrow \forall v.\,P\ v$ [tr_ind]

4.2.1 Structured Assembly Language

Validation of the translation from high-level language programs to low-level codes is believed to be difficult due to inherent nonmodularity the of low-level programs. This is attributed to low-level code being flat and to the prominent presence of unrestricted jumps.

Fortunately, although low-level code seems to be just flat finite sets of instructions, it is structured by finite unions naturally: a compilation produces code structurally by combining smaller pieces of code together to generate larger code. Technically, we can formulate a structured version for a piece of low-level instructions and develop compositional natural semantics for it. With this spirit, Saabas and Uustalu [99] propose a compositional natural semantics and a Hoare logic for a structured low-level language. Siminarly, Tan and Appel [110] propose a continuation-style compositional logic with a rather sophisticated interpretation of Hoare triples involving explicit fix-point approximations.

We introduce a structured assembly language (SAL) with a compositional (natural) semantics as the next intermediate representation. The translation from FIL to SAL is shown to be correct with respect to this compositional semantics. This language has the following grammar. For optimization purpose, some of the labels in a code may be omitted when the control flow is clear (see section 4.2).

$$sc ::= (\ell\ \{v := e\}\ \ell) \mid l\ \mathsf{ifgoto}\ cond\ \ell\ \ell \mid \ell\ \mathsf{goto}\ \ell \mid sc\ \uplus\ sc$$

The natural semantics of SAL is specified as evaluation rules which relate a piece of code (with entry label and exit label) with the functional expression it implements. Rule $\vdash\ \langle l_1\rangle\ S\ \langle l_2\rangle \Rightarrow (w, v)$ indicates: (1) Structure S computes a FIL expression w and stores the result of computation in v; (2) The control flow starts at label l_1 and ends at label l_2 (by convention $l_i \neq l_j$ if $i \neq j$). In this case, we say S is reducible to (w, v). Roughly (w, v) can be understood as $\mathsf{C}\ w\ (\lambda v.\ \ldots)$ where C is the CPS combinator defined in Section 3.2. A code being reducible to (v, v) means that it computes nothing but moving the control flow. Actually (v, v) is often abbreviated to be $()$. If a piece of SAL code c is reducible to (e, v),

then we claim that c implements FIL expression $\mathsf{let}\ v = e\ \mathsf{in}\ v$, and the translation from this expression to c is correct.

The idea behind this natural semantics is that an assembly program can be structured as a union of labeled structures such that the control flow is represented by the jumps between labels of these structures. Since the composition of these labeled structures is flat and the only connection between them is labels, the gap between high-level programs (which exhibit complicated control flow structures) and low-level assembly code (which is flat) is met.

As shown in Figure 4.4, a SAL program is built by composing labeled structures according to their entry labels and exit labels. Most of these rules are self-explanatory. Rule loop says if a round of computation of the body of a loop returns value $e[v]$, and subsequent rounds takes $e[v]$ as arguments and computes $f[e[v]]$, then the effect of these rounds together is to compute $f[v]$. Clearly this rule characterizes the behavior of tail recursions.

Based on the basic rules, we derive some advanced rules for more complicated control flow structures such as conditional jumps, tail recursions, and function calls. The proof of rule conditional goes by case analysis on the condition c; so does the proof of rule tr. The proof of rule fun_call is based on the fact that the preprocessing and postprocessing for a function call take care of argument passing

$$\frac{}{\vdash \langle l_1\rangle\ l_1\ \{v := w\}\ l_2\ \langle l_2\rangle \Rightarrow (w, v)}\ \mathsf{inst}$$

$$\frac{\vdash \langle l_1\rangle\ S_1\ \langle l_2\rangle \Rightarrow ()\qquad \vdash \langle l_2\rangle\ S_2\ \langle l_3\rangle \Rightarrow e}{\vdash \langle l_1\rangle\ S_1 \uplus S_2\ \langle l_3\rangle \Rightarrow e}\ \mathsf{nop}$$

$$\frac{\vdash \langle l_1\rangle\ S_1\ \langle l_2\rangle \Rightarrow e_1\qquad \vdash \langle l_3\rangle\ S_2\ \langle l_4\rangle \Rightarrow e_2}{\vdash \langle l_1\rangle\ S_1 \uplus S_2\ \langle l_2\rangle \Rightarrow e_1}\ \mathsf{skip}$$

$$\frac{\vdash \langle l_1\rangle\ S_1\ \langle l_2\rangle \Rightarrow (e, v)\qquad \vdash \langle l_2\rangle\ S_2\ \langle l_3\rangle \Rightarrow (f\ v, w)}{\vdash \langle l_1\rangle\ S_1 \uplus S_2\ \langle l_3\rangle \Rightarrow (\mathsf{let}\ v = e\ \mathsf{in}\ f\ v, w)}\ \mathsf{seq}$$

$$\frac{}{\vdash \langle l_1\rangle\ \mathsf{ifgoto}\ \top\ l_2\ l_3\ \langle l_2\rangle \Rightarrow ()}\ \mathsf{ift}\qquad \frac{}{\vdash \langle l_1\rangle\ \mathsf{ifgoto}\ \bot\ l_2\ l_3\ \langle l_3\rangle \Rightarrow ()}\ \mathsf{iff}$$

$$\frac{}{\vdash \langle l_1\rangle\ l_1\ \mathsf{goto}\ l_2\ \langle l_2\rangle \Rightarrow ()}\ \mathsf{goto}$$

$$\frac{\vdash \langle l_1\rangle\ S\ \langle l_1\rangle \Rightarrow (e[v], v)\qquad \langle l_1\rangle\ S\ \langle l_2\rangle \Rightarrow (f[e[v]], v)}{\vdash \langle l_1\rangle\ S\ \langle l_2\rangle \Rightarrow (f[v], v)}\ \mathsf{loop}$$

Figure 4.4: Compositional semantics of SAL.

and result passing, respectively.

$$\frac{\vdash \langle l_2\rangle\ S_1\ \langle l_4\rangle \Rightarrow (e_1,\, v) \qquad \vdash \langle l_3\rangle\ S_2\ \langle l_4\rangle \Rightarrow (e_2,\, v)}{\vdash \langle l_1\rangle\ (l_1\ \mathsf{ifgoto}\ c\ l_2\ l_3) \uplus S_2 \uplus S_1\ \langle l_4\rangle \Rightarrow (\mathsf{if}\ c\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2,\, v)}\ \mathsf{conditional}$$

$$\frac{\neg c\ v \Longrightarrow \langle l_3\rangle\ S\ \langle l_4\rangle \Rightarrow (f\ v,\, v)}{\vdash \langle l_1\rangle\ (l_1\ \mathsf{ifgoto}\ (c\ v)\ l_2\ l_3) \uplus S \uplus (l_4\ \mathsf{goto}\ l_1)\ \langle l_2\rangle \Rightarrow (\mathsf{tr}\ c\ f\ v,\, v)}\ \mathsf{tr}$$

$$\frac{\vdash \langle l_2\rangle\ S\ \langle l_3\rangle \Rightarrow (f\ w_1,\, v_1)}{\vdash \langle l_1\rangle\ (l_1\ \{w_1 := w_2\}\ l_2) \uplus S \uplus (l_3\ \{v_2 := v_1\}\ l_4) \Rightarrow (f\ w_2,\, v_2)}\ \mathsf{fun\_call}$$

The detailed derivation of these advanced rules is shown in Figure 4.5.

These rules immediately validate the following rewrites whose repeated application will convert a FIL program to an equivalent SAL program. Notation l_{+i} stands for the i^{th} new label introduced during the conversion.

[conv_exp]	$\mathsf{conv}\ (l\ (e, v)\ l') \longleftrightarrow (l,\ \{v := e\},\ l')$
[conv_let]	$\mathsf{conv}\ (l\ (\mathsf{let}\ v = e\ \mathsf{in}\ f\ v,\ w)\ l') \longleftrightarrow$
	$(\mathsf{conv}\ (l\ (e, v)\ l_{+1})) \uplus \mathsf{conv}\ (l_{+1}\ (f\ v, w)\ l')$
[conv_cond]	$\mathsf{conv}\ (l\ (\mathsf{if}\ c\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, v)\ l') \longleftrightarrow$
	$(l\ \mathsf{ifgoto}\ c\ l_{+1}\ l_{+2}) \uplus \mathsf{conv}\ (l_{+2}\ (e_2, v)\ l') \uplus \mathsf{conv}\ (l_{+1}\ (e_1, v)\ l')$
[conv_tr]	$\mathsf{conv}\ (l\ (\mathsf{tr}\ c\ f\ v,\ v)\ l') \longleftrightarrow$
	$(l\ \mathsf{ifgoto}\ c\ l'\ l) \uplus \mathsf{conv}\ (l\ (f\ v, v)\ l_{+1}) \uplus (l_{+1}\ \mathsf{goto}\ l)$
[conv_app]	$\mathsf{conv}\ (l\ (f\ w_2,\ v_2)\ l') \longleftrightarrow$
	$(l\ \{w_1 := w_2\}\ l_{+1}) \uplus (\mathsf{conv}\ (l_{+1}\ bd\ l_{+2})) \uplus (l_{+2}\ \{v_2 := v_1\}\ l')$
	where v_1, bd, and w_1 are the input, body,
	and output of f, respectively.

4.2.2 Machine Code Generation

This phase pretty-prints SAL programs into assembly code with respect to the instruction set of the target machine. Since we do not specify the semantics of the machine language, this conversion does not go by proof. However, the high similarity between SAL and realistic assembly language makes the correctness of this conversion easy to check (*e.g.* by hand).

One optimization in this phase is to eliminate labels that are not the targets of existing jumps. For instance, internal labels within a block consisting of sequential assignment instructions can be removed safely. And, the exit label of a structure is superfluous when the control flow after its execution goes directly to the next structure. Furthermore, since a ifgoto instruction is always followed immediately by its false block, it is safe to remove its exit label pointing to the false block.

Derivation of rule conditional

$$asm_1 = \vdash\ \langle l_1\rangle\ l_1\ \mathsf{ifgoto}\ \top\ l_2\ l_3\ \langle l_2\rangle\ \Rightarrow\ () \qquad asm_2 = \vdash\ \langle l_3\rangle\ S_2\ \langle l_4\rangle\ \Rightarrow\ (e_2, v)$$
$$asm_3 = \vdash\ \langle l_2\rangle\ S_1\ \langle l_4\rangle\ \Rightarrow\ (e_1, v) \qquad asm_4 = \vdash\ \langle l_1\rangle\ l_1\ \mathsf{ifgoto}\ \bot\ l_2\ l_3\ \langle l_3\rangle\ \Rightarrow\ ()$$
$$lem_1 = \langle l_1\rangle\ (l_1\ \mathsf{ifgoto}\ \top\ l_2\ l_3) \uplus S_2\ \langle l_2\rangle\ \Rightarrow\ ()$$
$$thm_1 = \vdash\ \langle l_1\rangle\ (l_1\ \mathsf{ifgoto}\ \top\ l_2\ l_3) \uplus S_2 \uplus S_1\ \langle l_4\rangle\ \Rightarrow\ (e_1, v)$$
$$lem_2 = \vdash\ \langle l_1\rangle\ (l_1\ \mathsf{ifgoto}\ \bot\ l_2\ l_3) \uplus S_2\ \langle l_4\rangle\ \Rightarrow\ (e_2, v)$$
$$thm_2 = \vdash\ \langle l_1\rangle\ (l_1\ \mathsf{ifgoto}\ \bot\ l_2\ l_3) \uplus S_2 \uplus S_1\ \langle l_4\rangle\ \Rightarrow\ (e_2, v)$$

$$\dfrac{\dfrac{\dfrac{asm_1 \qquad asm_2}{lem_1}\ \text{skip} \qquad asm_3}{thm_1}\ \text{nop} \qquad \dfrac{\dfrac{asm_4 \qquad asm_2}{lem_2}\ \text{nop} \qquad asm_3}{thm_2}\ \text{skip}}{\langle l_1\rangle\ (l_1\ \mathsf{ifgoto}\ c\ l_2\ l_3)\ \uplus\ S_2\ \uplus\ S_1\ \langle l_4\rangle\ \Rightarrow\ (\mathsf{if}\ c\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2,\ v)}$$

Derivation of rule tr

$$body = \langle l_1\rangle\ (l_1\ \mathsf{ifgoto}\ (c\ v)\ l_2\ l_3) \uplus S \uplus (l_4\ \mathsf{goto}\ l_1)\ \langle l_2\rangle$$
$$asm_1 = \vdash\ \langle l_1\rangle\ l_1\ \mathsf{ifgoto}\ \top\ l_2\ l_3\ \langle l_2\rangle\ \Rightarrow\ () \qquad asm_2 = \vdash\ \langle l_3\rangle\ S\ \langle l_4\rangle\ \Rightarrow\ (e, v)$$
$$asm_3 = \vdash\ \langle l_4\rangle\ l_4\ \mathsf{goto}\ l_1 \langle l_1\rangle \qquad lem_1 = \vdash\ \langle l_1\rangle\ (l_1\ \mathsf{ifgoto}\ \top\ l_2\ l_3) \uplus S\ \langle l_2\rangle\ \Rightarrow\ ()$$
$$thm_1 = \vdash\ c\ v \Longrightarrow body\ \Rightarrow\ () \qquad thm_2 = \vdash\ c\ v \Longrightarrow body\ \Rightarrow\ (\mathsf{tr}\ c\ f\ v, v)$$
$$asm_4 = \vdash\ \neg c\ v\ \Longrightarrow body\ \Rightarrow\ (\mathsf{tr}\ c\ f\ (f\ v), v)$$
$$asm_5 = \vdash\ \neg c\ v\ \Longrightarrow \langle l_1\rangle\ (l_1\ \mathsf{ifgoto}\ (c\ v)\ l_2\ l_3)\ \uplus\ S\ \uplus\ (l_4\ \mathsf{goto}\ l_1)\ \langle l_1\rangle\ \Rightarrow\ (f\ v, v)$$
$$thm_3 = \vdash\ \neg c\ v\ \Longrightarrow body\ \Rightarrow\ (\mathsf{tr}\ c\ f\ v, v)$$

$$\dfrac{\dfrac{\dfrac{asm_1 \qquad asm_2}{lem_1}\ \text{skip} \qquad asm_3}{\dfrac{thm_1}{thm_2}\ \text{tr\_def}}\ \text{nop} \qquad \dfrac{asm_4 \qquad asm_5}{thm_3}\ \text{loop, tr\_def}}{\vdash\ body\ \Rightarrow\ (\mathsf{tr}\ c\ f\ v, v)}\ \text{case}$$

Derivation of rule fun_call

$$pre = l_1\ \{w_1 := w_2\}\ l_2 \qquad post = l_3\ \{v_2 := v_1\}\ l_4$$

$$\dfrac{\dfrac{\dfrac{\vdash\ \langle l_1\rangle\ pre\ \langle l_2\rangle\ \Rightarrow\ (w_2, w_1) \qquad \vdash\ \langle l_2\rangle\ S\ \langle l_3\rangle\ \Rightarrow\ (f\ w_1,\ v_1)}{\vdash\ \langle l_1\rangle\ pre \uplus S\ \langle l_3\rangle\ \Rightarrow\ (\mathsf{let}\ w_1 = w_2\ \mathsf{in}\ f\ w_1, v_1)}\ \text{seq} \qquad \langle l_3\rangle\ \text{post}\ \langle l_4\rangle\ \Rightarrow\ (v_1, v_2)}{\vdash\ \langle l_1\rangle\ pre \uplus S \uplus post\ \langle l_4\rangle\ \Rightarrow\ (\mathsf{let}\ v_1 = (\mathsf{let}\ w_1 = w_2\ \mathsf{in}\ f\ w_1)\ \mathsf{in}\ v_1,\ v_2)}\ \text{seq}}{\vdash\ \langle l_1\rangle\ pre \uplus S \uplus post\ \langle l_4\rangle\ \Rightarrow\ (f\ w_2, v_2)}\ \text{let\_def}$$

Figure 4.5: Derivation of composite rules.

$$
\begin{array}{ll}
\text{[elim\_blk\_lab]} & (l_1\ \{v_1 := w_1\}\ l_2) \uplus (l_2\ \{v_2 := w_2\}\ l_3) \longrightarrow \\
& l_1\ (\{v_1 := w_1\} \uplus\ \{v_2 := w_2\})\ l_3 \\
\text{[elim\_seq\_lab]} & (l_1\ S_1\ l_2) \uplus (l_2\ S_2\ l_3) \longrightarrow (l_1\ S_1)\ \uplus\ (l_2\ S\ l_3) \\
\text{[elim\_ifgoto\_lab]} & (\mathsf{ifgoto}\ c\ l_2\ l_3)\ \uplus\ (l_3\ S\ l_4) \longrightarrow (\mathsf{ifgoto}\ c\ l_2)\ \uplus (l_3\ S\ l_4)
\end{array}
$$

The process of producing assembly from an optimized SAL program is trivial: (1) An assignment instruction with a single variable as target is replaced with a corresponding machine instruction; (2) A goto instruction is inserted for the exit label of a structure; (3) An ifgoto instruction is replaced with a comparison instruction followed by a jump. When both the target and source of an assignment are tuple and the machine model does not support parallel move, this assignment can be implemented by first pushing the value of source into the stack and then loading it back from the stack. Clearly, this process can be used to generate ARM assembly code.

4.3 Examples

4.3.1 Compilation of TEA

The TFL function of the TEA [122] block cipher includes the following first order logic functions:

```
ShiftXor (x,s,k0,k1) =
  ((x << 4) + k0) # (x + s) # ((x >> 5) + k1)

Round ((y,z),(k0,k1,k2,k3),s):state =
  let s' = s + DELTA in
  let y' = y + ShiftXor(z, s', k0, k1)
  in ((y', z + ShiftXor(y', s', k2, k3)), (k0,k1,k2,k3),s')

Rounds (n,s:state) =
  if n=0w then s else Rounds (n-1w, Round s)

TEAEncrypt (keys,txt) =
   let (cipheredtxt,keys,sum) = Rounds(32w,(txt,keys,0w))
   in cipheredtxt
```

Our compiler generates the following validated object code (pretty-printed to be in conventional ARM format).

```
Name : TEAEncrypt
Arguments : r0 r1 r2 r3 r4 r5
Modified Registers : r0 r1 r2 r3 r4 r5 r6
Returns : r6 r5
Body:
0:  mov ip, sp
1:  stmfd sp!, {fp,ip,lr,pc}
2:  sub fp, ip, #1i
3:  sub sp, sp, #7i
4:  mov r10, #0iw
5:  str r10, [sp]
6:  sub sp, sp, #1i
```

```
 7:  stmfd sp!, {r4,r5,r0,r1,r2,r3}
 8:  mov r10, #32iw
 9:  str r10, [sp]
 10:  sub sp, sp, #1i
 11:  bl + (6)
 12:  add sp, sp, #8i
 13:  ldmfd sp, {r6,r5,r4,r3,r2,r1,r0}
 14:  add sp, sp, #7i
 15:  sub sp, fp, #3i
 16:  ldmfd sp, {fp,sp,pc}
 17:  mov ip, sp
 18:  stmfd sp!, {r0,r1,r2,r3,r4,r5,r6,r7,
                  r8,r9,fp,ip,lr,pc}
 19:  sub fp, ip, #1i
 20:  sub sp, sp, #7i
 21:  ldmfd ip, {r0,r8,r5,r4,r3,r2,r6,r7}
 22:  add ip, ip, #8i
 23:  cmp r0, #0iw
 24:  beq + (37)
 25:  sub r1, r0, #1iw
 26:  str r1, [fp, #~11]
 27:  add r1, r7, #2654435769iw
 28:  str r1, [fp, #~12]
 29:  sub sp, sp, #1i
 30:  stmfd sp!, {r4,r3}
 31:  ldr r10, [fp, #~12]
 32:  str r10, [sp]
 33:  str r5, [sp, #~1]
 34:  sub sp, sp, #2i
 35:  bl + (32)
 36:  add sp, sp, #4i
 37:  ldr r1, [sp, #1]
 38:  add sp, sp, #1i
 39:  add r9, r8, r1
 40:  sub sp, sp, #1i
 41:  stmfd sp!, {r2,r6}
 42:  ldr r10, [fp, #~12]
 43:  str r10, [sp]
 44:  str r9, [sp, #~1]
 45:  sub sp, sp, #2i
 46:  bl + (21)
 47:  add sp, sp, #4i
 48:  ldr r1, [sp, #1]
 49:  add sp, sp, #1i
 50:  add r1, r5, r1
 51:  ldr r10, [fp, #~12]
 52:  str r10, [sp]
 53:  sub sp, sp, #1i
 54:  stmfd sp!, {r9,r1,r4,r3,r2,r6}
 55:  ldr r10, [fp, #~11]
 56:  str r10, [sp]
 57:  sub sp, sp, #1i
 58:  ldmfd sp, {r0,r8,r5,r4,r3,r2,r6,r7}
 59:  add sp, sp, #8i
 60:  bal - (37)
 61:  mov r1, r6
 62:  mov r0, r7
 63:  add sp, fp, #16i
 64:  stmfd sp!, {r8,r5,r4,r3,r2,r1,r0}
 65:  sub sp, fp, #13i
 66:  ldmfd sp, {r0,r1,r2,r3,r4,r5,r6,r7,
                  r8,r9,fp,sp,pc}
 67:  mov ip, sp
 68:  stmfd sp!, {r0,r1,r2,r3,r4,fp,ip,lr,pc}
 69:  sub fp, ip, #1i
 70:  ldmfd ip, {r0,r1,r2,r3}
 71:  add ip, ip, #4i
 72:  lsl r4, r0, #4i
 73:  add r2, r4, r2
 74:  add r1, r0, r1
 75:  eor r1, r2, r1
 76:  asr r0, r0, #5i
 77:  add r0, r0, r3
 78:  eor r0, r1, r0
 79:  add sp, fp, #6i
 80:  str r0, [sp]
 81:  sub sp, sp, #1i
 82:  sub sp, fp, #8i
 83:  ldmfd sp, {r0,r1,r2,r3,r4,fp,sp,pc}
```

4.3.2 A Detailed Example

We show below the real ouput in HOL-4 of the correctness statement for the simple factorial function.

$$fact\ (x, a) \doteq \texttt{if}\ x = 0w\ \texttt{then}\ a\ \texttt{else}\ fact\ (x - 1w, x \times a)$$

```
|- !st.
(get_st (run_arm
                [((CMP,NONE,F),NONE,[REG 0; WCONST 0w],NONE);
                 ((B,SOME EQ,F),NONE,[],SOME (POS 6));
                 ((SUB,NONE,F),SOME (REG 3),[REG 0; WCONST 1w],NONE);
                 ((MUL,NONE,F),SOME (REG 2),[REG 0; REG 1],NONE);
                 ((MOV,NONE,F),SOME (REG 0),[REG 3],NONE);
                 ((MOV,NONE,F),SOME (REG 1),[REG 2],NONE);
                 ((B,SOME AL,F),NONE,[],SOME (NEG 6));
                 ((MOV,NONE,F),SOME (REG 2),[REG 1],NONE)]
 ((0,0w,st),{}))<MR R2> = fact(st<MR R0>,st<MR R1>)
```

We show here how our mechanical verifier (associated with Back-end I) applies symbolic simulation and projective Hoare rules to compile the factorial function. An *annotated ir* tree for this function is generated during the compilation.

```
SC(TR((REG 0, eq, WCONST0i),
        \konst{BLK}([{dst = [REG 3], oper = msub, src = [REG 0, WCONST1i]},
             {dst = [REG 2], oper = mmul, src = [REG 0, REG 1]},
             {dst = [REG 0], oper = mmov, src = [REG 3]},
             {dst = [REG 1], oper = mmov, src = [REG 2]}],
            {context = [], fspec = |- T, ins = PAIR(REG 0, REG 1),
             outs = PAIR(REG 0, REG 1)}),
        {context = [NA], fspec = |- T, ins = PAIR(REG 0, REG 1),
         outs = PAIR(REG 0, REG 1)}),
      BLK([{dst = [REG 2], oper = mmov, src = [REG 1]}],
          {context = [], fspec = |- T, ins = PAIR(REG 0, REG 1),
           outs = REG 2}),
      {context = [], fspec = |- T, ins = PAIR(REG 0, REG 1), outs = REG 2})
```

The verifier applies rules according to the structure of this *annotated ir*. At first, it descends to the body of the TR structure and simulates this block symbolically to get a specification:

```
|- let ir = BLK [MSUB R3 (MR R0) (MC 1w); MMUL R2 (MR R0) (MR R1);
                 MMOV R0 (MR R3); MMOV R1 (MR R2)]
   in
   PSPEC ir ((\st. T),(\st. T)) (\st. T)
    ((\st. (st<MR R0>,st<MR R1>)), (\(v0,v1). (v0 + 4294967295w,v0 * v1)),
    (\st. (st<MR R0>,st<MR R1>))) /\ WELL_FORMED ir : thm
```

Then, the `tr` rule is applied to obtain the following:

```
|- let ir = TR (REG 0,EQ,WCONST 0w)
              (BLK [MSUB R3 (MR R0) (MC 1w); MMUL R2 (MR R0) (MR R1);
                    MMOV R0 (MR R3); MMOV R1 (MR R2)])
   in
      PSPEC ir ((\st. T),(\st. T)) (\st. T)
        ((\st. (st<MR R0>,st<MR R1>)),
         WHILE ($~ o (\(v0,v1). v0 = 0w))
         (\(v0,v1). (v0 + 4294967295w,v0 * v1)),
         (\st. (st<MR R0>,st<MR R1>))) /\ WELL_FORMED ir : thm
```

Next, the block following the `TR` structure is simulated to generate another specification.

```
|- let ir = BLK [MMOV R2 (MR R1)] in
       PSPEC ir ((\st. T),(\st. T)) (\st. T)
         ((\st. (st<MR R0>,st<MR R1>)),(\(v0,v1). v1),(\st. st<MR R2>)) /\
       WELL_FORMED ir : thm
```

And then, the `sc` rule is applied to get the final specification.

```
|- let ir = SC
             (TR (REG 0,EQ,WCONST 0w)
                (BLK [MSUB R3 (MR R0) (MC 1w); MMUL R2 (MR R0) (MR R1);
                      MMOV R0 (MR R3); MMOV R1 (MR R2)]))
             (BLK [MMOV R2 (MR R1)])
   in
   PSPEC ir ((\st. T),(\st. T)) (\st. T)
            ((\st. (st<MR R0>,st<MR R1>)), (\(v0,v1). v1) o
```

```
  WHILE ($~ o (\(v0,v1). v0 = 0w))
        (\(v0,v1). (v0 + 4294967295w,v0 * v1)),(\st. st<MR R2>)) /\
WELL_FORMED ir : thm
```

Finally, the semantics function contained in this specification is proved to be equal to the source function with the assistance of preproved theorems about the `WHILE` combinator.

```
|- (\(v0,v1). v1) o WHILE ($~ o (\(v0,v1). v0 = 0w))
                          (\(v0,v1). (v0 + 4294967295w,v0 * v1)) =
    fact : thm
```

CHAPTER 5

ANALYZING PARALLEL PROGRAMS: MPI PROGRAMS

Application Programming Interfaces (API) (also known as libraries) are an important part of modern programming – especially concurrent programming. APIs allow significant new functionality (*e.g.* communication and synchronization) to be provided to programmers without changing the underlying programming language. APIs such as the Message Passing Interface (MPI, [78]) have been in existence for nearly two decades, adapting to the growing needs of programmers for new programming primitives, and growing in the number of primitives supported. The immense popularity of MPI is attributable to the balance it tends to achieve in terms of portability, performance, simplicity, symmetry, modularity, composability, and completeness [38]. While MPI itself has evolved, its basic concepts have essentially remained the same. This has allowed the creation of important long-lived codes — such as weather simulation codes [33]. Despite these successes, *MPI does not have a formal specification.* In this chapter, we present the first formal specification for a significant subset of MPI 2.0.

5.1 Motivation and Background

MPI [111] has become a *de facto* standard in High Performance Computing (HPC) and is being actively developed and supported through several implementations [32, 37, 108]. However, for several reasons, even experienced programmers sometimes misunderstand MPI calls. First, MPI calls are traditionally described in natural languages. Such descriptions are prone to being misinterpreted. Another common approach among programmers is to discover MPI's "intended behavior" by conducting ad hoc experiments using MPI implementations. Such experiments cannot reveal all intended behaviors of an MPI call, and may even be misleading. A formalization of the MPI standard can potentially help avoid these misunderstandings, and also *help define what is an acceptable MPI implementation.*

Engineering a formal specification for a nontrivial concurrency API requires the right combination of rigor, executability, and traceability. A formal specification must also be written as an elaboration of a well-written informal specification. It must also be as direct and declarative in nature, *i.e.* it must not be described in terms of what a specific scheduler might do or rely upon detailed data structures that suggest an actual implementation. Our formal semantics for MPI is written with these goals in mind. At first glance, it may seem that creating a formal specification for MPI which has over 300 fairly complex functions is almost an impossible task. However, as explained in [38], the large size of MPI is somewhat misleading. The primitive concepts involved in MPI are, relatively speaking, quite parsimonious. Our formal specification attempts to take advantage of this situation by first defining a collection of primitives, and then defining MPI calls in terms of these primitives.

Besides contributing directly to MPI, we hope that our work will address the growing need to properly specify and validate future concurrency APIs. In a modern context, APIs allow programmers to harness the rapidly growing power and functionality of computing hardware through new message transfer protocols such as one-sided communication [78] and new implementations of MPI over modern interconnects [52]. Given the explosive growth in concurrency and multicore computing, one can witness a commensurate growth in the number of concurrency APIs being proposed. Among the more recently proposed APIs are various Transactional Memories [42], OpenMP [19], Ct [47], Thread Building Blocks [94], and Task Parallel Library [56]. There is also a high degree of interest in light weight APIs such as the Multicore Communications API (MCAPI) [71] intended to support core-to-core communication in a systems-on-chip multicore setting. One could perhaps draw lessons from exercises such as ours and ensure that for these emerging APIs, the community would create formal specifications contemporaneously with informal specifications. As opposed to this, a formal specification for MPI has been late by nearly two decades in arriving on the scene, because none of the prior work meets our goals for a rigorous specification for MPI.

Besides developing formal specifications, we must also constantly improve the mechanisms that help derive value from formal specifications. For instance, formal

specifications can help minimize the effort to *understand an API*. Concurrency APIs possess many nonintuitive but legal behaviors: how can formal specifications help tutor users of the API as to what these are? Second, it is quite easy to end up with an incorrect or incomplete formal specification. How do we best identify the mistakes or omissions in a formal specification? Third, it is crucial that formal specifications offer assistance in validating or verifying API implementations, especially given that these implementations tend to change much more rapidly than the API semantics themselves change. While we only provide preliminary answers to these issues in this chapter, our hope is that the availability of a formal specification is the very first step in being able to approach these more formidable problems. Last but not least, many scientists believe that the growth in complexity of APIs can have undesirable or unexpected consequences with respect to the more tightly controlled growth of programming language semantics; see [17] for related discussions. We strongly believe that these discussions point to an even stronger need for formal specifications of concurrency APIs, and as a next step to our work, they suggest examining how API formal specifications interact with language and compiler semantics.

5.1.1 Background

The work by Palmer *et al.* [86] presented the formal specification of around 30% of the 128 MPI-1.0 functions (mainly for point-to-point communication) in the specification language TLA+ [112]. TLA+ enjoys wide usage in industry by engineers (*e.g.* in Microsoft [113] and Intel [9]), and is relatively easy to learn. Additionally, in order to help practitioners access our specification, They built a C front-end in the Microsoft Visual Studio (VS) environment, through which users can submit and run short MPI programs with embedded assertions (called litmus tests). Such tests are turned into TLA+ code and run through the TLC model checker [112], which searches all the reachable states to check properties such as deadlocks and user-defined invariants. This permits practitioners to play with (and find holes in) the semantics in a formal setting. [86] shows that this rather simple approach is surprisingly effective for querying a standard and obtaining all possible execution outcomes (some of which are entirely unexpected), as computed by the underlying TLC model checker. In comparison, a programmer experimenting with an actual MPI implementation will not have the benefit of search that a

model checker provides, and be able to check assertions only on executions that materialize in a given MPI implementation along with its (fixed) scheduler.

This chapter extends the work reported in [59, 86], and in addition covers considerably more ground. In particular, we now have a formal specification for nearly 200 MPI functions, including point-to-point calls, MPI data types, collective communication, communicators, process management, one-sided communication, and IO.

Space restrictions prevent us from elaborating on all these aspects: this chapter covers the first three aspects as in a more detailed journal version [65], and a companion technical report [64] covers the rest. (Note: We have not extended the C front-end described in [86] to cover these additional MPI functions.) We have extensively tested our formal specification, as discussed in Section 5.3. Using our formal specification, we have justified a tailored Dynamic Partial Order Reduction algorithm (Section 5.4).

In order to make our specification faithful to the English description, we (i) organize the specification for *easy traceability*: many clauses in our specification are cross-linked with [111] to particular page/line numbers; (ii) provide comprehensive unit tests for MPI functions and a rich set of litmus tests for tricky scenarios; (iii) relate aspects of MPI to each other and verify the self-consistency of the specification; and (iv) provide a programming and debugging environment based on TLC, Phoenix, and Visual Studio to help engage expert MPI users (who may not be formal methods experts) into experimenting with our semantic definitions.

5.1.2 Related Work

The IEEE Floating Point standard [46] was initially conceived as a standard that helped minimize the danger of nonportable floating point implementations, and now has incarnations in various higher-order logic specifications (*e.g.* [41]), finding routine applications in *formal proofs* of modern microprocessor floating point hardware circuits. Formal specifications using TLA+ include Lamport's Win32 Threads API specification [113] and the RPC Memory Problem specified in TLA+ and formally verified in the Isabelle theorem prover by Lamport, Abadi, and Merz [1]. In [49], Jackson presents a lightweight object modeling notation called Alloy, which has tool support [50] in terms of formal analysis and testing based on Boolean satisfiability methods. The approach taken in Alloy is extremely

complementary to what we have set out to achieve through our formal specifications. In particular, their specification of the Java Memory Model is indicative of the expressiveness of Alloy. Abstract State Machines (ASMs) [6] have been used for writing formal specifications of concurrent systems, for instance [55].

Bishop *et al.* [13, 14] formalized in the HOL theorem prover [44] three widely-deployed implementations of the TCP protocol: FreeBSD 4.6-RELEASE, Linux 2.4.20-8, and Windows XP Professional SP1. Analogous to our work, the specification of the interactions between objects are modeled as transition rules. The fact that implementations other than the standard itself are specified requires repeating the same work for different implementations. They perform a vast number of conformance tests to validate the specification. Test programs in a concrete implementation are instrumented and executed to generate execution trances, each of which is then symbolically executed with respect to the formal operational semantics. Constraint solving is used to handle nondeterminism in picking rules or determining possible values in a rule. We also rely on testing for validation check. As it is the standard that we formalize, we need to write all the test cases by hand.

Norrish [84] formalized in HOL [44] a structural operational semantics and a type system of the majority of the C language, covering the dynamic behavior of C programs. Semantics of expressions, statements, and declarations are modeled as transition relations. The soundness of the semantics and the type system is proved formally. In addition, a set of Hoare rules are derived from the operational semantics to assist property verification. In contrast, our specification defines the semantics in a more declarative style and does not encode the correctness requirement into a type system.

Two other related works in terms of writing executable specifications are the Symbolic Analysis Laboratory (SAL) approach [100] and the use of the Maude rewrite technology [70]. The use of these frameworks may allow us to employ alternative reasoning techniques: using decision procedures (in case of SAL), and using term rewriting (in case of Maude). These will be considered during our future work.

Georgelin and Pierre [34] specify some of the MPI functions in LOTOS [27]. Siegel and Avrunin [104] describe a finite state model of a limited number of MPI point-to-point operations. This finite state model is embedded in the SPIN

model checker [45]. They [105] also support a limited partial-order reduction method – one that handles wild-card communications in a restricted manner, as detailed in [87]. Siegel [103] models additional 'nonblocking' MPI primitives in Promela. None of these efforts: (i) approach the number of MPI functions we handle, (ii) have the same style of high-level specifications (TLA+ is much closer to mathematical logic than finite-state Promela or LOTOS models), (iii) have a model extraction framework starting from C/MPI programs, and (iv) have a practical way of displaying error traces in the user's C code.

5.2 Specification

TLA+ provides built-in support for sets, functions, records, strings, and sequences. To model MPI objects, we extend the TLA+ library by defining advanced data structures including maps and ordered sets (`oset`). For instance, MPI groups and I/O files are represented by ordered sets.

The approximate sizes (excluding comments and blank lines) of the major parts in the current specification are shown in Table 5.1, where #funcs and #lines give the number of MPI primitives and code lines, respectively. We do not model functions whose behavior depends on the underlying operating system. For deprecated items, we only model their replacement.

5.2.1 Data Structures

The data structures modeling explicit and opaque MPI objects are shown in Figure 5.1. Each process contains a set of local objects such as the local memory

Table 5.1: Size of the MPI 2.0 specification (excluding comments and blank lines).

Main Module	**#funcs(#lines)**
Point-to-point Communication	35(800)
Userdefined Datatype	27(500)
Group and Communicator Management	34(650)
Intracollective Communication	16(500)
Topology	18(250)
Environment Management in MPI 1.1	10(200)
Process Management	10(250)
One-sided Communication	15(550)
Intercollective Communication	14(350)
I/O	50(1100)
Interface and Environment in MPI 2.0	35(800)

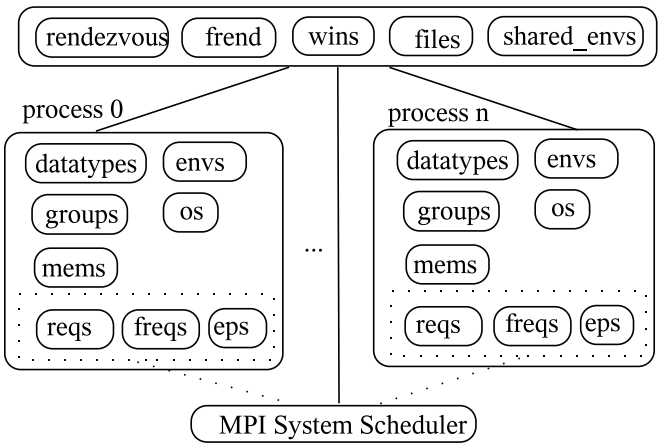


Figure 5.1: MPI objects and their interaction

object `mems`. Multiple processes coordinate with each other through shared objects `rendezvous`, `wins`, and so on. The message passing procedure is simulated by the *MPI system scheduler (MSS)*, which matches requests at origins and destinations and performs message passing. MPI primitive calls at different processes make transitions nondeterministically.

Request object `reqs` is used in point-to-point communications. A similar file request object `freqs` is for parallel I/O communications. Objects `groups` and `comms` model the groups and (intra- or inter-) communicators, respectively. In addition to the group, a communicator also includes virtual topology and other attributes. Objects `rendezvous` and `frend` objects are for collective communications and shared file operations, respectively. Objects `epos` and `wins` are used in one-sided communications.

Other MPI objects are represented as components in a shared environment `shared_envs` and local environments `envs`. The underlying operating system is abstracted as `os` in a limited sense, which includes the objects visible to the MPI system such as physical files on the disk. We define a separate object `mems` for the physical memory at processes.

5.2.2 Notations

Our presentation uses notations extended and abstracted from TLA+. The basic concept in TLA+ is functions. We write $f[v]$ for the value of function f applied to v; this value is specified only if v is in f's domain `DOM` f. Notation

$[S \to T]$ specifies the set of all functions f such that $\texttt{DOM}\ f = S$ and $f[v] \in T$ for all $v \in S$. For example $[\texttt{int} \to \texttt{nat}]$ denotes all functions from integers to natural numbers. This notation is usually used to specify the type of a function.

Functions may be described explicitly with the construct $[x \in S \mapsto e]$ such that $f[x] = e$ for $x \in S$. For example, the function f_{double} that doubles input natural numbers can be specified as $[x \in \texttt{nat} \mapsto 2x]$. Obviously, $f_{double}[1] = 2$ and $f_{double}[4] = 8$. Notation $[f\ \texttt{EXCEPT}\ ![e_1] = e_2]$ defines a function f' such that f' is the same as f except $f'[e_1] = e_2$. An @ appearing in e_2 represents the old value of $f[e_1]$. For example, $[f_{double}\ \texttt{EXCEPT}\ ![3] = @ + 10]$ is the same as f_{double} except that it returns 16 for input 3.

Tuples, arrays, records, sequences, and ordered sets are special functions with finite domains. They differ mainly in the operators defined over these data structures. An n-tuple is written as $\langle e_1, \ldots, e_n \rangle$, which defines a function f with domain $\{1, \ldots, n\}$ such that $f[i] = e_i$ for $1 \leq i \leq n$. Its i^{th} component is given by $\langle e_1, \ldots, e_n \rangle[i]$. An array resembles a tuple except that its index starts from 0 rather than 1 so as to conform to the convention of the C language. Records can be written explicitly as $[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$, which is actually a function mapping field h_i to value e_i. For instance tuple $\langle 1, 4, 9 \rangle$, record $[1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 9]$, and function $[x \in \{1, 2, 3\} \mapsto x^2]$ are equivalent. Similar to function update, $[r\ \texttt{EXCEPT}\ !.h = e]$ represents a record r' such that r' is the same as r except $r'.h = e$, where $r.h$ returns the h-field of record r.

A (finite) sequence is represented as a tuple. Operators are provided to obtain the head or tail elements, append elements, concatenate two sequences, and so on. An ordered set is analogous to a usual set except it consists of distinct elements. It may be interpreted as a function too: its domain is $[0, n-1]$ where n is the number of elements (*i.e.* the cardinality), and its range contains all the elements.

The basic temporal logic operator used to define transition relations is the next state operator, denoted using $'$ or *prime*. For example, $s' = [s\ \texttt{EXCEPT}\ ![x] = e]$ indicates that the next state s' is equal to the original state s except that x's value is changed to e.

For illustration, consider a stop watch that displays hour and minute. A typical behavior of the clock is a sequence $[hr \mapsto 0, mnt \mapsto 0]$, $[hr \mapsto 0, mnt \mapsto 1]$, $\ldots$, $[hr \mapsto 0, mnt \mapsto 59]$, $[hr \mapsto 1, mnt \mapsto 0]$, $\ldots$, where $[hr \mapsto i, mnt \mapsto j]$ is a state with hour i and minute j. Its next-state relation is a formula expressing the relation between

the values of hr and mnt. It asserts that mnt equals $mnt + 1$ if $mnt \neq 59$. When mnt is 59, mnt is reset to 0, and hr increased by 1.

$$
\begin{array}{ll}
time' = & \texttt{let } c = (time[mnt] \neq 59) \texttt{ in} \\
& [time \texttt{ EXCEPT } ![mnt] = \\
& \quad \texttt{if } c \texttt{ then } @ + 1 \texttt{ else } 0,\ ![hr] = \texttt{if } \neg c \texttt{ then } @ + 1 \texttt{ else } @]
\end{array}
$$

To make the specification succinct, we introduce some other commonly used notations. Note that $\top$ and $\bot$ denote Boolean value $true$ and $false$, respectively; and ϵ and α denote the null value and an arbitrary value, respectively. Notation $\Gamma_1 \diamond x_k \diamond \Gamma_2$ specifies a queue where x is the k^{th} element, Γ_1 contains the elements before x, and Γ_2 contains the elements after x. When it appears in the precondition of a transition rule, it should be interpreted in a *pattern-matching* manner such that Γ_1 returns the first $k-1$ elements, x is the k^{th} element, and Γ_2 returns the rest elements.

$\Gamma_1 \diamond \Gamma_2$	the concatenation of queue Γ_1 and Γ_2
$\Gamma_1 \diamond x_k \diamond \Gamma_2$	the queue with x being the k^{th} element
$\Gamma_1 \sqsubseteq \Gamma_2$	Γ_1 is a sub-queue (sub-array) of Γ_2
$\top, \bot, \epsilon$ and α	true, false, null value and arbitrary value
$f_1 = f \ \uplus \ (x, v)$	$\texttt{DOM}(f_1) = \texttt{DOM}(f) \cup \{x\} \ \wedge \ x \notin \texttt{DOM}(f) \ \wedge \ f_1[x] = v$ $\wedge \forall y \in \texttt{DOM}(f) : f_1[y] = f[y]$
$f\|_x$	the index of element x in function f, *i.e.* $f[f\|_x] = x$
$c \ ? \ e_1 \ : \ e_2$	An abbreviation for $\texttt{if } c \texttt{ then } e_1 \texttt{ else } e_2$
$\texttt{size}(f)$ or $\|f\|$	the number of elements in function f

Similar to the separating operator $*$ in separation logic [95], operator $\uplus$ divides a function into two parts with disjoint domains. For example, function $[x \in \{1,2,3\} \mapsto x^2]$ can be written as $[x \in \{1,2\} \mapsto x^2] \uplus (3,9)$ or $[x \in \{1,3\} \mapsto x^2] \uplus (2,4)$. This operator is especially useful when representing the content of a function.

TLA+ allows the specification of MPI primitives in a declarative style. For illustration we show below a helper (auxiliary) function used to implement the `MPI_COMM_SPLIT` primitive, where *group* is an ordered set of processes, *colors* and *keys* are arrays. Here, `DOM`, `RNG`, `CARD` return the domain, range, and cardinality of an ordered set, respectively. This code directly formalizes the English description (see page 147 in [111]): "This function partitions the group into disjoint subgroups, one for each value of `color`. Each subgroup contains all processes of the same color. Within each subgroup, the processes are ranked in the order defined by

`key`, with ties broken according to their rank in the old group. When the process supplies the color value `MPI_UNDEFINED`, a null communicator is returned." In contrast, it is impossible to write such declarative specification in the C language.

$$
\begin{array}{ll}
 & \texttt{Comm\_split}(group, colors, keys, proc) \doteq \\
1: & \texttt{let } rank = group|_{proc} \texttt{ in} \\
2: & \texttt{if } (colors[rank] = \texttt{MPI\_UNDEFINED}) \texttt{ then MPI\_GROUP\_NULL} \\
3: & \texttt{else} \\
4: & \quad \texttt{let } same\_colors = \{k \in \texttt{DOM}(group) : colors[k] = colors[rank]\} \texttt{ in} \\
5: & \quad \texttt{let } sorted\_same\_colors = \\
6: & \quad\ \texttt{choose } g \in [\texttt{DOM}(same\_colors) \to \texttt{RNG}(same\_colors)] : \\
7: & \quad\ \wedge\ \texttt{RNG}(g) = same\_colors \\
8: & \quad\ \wedge\ \forall i, j \in same\_colors : g|_i < g|_j \Rightarrow \\
9: & \qquad\ (keys[i] < keys[j]\ \vee\ (keys[i] = keys[j]\ \wedge\ i < j)) \\
10: & \quad \texttt{in } [i \in \texttt{DOM}(sorted\_same\_colors) \mapsto group[sorted\_same\_colors[i]]]
\end{array}
$$

After collecting the color and key information from all other processes, a process *proc* calls this function to create the group of a new communicator. Line 1 calculates *proc*'s rank in the group; line 4 obtains an ordered set of the ranks of all the processes with the same color as *proc*; lines 5-9 sort this rank set in the ascending order of keys, with ties broken according to the ranks. Specifically, lines 6-7 pick an ordered set g with the same domain and range as *same_colors*; lines 8-9 indicates that, in g, rank i shall appear before rank j (*i.e.* $g_{|i} < g_{|j}$) if the key at i is less than that at j. This specification may be a little tricky as we need to map a process to its rank before accessing its color and key. This merits our formalization which explicitly describes all the details. For illustration, suppose $group = \langle 2, 5, 1\rangle$, $colors = \langle 1, 0, 0\rangle$, and $keys = \langle 0, 2, 1\rangle$, then the call of this function at process 5 creates a new group $\langle 1, 5\rangle$.

5.2.2.1 Operational Semantics

The formal semantics of an MPI primitive is modeled by a state transition. A system state consists of explicit and opaque objects mentioned in 5.2.1. An object may involve multiple processes; we write $\texttt{obj}_p$ for the object `obj` at process p. For example, $\texttt{reqs}_p$ refers to the request object (for point-to-point communications) at process p.

We use notation $\stackrel{\circ}{=}$ to define the semantics of an MPI primitive, and $\doteq$ to introduce a helper function. The precondition *cond* of a transition, if exists, is specified by "`requires` {*cond*}." An error is reported if this precondition is

violated. The body of a transition is expressed by a rule of format $\frac{guard}{action}$, where *guard* specifies the requirement for the transition to be triggered, and *action* defines how the MPI objects are updated after the transition. When the guard is satisfied, the action is enabled and may be performed. Otherwise, the rule is blocked and the action will be delayed. A true guard will be omitted, meaning that the transition is always enabled.

For instance, the semantics of `MPI_Buffer_detach` is shown below. A buffer object contains several fields: $buff$ and $size$ record the start address in the memory and the size, respectively; *capacity* and *max_capacity* record the available space and maximum space, respectively. The values of these fields are set when the buffer is created. The precondition of the `MPI_Buffer_detach` rule enforces that process p's buffer must exist; the guard indicates that the transition will block until all messages in the buffer have been transmitted (*i.e.* the entire space is available); the action is to write the buffer address and the buffer size into p's local memory, and deallocate the space occupied by the buffer.

$$\texttt{MPI\_Buffer\_detach}(buff, size, p) \triangleq$$
$$\texttt{requires } \{\texttt{buffer}_p \neq \epsilon\}$$
$$\frac{\texttt{buffer}_p.capacity = \texttt{buffer}_p.max\_capacity}{\texttt{mems}'_p = [\texttt{mems}_p \texttt{ EXCEPT } ![buff] = \texttt{buffer}_p.buff,\ ![size] = \texttt{buffer}_p.size] \ \wedge\ \texttt{buffer}'_p = \epsilon}$$

It may be desirable to specify only the objects and components that are affected by the transition such that those not appeared in the action are assumed to be unchanged. Thus, the action of the above rule can be written as follows. We will use this lighter notation throughout the rest of Section 5.2.

$$\texttt{mems}'_p[buff] = \texttt{buffer}_p.buff \ \wedge\ \texttt{mems}'_p[size] = \texttt{buffer}_p.size \ \wedge\ \texttt{buffer}'_p = \epsilon$$

5.2.3 Quick Overview of the Methodology

We first give a simple example to illustrate how MPI programs and MPI primitives are modeled. Consider the following program involving two processes:

$$\begin{array}{ll} P0: & \texttt{MPI\_Send}(buf_s, 2, \texttt{MPI\_INT}, 1, 10, \texttt{MPI\_COMM\_WORLD}) \\ & \texttt{MPI\_Bcast}(buf_b, 1, \texttt{MPI\_FLOAT}, 0, \texttt{MPI\_COMM\_WORLD}) \\ P1: & \texttt{MPI\_Recv}(buf_r, 2, \texttt{MPI\_INT}, 0, \texttt{MPI\_ANY\_TAG}, \texttt{MPI\_COMM\_WORLD}) \\ & \texttt{MPI\_Bcast}(buf_b, 1, \texttt{MPI\_FLOAT}, 0, \texttt{MPI\_COMM\_WORLD}) \end{array}$$

This program is converted by our compiler into the following TLA+ code (*i.e.* the model of this program), where the TLA+ code of MPI primitives will be presented in subsequent sections. An extra parameter is added to an MPI primitive to specify the process it belongs to. In essence, a program model is a transition system consisting of transition rules. When the guard of a rule is satisfied, this rule is enabled and ready for execution. Multiple enabled rules are executed in a nondeterministic manner. The control flow of a program at process p is represented by the pc values: $pc[p]$ stores the current values of the program pointer. The pc values are integer-value labels such as L_1, L_2, and so forth. A blocking call is modeled by its nonblocking version followed by a wait operation, *e.g.* `MPI_Send` $\triangleq$ `(MPI_Isend; MPI_Wait)`. The compiler treats $request_0$ and $status_0$ as references to memory locations. For example, suppose reference $request_0$ has address 5, then the value it points to is $\texttt{mems}_p[request_0]$ (*i.e.* $\texttt{mems}_p[5]$). As all variables in the source C program are mapped to memory locations.

p0's transition rules

$$\begin{array}{ll}
\vee & \wedge\ pc[0] = L_1\ \wedge\ pc'[0] = L_2 \\
& \wedge\ \texttt{MPI_Isend}(buf_s, 2, \texttt{MPI_INT}, 1, 10, \texttt{MPI_COMM_WORLD}, request_0, 0) \\
\vee & \wedge\ pc[0] = L_2\ \wedge\ pc'[0] = L_3\ \wedge\ \texttt{MPI_Wait}(request_0, status_0, 0) \\
\vee & \wedge\ pc[0] = L_3\ \wedge\ pc'[0] = L_4\ \wedge\ \texttt{MPI_Bcast}_{init}(buf_b, 1, \texttt{MPI_FLOAT}, 0, \texttt{MPI_COMM_WORLD}, 0) \\
\vee & \wedge\ pc[pid] = L_4\ \wedge\ pc'[0] = L_5\ \wedge\ \texttt{MPI_Bcast}_{wait}(buf_b, 1, \texttt{MPI_FLOAT}, 0, \texttt{MPI_COMM_WORLD}, 0)
\end{array}$$

p1's transition rules

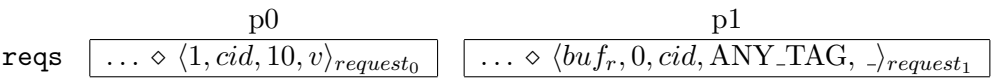
$$\begin{array}{ll}
\vee & \wedge\ pc[1] = L_1\ \wedge\ pc'[1] = L_2 \\
& \wedge\ \texttt{MPI_Irecv}(buf_r, 2, \texttt{MPI_INT}, 0, \texttt{MPI_ANY_TAG}, \texttt{MPI_COMM_WORLD}, request_1, 1) \\
\vee & \wedge\ pc[1] = L_2\ \wedge\ pc'[1] = L_3\ \wedge\ \texttt{MPI_Wait}(request_1, status_1, 1) \\
\vee & \wedge\ pc[1] = L_3\ \wedge\ pc'[1] = L_4\ \wedge\ \texttt{MPI_Bcast}_{init}(buf_b, 1, \texttt{MPI_FLOAT}, 0, \texttt{MPI_COMM_WORLD}, 1) \\
\vee & \wedge\ pc[1] = L_4\ \wedge\ pc'[1] = L_5\ \wedge\ \texttt{MPI_Bcast}_{wait}(buf_b, 1, \texttt{MPI_FLOAT}, 0, \texttt{MPI_COMM_WORLD}, 1)
\end{array}$$

An enabled rule may be executed at any time. Suppose the program pointer of process $p0$ is L_1, then the `MPI_Isend` rule may be executed, modifying the program pointer to L_2. As indicated below, it creates a new send request of format $\langle destination, communicator\_id, tag, value\rangle_{request\_id}$, and appends it to $p0$'s request queue $\texttt{reqs}_0$. Value v is obtained by $read\_data(\texttt{mems}_0, buf_s, 2, \text{MPI\_INT})$, which reads from the memory two consecutive integers starting from address buf_s.

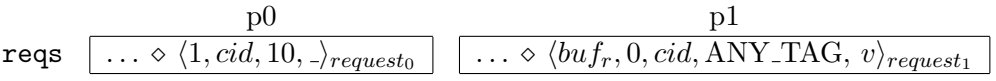
	process $p0$	process $p1$
`reqs`	$\ldots \diamond \langle 1, cid, 10, v\rangle_{request_0}$	$\ldots$

Similarly, when the `MPI_Irecv` rule at process $p1$ is executed, a new receive request of format $\langle buffer, source, communicator\_id, tag, \_\rangle_{request\_id}$ is appended to

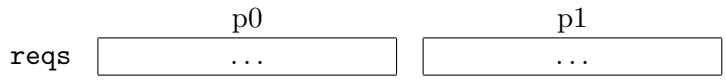
$\texttt{reqs}_1$, where _ indicates that the data value is yet to be received.



The MPI System Scheduler matches the send request and the receive request, and transfers the data value v from $p0$ to $p1$. After the transferring, the value fields in the send and receive requests become _ and v, respectively.



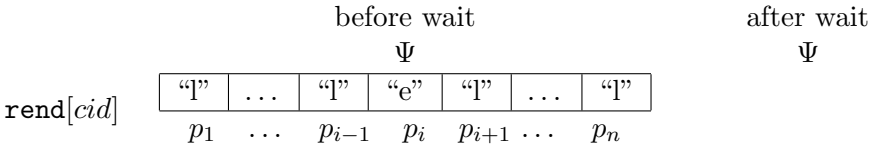
If the send is not buffered at $p0$, then the `MPI_Wait` call will be blocked until the data v is sent. After that, the send request is removed from the queue. Analogously, the `MPI_Wait` rule at $p1$ is blocked until the incoming value arrives. Then, v is written into $p1$'s local memory and this request is removed.



In our formalization, each process divides a collective primitive call into two phases: an "init" phase that initializes the call, and a "wait" phase that synchronizes the communication with other processes. Processes synchronize with each other through the `rendezvous` (or `rend` for short) object which records the status of the communication (denoted by Ψ) and the data sent by the processes (denoted by S_v). For a communicator with context ID cid, there exists an individual rendezvous object $\texttt{rend}[cid]$. In the "init" phase, process p_i is able to proceed only if it is not in the domain the status component (*i.e.* p_i is not participating the communication). It updates its status to "e" ("entered") and stores its data in the rendezvous. In the given example, after the "init" phases of the broadcast at process 0 and 1 are over, the rendezvous pertaining to communicator `MPI_COMM_WORLD` becomes $\langle[0 \mapsto \text{“e”}, 1 \mapsto \text{“e”}], [0 \mapsto \text{val}]\rangle$, where $val = read\_data(\texttt{mems}_0, buf_b, 1, \texttt{MPI\_FLOAT})$.

$$\frac{p_i \notin \texttt{DOM}(\Psi)}{\texttt{rend}'[cid] = \langle \Psi \uplus (p_i, \text{“e”}),\ S_v \uplus (p_i, val)\rangle} \quad \texttt{syn}_{\texttt{init}}(cid, val, p_i) \doteq \text{process } p_i \text{ joins the communication and stores data } v \text{ in } \texttt{rend}$$

In the "wait" phase, if the communication is synchronizing, then process p_i has to wait until all other processes finish their "init" phases. If p_i is the last process that leaves the communication, then the `rend` object will be deleted; otherwise, p_i just updates its status to "l" ("left").



$$\texttt{syn}_{\texttt{wait}}(cid, p_i) \doteq \text{ process } p \text{ leaves the synchronizaing communication}$$

$$\frac{\texttt{rend}[cid] = \langle \Psi \uplus (p_i, \text{"e"}), S_v \rangle \ \wedge \ \forall k \in \texttt{comms}_{p_i}[cid].group : k \in \texttt{DOM}(\Psi)}{\begin{array}{rl} \texttt{rend}'[cid] = & \texttt{if } \forall k \in \texttt{comms}_{p_i}[cid].group : \Psi[k] = \text{"l"} \\ & \texttt{then } \epsilon \texttt{ else } \langle \Psi \uplus (p_i, \text{"l"}),\ S_v \rangle \end{array}}$$

These simplified rules illustrate how MPI point-to-point and collective communications are modeled. The standard rules are given in Section 5.2.4 and 5.2.5.

5.2.4 Point-to-point Communication

The semantics of core point-to-point communication primitives are shown in Figures 5.2, 5.3 and 5.4. Figure 5.5 gives an example illustrating the "execution" of point-to-point communication primitives. Readers should refer to the semantics when reading through this section.

New send and receive requests are appended to the request queues. A send request contains information about the destination process (dst), the context ID of the communicator (cid), the tag to be matched (tag), the data value to be send ($value$), and the status (omitted here) of the message. This request also includes Boolean flags indicating whether the request is persistent, active, live, canceled, and deallocated or not. For brevity, we do not show the last three flags when presenting the content of a request in the queue. In addition, in order to model the ready send, we include in the send request a field $prematch$ of format $\langle destination, request\_index \rangle$ which points to the receive request matching this send request. A receive request contains similar fields plus the buffer address and a field to store the incoming data. Initially, the data value is missing (represented by the "_" in the data field); an incoming message from a sender will replace the "_" with the data it carries. Notation $v_{\_}$ denotes either data value v arrives or the data is still missing. For example, $\langle buf, 0, 10, *, \_, \top, \top, \langle 0, 5 \rangle \rangle_2^{recv}$ is a receive

Data Structures

$send\ request$: important fields + less important fields
$\langle dst : \texttt{int},\ cid : \texttt{int},\ tag : \texttt{int},\ value,\ pr : \texttt{bool},\ active : \texttt{bool},\ prematch\rangle^{mode}$ +
$\langle cancelled : \texttt{bool},\ dealloc : \texttt{bool},\ live : \texttt{bool}\rangle$
$recv\ request$: important fields + less important fields
$\langle buf : \texttt{int},\ src : \texttt{int},\ cid : \texttt{int},\ tag : \texttt{int},\ value,\ pr : \texttt{bool},\ active : \texttt{bool},\ prematch\rangle^{recv}$
$+ \langle cancelled : \texttt{bool},\ dealloc : \texttt{bool},\ live : \texttt{bool}\rangle$

$\texttt{ibsend}(v, dst, cid, tag, p) \triangleq$ buffer send
$\texttt{requires}\ \{\texttt{size}(v) \leq \texttt{buffer}_p.vacancy\}$ check buffer availability
$\texttt{reqs}'_p = \texttt{reqs}_p \diamond \langle dst, cid, tag, v, \bot, \top, \epsilon\rangle^{bsend} \wedge$ append a new send request
$\texttt{buffer}'_p.vacancy = \texttt{buffer}_p.vacancy - \texttt{size}(v)$ allocate buffer space

$(\langle p, dst, tag_p, \omega_p, k_p\rangle \asymp \langle src, q, tag_q, \omega_q, k_q\rangle) \doteq$ match send and receive requests
$\texttt{if}\ \omega_p = \epsilon \wedge \omega_q = \epsilon\ \texttt{then}\ tag_q \in \{tag_p, \texttt{ANY\_TAG}\} \wedge q = dst \wedge src \in \{p, \texttt{ANY\_SOURCE}\}$
$\texttt{else}\ \omega_p = \langle q, k_q\rangle \wedge \omega_q = \langle p, k_p\rangle$ prematched requests

$\texttt{irsend}(v, dst, cid, tag, p) \triangleq$ ready send
$\texttt{requires}\ \{\ \exists q : \exists\langle src, cid, tag_1, \_, pr_1, \top, \epsilon\rangle_k^{recv} \in \texttt{reqs}_q : \langle p, dst, tag, \epsilon, \texttt{size}(\texttt{reqs}_p)\rangle \asymp \langle src, q, tag_1, \epsilon, k\rangle\ \}$ a matching recv exists?
$\texttt{reqs}'_p = \texttt{reqs}_p \diamond \langle dst, cid, tag, v, \bot, \top, \langle q, k\rangle\rangle^{rsend} \wedge \texttt{reqs}'_q.\omega = \langle p, \texttt{size}(\texttt{reqs}_p)\rangle$

$\texttt{isend} \triangleq \texttt{if}\ use\_buffer\ \texttt{then ibsend else issend}$ standard mode send

$\texttt{irecv}(buf, src, cid, tag, p) \triangleq \quad \texttt{reqs}'_p = \texttt{reqs}_p \diamond \langle buf, src, cid, tag, \_, \bot, \top, \epsilon\rangle^{recv}$

$\texttt{MPI\_Isend}(buf, count, dtype, dest, tag, comm, request, p) \triangleq$ standard immediate send
$\texttt{let}\ cm = \texttt{comms}_p[comm]\ \texttt{in}$ the communicator
$\wedge\ \texttt{isend}(read\_data(\texttt{mems}_p, buf, count, dtype), cm.group[dest], cm.cid, tag, p)$
$\wedge\ \texttt{mems}'_p[request] = \texttt{size}(\texttt{reqs}_p)$ set the request handle

$\texttt{MPI\_Irecv}(buf, count, dtype, source, tag, comm, request, p) \triangleq$ immediate receive
$\texttt{let}\ cm = \texttt{comms}_p[comm]\ \texttt{in}$ the communicator
$\quad \texttt{irecv}(buf, cm.group[dest], cm.cid, tag, p) \wedge \texttt{mems}'_p[request] = \texttt{size}(\texttt{reqs}_p)$

$\texttt{wait\_one}(request, status, p) \doteq$ wait for one request to complete
$\texttt{if}\ \texttt{reqs}_p[\texttt{mems}_p[request]].mode = recv$
$\texttt{then}\ \texttt{recv\_wait}(request)$ for receive request
$\texttt{else}\ \texttt{send\_wait}(request)$ for send request

$\texttt{MPI\_Wait}(request, status, p) \triangleq$ the top level wait function
$\texttt{if}\ \texttt{mems}_p[request] \neq \texttt{REQUEST\_NULL}\ \texttt{then}\ \texttt{wait\_one}(request, status, p)$
$\texttt{else}\ \texttt{mems}'_p[status] = empty\_status$ the handle is null, return an empty status

Figure 5.2: Modeling point-to-point communications (I)

$\texttt{transfer}(p, q) \triangleq$ message transferring from process p to process q

$\quad \wedge\ \texttt{reqs}_p = \Gamma_1^p \diamond \langle dst, cid, tag_p, v, pr_p, \top, \omega_p \rangle_i^{send} \diamond \Gamma_2^p$

$\quad \wedge\ \texttt{reqs}_q = \Gamma_1^q \diamond \langle buf, src, cid, tag_q, \_, pr_q, \top, \omega_q \rangle_j^{recv} \diamond \Gamma_2^q \wedge$

$\quad \wedge$ match the requests in a FIFO manner

$\qquad \langle p, dst, tag_p, \omega_p, i \rangle \rightarrowtail \langle src, q, tag_q, \omega_q, j \rangle \wedge$

$\qquad \nexists \langle dst, cid, tag_1, v, pr_1, \top, \omega_1 \rangle_m^{send} \in \Gamma_1^p :$

$\qquad\quad \nexists \langle buf, src_2, cid, tag_2, \_, pr_2, \top, \omega_2 \rangle_n^{recv} \in \Gamma_1^q :$

$\qquad\qquad \vee\ \langle p, dst, tag_1, \omega_1, m \rangle \rightarrowtail \langle src, q, tag_q, \omega_q, j \rangle$

$\qquad\qquad \vee\ \langle p, dst, tag_p, \omega_p, i \rangle \rightarrowtail \langle src_2, q, tag_2, \omega_2, n \rangle$

$\qquad\qquad \vee\ \langle p, dst, tag_1, \omega_1, m \rangle \rightarrowtail \langle src_2, q, tag_2, \omega_2, n \rangle$

$\quad \wedge\ \texttt{reqs}'_p =$ send the data

$\qquad \texttt{let}\ b = \texttt{reqs}_p[i].live\ \texttt{in}$

$\qquad\quad \texttt{if}\ \neg b\ \wedge\ \neg \texttt{reqs}_p[i].pr\ \texttt{then}\ \Gamma_1^p \diamond \Gamma_2^p$

$\qquad\quad \texttt{else}\ \Gamma_1^p \diamond \langle dst, cid, tag_p, \_, pr_p, b, \omega_p \rangle^{send} \diamond \Gamma_2^p$

$\quad \wedge\ \texttt{reqs}'_q =$ receive the data

$\qquad \texttt{let}\ b = \texttt{reqs}_q[j].live\ \texttt{in}$

$\qquad\quad \texttt{if}\ \neg b\ \wedge\ \neg \texttt{reqs}_q[j].pr\ \texttt{then}\ \Gamma_1^q \diamond \Gamma_2^q$

$\qquad\quad \texttt{else}\ \Gamma_1^q \diamond \langle buf, p, cid, tag_q, v, pr_q, b, \omega_q \rangle^{recv} \diamond \Gamma_2^q$

$\quad \wedge\ \neg \texttt{reqs}_q[j].live \Rightarrow \texttt{mems}'_q[buf] = v$ write the data into memory

$\texttt{recv\_wait}(request, status, p) \triangleq$ wait for a receive request to complete

$\texttt{let}\ req\_index = \texttt{mems}_p[request]\ \texttt{in}$

$\wedge\ \texttt{reqs}'_p[req\_index].live = \bot$ indicate the wait has been called

$\wedge$

$\quad \vee\ (\neg \texttt{reqs}_p[req\_index].active \Rightarrow \texttt{mems}'_p[status] = empty\_status)$

$\quad \vee$ the request is still active

$\qquad \texttt{let}\ \Gamma_1 \diamond \langle buf, src, cid, tag, v\_, pr, \top, \omega \rangle_{req\_index}^{recv} \diamond \Gamma_2 = \texttt{reqs}_q\ \texttt{in}$

$\qquad \texttt{let}\ b = pr\ \wedge\ \neg \texttt{reqs}_p[req\_index].dealloc\ \texttt{in}$

$\qquad \texttt{let}\ new\_reqs =$

$\qquad\quad \texttt{if}\ b\ \texttt{then}\ \Gamma_1 \diamond \langle buf, src, cid, tag, v\_, pr, \bot, \omega \rangle^{recv} \diamond \Gamma_2$ deactivate the request

$\qquad\quad \texttt{else}\ \Gamma_1 \diamond \Gamma_2$ remove the request

$\qquad \texttt{in}$

$\qquad \texttt{let}\ new\_req\_index = \texttt{if}\ b\ \texttt{then}\ req\_index\ \texttt{else}\ \texttt{REQUEST\_NULL}\ \texttt{in}$ handle update

$\qquad \texttt{if}\ \texttt{reqs}_q[req\_index].cancelled\ \texttt{then}$

$\qquad\quad \texttt{mems}'_p[status] = get\_status(\texttt{reqs}_p[req\_index]) \wedge$

$\qquad\quad \texttt{reqs}'_p = new\_reqs\ \wedge\ \texttt{mems}'_p[request] = new\_req\_index$

$\qquad \texttt{else if}\ src = \texttt{PROC\_NULL}\ \texttt{then}$

$\qquad\quad \texttt{mems}'_p[status] = null\_status\ \wedge\ \texttt{reqs}'_p = new\_reqs \wedge$

$\qquad\quad \texttt{mems}'_p[request] = new\_req\_index$

$\qquad \texttt{else}$

$\qquad\qquad \dfrac{v\_ \neq \_}{\texttt{mems}'_p[status] = get\_status(\texttt{reqs}_p[req\_index])\ \wedge\ \texttt{mems}'_p[buf] = v\_ \wedge \texttt{reqs}'_p = new\_reqs\ \wedge\ \texttt{mems}'_p[request] = new\_req\_index}$

Figure 5.3: Modeling point-to-point communications (II)

$\texttt{send\_wait}(request, status, p) \triangleq$ wait for a receive request to complete
$\texttt{let}\ req\_index = \texttt{mems}_p[request]\ \texttt{in}$
$\wedge\ \texttt{reqs}'_p[req\_index].live = \bot$ indicate the wait has been called
$\wedge$
$\quad \vee (\neg \texttt{reqs}_p[req\_index].active \Rightarrow \texttt{mems}'_p[status] = empty\_status)$
$\quad \vee$ the request is still active
$\quad \texttt{let}\ \Gamma_1 \diamond \langle dst, cid, tag, v\_, pr, \top, \omega\rangle^{mode}_{req\_index} \diamond \Gamma_2 = \texttt{reqs}_q\ \texttt{in}$
$\quad \texttt{let}\ b = pr\ \wedge\ \neg \texttt{reqs}_p[req\_index].dealloc\ \vee\ v\_ \neq \_\ \texttt{in}$
$\quad \texttt{let}\ new\_reqs =$
$\quad\quad \texttt{if}\ \neg b\ \texttt{then}\ \Gamma_1 \diamond \Gamma_2$ remove the request
$\quad\quad \texttt{else}\ \Gamma_1 \diamond \langle buf, src, cid, tag, v\_, pr, \bot, \omega\rangle^{recv} \diamond \Gamma_2$ deactive the request
$\quad \texttt{in}$
$\quad \texttt{let}\ new\_req\_index = \texttt{if}\ b\ \texttt{then}\ req\_index\ \texttt{else}\ \texttt{REQUEST\_NULL}\ \texttt{in}$
$\quad \texttt{let}\ action =$ update the queue, the status and the request handle
$\quad\quad \wedge\ \texttt{mems}'_p[status] = get\_status(\texttt{reqs}_p[req\_index])$
$\quad\quad \wedge\ \texttt{reqs}'_p = new\_reqs\ \wedge\ \texttt{mems}'_p[request] = new\_req\_index$
$\quad \texttt{in}$
$\quad\quad \texttt{if}\ \texttt{reqs}_q[req\_index].cancelled\ \texttt{then}\ action$
$\quad\quad \texttt{else if}\ dst = \texttt{PROC\_NULL}\ \texttt{then}\ \texttt{mems}'_p[status] = null\_status \wedge$
$\quad\quad\quad\quad \texttt{reqs}'_p = new\_reqs\ \wedge\ \texttt{mems}'_p[request] = new\_req\_index$
$\quad\quad \texttt{else if}\ mode = ssend\ \texttt{then}$ synchronous send requires a matching receive
$\quad\quad\quad \dfrac{\exists q : \exists \langle src_1, cid, tag_1, \_, pr_1, \top, \omega_1\rangle^{recv}_k \in \Gamma_1 : \langle dst, p, tag, \omega, req\rangle \asymp \langle src_1, q, tag_1, \omega_1, k\rangle}{action}$
$\quad\quad \texttt{else if}\ mode = bsend\ \texttt{then}$
$\quad\quad\quad action\ \wedge\ \texttt{buffer}'.capacity = \texttt{buffer}.capacity - \texttt{size}(v\_)$
$\quad\quad \texttt{else}$ if no buffer is used then wait until the value is sent
$\quad\quad\quad \dfrac{\neg use\_buffer \Rightarrow (v\_ = \_)}{action}$

$has\_completed(req\_index, p) \doteq$ whether a request has completed
$\vee\ \exists \langle buf, src, cid, tag, v, pr, \top, \omega\rangle^{recv} = \texttt{reqs}_p[req\_index]$ the data v have arrived
$\vee\ \exists \langle dst, cid, tag, v\_, pr, \top, \omega\rangle^{mode} = \texttt{reqs}_p[req\_index] :$
$\quad \vee\ mode = bsend$ the data are buffered
$\quad \vee\ mode = rsend\ \wedge\ (use\_buffer\ \vee\ (v\_ = \_))$ the data is out
$\quad \vee\ mode = ssend\ \wedge$ there must exist a matching receive
$\quad\quad \exists q : \exists \langle buf_1, src_1, cid, tag_1, \_, pr_1, \top, \omega_1\rangle^{recv}_k \in \texttt{reqs}_q :$
$\quad\quad\quad \langle dst, p, tag, \omega, req\rangle \asymp \langle src_1, q, tag_1, \omega_1, k\rangle$

$\texttt{wait\_any}(count, req_{array}, index, status, p) \triangleq$ wait for any request in req_{array}
$\texttt{if}\ \forall i \in 0\,..\,count - 1 : req_{array}[i] = \texttt{REQUEST\_NULL}\ \vee\ \neg \texttt{reqs}_p[req_{array}[i]].active$
$\texttt{then}\ \texttt{mems}'_p[index] = \texttt{UNDEFINED}\ \wedge\ \texttt{mems}'_p[status] = empty\_status$
$\texttt{else}\ \dfrac{\exists\, i : has\_completed(req_{array}[i], p)}{\begin{array}{l}\texttt{mems}'_p[index] = \texttt{choose}\ i : has\_completed(req_{array}[i], p) \wedge \\ \texttt{mems}'_p[status] = get\_status(\texttt{reqs}_p[req_{array}[i]])\end{array}}$

Figure 5.4: Modeling point-to-point communications (III)

MPI calls at processes p_0, p_1 and p_2:

p_0	p_1	p_2
$\texttt{Issend}(v_1, dst = 1, cid = 5,$ $tag = 0, req = 0)$	$\texttt{Irecv}(b, src = 0, cid = 5,$ $tag = *, req = 0)$	$\texttt{Irecv}(b, src = *, cid = 5,$ $tag = *, req = 0)$
$\texttt{Irsend}(v_2, dst = 2, cid = 5,$ $tag = 0, req = 1)$	$\texttt{Wait}(req = 0)$	$\texttt{Wait}(req = 0)$
$\texttt{Wait}(req = 0)$		
$\texttt{Wait}(req = 1)$		

Execution states at processes p_0, p_1 and p_2:

step	$\texttt{reqs}_0$	$\texttt{reqs}_1$	$\texttt{reqs}_2$
1	$\langle 1, 5, 0, v_1, \bot, \top, \epsilon \rangle_0^{ss}$		
2	$\langle 1, 5, 0, v_1, \bot, \top, \epsilon \rangle_0^{ss}$	$\langle b, 0, 5, *, \_, \bot, \top, \epsilon \rangle$	
3	$\langle 1, 5, 0, v_1, \bot, \top, \epsilon \rangle_0^{ss}$	$\langle b, 0, 5, *, \_, \bot, \top, \epsilon \rangle_0^{rc}$	$\langle b, *, 5, *, \_, \bot, \top, \epsilon \rangle_0^{rc}$
4	$\langle 1, 5, 0, v_1, \bot, \top, \epsilon \rangle_0^{ss} \diamond$ $\langle 2, 5, 0, v_2, \bot, \top, \langle 2, 0 \rangle \rangle_1^{rs}$	$\langle b, 0, 5, *, \_, \bot, \top, \epsilon \rangle_0^{rc}$	$\langle b, *, 5, *, \_, \bot, \top, \langle 0, 1 \rangle \rangle_0^{rc}$
5	$\langle 1, 5, 0, \_, \bot, \top, \epsilon \rangle_0^{ss} \diamond$ $\langle 2, 5, 0, v_2, \bot, \top, \langle 2, 0 \rangle \rangle_1^{rs}$	$\langle b, 0, 5, *, v_1, \bot, \top, \epsilon \rangle_0^{rc}$	$\langle b, *, 5, *, \_, \bot, \top, \langle 0, 1 \rangle \rangle_0^{rc}$
6	$\langle 2, 5, 0, v_2, \bot, \top, \langle 2, 0 \rangle \rangle_1^{rs}$	$\langle b, 0, 5, *, v_1, \bot, \top, \epsilon \rangle_0^{rc}$	$\langle b, *, 5, *, \_, \bot, \top, \langle 0, 1 \rangle \rangle_0^{rc}$
7	$\langle 2, 5, 0, \_, \bot, \top, \langle 2, 0 \rangle \rangle_1^{rs}$	$\langle b, 0, 5, *, v_1, \bot, \top, \epsilon \rangle_0^{rc}$	$\langle b, *, 5, *, v_2, \bot, \top, \langle 0, 1 \rangle \rangle_0^{rc}$
8		$\langle b, 0, 5, *, v_1, \bot, \top, \epsilon \rangle_0^{rc}$	$\langle b, *, 5, *, v_2, \bot, \top, \langle 0, 1 \rangle \rangle_0^{rc}$
9		$\langle b, 0, 5, *, v_1, \bot, \top, \epsilon \rangle_0^{rc}$	
10			

The execution order:

$1 : \texttt{issend}(v_1, 1, 5, 0, p_0)$	$2 : \texttt{irecv}(b, 0, 5, *, p_1)$
$3 : \texttt{irecv}(b, *, 5, *, p_2)$	$4 : \texttt{irsend}(v_2, 2, 5, 0, p_0)$
$5 : \texttt{transfer}(p_0, p_1)$	$6 : \texttt{wait}(0, p_0)$
$7 : \texttt{transfer}(p_0, p_2)$	$8 : \texttt{wait}(1, p_0)$
$9 : \texttt{wait}(0, p_2)$	$10 : \texttt{wait}(0, p_1)$

Figure 5.5: A point-to-point communication program and one of its possible executions.
Process p_0 sends messages to p_1 and p_2 in synchronous send mode and ready send mode, respectively. The scheduler first forwards the message to p_1, then to p_2. A request is deallocated after the wait call on it. Superscripts ss, rs, and rc represent *ssend*, *rsend*, and *recv*, respectively. The execution follows from the semantics shown in Figures 5.2, 5.3, and 5.4.

in the request queue is 2.

MPI offers four send modes. A standard send may or may not buffer the outgoing message (represented by a global flag use_buffer). If buffer space is available, then it behaves the same as a send in the buffered mode; otherwise, it acts as a synchronous send. We show below the specification of `MPI_IBsend`. As *dtype* and *comm* are the references (pointers) to datatype and communicator objects, their values are obtained by $\texttt{datatypes}_p[dtype]$ and $\texttt{comms}_p[comm]$. Helper function `ibsend` creates a new send request, appends it to p's request queue, and puts the data in p's send buffer $\texttt{buffer}_p$. The request handle points to the last request in the queue.

$\texttt{MPI\_IBsend}(buf, count, dtype, dest, tag, comm, request, p) \triangleq$ top level definition
$\texttt{let } cm = \texttt{comms}_p[comm] \texttt{ in}$ the communicator
$\wedge\ \texttt{ibsend}(read\_data(\texttt{mems}_p, buf, count, \texttt{datatypes}_p[dtype]), cm.group[dest], cm.cid, tag, p)$
$\wedge\ \texttt{mems}'_p[request] = \texttt{size}(\texttt{reqs}_p)$ set the request handle

`MPI_Recv` is specified in a similar way. The MPI System Scheduler transfers values from a send request to its matching receive request. Relation $\rightleftharpoons$ defines the meaning of *matching*. Two cases are considered:

- The send is in ready mode. When a send request req_s is added into the queue, it is prematched to a receive request req_r such that the *prematch* field (abbreviated as ω) of req_s stores the tuple $\langle$*destination process, destination request index*$\rangle$, and req_r's *prematch* field stores the tuple $\langle$*source process, source request index*$\rangle$. req_s and req_r match iff these two tuples match.

- The send is in other modes. The send request and receive request are matched if relevant information (*e.g.* source, destination, context ID and tag) matches. The source and tag in the receive request may be `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, respectively.

It is the `transfer` rule (see Figure 5.3) that models message passing. Messages from the same source to the same destination must be matched in a FIFO order: only the first send request in the send queue and the first matching receive request in the receive queue will participate in the transferring. The FIFO requirement is enforced by the following predicate which indicates that there exist no prior send requests and prior receive requests that match.

$$\nexists\langle dst, cid, tag_1, v, pr_1, \top, \omega_1\rangle^{send}_m \in \Gamma^p_1 : \nexists\langle buf, src_2, cid, tag_2, \_, pr_2, \top, \omega_2\rangle^{recv}_n \in \Gamma^q_1 :$$
$$\vee\ \langle p, dst, tag_1, \omega_1, m\rangle \neq \langle src, q, tag_q, \omega_q, j\rangle\ \vee\ \langle p, dst, tag_p, \omega_p, i\rangle \neq \langle src_2, q, tag_2, \omega_2, n\rangle$$
$$\vee\ \langle p, dst, tag_1, \omega_1, m\rangle \neq \langle src_2, q, tag_2, \omega_2, n\rangle$$

When the transfer is done, the value field in the receive request req_j is filled with the incoming value v, and that in the send request req_i becomes _ to indicate that the value has been sent out. If the request is not persistent and not live (*i.e.* the corresponding `MPI_Wait` has been called), then it will be removed from the request queue.

The `MPI_Wait` call returns when the operation associated with request *request* is complete. If *request* is a null handle, then an empty status is returned; otherwise, the helper function `wait_one` is invoked to pick the appropriate wait function according to the request's type.

Let us look closer at the definition of `recv_wait` (see Figure 5.3). First of all, after the call, the request is not "live" any more; thus, the *live* flag becomes false. When the call is made with an inactive request, it returns immediately with an empty status. If the request is persistent and not marked for deallocation, then the request becomes inactive after the call; otherwise, it is removed from the request queue and the corresponding request handle is set to `MPI_REQUEST_NULL`.

If the request has been marked for cancellation, then the call completes without writing the data into memory. If the source process is a null process, then the call returns immediately with a null status where source = `MPI_PROC_NULL`, tag = `MPI_ANY_TAG`, and count = 0. Finally, if the value has been received (*i.e.* $v_{\_} \neq \_$), then the value v is written to process p's local memory and the status object is updated accordingly.

The completion of a request is modeled by the *has_completed* predicate. A receive request completes when the value has been received. A send request in the buffer mode completes when the value has been buffered or transferred. This function is used to implement multiple communication primitives. For instance, `MPI_Waitany` blocks until one of the requests completes.

5.2.5 Collective Communication

Collective communications are based on a protocol shown in Figure 5.6. An example illustrating the "execution" of these primitives is shown in Figure 5.7.

Data Structures
rendezvous for a communication :
$\langle status : [\texttt{int} \to \{\text{“e”}, \text{“l”}\}],\ sdata,\ data : [\texttt{int} \to value]\rangle$ `array`

process p joins the communication and stores v_s and v in the rendevous
$\texttt{syn}_{\texttt{put}}(cid, v_s, v, p) \doteq$
`if` $cid \notin$ `DOM rend then` $\texttt{rend}'[cid] = \langle [p \mapsto \text{“e”}], v_s, [p \mapsto v]\rangle$
`else if` $\forall slot \in \texttt{rend}[cid] : p \in$ `DOM` $(slot.status)$ `then`
$\texttt{rend}'[cid] = \texttt{rend}[cid] \diamond \langle [p \mapsto \text{“e”}], v_s, [p \mapsto v]\rangle$
`else`
$$\frac{\texttt{rend}[cid] = \Gamma_1 \diamond \langle \Psi, \alpha, S_v\rangle \diamond \Gamma_2 \ \wedge\ p \notin \texttt{DOM}\ \Psi \ \wedge\ \forall slot \in \Gamma_1 : p \in \texttt{DOM}\ (slot.status)}{\texttt{rend}'[cid] = \Gamma_1 \diamond \langle \Psi \uplus (p, \text{“e”}),\ v_s,\ S_v \uplus (p, v)\rangle \diamond \Gamma_2}$$

$\texttt{syn}_{\texttt{init}}(cid, p) \doteq \texttt{syn\_put}(cid, \epsilon, \epsilon, p)$ no data are stored

$\texttt{syn}_{\texttt{write}}(cid, v, p) \doteq \texttt{syn\_put}(cid, \epsilon, v, p)$ no shared data are stored

$\texttt{syn}_{\texttt{wait}}(cid, p) \doteq$ process p leaves the synchronizaing communication
$$\frac{\begin{array}{l}\texttt{rend}[cid] = \Gamma_1 \diamond \langle \Psi \uplus (p, \text{“e”}), v_s, S_v\rangle \diamond \Gamma_2 \wedge \\ \forall k \in \texttt{comms}_p[cid].group : k \in \texttt{DOM}\ \Psi \ \wedge\ \forall slot \in \Gamma_1 : slot.status[p] \neq \text{“e”}\end{array}}{\begin{array}{ll}\texttt{rend}'[cid] = & \texttt{if}\ \forall k \in \texttt{comms}_p[cid].group : k \in \texttt{DOM}\ \Psi \wedge \Psi[k] = \text{“l”}\ \texttt{then}\ \Gamma_1 \diamond \Gamma_2 \\ & \texttt{else}\ \Gamma_1 \diamond \langle \Psi \uplus (p, \text{“l”}),\ v_s,\ S_v\rangle \diamond \Gamma_2\end{array}}$$

$\texttt{asyn}_{\texttt{wait}}(cid, p) \doteq$ process p leaves the nonsynchronizaing communication
$$\frac{\texttt{rend}[cid] = \Gamma_1 \diamond \langle \Psi \uplus (p, \text{“e”}), v_s, S_v\rangle \diamond \Gamma_2 \ \wedge\ \forall slot \in \Gamma_1 : slot.status[p] \neq \text{“e”}}{\begin{array}{ll}\texttt{rend}'[cid] = & \texttt{if}\ \forall k \in \texttt{comms}_p[cid].group : k \in \texttt{DOM}\ \Psi \wedge \Psi[k] = \text{“l”}\ \texttt{then}\ \Gamma_1 \diamond \Gamma_2 \\ & \texttt{else}\ \Gamma_1 \diamond \langle \Psi \uplus (p, \text{“l”}),\ v_s,\ S_v\rangle \diamond \Gamma_2\end{array}}$$

Figure 5.6: The basic protocol for collective communications

p_0	p_1	p_2
$\texttt{syn}_{\texttt{put}}(cid = 0, sdata = v_s, data = v_0)$	$\texttt{syn}_{\texttt{init}}(cid = 0)$	$\texttt{syn}_{\texttt{write}}(cid = 0, data = v_2)$
$\texttt{asyn}_{\texttt{wait}}(cid = 0)$	$\texttt{syn}_{\texttt{wait}}(cid = 0)$	$\texttt{syn}_{\texttt{wait}}(cid = 0)$
$\texttt{syn}_{\texttt{init}}(cid = 0)$		

step	*event*	rend[0]
1	$\texttt{syn}_{\texttt{put}}(0, v_s, v_0, p_0)$	$\langle [0 \mapsto \text{“e”}], v_s, [0 \mapsto v_0]\rangle$
2	$\texttt{syn}_{\texttt{init}}(0, p_1)$	$\langle [0 \mapsto \text{“e”}, 1 \mapsto \text{“e”}], v_s, [0 \mapsto v_0]\rangle$
3	$\texttt{asyn}_{\texttt{wait}}(0, p_0)$	$\langle [0 \mapsto \text{“l”}, 1 \mapsto \text{“e”}], v_s, [0 \mapsto v_0]\rangle$
4	$\texttt{syn}_{\texttt{init}}(0, p_0)$	$\langle [0 \mapsto \text{“l”}, 1 \mapsto \text{“e”}], v_s, [0 \mapsto v_0]\rangle \diamond \langle [0 \mapsto \text{“e”}], \epsilon, \epsilon\rangle$
5	$\texttt{syn}_{\texttt{write}}(0, v_2, p_2)$	$\langle [0 \mapsto \text{“l”}, 1 \mapsto \text{“e”}, 2 \mapsto \text{“e”}], v_s,$ $[0 \mapsto v_0, 2 \mapsto v_2]\rangle \diamond \langle [0 \mapsto \text{“e”}], \epsilon, \epsilon\rangle$
6	$\texttt{syn}_{\texttt{wait}}(0, p_2)$	$\langle [0 \mapsto \text{“l”}, 1 \mapsto \text{“e”}, 2 \mapsto \text{“l”}], v_s,$ $[0 \mapsto v_0, 2 \mapsto v_2]\rangle \diamond \langle [0 \mapsto \text{“e”}], \epsilon, \epsilon\rangle$
7	$\texttt{syn}_{\texttt{wait}}(0, p_1)$	$\langle [0 \mapsto \text{“e”}], \epsilon, \epsilon\rangle$

Figure 5.7: An example using the collective protocol (with $cid = 0$).

Processes participating in a collective communication coordinate with each other through rendezvous objects. Each communicator with context id *cid* is associated with object `rend`[*cid*], which consists of a sequence of communication slots. In each slot, the *status* field records the status of each process: "e" ("entered") or "l" ("left"); the *shared_data* field stores the data shared among all processes; and *data* stores the data sent by each process. We use notation Ψ to represent *status*'s content.

Many collective communications are synchronizing, while the rest (such as `MPI_Bcast`) can be either synchronizing or nonsynchronizing. A collective primitive is implemented by a loose synchronization protocol: in the first "init" phase $\texttt{syn}_{put}$, process p checks whether there exists a slot such that p has not participated in. A negative answer means that p is initializing a new communication; thus, p creates a new slot, sets its status to be 'e', and stores its value v in this slot. If there are multiple slots that p has not joined into (*i.e.* p is not in the domains of these slots), then p registers itself in the first one. This phase is the same for both synchronizing and nonsynchronizing communications. Rules $\texttt{syn}_{init}$ and $\texttt{syn}_{write}$ are the simplified cases of $\texttt{syn}_{put}$.

After the "init" phase, process p proceeds to its "wait" phase. Among all the slots, p locates the first one it has entered but not left. If the communication is synchronizing, then p has to wait until all other processes finish their "init" phases; otherwise, it proceeds. If p is the last process that leaves, then the entire collective communication is over and the communication slot can be removed from the queue; otherwise, p just updates its status to 'left'.

These protocols are used to specify collective communication primitives. For example, `MPI_Bcast` is implemented by two transitions: $\texttt{MPI\_Bcast}_{init}$ and $\texttt{MPI\_Bcast}_{wait}$. The root first sends its data to the rendezvous in $\texttt{MPI\_Bcast}_{init}$, then it calls either the $\texttt{asyn}_{wait}$ rule or the $\texttt{syn}_{wait}$ rule depending on whether the primitive is synchronizing. In the synchronizing case, the wait returns immediately without waiting for the completion of other processes. On the other hand, a nonroot process always calls the $\texttt{syn}_{wait}$ rule because it must wait for the data from the root to "reach" the rendezvous.

$$\texttt{bcast}_{\texttt{init}}(buf, v, root, comm, p) \triangleq \text{ the root broadcasts data to processes}$$
$$(comm.group[root] = p)\ ?\ \texttt{syn}_{\texttt{put}}(comm.cid, v, \epsilon, p)\ :\ \texttt{syn}_{\texttt{init}}(comm.cid, p)$$

$$\begin{array}{l}\texttt{bcast}_{\texttt{wait}}(buf, v, root, comm, p) \triangleq \\ \quad \texttt{if } comm.group[root] = p \texttt{ then} \\ \qquad \textit{need\_syn} \text{ is a global flag whose value is set by the user} \\ \qquad \textit{need\_syn} \ ? \ \texttt{syn}_{\texttt{wait}}(comm.cid, p) \ : \ \texttt{asyn}_{\texttt{wait}}(comm.cid, p) \\ \quad \texttt{else } \texttt{syn}_{\texttt{wait}}(comm.cid, p) \ \wedge \ \texttt{mems}'_p[buf] = \texttt{rend}_p[comm.cid].sdata\end{array}$$

MPI-2 extends many MPI-1 collective primitives to intercommunicators. An intercommunicator contains a local group and a remote group. To model this, we replace $\texttt{comms}_p[cid].group$ with $\texttt{comms}_p[cid].group \cup \texttt{comms}_p[cid].remote\_group$ in the rules shown in Figure 5.6.

5.3 Evaluation and Program Verification

How to ensure that our formalization is faithful with the English description? To attack this problem, we rely heavily on testing in our formal framework. We provide comprehensive unit tests and a rich set of short litmus tests of the specification. Generally it suffices to test local, collective, and asynchronous MPI primitives on one, two, and three processes, respectively. These test cases, which include many simple examples in the MPI reference, are hand-written directly in TLA+ and modeled checked using TLC. Although typically, test cases are of only dozens of lines of code, they are able to expose most of the formalization errors.

Another set of test cases are built to verify the *self-consistency* of the specification modeled after [116] where self-consistency rules are used as performance guidelines. It is possible to relate aspects of MPI to each other, *e.g.* explain certain MPI primitives in terms of other MPI primitives.

For example, a message of size $k \times n$ can be divided into k submessages sent separately; a collective primitive can be replaced by the combination of several point-to-point or one-sided primitives. We introduce relation $\texttt{MPI\_}A \simeq \texttt{MPI\_}B$ to indicate that A and B have the same functionality. This relation helps us to design test cases to test the specification of some MPI primitives. To verify these relations, we design test cases with concrete inputs and run the TLC to make sure that the same outputs are obtained. We plan to prove them formally in the Isabelle/TLA tool.

$$\begin{array}{c}\texttt{MPI\_A(k} \times \texttt{n)} \simeq (\texttt{MPI\_A(n)}_1; \ \cdots \ ; \ \texttt{MPI\_A(n)}_k) \\ \texttt{MPI\_A(k} \times \texttt{n)} \simeq (\texttt{MPI\_A(k)}_1; \ \cdots \ ; \ \texttt{MPI\_A(k)}_n) \\ \texttt{MPI\_Bcast(n)} \simeq (\texttt{MPI\_Send(n)}; \ \cdots \ ; \ \texttt{MPI\_Send(n)}) \\ \texttt{MPI\_Gather(n)} \simeq (\texttt{MPI\_Recv(n/p)}_1; \ \cdots \ ; \ \texttt{MPI\_Recv(n/p)}_p)\end{array}$$

It should be noted that we have not modeled all the details of the MPI standard, which include:

- *Implementation details.* To the greatest extent, we have avoided asserting implementation-specific details in our formal semantics. One obvious example is the **info** object is ignored.
- *Physical Hardware.* The underlying physical hardware is invisible in our model. Thus, we do not model hardware related primitives like `MPI_Cart_map`.
- *Profiling Interface.* The MPI profiling interface is to permit the implementation of profiling tools. It is irrelevant to the semantics of MPI primitives.

5.3.1 Issues Raised by Modeling

While creating the model, we become aware of some specific issues that have not been discussed in the standard. For example, `MPI_Probe` on process j becomes enabled when there is a matching request posted on process j; `MPI_Cancel` attempts to cancel the corresponding communication. The standard says the message may still complete, and it is up to the user to program appropriately. In this context, we identify some specific issues: (i) There are numerous ways that `MPI_Probe` and `MPI_Cancel` can interact, resulting in an undefined system state. In particular, any time a message is probed successfully, it is not specified whether it is still possible for the message to be canceled or if the message must at that point be delivered. (ii) `MPI_Cancel` also creates an undefined state when used with ready mode send. Consider an execution trace: `MPI_Irecv`; `MPI_Irsend`; `MPI_Cancel`; $\cdots$. If the ready send is successful, can the receive still be canceled? and (iii) Continuing with Cancel, what happens if the null request is canceled?

5.4 An Application: Soundness Proof

The main problem of model checking MPI programs is the state space explosion problem. This problem may be mitigated by using partial order reduction techniques. A sound partial order reduction guarantees that if there is a property violation in the full state space, that violation will be discovered by the model checker while enumerating a subset of the state space.

The Verification Group in Utah has developed several partial order reduction (DPOR) algorithms [87, 88, 118] to model check MPI programs. For instance, the

ISP checker [118, 121] exploits the out-of-order completion semantics of MPI by issuing MPI calls according to match-sets which are ample 'big-step' moves. The core of a DPOR algorithm is to base on an dependence analysis to determine when it is safe to execute only a subset of the enabled calls. Such dependence information is computed based on the semantics of MPI calls. In this section, we show how to justify the dependence definition in these DPOR algorithms.

The goal is to prove the soundness of the *complete-before* relation $\prec$ defined in [118]. Relation $\prec$ specifies the order enforced on the completion of MPI calls. An MPI immediate send $S_{i,j}(k, \langle i, j\rangle, \ldots)$, where k is the process targeted, i, j is the request handle used to track the processes of this send, completes when it matches a receive (*e.g.* by the MPI System Scheduler). An MPI immediate receive $S_{i,j}(k, \langle i, j\rangle, \ldots)$, where k is the process from which the message is sent ($k = *$ means a 'wildcard receive'), completes when it receives the message. A barrier operation $B_{i,j}$ completes when all participants exit the synchronization. A wait operation $W_{i,j}\langle i, j\rangle$ completes when the corresponding send (receive) operation completes and the data has been sent out (copied into the target process's memory).

The formal definition of the completes-before relation is given as eight rules:

(Css-kk) $\forall i, j_1, j_2, k:\ j_1 < j_2 \Rightarrow S_{i,j_1}(k, \ldots) \prec S_{i,j_2}(k, \ldots)$
(Crr-kk) $\forall i, j_1, j_2, k:\ j_1 < j_2 \Rightarrow R_{i,j_1}(k, \ldots) \prec R_{i,j_2}(k, \ldots)$
(Crr-*k) $\forall i, j_1, j_2, k:\ j_1 < j_2 \Rightarrow R_{i,j_1}(*, \ldots) \prec R_{i,j_2}(k, \ldots)$
(Crr-**) $\forall i, j_1, j_2, k:\ j_1 < j_2 \Rightarrow R_{i,j_1}(*, \ldots) \prec R_{i,j_2}(*, \ldots)$
(Csw) $\forall i, j_1, j_2, k:\ j_1 < j_2 \Rightarrow S_{i,j_1}(k, \langle i, j_1\rangle) \prec W_{i,j_2}(\langle i, j_1\rangle)$
(Crw) $\forall i, j_1, j_2, k:\ j_1 < j_2 \Rightarrow R_{i,j_1}(k, \langle i, j_1\rangle) \prec W_{i,j_2}(\langle i, j_1\rangle)$
(Cb) $\forall i, j_1, j_2, k:\ j_1 < j_2 \Rightarrow B_{i,j_1} \prec any_{i,j_2}(\ldots)$
(Cw) $\forall i, j_1, j_2, k:\ j_1 < j_2 \Rightarrow W_{i,j_1}(\ldots) \prec any_{i,j_2}(\ldots)$

Now, we proceed to prove the correctness of these rules with respect to our formal semantics. As in [118], we abstract away such fields as communicator ID, tag, prematch, value, and flags. First of all, rule `Csw` and rule `Crw` are valid because a blocking send or receive operation is modeled by a nonblocking operation followed by a wait operation. As indicated in the semantics, a nonblocking operation sets the active flag of the request, and the corresponding wait operation can return only if this flag is set. Hence, these two operations cannot execute out of order.

5.4.1 Send and Receive

Consider the `Css-kk` rule, which specifies the order of two immediate sends from process i to process k. Assume that the request queue at process i contains two active send requests $S_{i,j_1}(k,\ldots)$ and $S_{i,j_2}(k,\ldots)$:

$$\langle k,\ldots\rangle_{j_1}^{send} \diamond \langle k,\ldots\rangle_{j_2}^{send}$$

Suppose for contradiction that request j_2 may complete before request j_1. In order for j_2 to complete, there must exist a receive request $\langle buf,i,\ldots\rangle_n^{recv}$ at process k that matches this send request, and the following condition specified in the `transfer` rule must hold (note that the first request $\langle k,\ldots\rangle_{j_1}^{send}$ is in Γ_1^i):

$$\nexists\langle k,\ldots\rangle_m^{send} \in \Gamma_1^i : \langle i,k,\ldots,m\rangle \eqsim \langle i,k,\ldots,n\rangle$$

However, if m equals to j_1, then this condition is false immediately because request j_1 matches the receive request. This contradiction implies the correctness of rule `Css-kk`. Rule `Crr-kk` can be proved in a similar way.

Let us look at rule `Crr-*k` and rule `Crr-**`, where the first receive is a wildcard receive. Assume that the request queue at process i contains two active receive requests $R_{i,j_1}(*,\ldots)$ and $R_{i,j_2}(k_*,\ldots)$. In the second receive, either $k_* = k$ (*i.e.* the source is process k) or $k_* = *$ (*i.e.* it is a wildcard receive):

$$\langle buf_1,*,\ldots\rangle_{j_1}^{recv} \diamond \langle buf_2,k_*,\ldots\rangle_{j_2}^{recv}$$

If request j_2 completes before request j_1, then there must exist a send request $\langle i,\ldots\rangle_n^{send}$ at a process p (which may be k) that matches this receive request, and the FIFO condition in the `transfer` rule must hold. In other words, we have

$$\begin{array}{l}\langle p,i,\ldots,n\rangle \eqsim \langle k_*,i,\ldots,j_2\rangle \wedge \\ \nexists\langle buf,q,\ldots\rangle_m^{recv} \in \Gamma_1^i : \langle p,i,\ldots,n\rangle \eqsim \langle q,i,\ldots,m\rangle\end{array}$$

Let m equal to j_1, then the second condition requires us to prove that $\langle p,i,\ldots,n\rangle \eqsim \langle *,i,\ldots,j_1\rangle$ is false. Using the definition of $\eqsim$ (where the prematch fields are empty),

$$\begin{array}{l}(\langle p,dst,\ldots,k_p\rangle \eqsim \langle src,q,\ldots,k_q\rangle) \doteq \\ \quad q = dst \ \wedge \ src \in \{p,*\} \ \text{the source and target must match}\end{array}$$

after simplification we have

$$k_* \in \{p, *\} \ \wedge \ \neg(* \in \{p, *\}),$$

which is obviously false. Thus, request j_2 cannot complete before j_1, which implies the correctness of these two rules.

On the other hand, the rule $\forall i, j_1, j_2, k: \ j_1 < j_2 \Rightarrow R_{i,j_1}(k, \ldots) \ \prec R_{i,j_2}(*, \ldots)$ is invalid. If we perform the same contradiction proof as shown above, then finally, we will get a predicate not leading to a contradiction: $* \in \{p, *\} \ \wedge \ \neg(k \in \{p, *\})$. This predicate is true when $k \neq p$, *i.e.* a process other than k sends the message.

5.4.2 Barrier

Rule `Cb` specifies that any MPI call starting after a barrier operation will complete after the barrier. This rule is valid because the barrier function has blocking semantics: the "wait" phase of a barrier operation B_{i,j_1} at process i will be blocked until i leaves the synchronizing communication. Thus, only after B_{i,j_1} returns will a subsequent MPI call any_{i,j_2} start and then complete. Similarly, rule `Cw` is valid because `Wait` also has blocking semantics.

On the other hand, the rule $\{\forall i, j_1, j_2, k: \ j_1 < j_2 \Rightarrow any_{i,j_1}(\ldots) \prec B_{i,j_2}\}$ is invalid. This can be explained easily with the formal semantics. Recall that B_{i,j_2} is implemented as B_{i,j_2}_init followed by B_{i,j_2}_wait. Suppose any_{i,j_1} is a send operation, as the barrier and send operate on different MPI objects (*i.e.* `rend` and `reqs` respectively), the B_{i,j_2}_wait needs not to wait for the completion of the send. Hence, the following sequence is possible, implying that $send_{i,j_1}(\ldots) \prec B_{i,j_2}$ is false.

$$send_{i,j_1} \text{ starts} < B_{i,j_2}\_init < B_{i,j_2}\_wait < send_{i,j_1} \text{ completes}$$

CHAPTER 6

VERIFYING PARALLEL PROGRAMS: CUDA KERNELS

There is an explosive growth of interest in *Graphical Processing Units (GPU)* for speeding up computations occurring at all application scales [30, 53]. GPUs are used in iPhones for video processing, and on desktop computers for extracting features from medical images. All future supercomputers will employ GPUs. The main attraction of GPUs is that *when properly programmed*, they can yield anywhere from 20 to 100 times more performance compared to standard CPU based multicores. Unfortunately, obtaining this performance requires heroic acts of programming; to name a few: (i) one must keep all the fine-grained GPU threads busy; (ii) one must ensure coalesced [53] data movements from the *global memory* (that is accessed commonly by CPUs and GPUs) to the *shared memory* (that is accessed commonly by the GPU threads); and (iii) one must minimize bank conflicts when the GPU threads step through the shared memory. Data races and incorrect barrier placements are frequently introduced during CUDA programming. Few tools are available to verify CUDA programs. The emulator that comes with GPUs assumes concrete inputs and executes only a miniscule fraction of all possible schedules. Bugs often escape, either crashing or deadlocking the GPU hardware, often requiring a hardware reboot.

GPU kernels are comprised of light-weight threads. Their Single Instruction Multiple Data (SIMD) organization bears little resemblance to thread programs written in C/Java with their heterogeneous and heavy-weight threads, and use of synchronization primitives such as locks/monitors. This requires a fundamentally new approach for analyzing CUDA kernels. This chapter's main result is that while Satisfiability Modulo Theories (SMT [107]) techniques are a natural choice for analyzing CUDA kernels, many innovations are essential before such analysis can scale. Efficient techniques for encoding concurrent interleavings and analyzing barrier placement must be developed. One must try to exploit the "mostly

deterministic" style of programming and avoiding interleaving generation. It is efficient to divide up the analysis over barrier intervals. Finally, techniques for efficiently handling loops (rather than simply unrolling them) must be developed. We now begin with a few CUDA examples and elaborate our innovations.

6.1 Overview

6.1.1 Illustration of CUDA

A CUDA kernel is launched as a 1D or 2D *grid* of *thread blocks*. The total size of a 2D grid is `gridDim.x` × `gridDim.y`. The coordinates of a (thread) block are ⟨`blockIdx.x`, `blockIdx.y`⟩. The dimensions of each thread block are `blockDim.x` and `blockDim.y` (assuming 1D or 2D blocks in this thesis). Each block contains `blockDim.x` × `blockDim.y` threads, each with coordinates ⟨`threadIdx.x`, `threadIdx.y`⟩. These threads can share information via *shared memory*, and synchronize via *barriers* (__syncthreads()). Threads belonging to distinct blocks must use the much slower *global memory* to communicate. *This chapter focuses on shared memory races.* Consider a simple example of a CUDA kernel to add `b` to all the elements of a shared array `a` of size `N`:

```
void __global__ kernel (int *a, int b) {
int idx = blockIdx.x * blockDim.x + threadIdx.x;
if (idx < N) a[idx] = a[idx] + b;}
```

Basically, each thread accesses a different array location and adds `b` to it in parallel; *there are no data races.* Now imagine the programmer wanting to update each array location with `b` added to the previous array location. The programmer may not simply change the last line to `a[idx] = a[idx-1] + b;` because there will be data races between adjacent threads. The programmer may, however, change the code to the following:

```
void __global__ kernel1 (int *a, int b) {
__shared__ int temp[N];
int idx = blockIdx.x * blockDim.x + threadIdx.x;
if (idx < N) temp[idx] = a[idx-1] + b;
__syncthreads(); // A barrier
if (idx < N) a[idx] = temp[idx];}
```

What if the barrier is removed from this code? Obviously, the accesses of $a[idx]$ and $a[idx-1]$ by different threads may cause a race. This can be detected by examining the symbolic models of two threads as following, where private variables

in a thread are superscripted by the thread id, *bid* and *bdim* are the short hands for `blockIdx` and `blockDim`, respectively. Threads t_1 and t_2 are assumed to be in the same block. Formally, a race occurs if predicate $t_1.x \neq t_2.x \wedge id^{t_1} < N \wedge id^{t_2} < N \wedge idx^{t_1} - 1 = idx^{t_2}$ holds. As all variables have symbolic values, we can consult with a constraint solver to determine whether this predicate is satisfiable. If so, then the solver would return a concrete counter example. If the barrier is present, then we only need to check whether the writes to $a[idx^{t_1}]$ and $a[idx^{t_2}]$ conflict. Since $t_1.x \neq t_2.x$ implies $idx^{t_1} \neq idx^{t_2}$ for $t_1.x < bdim.x$ and $t_2.x < bdim.x$, these two writes will not result in a race.

thread t_1	thread t_2
$idx^{t_1} = bid.x * bdim.x + t_1.x$	$idx^{t_2} = bid.x * bdim.x + t_2.x$
$if\ (idx^{t_1} < N)$ read $a[idx^{t_1} - 1]$	$if\ (idx^{t_2} < N)$ read $a[idx^{t_2} - 1]$
$if\ (idx^{t_1} < N)$ write $a[idx^{t_1}]$	$if\ (idx^{t_2} < N)$ write $a[idx^{t_2}]$

As another example, the `scalarProdGPU` (Figure 6.1) kernel computes the scalar product of `vN` pairs of vectors with `eN` elements in each vector (both sequential and CUDA parallel versions are shown). This kernel coalesces global memory accesses, minimizes bank conflicts, avoids redundant barriers, and reduces serial penalties through tree summation. Without such hand-crafting steps, kernels such as this will perform poorly. In this chapter, we present our tool, Prover for User GPU Functions (PUG, see Figure 6.2), which helps detectbugs introduced during kernel design.

6.1.2 Internal Architecture of PUG

PUG takes a kernel program written in C (called Kernel C) as input. It first uses the Rose Compiler [98] to parse the kernel and generates an immediate format, then produces an SMT expression according to the configuration information supplied (*e.g.* the properties to be checked or the number of threads). We consider only two threads with symbolic identifiers (IDs) for race and synchronization checking. Users must specify the number of threads for assertion (user-defined property) checking. The PUG generated SMT expressions are processed by an SMT solver (currently Yices [124]) for satisfiability checking. If the expression is satisfiable, the solver will return a concrete counter-example; otherwise, the kernel is deemed free of the bugs targeted by our analysis.

```
void scalarProdSeq // Sequential version
(float *d_C, float *d_A, float *d_B, int vN, int eN) {
1:  for(int vec = 0; vec < vN; vec++){
2:    int vBase = eN * vec; int vEnd  = vBase + eN;
3:    double sum = 0;
4:    for(int pos = vBase; pos < vEnd; pos++)
5:      sum += d_A[pos] * d_B[pos];
6:    d_C[vec] = (float)sum;
7:  }}

// Parallel version: Nvidia CUDAZone site
__global__ void scalarProdGPU (float *d_C, float *d_A,
  float *d_B, int vN, int eN) {
1:  __shared__ float acc[ACC_N];
2:
3:  for(int vec = blockIdx.x; vec < vN; vec += gridDim.x) {
4:    int vBase = eN * vec; int vEnd  = vBase + eN;
5:
6:    for(int i = threadIdx.x; i < ACC_N; i += blockDim.x){
7:     float sum = 0;
8:     for(int pos = vBase + i; pos < vEnd; pos += ACC_N)
9:       sum += d_A[pos] * d_B[pos];
10:     acc[i] = sum;
11:   }
12:
13:   for(int stride = ACC_N / 2; stride > 0; stride >>= 1) {
14:     __syncthreads();
15:     for(int i = threadIdx.x; i < stride; i += blockDim.x)
16:       acc[i] += acc[stride + i];
17:   }
18:
19:   if(threadIdx.x == 0) d_C[vec] = acc[0];
20:  }}
```

Figure 6.1: Scalar product: sequential and CUDA parallel versions.

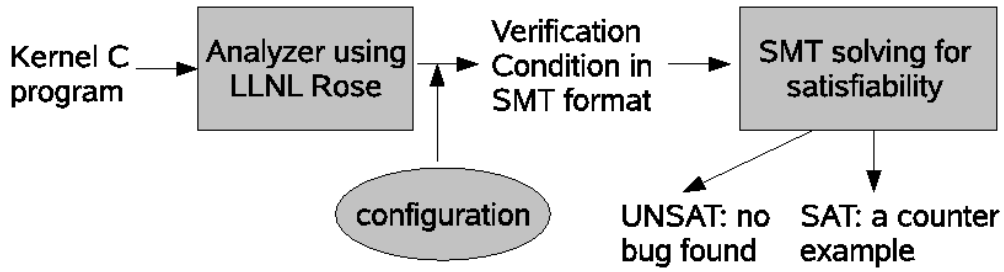


Figure 6.2: The internal architecture of PUG.

6.1.3 Organization

We now list some of our novel contributions, each of which is later elaborated in its own section.

• PUG employs a C front-end based on the LLNL Rose [98] framework (with customized extensions). It handles many CUDA C features including: (i) arrays and records, (ii) loops, conditional statements, and function calls, (iii) variable aliases due to pointer expressions, and (iv) lexical scopes. Many features such as heap allocation and recursive calls are not allowed in CUDA, simplifying our translation. See Section 6.2.

• We contribute a novel approach to capture all possible interleavings between CUDA threads as compact SMT formulae. In practice, working with this SMT representation is far more efficient than explicitly enumerating all schedules. See Section 6.3.

• We propose a way to model the semantics of barriers exactly. We generate SMT formulae that help verify that despite the presence of branches and loops, all barriers are well synchronized. See Section 6.4.

• While we have the ability to model all possible concurrent interleavings, *it is preferable to avoid resorting to this approach whenever possible.* Our observation that enables this optimization is based on the fact that in many cases, the existence of races between a given pair of variables is predicated on the existence of conflicts on other variables. The existence of conflicts can be checked over just one canonical interleaving – say the one that simply runs one thread till it blocks and then switching over to another. This helps dramatically improve the overall efficiency. We propose a way to further scale up this approach by analyzing one *barrier interval* (the portion before and after `__syncthreads()`) at a time. This *divide-and-conquer* approach also helps boost efficiency. See Section 6.5.

• The translation of loops can become extremely involved – especially if the loops are nested and they employ nonlinear strides. Our multipronged attack is as follows: (i) we normalize loops through program transformation into a unit-stride loop; (ii) we over-approximate loop computations; and (iii) we can automatically discover *compensating invariants* that compensate for nonlinear loop strides frequently found in practice. See Section 6.6.

• For many kernels, an SMT tool may generate a false alarm (false bug report) when it cannot determine how the kernel formal parameters are constrained by the

main program (caller). For example, PUG assures that the matrix multiplication kernel in the CUDA Programming Guide [24] works only when the size of matrix B is greater or equal to the block size. PUG is able to reveal such undocumented assumptions.

• We have obtained very encouraging results using PUG on real examples. As one example of its multiple uses, with respect to `scalarProdGPU`, we could obtain many valuable analysis results using PUG: (i) One may not remove the barrier on line 14 (it will result in a data race), but this single barrier suffices to remove all races with respect to the variables d_A, d_B, d_C, and acc. (ii) It is formally guaranteed that no bank conflicts (by different threads) occur in this example for all possible values of vN, eN, and ACC_N; (iii) Analysis by PUG helped us confirm the assumption that ACC_N must be a power of two; (iv) We could establish the equivalence of this kernel to `scalarProdSeq` for small instances of the problem parameters.

• We have also encountered examples where some kernels have benign races; *i.e.* they are still functionally correct. PUG has caught some serious (but nonobvious) bugs in beginner examples. It has also handled many large examples from the CUDA SDK site. All these examples and PUG itself are freely downloadable [60]. See Section 6.7.

This section describes the encoding of serial constructs; it gives the formal semantics of a kernel assuming no concurrency. Concurrency is handled in the next section.

The main syntax of Kernel C is given in Figure 6.3, and are illustrated by the kernel examples given so far. The notation $\langle term \rangle_{separator}$ (used in *fun_decl*, *block*, *etc.*) denotes a sequence of *term*'s separated by *separator*. Expression *exp* represents usual C expressions including assignments. Identifiers id_f and id_v represent names of functions and variables, respectively. Shared and global variables reside in the GPU and the CPU, respectively. A variable declared without modifier is local to each thread.

6.2 SMT-Encoding Sequential Constructs

We now present the encoding of sequential program structures. The encoding of concurrency will be presented in the next section.

prog	$::= \langle var\_decl \mid fun\_decl \rangle_;$	program
var_decl	$::= [md_v]\ ty\ id_v[= exp]$	variable
fun_decl	$::= ty\ id_f(\langle ty\ id_v \rangle_,) = block$	function
block	$::= \{\langle stmt \rangle_;\}$	basic block
stmt	::= `if` *exp block* [`else` *block*]	conditional
	| `for`(*exp*; *exp*; *exp*) *block*	loop
	| *block*	
	| *var_decl*	
	| *exp*	expression
	$\mid id_f(\langle exp \rangle_,)$	function call
ty	`int` | *ty* * | *ty*[] |	type
md_v	`shared` | `global`	modifier

Figure 6.3: Summary syntax of Kernel C

6.2.1 Basic Statements

Our encoding assigns SSA indexes to variables. Specifically, the following translation function Γ constructs a logical formula from single statements and expressions, where `next` and `cur` return the next and the current SSA indices of a variable, respectively, and $v \uplus ([i] \mapsto x)$ denotes the update of array v by setting the element at i to x. We also give below a simple example of applying Γ.

$$
\begin{array}{lcl}
\Gamma(e_1\ \texttt{op}\ e_2) & \doteq & \Gamma(e_1)\ \texttt{op}\ \Gamma(e_2) \\
\Gamma(v := e) & \doteq & v_{\texttt{next}(v)} = \Gamma(e) \\
\Gamma(v[e_1] := e_2) & \doteq & v_{\texttt{next}(v)} = v_{\texttt{cur}(v)}([\Gamma(e_1)] \mapsto \Gamma(e_2)) \\
\Gamma(v) & \doteq & v_{\texttt{cur}(v)}
\end{array}
$$

```
int k = 0;
int a[3];
int i = a[1] + k;
a[0] = i * k;
i++;
```

$$
\xrightarrow{\Gamma}\quad
\begin{array}{l}
k_1 = 0 \wedge \\
i_1 = a_0[1] + k_1 \wedge \\
a_1 = a_0([0] \mapsto i_1 * k_1) \wedge \\
i_2 = i_1 + 1
\end{array}
$$

6.2.2 Branches

The SSA indices of the variables updated in the two clauses of a conditional statement "if c blk_1 else blk_2" should be synchronized so that subsequent statements have a consistent view of their values. The following example gives an illustration: $i_1 = i_0$ is added into the first clause so that later on, i_0 is invisible and only variable i_1 will be referred. Here, notation `ite` stands for "`if then else`".

$$
\begin{array}{l}
\texttt{if } i > 0 \{ \\
\quad j = i * 10; \\
\quad k = j - i; \\
\} \\
\texttt{else} \\
\quad i = j + k;
\end{array}
\quad \xrightarrow{\Gamma} \quad
\begin{array}{l}
\texttt{ite}\ (i_0 > 0, \\
\qquad j_1 = i_0 * 10\ \wedge\ k_1 = j_1 - i_0\ \wedge \\
\qquad i_1 = i_0, \\
\qquad i_1 = j_0 + k_0\ \wedge \\
\qquad j_1 = j_0\ \wedge\ k_1 = k_0 \\
\qquad)
\end{array}
$$

Such synchronization is done at the join node by inserting the following formula into $\Gamma(blk_1)$ (and similarly to $\Gamma(blk_2)$), where $\texttt{cur}(blk, v)$ returns v's last SSA index in blk.

$$v_j = v_i \quad \text{for} \quad i = \texttt{cur}(blk_1, v),\ j = \texttt{cur}(blk_2, v) \quad \text{such that } i < j$$

6.2.3 Variable Aliasing

Variables may be aliased due to the use of pointers or references. Typically, when the formal parameters of a function are of pointer or reference types, the parameters are the aliases of the incoming actual arguments. When converting the programs, we map an alias to its corresponding variable and use the variable rather than the alias. For the alias updated in different paths, we add an `ite` expression at the join. Note that most aliases in CUDA kernels occur at function entry.

$$
\begin{array}{l}
\text{int a[3]; int *i = a;} \\
\text{int j = i[1] + a[2];} \\
\text{i[0]++;}
\end{array}
\quad \xrightarrow{\Gamma} \quad
\begin{array}{l}
j_1 = a_0[1] + a_0[2]\ \wedge \\
a_1 = a_0([0] \mapsto a_0[0] + 1)
\end{array}
$$

However, we do not model complicated pointer operations (*e.g.* pointer dereference) although it can be implemented by using a global array to represent the shared memory. Since typical CUDA programs exhibit very limited pointer arithmetic operations, PUG does not encounter this problem in practice.

6.2.4 Scopes and Function Calls

Each basic block has its own scope. A variable should be distinguished from another one with the same name but in a different scope. For this, a variable is prepended by its scope number: ${}^n v$ indicates that v is in scope n. The scope numbers of top-level variables are skipped. When a function is inlined, its body constitutes a new scope. In the following example, the top-level code consists of

an "if" statement, whose left clause (a basic block) contains a call to f. Note that j is passed as a pointer.

$$\begin{array}{l} \texttt{int f (int i, int* j) \{} \\ \texttt{\ int k = i - j;} \\ \texttt{\ return (i * k);} \\ \texttt{\}} \end{array} \quad \begin{array}{l} \texttt{if (i > 10) \{} \\ \texttt{\ int i = 2;} \\ \texttt{\ int j = f(i, j);} \\ \texttt{\}} \end{array} \quad \xrightarrow{\Gamma} \quad \begin{array}{l} \neg(i_0 > 10) \vee \\ {}^1i_1 = 2 \ \wedge \ {}^2i_1 = {}^1i_1 \ \wedge \\ {}^2k_1 = {}^2i_1 - {}^1j_1 \ \wedge \\ {}^1j_1 = {}^2i_1 * {}^2k_1 \end{array}$$

6.3 Encoding Concurrency

A variable with modifier `shared` is "shared" for all threads within a block. Private variables have no modifiers. We now illustrate the translation of `shared` variable updates.

6.3.1 2-thread Translation of `Shared` Updates

Suppose we have to translate a shared assignment $v = 1$. Note that two threads are being allowed to concurrently perform this assignment. Our approach is to treat v as an array indexed by *Schedule IDs* (SID$\in \{0, 1, 2, \ldots\}$). (If v were an array, we would simply add one more dimension to v indexed by SID.) An SID has the same root name as the variable, but has a subscript and a superscript. It is like a timestamp and combines two pieces of information: which thread is accessing it (superscript), and where in the code the access is occurring (subscript, forming the single static assignment or SSA index [82]). With these, the translation of $v = 1$ is as follows:

$$v = 1 \qquad v[v_1^{t_1}] = 1 \ \wedge \ v[v_1^{t_2}] = 1$$

Here, the SIDs $v_1^{t_1}$ and $v_1^{t_2}$ range over $\{0, 1\}$. To say that t_1 accesses (writes) into v first, we can throw in the constraint $v_1^{t_1} < v_1^{t_2}$. To say that either access order is possible, we do not throw in any constraint. Now, things get more interesting when we translate $v = v + 1$:

$$v = v + 1; \quad \xrightarrow{\Gamma} \quad \begin{array}{l} v[v_2^{t_1}] = v[v_1^{t_1}] + 1 \ \wedge \ v[v_2^{t_2}] = v[v_1^{t_2}] + 1 \\ \wedge \ (v_2^{t_1} > v_1^{t_1}) \ \wedge \ (v_2^{t_2} > v_1^{t_2}) \wedge \\ v[v_1^{t_1}] = v[v_1^{t_1} - 1] \ \wedge \ v[v_1^{t_2}] = v[v_1^{t_2} - 1] \end{array}$$

and further $v_1^{t_1}$, $v_1^{t_2}$, $v_2^{t_1}$, and $v_2^{t_2}$ should be pairwise distinct and must belong to the set $\{0, \ldots, 3\}$.

First, let us look at the "pairwise distinct" requirement. This can be elegantly modeled by using an uninterpreted function f. More specifically, consider two variables l and m that range over $v_1^{t_1}$, $v_1^{t_2}$, $v_2^{t_1}$, and $v_2^{t_2}$. Then, we can say $f(l) \neq f(m)$. Since f is a function, this forces $l \neq m$.

Now, what about the rest of the constraints? It is clear that $v[v_2^{t_1}] = v[v_1^{t_1}] + 1$ and $v[v_2^{t_2}] = v[v_1^{t_2}] + 1$ model how "assignment works." It is also clear that $v_2^{t_1} > v_1^{t_1}$ and $v_2^{t_2} > v_1^{t_2}$ model that the L-value is updated only after the R-value is obtained. Now, what about the R-value itself? This depends on "who wrote v last." This is precisely why we include $v[v_1^{t_1}] = v[v_1^{t_1} - 1]$ and $v[v_1^{t_2}] = v[v_1^{t_2} - 1]$. It is interesting that this system, in one fell swoop, models all the six schedules possible.

Suppose $v_1^{t_1} = 0$, $v_2^{t_1} = 3$, $v_1^{t_2} = 1$, and $v_2^{t_2} = 2$. Then we have expressed these constraints: $v[3] = v[0] + 1 \ \wedge \ v[2] = v[1] + 1 \ \wedge \ v[1] = v[0]$. In this example, we are modeling the following schedule that, overall, increments v by 1, and not 2: (i) $v[1] = v[0]$ models that thread t_2 also "enjoys" the initial value of v in addition to t_1 (we take $v[-1]$ to be the initial value of v, which is what t_1 gets); (ii) $v[2] = v[1] + 1$ models that thread t_2 now does the update of this v; (iii) finally, $v[3] = v[0] + 1$ models that t_1 now takes the value it had read "long ago," is incrementing that value, and depositing it into v.

6.3.2 An Advanced Example Showing Barrier Encoding

In Figure 6.4 we illustrate the advanced features of our encoding scheme through an example (details in [60]). In this kernel, `k` is allocated in the `shared` memory.

• To capture the semantics of barriers, we assign them a single SID (*e.g.* *bar*0 in our example) and constrain them with respect to SIDs of *all* threads.

• Each thread t has a *private* copy of local variables like v. They are referred to by v^t. Since its value is independent of the schedule, there is no SID associated with it.

• We can now derive inequalities to model all these facts (the cases under ORDER are the numbers we refer to here): (1) the program order within each thread must be respected; (2) all the SIDs of all threads constitute a natural number interval $[0, 4n+1]$ where n is the number of threads, and (3) all the SIDs must be distinct.

A valid schedule of the given example for two threads is depicted below (note that k is the only shared variable):

```
__global__ kernel (unsigned int* k) {
  unsigned int s[2][3] = {{0,1,2},{3,4,5}};
  unsigned int i = threadIdx.x;
  unsigned int j = k[i] - i;
  if (j < 3)
    { k[i] = s[j][0];  j = i + j; }
  else
    s[1][j && 0x11] = k[i] * j;
  __syncthreads();
  k[j] = s[1][2] + j;
}
```

$$
\begin{array}{l}
\texttt{TRANS(t)} \equiv \\
s_1^t[0] = \lambda i \in \{0,1,2\}.i \ \wedge \ s_1^t[1] = \lambda i \in \{0,1,2\}.i + 3) \ \wedge \\
i_1^t = t \ \wedge \ j_1^t = k[k_0^t][i_1^t] - i_1^t \ \wedge \\
\texttt{ite}(j_1^t < 3, \ k[k_1^t] = k[k_1^t - 1] \uplus ([i_1^t] \mapsto s_1^t[j_1^t][0]) \ \wedge \ j_2^t = i_1^t + j_1^t \\
\qquad\qquad \wedge \, s_2^t = s_1^t, \\
\qquad\qquad s_2^t = s_1^t \uplus ([1][j_1^t \# 0x11] \mapsto k[k_2^t][i_1^t] \times j_1^t) \ \wedge \\
\qquad\qquad j_2^t = j_1^t \ \wedge \ k[k_1^t] = k[k_1^t - 1]) \\
k[k_2^t] = k[k_2^t - 1] \ \wedge \ k[bar_0] = k[bar_0 - 1] \ \wedge \\
k[k_3^t] = k[k_3^t - 1] \uplus ([j_2^t] \mapsto s_2^t[1][2] + j_2^t)
\end{array}
$$

$$
\begin{array}{l}
\texttt{TRANS}(t_1, \cdots, t_n) \equiv \bigwedge_{i \in [1,n]} \texttt{TRANS}(t_i) \\
\texttt{ORDER}(t_1, \cdots, t_n) \equiv \\
\quad (1) \quad \bigwedge_{i \in [1,n]} (k_0^{t_i} < \{k_1^{t_i}, k_2^{t_i}\} < bar_0 < k_3^{t_i}) \\
\quad (2) \quad bar_0 < l \ \wedge \ \bigwedge_{i \in [1,n],\, j \in [0,3]} (k_j^{t_i} < l) \text{ where } l = 4n + 1. \\
\quad (3) \quad \texttt{rank}(bar_0) = 0 \ \wedge \ \bigwedge_{i \in [1,n],\, j \in [0,3]} (\texttt{rank}(k_j^{t_i}) = 4i + j)
\end{array}
$$

Figure 6.4: An advanced example illustrating the encoding of concurrency.

$$
\begin{array}{l}
k_0^{t_1} = 0 \ \wedge \ k_1^{t_1} = 1 \ \wedge \ k_0^{t_2} = 2 \ \wedge \ k_1^{t_2} = 3 \ \wedge \ k_2^{t_2} = 4 \ \wedge \\
k_2^{t_1} = 5 \ \wedge \ bar_0 = 6 \ \wedge \ k_3^{t_2} = 7 \ \wedge \ k_3^{t_1} = 8
\end{array}
$$

In [60], we present an approach to detect races by encoding *Access IDs* into the formulas. It guarantees that all valid schedules are investigated; a race exhibiting in any particular schedule will not be missed. However, it does not scale well [62]; thus, we have replaced it with the method described in Section 6.4, which needs to consider only one schedule as Feng and Leiserson [29] did for multithreaded programs represented by series-parallel DAGs.

6.4 Conditional Barriers and Conflicts

The presence of conditional statements makes it imperative that we have the precision of the SMT technology when we check whether all barriers are well-synchronized. It also influences the determination of whether races occur. Work

such as [2] which rely purely on static analysis can generate too many false alarms in codes where there are many conditionals.

To illustrate these ideas, consider the control-flow graph (CFG) given in Figure 6.5(a). This diagram shows how statements s_1 through s_4 are situated in some example program (in (a), s_2 itself is shown expanded in terms of `write k[i]` followed by the barrier `bar`). At first glance, this appears ill-synchronized: one thread may take the s_1 to s_4 path encountering no barriers while another may take the path through p_1 encountering a barrier. Our SMT techniques can determine whether these paths are feasible, and flag an error if so. PUG's approach to checking for well-synchronized barriers is as follows: either (i) two branches must execute the same number of barriers; or (ii) all threads must make the same decision on the condition.

In Figure 6.5(a), if all threads make the same decision on condition p_1 , *i.e.* $\forall t_1, t_2 : p_1^{t_1} = p_1^{t_2}$, then all threads will execute the same branch, which is synchronization safe even if the two branches contain different numbers of barriers. In Figure 6.5(b), both the left and the right branch contains only one barrier; thus, they are considered well-synchronized.

Now, assume that all barriers are well synchronized. We must now check for conflicting accesses that occur in programs involving conditionals. If, for instance, the formula $(i^{t_1} = i^{t_2}) \wedge p_1^{t_1} \wedge p_1^{t_2}$ is true in Figure 6.5(a), both threads can take the p_1 branch *and* conflict on the same k location, causing a race.

A more general analysis is captured by the CFG in Figure 6.5(b). The conflict

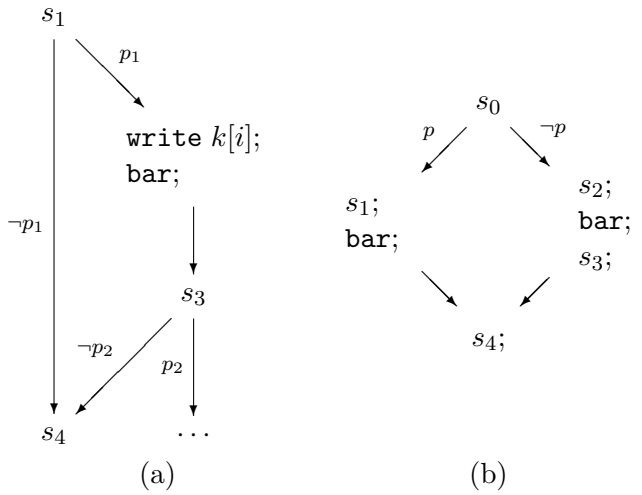


Figure 6.5: Example CFGs.

check includes the following expressions (here $\not\sim$ denotes *nonconflicting*). Also let us use $p \,?\, s$ to denote an expression s guarded by path condition p. Now, this CFG may be regarded as consisting of two *barrier intervals*: the first one containing s_0, $p \,?\, s_1$ and $\neg p \,?\, s_2$, and the second one containing s_4 and $\neg p \,?\, s_3$. Conflict freedom requires the pairwise comparison of the elements in each barrier interval:

$$\begin{array}{ll} p^{t_2} \Rightarrow s_0^{t_1} \not\sim s_1^{t_2} & \neg p^{t_2} \Rightarrow s_0^{t_1} \not\sim s_2^{t_2} \\ p^{t_1} \wedge \neg p^{t_2} \Rightarrow s_1^{t_1} \not\sim s_2^{t_2} & \neg p^{t_2} \Rightarrow s_4^{t_1} \not\sim s_3^{t_2} \end{array}$$

6.5 Serial Checking, Exploiting Barrier Intervals

CUDA programmers often intend to write *deterministic* programs whose final results are independent of the concurrent schedule. Thus, it is natural to seek analysis methods that also try to avoid having to generate schedules. Our insights are explained with respect to a simple example:

thread t_1	thread t_2
write $k[i]$;	read v;
$\ldots$;	$\ldots$;
write v;	read $k[j]$;

Let us ignore `write v` and `read v` for the moment. Suppose k is the only shared variable. Now, if both i and j are (control- and data-) dependent only on thread-local variables, then their values are the same in all schedules. In that case, in order to check whether write $k[i]$ and read $k[j]$ conflict, it suffices to examine only one arbitrary schedule that respects program order.

Suppose j depends on a shared variable v. Then, j's value in thread t_2 may be different in different schedules. *However, in this case, there exists a conflict on v.* Furthermore, this conflict can be detected by executing v's accesses according to one schedule (any schedule) that simply respects the program order. If we find two accesses within the same barrier interval that conflict, we are done detecting the conflict. This conflict does not go away under another schedule. For this reason, we say that k's conflict is *reduced* to v's.

Theorem 1 (Serializability) Consider each pair of accesses to shared variables where one access in the pair is a write. Suppose these access pairs can be shown

to be nonconflicting. Then, the entire code containing these accesses is race free and can be serialized.

PUG implements such conflict checks and is able to eliminate generating concurrency schedules in our realistic examples. We now show how the ideas in this theorem apply to programs that are decomposed in terms of barrier intervals.

6.5.1 Barrier Intervals (BI) and Incremental Modeling

CUDA intrablock thread executions exhibit a regular pattern: $\{t_1, \cdots, t_n\}$ execute $\rightarrow$ barrier $\rightarrow \{t_1, \cdots, t_n\}$ execute $\rightarrow \cdots$. Since an access before a barrier will never conflict with an access after this barrier, we may focus on the accesses between two consecutive barriers (so called a *barrier interval* or *BI*). If the accesses in a BI are nonconflicting, we build a transition constraint by serializing (sequentializing) them; then, we move on to the next BI and hope to repeat this treatment. This approach also goes hand in hand with our SMT solver Yices's [124] *incremental SMT solving* facility that reuses existing conflict clauses in the context when checking new expressions. As an illustration, we consider the following program *where shared variables are marked with a hat for readability.*

$$\begin{array}{lll} 1: \ j^t := \widehat{i}^t + t + 1; & 2: \ \_\_\text{synthreads}; & 3: \ e_1 = \widehat{k}^t[\widehat{i}^t]; \\ 4: \ \widehat{k}^t[j^t] = e_2; & 5: \ \_\_\text{synthreads}; & 6: \ \text{write } \widehat{i}^t \end{array}$$

Let us consider the case of two threads t_1 and t_2. The first BI consists of statement 1. Since there are no writes to shared variables, accesses to $\widehat{i}$ at t_1 and t_2 are nonconflicting. Both of them can be set to $i[0]$, *i.e.* their SIDs can both be forced to be 0. Using this approach, the transition relation up to statement 2 can be simplified and rewritten as follows (the js are private variables):

$$\texttt{TRANS}(t_1, t_2)_2 \equiv j_1^{t_1} = i[0] + t_1 + 1 \ \wedge \ j_1^{t_2} = i[0] + t_2 + 1$$

Now, the second BI consists of a read and a write to shared variable $\widehat{k}$. We need to determine whether their addresses may overlap for different threads. Given $\texttt{TRANS}(t_1, t_2)_2 \wedge t_1 \neq t_2$, expression $j_1^t = i[0]$ is unsatisfiable for $t \in \{t_1, t_2\}$. Therefore, the read and write of $\widehat{k}$ do not conflict. Also, we have $j_1^{t_1} = j_1^{t_2}$. Therefore, even the writes to $\widehat{k}$ are nonconflicting. We can follow the approach

used before and (re-)use the SIDs 0 and 1 for $\widehat{i}$ and $\widehat{j}$, respectively, and write the translation up to statement 4 as:

$$\begin{array}{l}\texttt{TRANS}(t_1,t_2)_4 \equiv \\ \quad \texttt{TRANS}(t_1,t_2)_2 \ \wedge \ \bigwedge_{t\in\{t_1,t_2\}}(\Gamma(e_1^t) = k[0][j_1^t]) \\ \quad \wedge\, k[1] = k[0] \uplus ([j_1^{t_1}] \mapsto \Gamma(e_2^{t_1})) \uplus ([j_1^{t_2}] \mapsto \Gamma(e_2^{t_2}))\end{array}$$

Things are fine if we keep the barrier (__syncthreads) at statement 5. Let us remove it and see what happens. Then, the second BI includes statement write $\widehat{i}^t$. Now, we do not know what write $\widehat{i}^t$ will write into $\widehat{i}^t$. It is possible that expression $j_1^t = \widehat{i}$ can be satisfied. The key observation is that *the conflict between statements 3 and 4 is reducible to a conflict between statements 3 and 6.*

The key point here is that we can keep building constraints without considering interleavings (just by following a canonical interleaving). If there is any race at all in the program, we will reach a point where there will be one conflict somewhere. Since we assume conflicts are rare, this optimistic approach has the ability to process many CUDA kernels successfully without finding any conflicts (and hence races).

In practice, instead of coalescing the SIDs among multiple threads, PUG builds the transitions in a *thread modular* manner: after constructing one single parameterized transition `TRANS(t)`, it instantiates the SIDs with concrete values so as to serialize the concurrent execution of all threads.

We give in Figure 6.6 the entire model of the example kernel in Section 6.3 for n threads. There are two BIs each of which contains only one write. The serialization makes t_i happen before t_j for $i < j$ for each BI. Hence, the SIDs of the writes in $t_1, t_2, \ldots, t_n$ in the first BI are $1, 2, \ldots, n$; and those in the second BI are $n+1, n+2, \ldots, 2n$. Clearly, this enforces that (1) within a BI, accesses in thread t_i happen before those in t_j for $i < j$; and (2) in a thread, accesses in BI i happen before those in BI j for $i < j$.

6.6 Loop Abstraction

While it is possible to unroll loops for precise checking, loop unrolling may not scale, especially with nested loops. Also, the loop bounds may involve symbolic values, making it impossible to perform loop unrolling. Consider the scalar product example shown in Figure 6.1. The outermost loop iterates through every

$$
\begin{array}{l}
\texttt{TRANS}(\texttt{t}_\texttt{x}, \texttt{n}) \equiv \\
s_1^t[0] = \lambda i \in \{0,1,2\}.i \ \wedge \ s_1^t[1] = \lambda i \in \{0,1,2\}.i + 3) \wedge \\
i_1^t = t \ \wedge \ j_1^t = k[x-1][i_1^t] - i_1^t \ \wedge \\
\texttt{ite}(j_1^t < 3, \ k[x] = k[x-1] \uplus ([i_1^t] \mapsto s_1^t[j_1^t][0]) \wedge \\
\qquad\qquad j_2^t = i_1^t + j_1^t \ \wedge \ s_2^t = s_1^t, \\
\qquad\quad s_2^t = s_1^t \uplus ([1][j_1^t \# 0x11] \mapsto k[x-1][i_1^t] \times j_1^t) \wedge \\
\qquad\qquad j_2^t = j_1^t \ \wedge \ k[x] = k[x-1]) \\
k[n+x] = k[n+x-1] \uplus ([j_2^t] \mapsto s_2^t[1][2] + j_2^t)
\end{array}
$$

$$
\texttt{TRANS}(t_1, \cdots, t_n) \equiv \bigwedge_{x \in [1,n]} \texttt{TRANS}(t_x, n)
$$

Figure 6.6: A serialized model of an example kernel.

pair of vectors. Each iteration first cycles through vectors with stride `ACC_N`, then performs tree-like reduction of the results. In practice, the grid size, the block size and the stride are large numbers, making it impractical to unroll, particularly the nested loops. One solution is to downscale the problem size by reducing these sizes to small numbers while preserving the program's behaviors (this is tedious if done manually). Another solution – the focus of this section – is to perform loop abstraction to reduce or even eliminate loop unrolling.

6.6.1 Loop Normalization

A standard result in program analysis [3] is that if the stride part of a loop is a linear function of the loop index i *i.e.* of format $i = i \pm e$ where e is an expression), then we can *normalize* such loops so they have a stride of one. For example, the loop header

$$\text{for (int i = lb; i} \leq \text{ub; i += stride)}$$

can be normalized to

$$\text{for (int i = 0; i} \leq \text{(ub - lb) / stride; i++),}$$

and each reference to i within the original loop is replaced by $i * stride + lb$. After normalization, the precise value range of the loop index is $[0, (ub - lb)/stride]$. When the stride is not a linear function on the loop index, we do not perform normalization to avoid making the range imprecise. Consider lines 13-17 of the example in Figure 6.1. Since the stride of the loop at line 13 is nonlinear, we

leave it alone. Since the stride of the loop at line 15 is linear, we change it. The transformation results in this code:

```
for(int stride = ACC_N/2; stride > 0; stride >>= 1) {
  __syncthreads();
  for(int i' = 0; i' < (stride-threadIdx.x)/blockDim.x; i'++) {
    int i = i' * blockDim.x + threadIdx.x;
    acc[i] += acc[stride + i];
  }}
```

To determine whether this code is conflict-free (no race on `acc` on line 16), we need to check, for threads t_1 and t_2, two cases:

• Whether ($i^{t_1} = i^{t_2}$). Luckily, this is false because i is initialized to `threadIdx.x` (different for different threads) and stays different.

• Or, whether ($i^{t_1} = stride^{t_2} + i^{t_2}$). This is also false because $i^{t_1} < stride^{t_2}$ holds.

The logical formula for conflict checking incorporates all this knowledge and also that $stride \in (0, ACC\_N/2]$ and $i \in [0, (stride - threadIdx.x)/blockDim.x)$; it also emerges unsatisfiable:

$$\bigwedge_{t\in\{t_1,t_2\}} \left(\begin{array}{l} stride^t > 0 \ \wedge \ stride^t \leq ACC_N/2 \ \wedge \\ i'^t \geq 0 \ \wedge \ i' < (stride^t - t)/blockDim.x \\ \ \wedge \ i^t = i'^t * blockDim.x + t \end{array} \right)$$
$$\wedge \ (t_1 \neq t_2) \ \wedge \ (i^{t_1} = i^{t_2} \ \vee \ i^{t_1} = stride^{t_2} + i^{t_2})$$

Similar analysis can also be applied to the loop at lines 6-11.

6.6.2 Automatic Refinement

In addition to the loop index, we need to handle the variables in the loop body. Consider the following example; the constraints generated for j, l, n, and k depend on whether they are loop carrying.

```
int m = 0; int k = a;
for(int i = lb; i < ub; i++)
{ int j = i * 2; int l = j + i;
  int n = m - l; k = j * k; ... }
```

A variable is *non-loop-carrying* if (1) it is the loop index variable, or (2) it is not updated in the loop, or (3) any of its updates (if there is any) involves only non-loop-carrying variables. We simplify our analysis by generating constraints only for non-loop-carrying variables, and over-approximate loop-carrying variables to have range $(-\infty, +\infty)$. In this example, i, j, m, n are non-loop-carrying while k

is loop-carrying. The formula $i \in [lb, ub) \wedge j = i{*}2 \wedge l = j{+}i \wedge n = m{-}i$ accurately specifies the value ranges of i, j, l, and n, while k (because it is loop-carrying) is over-approximated by leaving it unconstrained.

Sometimes, over-approximating the range of a loop-carrying variable may lead to false alarms. If j below is unconstrained, then a false race will be reported on $s[\texttt{threadIdx} * n + j]$.

```
int j = 1; int n = blockDim.x;
for(int i = n; i > 0; i >>= 1)
{ s[threadIdx * n + j] = ...; j = j * 2; }
```

To overcome this, PUG incorporates simple rules for syntactically deriving common invariants safely, and automatically adds them to the constraints. For the above example, PUG derives an invariant $i{*}j = n$, which follows from the relation between $*2$ and right shift ($\gg 1$). This implies $j < n$ and threadIdx $* n + j$ are different in different threads. PUG derives invariants for similar simple patterns involving $+$ and $-$, $*$ and $/$, and so on, but only for the variables used in the addresses of shared variables.

As another example, invariant $j = v + i * k$ can be derived for the following loop since j can be normalized to have the same stride as loop index i does.

```
int j = v;
for (int i = 0; i < ub; i++)
  { ...; j += k;}
```

6.6.3 Interiteration Race Checking

Within a loop, accesses to shared variables may conflict with themselves in previous iterations, thus causing *interiteration* conflicts. For example, in the following loop,

```
for(int i = lb; i < ub; i++)
{ __syncthreads(); acc[i+1+tid] += acc[i]; }
```

access $acc[i + 1 + tid]$ may not conflict with $acc[i + 1 + tid]$ and $acc[i]$ in the same iteration. However, if the barrier is removed, then $acc[i+1+tid]$ may conflict with $acc[(i - 1) + 1 + (tid + 1)]$, *i.e.* the access by a neighboring thread in the previous iteration.

PUG considers two cases:

- The loop body is not barriered. Different threads may be in different iterations, *i.e.* i's values in different threads may be regarded to be unrelated. If the barrier is removed in the above example, the constraint for conflict checks is as follows, which is clearly satisfiable for $t_1 \neq t_2$.

$$i^{t_1} \in [lb, ub] \ \wedge \ i^{t_2} \in [lb, ub] \wedge \\ (i^{t_1} + 1 + t_1 = i^{t_2} \ \vee \ i^{t_1} + 1 + t_1 = i^{t_2} + 1 + t_2)$$

- The loop body is barriered (*e.g.* ends with a barrier). If the body satisfies the synchronization correctness requirement described in Section 6.4, then all threads will always be in the same iteration. In other words, loop index variable i should have the same value at all threads (*i.e.* $i^{t_1} = i^{t_2}$) (we say i is *single valued*); and the following constraint is unsatisfiable for $t_1 \neq t_2$.

$$i \in [lb, ub] \ \wedge \ (i + 1 + t_1 = i \ \vee \ i + 1 + t_1 = i + 1 + t_2)$$

  Even after i is set to single valued, we may still need to consider two consecutive iterations. For the following code, PUG considers the possibility that accesses in s2 at iteration i conflict with those in s1 at iteration $i + 1$.

```
for(int i = lb; i < ub; i++)
  { s1; __syncthreads(); s2; }
```

In the scalar product example, the loop in lines 6-11 belongs to the first case, while the loop in lines 13-17 belongs to the second case. If the barrier at line 14 in the second loop is removed, then accesses on $acc[i]$ and $acc[stride + i]$ may conflict when $stride^{t_1} \neq stride^{t_2}$.

6.7 Implementation and Experimental Results

As described earlier, PUG is based on the Rose framework for C program analysis. The user may input a file containing multiple kernels together with the main (CPU side) program. The kernel to be analyzed is syntactically flagged, and this kernel alone will be analyzed. Within the kernel of interest, the user may place `assert` assertions anywhere in the code, which will be checked during analysis.

Given an annotated program, PUG works in a push-button fashion and is totally syntax driven (similar to a precise type checker, more details in [60]). It first parses the program and triggers rules for each syntactic category, building

constraints in an intermediate format. For instance, for handling loops, it first checks if whether the loop body contains barriers. It then performs loop normalization and loop refinement, and analyzes the loop body which may contain multiple BIs. For a BI, PUG first checks whether there is a race (conflict); if so, it then reports the bug and terminates. Otherwise, it serializes all the accesses to shared variables and moves to the next BI.

Expressions in the intermediate language (IL) are converted to Yices' expressions for satisfiability checking. Yices' expressions are based on bit vectors (bounded integers). We found that the correctness of most CUDA kernels relied on the assumption that no overflows will occur in arithmetic operations. To model this, the user has the ability to request (through the "+O" flag) whether nonoverflow constraints must be incorporated (for unsigned bit vectors). Setting the +O flag causes PUG to generate and incorporate these additional constraints for + and ∗:

IL Expr.	Yices Expr.	Constraint
$e_1 + e_2$	$e_1 + e_2$	$e_1 < 2^n - 1 \ \wedge \ e_2 < 2^n - 1$
$e_1 * e_2$	$e_1 * e_2$	$e_1 < 2^{n/2} \ \wedge \ e_2 < 2^{n/2}$
$e_1 \,/\, e_2$	q	$e_2 * q + r = e_1 \ \wedge \ r < e_2$
$e_1 \,\%\, e_2$	r	$e_2 * q + r = e_1 \ \wedge \ r < e_2$

In addition, since Yices does not provide the "div" and "mod" operator directly, we implement them using multiplication and addition. Some optimizations are performed when e_1 or e_2 are constants. For example, $2^m * e_2$ and $e_1 \,/\, 2^m$ are converted to $e_2 \ll m$ and $e_1 \gg m$, respectively ($\gg$ is a shift operator).

The user may further use two more types of annotations within the kernel of interest:

• An `assume` that defines the problem configuration parameters and input constraints (*e.g.* whether a matrix is assumed to be square or what the input data constraints are). We capture this assume class as if it were a flag, "+C".

• In some examples, the user has to help PUG out by providing simple loop invariants or simple predicates on shared variables. These are assumed to be true (for now; future work will try to semi-automate their formal verification). These are shown as the "+R" flag. We do not include the syntactic invariants automatically generated by PUG into +R (these are guaranteed to be correct invariants).

We performed experiments using PUG on a machine with a single CPU (Intel Pentium-4 3.60 GHz processor with only 1 GB of memory). Our table of results in Table 6.1 shows which examples required these flags for verification to succeed, and not fail through false alarms. All the examples in this table are widely cited kernels from the CUDA SDK, and naturally, PUG found them all to be correct. "Pass" in this table asserts that (i) *All barriers were found to be well-synchronized*, and (ii) *No races were found*. When a benchmark program (*e.g.* `Reduction`) contains multiple kernels, we invoke them one by one – but in a single run – and report the total time of this run.

PUG has checked many more CUDA SDK kernels than shown in Table 6.1. While the computation of a large application is usually broken into multiple kernels, we have successfully checked some very large kernels (*e.g.* Eigenvalues, at 2,200 LOC). The translation time into IL and to the Yices constraints is negligible, and not counted in.

PUG is able to check most programs smoothly. The radix sort kernel is the most difficult one to analyze since the addresses of a few shared variable accesses cannot be resolved locally, *i.e.* they are control-dependent on the shared arrays which may be updated by multiple threads. This makes the checking difficult. In our present attack, we added +C constraints indicating the the shared arrays are (partially) sorted to overcome this limitation.

Table 6.1: Experimental results of checking some SDK kernel programs for synchronization errors, races, and bank conflicts.

Kernels	loc	+O	+C	+R	B.C.	Time(pass)
Bitonic Sort	65				LO	2.2
MatrixMult	102	*	*		HI	<1
Histogram64	136				LO	2.9
Sobel	130	*			HI	5.6
Reduction	315	*			HI	3.4
Scan	255	*	*	*	LO	3.5
Scan Large	237	*	*		LO	5.7
Nbody	206	*			HI	7.4
Bisect Large	1,400	*	*		HI	44
Radix Sort	1,150	*	*	*	LO	39
Eigenvalues	2,200	*	*	*	HI	68

6.7.1 Bank Conflict Checking

A fascinating direction to evolve PUG is in giving designers feedback on performance metrics. Thanks to our use of SMT, we can use the infrastructure for race checking to check for bank conflicts also. Specifically, access $k[i]$ and $k[j]$ incurs a race when $i = j$, and incurs a bank conflict when $i\%16 = j\%16$. Column "B.C." indicates how serious the bank conflict is, which is measured by the *percentage of the barrier intervals (BI) containing bank conflicts*: HI (High) and LO (Low) denote $\geq 50\%$ and $< 50\%$, respectively. Since only two threads are considered and the loops are not unrolled, these results are quite preliminary; yet, the promise is clear. We plan to give more accurate measurement in the future work.

6.7.2 Road-Testing PUG

We took 57 assignment submissions from a recently completed graduate GPU class taught in our department. The "Defects" column in Table 6.2 indicates how many kernels were found to be not well-parameterized — *i.e.* work only in certain configurations (*e.g.* the grids and blocks must have specific sizes). We had to manually find this out by guessing and trying different +C settings. This is a promising way to reverse-engineer unstated assumptions and provide feedback to a programmer to improve their kernel.

There were *three benign races* and *two fatal races* in these (presumably tested) codes. These fatal races can be attributed to missing barriers in the loop body or incorrect indexing at the boundary between two thread data spaces.

While PUG always does its set of automatic loop refinements, we were curious as to how many of these cases could have passed through without them. When we turned off automatic refinements, we found that only 17.5% of the kernels (measured in terms of loops, only 10.5% of the total number of loops) would have failed (by giving false alarms). Thus, it appears that for small to medium kernels represented by a class, about 90% of the kernels can be verified even without loop refinements.

Table 6.2: Experimental results of running PUG on class examples.

Defects	Race		Refinement	
	benign	fatal	over #kernel	over #loop
13 (23%)	3	2	17.5%	10.5%

6.7.3 Assertion Checking (Functional Correctness)

Users can specify the properties to be checked using our `assume` and `guarantee` directives. If a precondition $assume(P)$ and a postcondition $guarantee(Q)$ are specified, formula $P \wedge \neg Q$ is added into the constraint. For example, we can specify the correctness of the bitonic sort kernel

```
__global__ bitonic (int vals[]) {
  ...
  guarantee(i < j ⟹ vals[i] ≤ vals[j]);
}
```

Functional correctness check requires accurate models of the programs. PUG translates the program into a bounded one by unrolling the loops dynamically in the incremental modeling phase. The number of threads must be specified explicitly. Since CUDA programs are highly symmetric, we only need to consider a few threads.

Table 6.3 shows the SMT solving time in seconds. To speed up the checking we turn off the overflow detection, assign small values to the loop bounds, and use smaller bitvectors. Here, n denotes the number of threads; T.O denotes Time Out ($>$ 5 minutes). Correctness is proven for bug-free programs, and bugged programs are obtained by disabling some required constraints or specifying false assertions. Correctness check takes much longer time since the solver needs to prove unsatisfiability (*i.e.* absence of bugs) for all cases. In general, the degree of loop unrolling needed is proportional to the number of threads n, making the solving time blow up on n.

This checker identifies several "bugs" in these programs: (i) the "bitonic sort" is incorrect when the number of threads is not the power of 2; (ii) the "scalar product" is incorrect when ACC_N is not the power of 2; and (iii) the "matrix tranpose" is incorrect when the sizes of two input matrixes are smaller than the block size.

Table 6.3: Experimental results of property checking.

Kernels	n = 2		n = 4		n = 8	
	Corr.	Bug	Corr.	Bug	Corr.	Bug
simple reduct.	< 1	< 1	2.8	< 1	T.O	4.42
matrix transp.	< 1	< 1	1.9	< 1	28	6.5
bitonic sort	< 1	< 1	3.7	< 1	T.O	255
scalar product	< 1	< 1	6.9	2	T.O	137

As the property checker does not scale well with respect to the number of threads, it is intended to be used as a *unit* tester/verifier for functional correctness.

6.7.4 Performance Improvement

PUG utilizes Yices's *incremental SMT solving* technique to avoid evaluating an expression multiple times. This technique is primarily used to manage the built transitions. For example, when the solver is called for evaluating e over transitions $\mathbb{E}$ provided that path condition $\mathbb{C}$ holds, we first assert $\mathbb{E}$ and push the context containing $\mathbb{E}$ into Yices' context stack, then assert $\mathbb{C}$ and e to evaluate the entire expression. After that, when we want to evaluate e_1 on $\mathbb{E}$ and $\mathbb{C}_1$, we pop the context stack so as to restore the context containing the existing clauses for $\mathbb{E}$, then we assert $\mathbb{C}_1$ and e_1. This enables us to avoid evaluating $\mathbb{E}$ again.

We also apply a simple *slicing* algorithm to exclude useless transitions from the transition stack. A use-def analysis is performed to identify the variables which will be used by the addresses of shared variables. We do not build transitions for the assignments involving other variables. For instance, in the scalar product example of Figure 6.1, no transitions corresponding to the assignments on line 9 and line 16 will be added into the transition stack.

6.7.5 Some Limitations of PUG

Present day SMT solvers provide limited support for real numbers. PUG cannot prove the functional correctness of many CUDA applications that operate on float or double numbers. Fortunately, this does not limit PUG's conflict checking power because the addresses of shared variables involves only unsigned integers.

PUG may report false alarms if it fails to derive loop invariants for complicated program patterns. In this case, the user is required to provide sufficient invariants.

PUG cannot handle kernels containing complicated pointer arithmetic operations. In addition, PUG requires manual transformation of the source programs to Kernel C format (*e.g.* by converting "while" loops to "for" loops and eliminating advanced C++ features).

Although focusing on CUDA kernels, PUG can be easily extended to other domains such as lock based multithreaded programs. It is particularly suitable for checking such programs over relaxed memory models: we just need to loosen the

constraint on the accesses orders w.r.t the memory model. The main challenge, however, is to model involved APIs and system calls. One solution is to build light-weight models or abstract interpretations for these APIs as we did for MPI 2.0 [65].

6.8 Appendix: Details of the Static Checker

In this section, we present PUG's static checker. Only synchronization safe and conflict free programs can pass the checker (*i.e.* without causing the checking to be stuck). A (logical) expression refers to the expression built by converting a syntactic fragment in the source program. Generally, we write $\mathbb{L}$, $\mathbb{S}$, and $\mathbb{M}$ to represent a list, set, and map, respectively; we also write $\top$ and $\bot$ for Boolean value *true* and $false$, respectively.

6.8.1 Data Structures

The checker uses an analysis state $\sigma = (\mathbb{E}, \mathbb{C}, \mathbb{I}, \mathbb{R}, \mathbb{W}, \mathtt{b}, \mathtt{n})$ where

• transition stack $\mathbb{E}$ is a list of sets of expressions recording the transitions built so far;

• path condition $\mathbb{C}$ is a list of expressions recording the current path condition;

• SSA map $\mathbb{I}$ is list of maps recording the SSA indices and scope numbers of variables;

• read pattern $\mathbb{R}$ is a list of sets of guarded accesses recording the reads to shared variables;

• write pattern $\mathbb{W}$ is a list of sets of guarded accesses recording the writes to shared variables;

• flag `b` is a Boolean flag indicating whether the syntactic fragment under consideration is in a branch of a *s*ingle valued condition (*i.e.* all threads make the same decision on the condition); and

• flag `n` is an integer recording the current scope number.

6.8.1.1 List, Transition Stack and Path Condition

We use the notation $\langle e_1 \ddagger \ldots \ddagger e_n \rangle$ to represent a n-element list; we also use $\mathbb{L} \ddagger e$ for a list where e is the last element and $\mathbb{L}$ contains the rest elements. Although represented as a list, transition stack $\mathbb{E}$ is operated in a stack style (*e.g.* using push and pop operations). In addition, we write $\mathbb{L} \uplus e$ for adding expression e

into $\mathbb{L}$ such that (1) if the elements of $\mathbb{L}$ are sets, then e is inserted into the last set of $\mathbb{L}$; and (2) otherwise, e becomes the last element. Operator $\bigwedge$ depicts the conjunction of all the expressions in $\mathbb{M}$ or $\mathbb{C}$, and `size` gives the length of a list.

6.8.1.2 SSA Map

An item in the map $\mathbb{M}$ is represented by $a \mapsto v$, where a and v are the address and value, respectively. We use $\mathbb{I}[a]$ for a's value in $\mathbb{I}$, and $\mathbb{I}[a \mapsto v]$ for the update of the element at a to v.

To resolve the name conflict on variables in multiple scopes, an SSA map contains a list of maps, each of which corresponds to the variables declared in a scope. To handle variable aliasing such as variable v is an alias of v', an item in the map is specified as $v \mapsto (v', k, i)$, where k is the scope index and i is the SSA index. The read operation is actually defined as follows. The last map $\mathbb{M}$ is searched first; if v is not in $\mathbb{M}$, then the search goes into prior maps $\mathbb{I}$. Otherwise, if v is not an alias of another variable v', then indices k and i are returned; otherwise, v' indices are returned. The update operation is defined similarly.

$$
\begin{array}{l}
(\mathbb{I} \ddagger \mathbb{M})[v] \doteq \\
\texttt{if } v \mapsto (v', k, i) \in \mathbb{M} \texttt{ then } (\textsf{if } v = v' \textsf{ then } (k, i) \textsf{ else } (\mathbb{I} \ddagger \mathbb{M})[v']) \\
\texttt{else } \mathbb{I}[v]
\end{array}
$$

6.8.1.3 Access Pattern and Flag `b`

An access may be guarded by a path condition. We write $c \,?\, v$ for the access v guarded by path condition c, and simply v if there is no path condition.

Pattern $\langle \mathbb{S}_1 \ddagger \mathbb{S}_2 \ddagger \mathbb{S}_3 \rangle$ denotes that the accesses in $\mathbb{S}_1$ and $\mathbb{S}_2$ are separated by a barrier, so do those in $\mathbb{S}_2$ and $\mathbb{S}_3$. For example, for the CFG in Figure 6.5(b), the access patterns of the left and right branch are $\mathbb{A}_l = \langle \{p \,?\, s_1\} \ddagger \{\} \rangle$ and $\mathbb{A}_r = \langle \{\neg p \,?\, s_2\} \ddagger \{\neg p \,?\, s_3\} \rangle$, respectively. The pattern for the entire CFG is $\langle \{s_0,\ p \,?\, s_1,\ \neg p \,?\, s_2\} \ddagger \{s_4,\ \neg p \,?\, s_3\} \rangle$. We write $c \,?\, \mathbb{A}$ for the prepending of condition c to the path conditions of all the accesses in $\mathbb{A}$; and $\mathbb{A}_1 \cup \mathbb{A}_2$ for the union of $\mathbb{A}_1$ and $\mathbb{A}_2$.

$$
\begin{array}{l}
c \,?\, \langle \{p_1 \,?\, s_1, \ldots, p_n \,?\, s_n\} \ \ddagger \ \ldots \rangle \ \ddagger \ \{p'_1 \,?\, s'_n\} \doteq \\
\quad \langle \{(c \ddagger p_1) \,?\, s_1, \ldots, (c \ddagger p_n) \,?\, s_n\} \ \ddagger \ \ldots \ \ddagger \ \{(c \ddagger p'_1) \,?\, s'_n\} \rangle \\
\langle \mathbb{S}_1 \ddagger \ldots \ddagger \mathbb{S}_n \rangle \ \cup \ \langle \mathbb{S}'_1 \ddagger \ldots \ddagger \mathbb{S}'_n \rangle \doteq \langle (\mathbb{S}_1 \cup \mathbb{S}'_1) \ddagger \ldots \ddagger (\mathbb{S}_n \cup \mathbb{S}'_n) \rangle
\end{array}
$$

Flag `b` is set when a condition is not single valued. If it is false, then all threads will enter the same branch; thus, there is no need to compare the accesses in the different branches. Otherwise, accesses in the left branch may conflict with those in the right branch, which requires us to maintain and compare the access patterns of the two branches.

In the initial analysis state, $\mathbb{E}$, $\mathbb{R}$, and $\mathbb{W}$ contain an empty set, $\mathbb{I}$ contains an empty map, $\mathbb{C}$ is empty, flag `b` is false, and scope number `n` is 0. Note that only shared memory access is modeled in this paper; global memory access is handled similarly. Since there exists no global barrier, we only need to maintain access sets rather than access patterns.

6.8.2 Main Rules

We use relation $\sigma \succ p \rightarrow \sigma'$ to depict how an analysis state is updated when we examine a syntactic program fragment p. Figures 6.7 and 6.8 show a subset of the relation rules. Here, $\overline{\cap}$ means *does not intersect*. To distinguish an expression in the source language and a generated logical expression, we often write $\underline{e}$ to emphasize that e is a generated expression. Note that operator $\doteq$ is for introducing the abbreviation for a long expression.

6.8.2.1 Expression Evaluation

We write $\mathbb{E}^{t_1};\mathbb{E}^{t_2}$ for the conjunction of the transitions in $\mathbb{E}$ in thread t_1 and t_2; and $\mathbb{E}^{t_1};\mathbb{E}^{t_2} \vdash e$ for the evaluation of e upon $\mathbb{E}^{t_1};\mathbb{E}^{t_2}$. The indices of shared variables in t_2 should be adjusted in the conjunction. Specifically, there are three types of variables: local variables, which are private to threads; single value variables, which have the same values among all threads (used for synchronizing loop iterations); and shared variables. We denote their types as τ_{pr}, τ_{sv} and τ_{sh}, respectively. Only local variables need to be superscripted by the thread id. The following example shows that z_i in thread t_2 should be adjusted to z_{i+1}, where 1 is actually z' maximum index in thread t_1.

source	$\mathbb{E}$	$\mathbb{E}^{t_1};\mathbb{E}^{t_2}$
$x : \tau_{pr} = 1;$	$x_1 = 1 \wedge$	$x_1^{t_1} = 1 \ \wedge \ y_1 = 2 \wedge$
$y : \tau_{sv} = 2;$	$y_1 = 2 \wedge$	$z_1 = x_1^{t_1} + z_0 - y_1 \wedge$
$z : \tau_{sh} = x + z - y;$	$z_1 = x_1 + z_0 - y_1$	$x_1^{t_2} = 1 \ \wedge \ y_1 = 2 \wedge$
		$z_2 = x_1^{t_2} + z_1 - y_1$

Expression Evaluation:

$$\frac{t_1 \not\sim t_2 \qquad \mathbb{E}^{t_1}; \mathbb{E}^{t_2} \vdash (\bigwedge \mathbb{C}^{t_1} \wedge \bigwedge \mathbb{C}^{t_2} \wedge (e^{t_1} = e^{t_2}))}{\mathbb{E}, \mathbb{C} \vdash \downarrow e} \text{ SINGLE\_VALUE}$$

$$\frac{t_1 \not\sim t_2 \qquad \mathbb{E}^{t_1}; \mathbb{E}^{t_2} \vdash (\bigwedge \mathbb{C}^{t_1} \wedge \bigwedge \mathbb{C}^{t_2} \wedge \bigwedge_{v_1 \in \mathbb{S}, v_2 \in \mathbb{S}'} (v_1^{t_1} \not\sim v_2^{t_2}))}{\mathbb{E}, \mathbb{C} \vdash \mathbb{S} \mathbin{\overline{\cap}} \mathbb{S}'} \text{ DISJOINT}_{\text{set}}$$

$$\frac{\forall i \in [1, n] : \mathbb{E}, \mathbb{C} \vdash \mathbb{S}_i \mathbin{\overline{\cap}} \mathbb{S}'_i}{\mathbb{E}, \mathbb{C} \vdash \langle \mathbb{S}_1 \ddagger \ldots \ddagger \mathbb{S}_n \rangle \mathbin{\overline{\cap}} \langle \mathbb{S}'_1 \ddagger \ldots \ddagger \mathbb{S}'_n \rangle} \text{ DISJOINT}_{\text{pat}}$$

Expressions and Statements:

$$\frac{\mathbb{I}[v] = (k, i) \qquad \mathbb{R}_1 \doteq \mathsf{if}\ v : \tau_{sh}\ \mathsf{then}\ \mathbb{R} \uplus \{{}^k v_i\}\ \mathsf{else}\ \mathbb{R}}{\mathbb{I}, \mathbb{R}, v \vdash {}^k v_i, \mathbb{R}_1} \text{ VREF}$$

$$\frac{\mathbb{I}, \mathbb{R}, e_1 \vdash \underline{e}_1, \mathbb{R}_1 \qquad \mathbb{I}, \mathbb{R}, e_2 \vdash \underline{e}_2, \mathbb{R}_2}{\mathbb{I}, \mathbb{R}, \mathsf{op}\ e_1\ e_2 \vdash \mathsf{op}\ \underline{e}_1\ \underline{e}_2, \mathbb{R}_1 \cup \mathbb{R}_2} \text{ EXP}$$

$$\frac{}{(\mathbb{E}, \mathbb{C}, \mathbb{I} \ddagger \mathbb{M}, \mathbb{R}, \mathbb{W}, \mathtt{b}, \mathtt{n}) \succ ty\ v \to (\mathbb{E}, \mathbb{C}, \mathbb{I} \ddagger (\mathbb{M}[v \mapsto (v, n, 0)]), \mathbb{R}, \mathbb{W}, \mathtt{b}, \mathtt{n})} \text{ DECL}$$

$$\frac{\begin{array}{c}\mathbb{I}[v] = (k, i) \qquad \mathbb{I}, \mathbb{R}, e \vdash \underline{e}, \mathbb{R}_1 \qquad \mathbb{W}_1 \doteq \mathsf{if}\ v : \tau_{sh}\ \mathsf{then}\ \mathbb{W}\ \mathsf{else}\ \mathbb{W} \uplus \{{}^k v_{i+1}\} \\ \mathbb{E}_1 \doteq \mathsf{if}\ \mathtt{is\_lc}(v)\ \mathsf{then}\ \mathbb{E}\ \mathsf{else}\ \mathbb{E} \uplus ({}^k v_{i+1} = \underline{e})\end{array}}{(\mathbb{E}, \mathbb{C}, \mathbb{I}, \mathbb{R}, \mathbb{W}, \mathtt{b}, \mathtt{n}) \succ v := e \to (\mathbb{E}_1, \mathbb{C}, \mathbb{I}[v \mapsto (k, i+1)], \mathbb{R}_1, \mathbb{W}_1, \mathtt{b}, \mathtt{n})} \text{ ASSIGN}$$

$$\frac{\begin{array}{c}\mathbb{E}, \mathbb{C} \vdash \mathbb{S}_r \mathbin{\overline{\cap}} \mathbb{S}_w \qquad \mathbb{E}, \mathbb{C} \vdash \mathbb{S}_w \mathbin{\overline{\cap}} \mathbb{S}_w \qquad \mathbb{R}_1 \doteq \mathsf{if}\ \mathtt{b}\ \mathsf{then}\ \mathbb{R} \ddagger \mathbb{S}_r \ddagger \{\}\ \mathsf{else}\ \langle\{\}\rangle \\ \mathbb{W}_1 \doteq \mathsf{if}\ \mathtt{b}\ \mathsf{then}\ \mathbb{W} \ddagger \mathbb{S}_w \ddagger \{\}\ \mathsf{else}\ \langle\{\}\rangle\end{array}}{(\mathbb{E}, \mathbb{C}, \mathbb{I}, \mathbb{R} \ddagger \mathbb{S}_r, \mathbb{W} \ddagger \mathbb{S}_w, \mathtt{b}, \mathtt{n}) \succ \text{barrier} \to (\mathbb{E}, \mathbb{C}, \mathbb{I}, \mathbb{R}_1, \mathbb{W}_1, \mathtt{b}, \mathtt{n})} \text{ BARRIER}$$

Control Flow Structures:

$$\frac{(\mathbb{E}, \mathbb{C}, \mathbb{I} \ddagger \{\}, \mathbb{R}, \mathbb{W}, \mathtt{b}, \mathtt{n}+1) \succ \text{s} \to \sigma}{(\mathbb{E}, \mathbb{C}, \mathbb{I}, \mathbb{R}, \mathbb{W}, \mathtt{b}, \mathtt{n}) \succ \{\text{s}\} \to \sigma} \text{ BLOCK} \qquad \frac{\sigma \succ \text{s}_1 \to \sigma_1 \qquad \sigma_1 \succ \text{s}_2 \to \sigma_2}{\sigma \succ \text{s}_1; \text{s}_2 \to \sigma_2} \text{ SEQ}$$

$$\frac{\begin{array}{c}\neg \mathtt{contain\_barrier}(s) \\ (\mathbb{E} \uplus ({}^{n+1} i_0 \in [0, (\underline{ub} - \underline{lb})/\underline{sd}]) \uplus ({}^{n+1} i_1 = {}^{n+1} i_0 * \underline{sd} + \underline{lb}), \mathbb{C}, \\ \mathbb{I} \ddagger \{i : \tau_{pr} \mapsto (n+1, 1)\}, \mathbb{R}, \mathbb{W}, \mathtt{b}, \mathtt{n}+1) \succ \text{s} \to \sigma_1\end{array}}{(\mathbb{E}, \mathbb{C}, \mathbb{I}, \mathbb{R}, \mathbb{W}, \mathtt{b}, \mathtt{n}) \succ \text{for (int i = lb, i} \leq \text{ub, i += sd) \{s\}} \to \sigma_1} \text{ LOOP}_{\text{async}}$$

$$\frac{\begin{array}{c}\mathbb{E}, \mathbb{C} \vdash \downarrow (\underline{ub} - \underline{lb})/\underline{sd} \\ (\mathbb{E} \uplus ({}^{n+1} i_0 \in [0, (\underline{ub} - \underline{lb})/\underline{sd}]) \uplus ({}^{n+1} i_1 = {}^{n+1} i_0 * \underline{sd} + \underline{lb}), \mathbb{C}, \\ \mathbb{I} \ddagger \{i : \tau_{sv} \mapsto (n+1, 1)\}, \mathbb{R}, \mathbb{W}, \top, \mathtt{n}+1) \succ \text{s} \to \sigma_1\end{array}}{(\mathbb{E}, \mathbb{C}, \mathbb{I}, \mathbb{R}, \mathbb{W}, \mathtt{b}, \mathtt{n}) \succ \text{for (int i = lb, i} \leq \text{ub, i += sd) \{barrier; s\}} \to \sigma_1} \text{ LOOP}_{\text{sync}}$$

Figure 6.7: Representative rules used by the static checker (I).

BRANCH (Control Flow):

$$\frac{\begin{array}{c} d \doteq \mathbb{E}, \mathbb{C} \vdash \downarrow c \\ (\mathbb{E} \ddagger \{\}, \mathbb{C} \ddagger c, \mathbb{I}, \langle \mathbb{S}_r \rangle, \langle \mathbb{S}_w \rangle, d, \mathtt{n}) \succ s_1 \rightarrow (\mathbb{E} \ddagger \mathbb{S}_1, \mathbb{C}_1, \mathbb{I}_1, \mathbb{R}_1, \mathbb{W}_1, d, \mathtt{n}_1) \\ (\mathbb{E} \ddagger \{\}, \mathbb{C} \ddagger \neg c, \mathbb{I}, \langle \mathbb{S}_r \rangle, \langle \mathbb{S}_w \rangle, d, \mathtt{n}) \succ s_2 \rightarrow (\mathbb{E} \ddagger \mathbb{S}_2, \mathbb{C}_2, \mathbb{I}_2, \mathbb{R}_2, \mathbb{W}_2, d, \mathtt{n}_2) \\ \mathbb{R}'_1 \doteq c \,?\, \mathbb{R}_1 \qquad \mathbb{R}'_2 \doteq \neg c \,?\, \mathbb{R}_2 \qquad \mathbb{W}'_1 \doteq c \,?\, \mathbb{W}_1 \qquad \mathbb{W}'_2 \doteq \neg c \,?\, \mathbb{W}_2 \\ \neg d \;\Rightarrow\; \begin{array}{c} (\mathtt{size}(\mathbb{R})'_1 = \mathtt{size}(\mathbb{R})'_2) \wedge \\ (\mathbb{E}, \mathbb{C} \vdash \mathbb{R}'_1 \frown \mathbb{W}'_2 \;\wedge\; \mathbb{E}, \mathbb{C} \vdash \mathbb{R}'_2 \frown \mathbb{W}'_1 \;\wedge\; \mathbb{E}, \mathbb{C} \vdash \mathbb{W}'_1 \frown \mathbb{W}'_2) \end{array} \\ e \doteq \mathtt{ite}(b, \bigwedge \mathbb{S}_1 \wedge \bigwedge_{\mathbb{I}} (\mathbb{I}_2 - \mathbb{I}_1), \bigwedge \mathbb{S}_2 \wedge \bigwedge_{\mathbb{I}} (\mathbb{I}_1 - \mathbb{I}_2)) \\ \mathbb{I}' \doteq \mathbb{I}[\forall v \in \mathtt{DOM}(\mathbb{I}) : v \mapsto \mathsf{if}\ \mathbb{I}_1[v] > \mathbb{I}_2[v]\ \mathsf{then}\ \mathbb{I}_1[v]\ \mathsf{else}\ \mathbb{I}_2[v]] \\ \mathbb{R}' \doteq \mathbb{R} \ddagger (\mathbb{R}'_1 \cup \mathbb{R}'_2) \qquad \mathbb{W}' \doteq \mathbb{W} \ddagger (\mathbb{W}'_1 \cup \mathbb{W}'_2) \end{array}}{(\mathbb{E}, \mathbb{C}, \mathbb{I}, \mathbb{R} \ddagger \mathbb{S}_r, \mathbb{W} \ddagger \mathbb{S}_w, \mathtt{b}, \mathtt{n}) \succ \text{if c } s_1 \text{ else } s_2 \rightarrow (\mathbb{E} \uplus e, \mathbb{C}, \mathbb{I}', \mathbb{R}', \mathbb{W}', \mathtt{b}, \mathtt{max}(n_1, n_2))}$$

Function Call:

Function f's declaration: int f(int v) {body; return e_1}

$$\frac{\begin{array}{c} \mathbb{I}, \mathbb{R}, e \vdash \underline{e}, \mathbb{R}_1 \\ (\mathbb{E} \uplus ({}^{n+1}v_1 = \underline{e}), \mathbb{C}, \mathbb{I} \ddagger \{v \mapsto (n+1, 1)\}, \mathbb{R}_1, \mathbb{W}, \mathtt{b}, n+1) \succ \mathit{body} \rightarrow (\mathbb{E}_1, \mathbb{C}, \mathbb{I}_1, \mathbb{R}_2, \mathbb{W}_1, \mathtt{b}, \mathtt{n}_1) \\ (\mathbb{E}_1, \mathbb{C}, \mathbb{I}_1, \mathbb{R}_2, \mathbb{W}_1, \mathtt{b}, \mathtt{n}_1) \succ w = e_1 \rightarrow (\mathbb{E}_2, \mathbb{C}, \mathbb{I}_2, \mathbb{R}_3, \mathbb{W}_2, \mathtt{b}, \mathtt{n}_2) \end{array}}{{\scriptstyle (\mathbb{E},\mathbb{C},\mathbb{I},\mathbb{R},\mathbb{W},\mathtt{b},\mathtt{n}) \succ \mathrm{w} \,=\, \mathrm{f(e)} \rightarrow (\mathbb{E}_2,\mathbb{C}_2,\mathbb{I}_2,\mathbb{R}_3,\mathbb{W}_2,\mathtt{b},\mathtt{n}_2)}}$$

Function f's declaration: int f(int* v) {body; return e_1}

$$\frac{\begin{array}{c} (\mathbb{E}, \mathbb{C}, \mathbb{I}[v \mapsto (x, \mathbb{I}[x])], \mathbb{R}, \mathbb{W}, \mathtt{b}, n+1) \succ \mathit{body} \rightarrow (\mathbb{E}_1, \mathbb{C}, \mathbb{I}_1, \mathbb{R}_1, \mathbb{W}_1, \mathtt{b}, \mathtt{n}_1) \\ (\mathbb{E}_1, \mathbb{C}, \mathbb{I}_1, \mathbb{R}_1, \mathbb{W}_1, \mathtt{b}, \mathtt{n}_1) \succ w = e_1 \rightarrow (\mathbb{E}_2, \mathbb{C}, \mathbb{I}_2, \mathbb{R}_2, \mathbb{W}_2, \mathtt{b}, \mathtt{n}_2) \end{array}}{{\scriptstyle (\mathbb{E},\mathbb{C},\mathbb{I},\mathbb{R},\mathbb{W},\mathtt{b},\mathtt{n}) \succ \mathrm{w} \,=\, \mathrm{f(x)} \rightarrow (\mathbb{E}_2,\mathbb{C}_2,\mathbb{I}_2,\mathbb{R}_2,\mathbb{W}_2,\mathtt{b},\mathtt{n}_2)}} \;\text{FUNC\_CALL}_{\text{ptr}}$$

Figure 6.8: Representative rules used by the static checker (II).

Rule SINGLE_VALUE checks whether an expression e is single valued (denoted by $\downarrow e$). Rule $\texttt{DISJOINT}_{\text{set}}$ checks whether the accesses in $\mathbb{S}$ and $\mathbb{S}'$ conflict. Two accesses to the same variable v conflicts iff both their addresses and their path conditions agree, *e.g.* $c_1 \,?\, v[e_1] \sim c_2 \,?\, v[e_2] \doteq \bigwedge c_1 \wedge \bigwedge c_2 \wedge (e_1 = e_2)$. Rule $\texttt{DISJOINT}_{\text{pat}}$ performs a similar check on access patterns.

6.8.2.2 Expressions and Statements

Rule VREF creates a corresponding SSA variable when a variable is referred. The generated variable is attached with the scope index k. If the variable is a shared variable, then this access is recorded into $\mathbb{R}$. Rule EXP converts an effect-free expression in the source program; it also records all the reads from shared variables. Rule DECL handles variable declaration by registering the variable in the last map of $\mathbb{I}$. These rules are for variables of nonpointer types.

Rule ASSIGN handles an assignment to a variable (of nonpointer type). It increases the SSA index of the variable by one, records this write reference into $\mathbb{W}$ if the variable is a shared variable, and inserts the transition representing this assignment into the transition stack if the variable is not loop carrying (predicate is_lc tells whether a variable is loop carrying).

One of the most important rules, BARRIER, is applied when a barrier is encountered. It takes the last access sets in $\mathbb{R}$ and $\mathbb{W}$ and compares the accesses in them upon the current $\mathbb{E}$ and $\mathbb{C}$. If there is a conflict, then the rule is inapplicable and no new analysis state is produced, indicating the checking is stuck. Otherwise, two kinds of actions may be performed depending on the flag b:

- If b is false, then $\mathbb{R}$ and $\mathbb{W}$ are cleared since the accesses in them will never be used again. Note that the accesses after this barrier will not conflict with those before the barrier, *e.g.* the accesses in $\mathbb{R}$ and $\mathbb{W}$.

- If b is true, then we are in a branch of a non-single-valued condition. The accesses in $\mathbb{R}$ and $\mathbb{W}$ must be kept since they will be compared with those in the other branch.

In both cases an empty access set is appended to the patterns.

6.8.2.3 Control Flow Structures

Rules BLOCK and SEQ deal with basic blocks and the sequential composition of two statements, respectively. They are self-explanatory. When entering a new block, we increase the scope index n by 1.

Rules $\texttt{LOOP}_{\text{async}}$ and $\texttt{LOOP}_{\text{sync}}$ are examples of loop abstraction described in Section 6.6. After the normalization, the constraint on i's new value range along with an expression for the substitution of i is added into the transition stack. If the body contains no barriers, then interiteration checking is turned on by marking i's type to single valued. Otherwise, we first make sure that all threads execute the same number of loops; then, mark i to be single valued to enforce that different threads will be always in the same iteration. For brevity, we only present the simplified version of the actual $\texttt{LOOP}_{\text{sync}}$ rule — here, a barrier is assumed to locate at the beginning (or end) of the body. The actual rule makes no assumption on the position of the barrier and checks possible conflicts in two consecutive iterations. See Section 6.6 for more details. There is a possible refinement on i: for statement sequence "s1; for(int i = lb; i $\leq$ ub, i++) body; s2", statment s1 and s2 may conflict with the body only if $i = lb$ and $i = ub$, respectively; thus, the checker uses these constraints to replace $i \in [lb, ub]$ accordingly.

Rule BRANCH, the most sophisticated one, examines individually two branches of a conditional statement and combines the two post states. For each branch, the initial access pattern contains the last access set of the pattern before the branching. After a branch returns the update pattern, we prepend the condition c (or its negation) to all the accesses in this pattern.

If the condition is not single valued, we first check whether the two branches contain the same number of barriers (*i.e.* the left and right patterns are of the same size); if not, then a synchronization error is found. Otherwise, we check whether the left access pattern will conflict with the right access pattern. If no conflict is found, then the two post states are combined:

- As described before, an ite expression combining the left transition set $\mathbb{S}_1$ and right transition set $\mathbb{S}_2$ is added into the combined transition stack, where

$$\bigwedge_{\mathbb{I}}(\mathbb{I}_2 - \mathbb{I}_1) = \bigwedge_{v \in \texttt{DOM}(\mathbb{I})}\{v_j = v_i \mid \mathbb{I}_1[v] = i \wedge \mathbb{I}_2[v] = j \wedge i < j\}.$$

- The two SSA maps are combined. Note that only the variables in $\mathbb{I}$ need to be merged since the variables in the scopes of the two branches will not be used anymore.

$$I_1[v] > I_2[v] \doteq i_1 > i_2 \text{ for } I_1[v] = (k_1, i_1) \ \wedge \ I_2[v] = (k_2, i_2)$$

- The two access patterns are combined and appended to the original pattern.

The two rules for function calls are self-explanatory. For brevity, we only present the case of one argument. Note that when a variable is passed into the function body as a pointer, the variable aliasing mechanism described on Page 93 is used.

This checker is able to prove absence of conflicts efficiently. However, it may generate false alarms due to over-approximations of the loops, in which case refinement discussed in Section 6.6 is used to rule out false alarms.

CHAPTER 7

PARAMETERIZED VERIFICATION: CUDA KERNELS

The property (assertion) checker presented in the previous chapter can handle large kernels – but for a fixed number (*e.g.* two or three) of threads. It builds a symbolic model (as transition relation) according to the operational semantics of a kernel. Despite being very general, it suffers from the problem of blowing-up on the number of threads when checking functional correctness. In fact, the checking times out easily when the number of threads is greater than 4. This makes it very difficult to check the properties of CUDA kernels which are usually run in hundreds of threads. While two threads are often sufficient for conflict checking, functional correctness requires more threads.

We show in this chapter that by taking a different approach to SMT-encoding, we can obtain parameterized verification for an important class of kernels. We show that for many kernels, this method outperforms greatly our previous method in Chapter 6 which runs out of capacity when we scale the number of threads. *This parameterized method builds the symbolic model according to data dependency on shared arrays.* It tracks how data flow through the threads in consecutive computation rounds. Since only one (parameterized) thread is considered, this method is highly scalable. From one perspective, *it implicitly implements the Omega Test [93] using SMT techniques.* This checker ensures that no false alarms will be reported. Although sometimes constrained by the capacity of SMT solvers and under-approximation is employed, it is able to locate bugs very fast.

One of the main applications of our method is to check the equivalence of a kernel and its optimized version. This parameterized equivalence checker is particularly suitable for handling typical optimizations for CUDA kernels such as memory coalescing and bank conflict elimination.

We organize this chapter by first presenting the generic, nonparameterized approach extended from Chapter 6, then the parameterized one, and then comparing

their performance on realistic CUDA programs.

7.1 Background and Motivating Examples

Recall that a CUDA kernel is launched as a 1D or 2D *grid* of *thread blocks*. The total size of a 2D grid is `gridDim.x` × `gridDim.y`. The coordinates of a (thread) block are ⟨`blockIdx.x`, `blockIdx.y`⟩. The dimensions of each thread block are `blockDim.x` and `blockDim.y`. Each block contains `blockDim.x` × `blockDim.y` threads, each with coordinates ⟨`threadIdx.x`, `threadIdx.y`⟩.

The values of `gridDim` and `blockDim` determines the *configuration* of the system, *e.g.* the sizes of the grid and each block. For a thread, `blockIdx` and `threadIdx` give its block index in the grid and its thread index in the block, respectively. For brevity purposes, we use *gdim*, *bid*, *bdim*, and *tid* for *gridDim*, *blockIdx*, *blockDim*, and *threadIdx*, respectively. Clearly constraints $bid.* < gdim.*$ for $* \in \{x, y\}$ and $tid.* < bdim.*$ for $* \in \{x, y, z\}$ always hold.

Consider the following simple example with 2D blocks, which is a slightly simplified version of the "transpose" kernel in CUDA SDK 2.0 [23].

```
void naiveTranspose (int *odata, int* idata, int width, int height) {
  int xIndex = bid.x*bdim.x + tid.x; int yIndex = bid.y*bdim.y + tid.y;
  if (xIndex < width && yIndex < height) {
    int index_in  = xIndex + width * yIndex;
    int index_out = yIndex + height * xIndex;
    odata[index_out] = idata[index_in];
  }
  int i, j;       // for the postcondition
  postcond(i < width && j < height =>
    odata[i * height + j] == idata[j * width + i]);
}
```

The threads transpose the array in parallel: each thread reads *idata* at location $(bid.x * bdim.x + tid.x) + width * (bid.y * bdim.y + tid.y)$ and writes it to *odata* at the location $(bid.y * bdim.y + tid.y) + height * (bid.x * bdim.x + tid.x)$. The functional correctness of this kernel is specified in the postcondition: the element at location $j * width + i$ in the input array *idata* is put at location $i * height + j$ in the output array *odata*. This property should hold *for all valid configurations* as well as *all possible input values*.

This naive kernel suffers from completely noncoalesced writes, and can be more than 10x slower than the following optimized kernel for large matrices. The kernel below is optimized to ensure all global reads and writes are coalesced, and to avoid

bank conflicts in shared memory. The computations between two consecutive barriers constitute a *barrier interval (BI)* or *round*. This example contains two rounds of computations.

```
void OptimizedTranpose (int *odata, int *idata, int width, int height) {
  __shared__ float block[bdim.x][bdim.x+1];
  // read the matrix tile into shared memory
  int xIndex = bid.x*bdim.x + tid.x; int yIndex = bid.y*bdim.y + tid.y;
  if((xIndex < width) && (yIndex < height)) {
    int index_in = yIndex * width + xIndex;
    block[tid.y][tid.x] = idata[index_in];
  }
  __syncthreads();
  // write the transposed tile to global memory
  xIndex = bid.y * bdim.y + tid.x; yIndex = bid.x * bdim.x + tid.y;
  if ((xIndex < height) && (yIndex < width))
  {  int index_out = yIndex * height + xIndex;
    odata[index_out] = block[tid.x][tid.y]; }
}
```

We may use the same postcondition as the above one to specify the functional correctness of this optimized kernel. Moreover, the equivalence of these two kernels can be specified as: suppose the two kernels take the same inputs, *i.e.* the same *idata*, *width*, and *height*, then after execution, they produce the same outputs (in *odata*) *for all possible configurations*. The main challange here is to meet the requirement of being parameterized and symbolic: the result should hold for any number of threads and any input value.

7.1.1 Related Work

7.1.1.1 Parameterized Verification

Some techniques [22, 92] reduce the problem of verifying parameterized system with infinite states to that with finite-state abstractions. They use counter abstraction [92], which abstract process identities, or environmental abstraction [22], which give abstract counting for the number of processes satisfying a given predicate. Typically, these techniques either require manual effort to obtain the appropriate abstraction or are applicable to certain restricted systems and properties.

Some others [5,90] apply automatic induction to generate and verify invariants of the parameterized systems. In most cases, manual effort is required to obtain the invariant generation; however, Pnueli *et al.* [90] presented a way to automatically

compute the invariants given an appropriate abstraction relation.

The reduction from infinite states to equivalent finite states is based on finding an appropriate cut-off k of the parameter of the system. The goal is to establish that a property is satisfied by k processes if and only if it is so by any number ($> k$) processes. Emerson and Namjoshi [28] provided such cut-off values for parameterized systems with ring topology. A similar technique [39] is proposed to obtain tighter bounds of the cut-off for parameterized systems independent of the communication topology. Our technique considers only one parameterized thread and requires no symmetry reduction.

7.1.1.2 Equivalence Checking

Many approaches have been proposed for checking the equivalence of two sequential programs. For instance, equivalence checkers [102, 120] perform a dependence graph abstraction of programs containing affine loops. The basic idea is to match the dependence graphs by checking the associated relations using Omega test. Unfortunately, Omega test only supports linear arithmetic; and the lack of a powerful decision procedure makes them unable to handle trivial arithmetic transformations such as $2 * k[i] = k[i] + k[i]$. They can only handle programs with high similarities.

TVOC [8] first verifies loop transformations using a specific proof rule Permute, then verifies structure-preserving optimizations using some validation rules. It relies on extra information supplied by the compiler to generate verification conditions, which are dumped to an SMT solver for satisfiability check. Zaks and Pnueli [127] also used SMT solving to check structure-preserving optimizations. Their verifier attempts to find invariants (to match the variables in the source and target programs) over the conjunction of the models of the two programs. However, it is hard to identify sufficient invariants for aggressive optimizations.

These checkers work on sequential programs. An equivalence checking method for CUDA kernels is discussed in [69]. As mentioned before, it makes many assumptions on the input programs; it is not parameterized; and and no implementation of the checker is reported.

7.2 Nonparameterized Checking

Although CUDA kernels are concurrent programs executing in parallel, CUDA programmers often intend to write *deterministic* programs whose final results are independent of the concurrent schedule. We have presented the static checker [61] to determine whether a program is deterministic; and also proved that a deterministic program could be serialized such that the accesses on shared variables are executed in a sequential order. In this section, we give a different (and slightly better) order to sequentialize the shared variable accesses. Unlike the one in Chapter 6, this order requires only local information of the accesses.

7.2.1 Serializing Concurrent Executions

We now illustrate the translation of `shared` variable updates in concurrent executions. Suppose we have to translate a global assignment `v[tid.x] = tid.x + 1` where v is a shared array. Note that n threads are being allowed to concurrently perform this assignment. On the other hand, since no data race exists and the program is deterministic, we can specify an order in which the assignments are executed by assigning SSA indexes to v. A typical order is to have the threads execute the assignments with respect to their thread ids: thread 0 executes first, then thread 1 executes, ..., finally thread $n-1$ executes. Such order is called the *natural order*.

$$v_1[0] = 0 + 1 \ \wedge \ v_2[1] = 1 + 1 \ \wedge \ \ldots \ \wedge \ v_n[n-1] = n - 1 + 1$$

Now consider a more complicated example where v is the only shared variable. As usual, we assume that no data races occur on v. In the first round, all threads execute `v[i] = v[j] + tid.x`. After all threads finish this assignment, the second round containing `v[k]++` starts execution.

v[i] = v[j] + tid.x; __syncthreads(); v[k]++;

The natural order generates the following constraint.

Thread t_0	...	Thread t_{n-1}
$v_1[i] = v_0[j] + 1$	...	$v_n[i] = v_{n-1}[j] + n$
$v_{n+1}[k] = v_n[k] + 1$	...	$v_{2n}[k] = v_{2n-1}[k] + 1$

Formally, the combined transition system for n threads is

$$\texttt{trans}(t_x, n) \equiv v_{x+1}[i] = v_x[j] + 1 \ \wedge \ v_{n+x+1}[k] = v_{n+x}[k] + 1$$
$$\texttt{TRANS}(t, n) \equiv \bigwedge_{x \in [0,n-1]} \texttt{trans}(t_x, n) \ .$$

In Figure 7.1 we give the model of the `naiveTranpose` kernel. Each thread has a *private* copy of local variables such as $xIndex$. They are referred to by $xIndex^{s_i}$ in each thread s_i. Similarly, we can obtain the model $\texttt{TRANS}^t(t, n)$ for the optimized kernel (we use s and t to refer to the source (naive) and target (optimized) kernel, respectively). The encoding of the postcondition is trivial and not shown here.

7.2.1.1 Equivalence Checking and Property Checking

Given the models $\texttt{TRANS}^s$ and $\texttt{TRANS}^t$ for two kernels with inputs $\vec{i}$ and output $\vec{o}$, the kernels are equivalent if and only if the following constraint holds. We subscript the variables in the source and target kernel with s or t, respectively.

$$\forall n. \, \texttt{TRANS}^s(s, n) \wedge \texttt{TRANS}^t(t, n) \wedge (\vec{i^s} = \vec{i^t}) \Rightarrow (\vec{o^s} = \vec{o^t}) \ .$$

Unfortunately, an SMT solver is unable to handle this quantified formula since the definition of `TRANS` is recursive over the number of threads and the solver requires a concrete n to unroll the recursion. This also forbids using induction (*e.g.* k-induction [89]) to to perform the proof. Moreover, the fact that it conjuncts the models of n threads makes it suffer from the blow-up problem.

In addition to equivalence checking, PUG also checks the properties specified as assertions (*e.g.* in the postconditions). The assertion language supports the definition of Boolean formula using C's syntax. Moreover, one of its main features is to allow the definition of loops so as to handle recursive properties and variables with symbolic values. For instance, consider a reduction kernel which computes

$$\begin{aligned}
&\texttt{trans}(s_i, n) \equiv \\
&\ \text{xIndex}_1^{s_i} = bid^{s_i}.x * bdim.x + s_i.x \ \wedge \ \text{yIndex}_1^{s_i} = bid^{s_i}.y * bdim.y + s_i.y \ \wedge \\
&\ \texttt{ite}(\text{xIndex}_1^{s_i} < \text{width}_0 \ \wedge \ \text{yIndex}_1^{s_i} < \text{height}_0, \\
&\qquad \text{index_in}_1^{s_i} = \text{xIndex}_1^{s_i} + \text{width}_0 * \text{yIndex}_1^{s_i} \ \wedge \\
&\qquad \text{index_out}_1^{s_i} = \text{yIndex}_1^{s_i} + \text{height}_0 * \text{xIndex}_1^{s_i} \ \wedge \\
&\qquad \text{odata}_{i+1}^{s} = \text{odata}_i^{s} \uplus ([\text{index_out}_1^{s_i}] \mapsto \text{idata}_0^{s}[\text{index_in}_1^{s_i}]), \\
&\qquad \text{odata}_{i+1}^{s} = \text{odata}_i^{s}) \\
&\texttt{TRANS}^s(s, n) \equiv \bigwedge_{i \in [0,n-1]} \texttt{trans}(s_i, n)
\end{aligned}$$

Figure 7.1: A unparameterized model of the `naiveTranpose` kernel.

the sum of the elements in the input array *idata* and store this sum in *odata* after the computation. A postcondition specifying the functional correctness is as following, where n is the number of elements in *idata*.

```
for (i = 1; i ≤ n; i++) {odata += idata[i];}
```

In some cases, the functional correctness can be specified recursively. Consider a `scan` kernel which computes the parallel prefix sum of the input elements. We show below a valid postcondition.

```
g_odata[0] = 0 ∧
(0 < i < n - 1 ⇒ g_odata[i+1] = g_odata[i] + g_idata[i])
```

7.3 Parameterized Checking

This section describes how to perform parameterized encoding. The key is to calculate the value of an output element regardless of the number of threads.

7.3.1 Single Conditional Assignment

Our method builds a symbolic model according to the accesses on shared arrays. We first present a method which eliminates all the intermediate variables so that only the accesses on shared arrays are left (an optimization is presented in Section 7.3.3). For example, the body of the `naiveTranspose` contains a conditional assignment (CA) to *odata* as follows.

```
if (bid.x * bdim.x + tid.x < width && bid.y * bdim.y + tid.y < height)
  odata[(bid.y * bdim.y + tid.y) + height * (bid.x * bdim.x + tid.x)] =
    idata[(bid.x * bdim.x + tid.x) + width * (bid.y * bdim.y + tid.y)];
```

This can be interpreted by a mapping from *odata* to *idata*. Let $c(tid)$, $addr_d(tid)$ and $addr_s(tid)$ denote the condition $bid.x * bdim.x + tid.x < width \wedge bid.y * bdim.y + tid.y < height$, the destination address $(bid.y * bdim.y + tid.y) + height * (bid.x * bdim.x + tid.x)$, and the source address $(bid.x * bdim.x + tid.x) + width * (bid.y * bdim.y + tid.y)$, respectively. This CA can be denoted as $c \ ? \ odata[addr_d] := idata[addr_s]$, where $odata[addr_d]$ and $idata[addr_s]$ are called the *range* and *domain* of the CA, respectively. Now, consider the k^{th} element in the output array, $odata[k]$. Its value comes from either (1) $idata[addr_s(s_i)]$ for some s_i provided that $k = addr_d(s_i)$ and the guard holds (there is only one such s_i since no race occurs on *idata*); or (2)

the old value of $odata[k]$ if $\nexists s_i : k = addr_d(s_i) \ \wedge \ c(s_i)$. For brevity, we write $p(s_i)$ for the predicate $(k = addr_d(s_i)) \ \wedge \ c(s_i)$. The following diagram indicates how $odata[k]$ is computed: if p holds for thread s_1, then $odata[i] = idata[addr_s(s_1)]$, otherwise thread s_2 is investigated, and so on. Here, we use the "xor" operator $\oplus$ to emphasize that at most one thread satisfies p. If no thread satisfies p, then the old value of $odata[k]$ is used. As before we will use SSA indices to subscript the accesses, *e.g.* $odata_1$ denote the first write to $odata$.

$$
\begin{array}{ccccccc}
 & & & odata[k] = & & & \\
p(s_1) & & p(s_2) & \ldots & p(s_n) & & \text{else} \\
\circ & \oplus & \circ & \ldots & \circ & \oplus & \circ \\
idata[addr_s(s_1)] & & idata[addr_s(s_2)] & \ldots & idata[addr_s(s_n)] & & odata_{old}[k]
\end{array}
$$

This seems to require the enumeration of n threads. However, since there exists no conflict, at most one thread will satisfy p. Therefore, we can build an SMT constraint considering only one thread (with symbolic ID s_i).

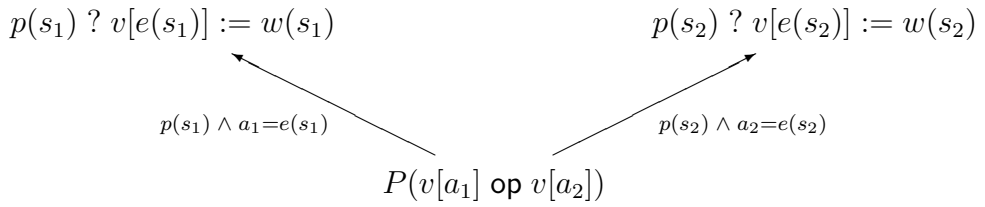
$$
\begin{array}{l}
(\exists s_i : p(s_i)) \Rightarrow odata_1[i] = idata_0[addr_s(s_i)] \text{ for that } s_i \\
(\forall s_i : \neg p(s_i)) \Rightarrow odata_1[k] = odata_0[k]
\end{array}
$$

Note that, for a given s_i, $\neg p(s_i)$ does not necessarily indicates that $odata[k]$ takes its old value — only if there exists no such s_i will the value of $odata[k]$ be unchanged. Thus, we cannot conclude that $odata_1[k] = \texttt{ite}(p(s_i),\ idata_0[addr_s(s_i)],\ odata_0[k])$. Instead, $odata_1[k] = odata_0[k]$ only if p does not hold for all s_i.

Unfortunately, existing SMT solvers often fail to handle quantified formulas (they return an inconclusive answer "unknown"). To overcome this limitation and make sure that our verifier gives conclusive answers, we derive unquantified formulas from the quantified ones and use them as the constraints. From the first formula, we can derive $p(s_i) \Rightarrow odata_1[k] = idata_0[addr_s(s_i)]$ for a fresh variable s_i, which indicates that, for any s_i, if $p(s_i)$ is true then $odata[k]$'s value comes from $idata_0[addr_s(s_i)]$. The absence of conflicts enables us to eliminate the $\exists$ quantifier by introducing the fresh variable s_i. For the second formula, we apply the approach detailed in section 7.3.4. It should be noted that such conversions are under-approximations: if PUG reports a bug, then this bug is real; if a kernel is correct, then PUG will not report a bug. However, PUG may fail to reveal some bugs in a kernel. We call the derived formulas *Verification Conditions*. Section 7.3.4 gives more discussions.

7.3.2 Instantiation of Conditional Assignments

Now consider a more complicated case where an expression contains multiple instances of an shared variables. For example, $v[a_1]$ op $v[a_2]$, where op is a binary operator, reads variable v twice at addresses a_1 and a_2, respectively. Right before these reads there exists a CA $p\ ?\ v[e] := w$. The question is: what is the value of $v[a_1]$ op $v[a_2]$ in terms of w? Or more specifically, suppose P is a predicate on $v[a_1]$ op $v[a_2]$; what is its value in terms of w?



For the first read $v[a_1]$, we introduce a fresh variable s_1 to denote the ID of the thread writing the value to $v[a_1]$. In other words, $(p(s_1)\ \wedge\ a_1 = e(s_1)) \Rightarrow v[a_1] = w(s_1)$. For the second read $v[a_2]$, note that we cannot use the same s_1 because the write may come from another thread. Thus, we introduce another fresh variable s_2 for the thread writing to $v[a_2]$ such that $(p(s_2)\ \wedge\ a_2 = e(s_2)) \Rightarrow v[a_2] = w(s_2)$. These two formulas connect expression $v[a_1]$ op $v[a_2]$ with w such that the value of this expression can be obtained from two instantiations (one for s_1 and the other for s_2) of w. In general, if an expression contains n reads from variable v, then n fresh variables and n formulas are created.

Considering only these two formulas, one verification condition for $P(v[a_1]\ \mathsf{op}\ v[a_2])$ is shown below. It reduces the checking on $v[a_1]$ and $v[a_2]$ to that on $w_1(s_1)$ and $w_2(s_2)$.

$$p(s_1)\ \wedge\ a_1 = e(s_1)\ \wedge\ p(s_2)\ \wedge\ a_2 = e(s_2)\ \Rightarrow\ P(w(s_1)\ \mathsf{op}\ w(s_2))$$

For instance, consider the optimized `Transpose` kernel. Let $X(i)$ and $Y(i)$ be $bid.x * bdim.x + i$ and $bid.y * bdim.y + i$, respectively. This kernel contains two CAs.

```
if (X(tid.x) < width && Y(tid.y) < height)
  block[tid.y][tid.x] = idata[Y(tid.y) * width + X(tid.x)];

if (Y(tid.x) < height && X(tid.y) < width)
  odata[X(tid.y) * height + Y(tid.x)] = block[tid.x][tid.y];
```

The value of the i^{th} output element $odata[i]$ may be tracked back to an element in the *block* first, then to an element in the *idata*. That is, it can be obtained by the sequential composition of the two CAs. We instantiate the tids in the first and second assignments to be t_1 and t_2, respectively (recall that we use t rather than s for this optimized kernel). An important point here is to match the first CA's range $block_1^t[t_2.x][t_2.y]$ and the CA's domain $block_1^t[t_1.y][t_1.x]$ using constraint $t_2.x = t_1.y \ \wedge \ t_2.y = t_1.x$.

$$
\begin{array}{l}
i = X(t_2.y) * height + Y(t_2.x) \ \wedge \ (Y(t_2.x) < height \wedge X(t_2.y) < width) \ \Rightarrow \\
\quad odata_1^t[i] = block_1^t[t_2.x][t_2.y] \\
(t_2.x = t_1.y \ \wedge \ t_2.y = t_1.x) \ \wedge \ (X(t_1.x) < width \wedge Y(t_1.y) < height) \ \Rightarrow \\
\quad block_1^t[t_2.x][t_2.y] = idata_0^t[Y(t_1.y) * width + X(t_1.x)]
\end{array}
$$

We may dig deeper into these formulas. Suppose $X(t.x), Y(t.y) < \mathsf{min}(width, height)$ holds for any t, *i.e.* thread t accesses data within the bounds of the 2-D input array with height *height* and *width*, then the above two formulas become

$$
\begin{array}{l}
i = X(t_2.y) * height + Y(t_2.x) \ \Rightarrow odata_1^t[i] = block_1^t[t_2.x][t_2.y] \\
(t_2.x = t_1.y \ \wedge \ t_2.y = t_1.x) \ \Rightarrow \\
\qquad block_1^t[t_2.x][t_2.y] = idata_0^t[Y(t_1.y) * width + X(t_1.x)] \ .
\end{array}
$$

If each block of threads is a square such that $bdim.x = bdim.y$, then we can derive the following formula justifying the correctness of the optimized kernel – the input array is correctly transposed *no matter how many threads are considered.* Note that this kernel is designed with implicit assumptions that (1) each block is square; and (2) only those threads with tid t satisfying $X(t.x), Y(t.y) < \mathsf{min}(width, height)$ should participate in the computation. Our encoding models exactly this design and reveals hidden assumptions. For example, PUG reports the bugs when the block is not square; and passes the checking for valid configurations.

$$odata_1^t[X(t_1.x) * height + Y(t_1.y)] = idata_0^t[Y(t_1.y) * width + X(t_1.x)]$$

The equivalence of the two example kernels requires $odata_1^s[i] = odata_1^t[i]$ provided that all above constraints hold and $idata_0^s = idata_0^t$. Note that only $odata[i]$ is instantiated only once for each kernel. We need to reducing the checking on $odata_1^s[i]$ and $odata_1^t[i]$ to that on the elements in the input array *idata*.

7.3.3 Barrier Interval and Control Flow

The statements between two consecutive barriers are within a *Barrier Interval* (BI). Since there are no conflicts in a BI, the writes to the same shared variable will not be on the same address. We may use this fact to simplify the generated constraints. Consider the following diagram where BI 1 contains multiple writes to v and in BI 2 property P reads v.

BI 1	$p_1 \ ? \ v[e_1] := w_1$ $p_2 \ ? \ v[e_2] := w_2$ $\dots$ $p_n \ ? \ v[e_2] := w_n$
BI 2	$P(v[a])$

The nonconflicting assumption indicates that there exists at most one $v[e_i]$ which would match $v[a]$. Thus, instead of writing a pair of constraints for each CA, we can combine all the CA constraints to be an embedded `ite` expression (here, v_1 and v_0 represent v's value right before BI 1 and BI 2, respectively). The main benefit is now we have only one quantified formula rather than n ones. In some cases (*e.g.* the two `Transpose` kernels), the quantified formula is not needed at all because $v[a]$'s value comes from one of the writes in BI 1.

$$
\begin{array}{l}
\mathsf{ite}(a = e_1 \ \wedge \ p_1, \ P(w_1), \\
\quad \mathsf{ite}(a = e_2 \ \wedge \ p_2, \ P(w_2), \\
\qquad \dots)) \quad \text{and} \\
(\nexists i \in [1, n] : a = e_i \ \wedge \ p_i) \Rightarrow P(v_0[a])
\end{array}
$$

A further optimization we employed is to keep the control flow of the BI and not eliminate all intermediate variables. The program below (the left column) contains two conditional jumps. Instead of flattening this program to generate three CAs: $c_1 \wedge c_2 \ ? \ v[e_1] = w_1$, $c_1 \wedge \neg c_2 \ ? \ v[e_2] = w_2$ and $\neg c_2 \ ? \ v[e_3] = w_3$, we keep this control flow structures and generate the constraint as shown on the right. This representation, which mimics those in Chapter 6, reduces substantially the size of the constraints and make them much more readable.

```
if (c1) {                    ite(c1,
  if (c2) v[e1] = w1;          ite(c2,
  else    v[e2] = w2;                a = e1 ⇒ P(w1),
}                                    a = e2 ⇒ P(w2),
else v[e3] = w3;               a = e3 ⇒ P(w3)))
```

7.3.4 Quantified Formulas

We try to convert a quantified formula into an equivalent quantifier-free formula whenever possible. The quantified formulas we encounter so far are of the following format, where t is the thread id with domain $[1..n]$, f is a function of t, c is the guard on t, a is an expression not involving t, and P is a predicate indicating the value of a variable is unchanged.

$$(\forall t \in [1..n] : \neg(a = f(t) \ \wedge \ c(t))) \ \Rightarrow \ P$$

We introduce a function $g : \text{int} \rightarrow \text{int}$ by defining $g(t) = a$ if $(a = f(t)) \ \wedge \ c(t)$ and $g(t) = \mathsf{undefined}$ otherwise. That is, $g(t)$ returns the address a satisfying $(a = f(t)) \ \wedge \ c(t)$. Let the integer space $\mathbb{S}$ be $\{g(t) \mid t \in [1..n]\}$, *i.e.* the set of all addresses obtained by applying g on the thread IDs. In a typical CUDA kernel, function g is often an increasing or decreasing function. Without loss of generality we assume g is increasing. Usually the space $\mathbb{S}$ is discrete such that $\forall t \in [1..n) : \exists v : a(i) < v < a(i+1)$. The fact that there exists no t satisfying $a = g(t)$ is equivalent to there exists a t such that a falls between $g(t)$ and $g(t+1)$ (here we need to extend g's definition to $t = 0$ and $t = n + 1$).

$$(\forall t \in [1..n] : \neg(a = g(t))) \Longleftrightarrow (\exists t \in [0\,..\,n] : g(t) < a < g(t+1))$$

Moreover, there exists at most one such t since g is an increasing function. In order to obtain an unquantified verification condition, we can introduce a fresh variable t to eliminate the $\exists$ quantifier to obtain the final verification condition.

$$t \in [0\,..\,n] : g(t) < a < g(t+1) \ \Rightarrow \ P$$

It is not hard to see that the g functions for the two `Transpose` kernels are increasing and their quantified formulas can be converted in this manner. In fact, under valid configurations (*e.g.* the block is of square size), their spaces $\mathbb{S}$ are continuous over the thread IDs; thus, the quantified formulas will never be used and can be safely removed.

7.3.4.1 Fast Bug Hunting

On the other hand, if the quantifier elimination is impossible, then we will further loose the requirement of *proving* the properties: our goal is to locate

the property violation quickly by ignoring the quantified formula. Of course, the checker must be trustworthy such that if it reports a bug, then the kernel is indeed buggy.

Consider the following sequence. Even the quantified formulas are unconvertible, we know conclusively that $P(f(w))$ should be true if both $e_3 = e_4 \wedge p_2$ and $e_2 = e_1 \wedge p_1$ hold. Thus, any violation of the predicate $(e_3 = e_4 \wedge p_2 \wedge e_2 = e_1 \wedge p_1) \Rightarrow P(f(w))$ reveals a real bug. PUG is able to find such bugs fast.

$$p_1 \ ? \ v[e_1] := w; \qquad p_2 \ ? \ v[e_3] := f(v[e_2]); \qquad \text{assert } P(v[e_4])$$

7.3.4.2 Coverage

One may complain that our parameterized method suffers from under-approximation due to the insufficient handling of quantified formulas. Yet our encoding ensures that all (conditional) assignments are covered. With the quantifier elimination technique and those described in Section 7.3.3, all combinations (as conjunctions) of the CAs in different BIs are encoded. In practice PUG will miss none or only very few bugs of many kernels.

7.3.4.3 Omega Test

Omega Test [93] may be used to match the address of a read and the range of a CA by building a relation (over the thread IDs) from the address to the range $\{address \rightarrow range \mid cond\}$. The main advantage is that it will not generate quantified formulas. However, Omega Test only supports linear expressions while nonlinear expressions prevail in CUDA kernels (*e.g.* in the two `Transpose` kernels). Our SMT-based method can be regarded as an alternative to Omega Test to handle nonlinear expressions.

7.3.5 Loops

So far, our method works well for kernels containing no loops. When a loop is present, a naive solution is to fully unroll the loop. However, loop unrolling may not scale, especially with nested loops. Also, the loop bounds may involve symbolic values, making it impossible to perform loop unrolling without assigning concrete values to relevant inputs. Our solution is to align the loops or down-size the iteration space.

The loop problem becomes much less severe in equivalence checking. Typical CUDA optimizations often preserve the loop structures of the source kernel such that we may just need to compare the bodies of the loops. Similar assumption is made in [127]. For example, we can optimize the following loop where *sdata* is a shared array

```
for(unsigned int k = bdim.x / 2; k > 0; k >>= 2) {
  if ((tid.x % (2*k)) == 0)
    sdata[tid.x] += sdata[tid.x + k];
  __syncthreads();
}
```

to the one below by eliminating the slow modulo arithmetic.

```
for(unsigned int k = 1; k < bdim.x; k *= 2) {
  int index = 2 * k * tid.x;
  if (index < bdim.x)
    sdata[index] += sdata[index + k];
  __syncthreads();
}
```

Since the operator + in the body is commutative and associative, the two loop headers can be normalized to be the same. Then, the two respective CAs are as follows, which will be used for equivalence checking discussed in previous sections.

$$s.x \,\%\, (2 * k) = 0 \quad ? \quad s.x := s.x + k$$
$$2 * k * t.x < bdim.x \quad ? \quad 2 * k * t.x := 2 * k * t.x + k$$

When the loop alignment fails, we unroll the loops fully. This happens when optimizations other than memory coalescing and bank conflict elimination are applied. We plan to port the method in [8] to deal with other typical loop transformations.

7.3.5.1 Symmetry Reduction

In many cases, the loop bounds in a CUDA kernel depends on the size of a block. Since the principle [24] of designing a CUDA kernel is to have it run on an arbitrary block size (or with little restriction), we are able to reduce the block size to a reasonable value and then run PUG. Currently, finding the appropriate size is done manually. We plan to develop an automatic symmetry reduction approach to identify, for a property p, the minimum number of threads n for which p should be checked. When p holds on n, then p holds on all n' such that $n' > n$.

7.4 Experimental Results (Equivalence Checking)

The parameterized checker uses Z3 [126] as the SMT solver. Z3' expressions are based on bit vectors (bounded integers); thus, the solving time depends on the number of bits.

We performed experiments on a laptop with an Intel Core(TM)2 Duo 1.60GHz processor and 2GB memory to check some representative kernels in CUDA SDK 2.0 Suite [23], each of which contains both unoptimized and optimized kernels. Table 7.1 shows the SMT solving time in seconds. Here n denotes the number of GPU threads. The Transpose kernels are not equivalent when n is not a square of a number; we mark these cases with the $*$. The reduction kernels contains loops whose upper bounds depend on n, making the generic method blow up on n. Notation $16b$ indicates that 16-bit bit-vectors are used; T.O denotes Time Out (> 5 minutes). These benchmark programs contains intensive multiplication operations, thus are quite sensitive to the size of bit-vectors. This may cause even the parameterized method to time out. In this case, we concretize some symbolic variables (*i.e.* give them concrete values, indicated by the "+C." flag) and then compare the results.

Our testing addresses two kinds of bugs. The first kind is due to incorrect configurations for running the kernels. For example, the block size for the Tranpose kernel is not square; or the value of ACCN is not the power of 2 in the Scalar Product kernel. PUG is able to reveal these hidden implications by reporting the bugs. The second kind is the bugs we introduce intentionally to the correct kernels, *e.g.* by modifying the addresses of accesses on shared variables or the guards of conditional statements.

Table 7.2 compares the two approaches on finding the bug taken the longest time to locate. Not surprisingly, the parameterized method shows dramatic improvements.

Table 7.1: Comparing the two methods in equivalence checking.

Kernel	**Nonparameterized**				**Parameterized**	
	n = 4	8	16(+C.)	32(+C.)	-C.	+C.
Transpose (8b)	<1	<1*	7.3	15.4*	T.O	<0.1
Transpose (16b)	28	<1*	T.O(1.2)	37(14.3)*	T.O	<0.1
Transpose (32b)	T.O	1.5*	T.O(4.3)	T.O(31)	T.O	0.16
Reduction (8b)	1	41	T.O(T.O)	T.O(T.O)	0.2	0.2
Reduction (12b)	21	T.O	T.O	T.O	15	11

Table 7.2: Comparing the two methods in bug finding.

Method / Kernel	**Transpose**		**Reduction**		
	16b	32b	8b	16b	32b
Nonparam.(n = 4)	0.16	0.54	0.2	0.3	0.8
Nonparam.(n = 8)	0.53	1.8	3.4	7	9.2
Nonparam.(n = 16)	2.7	7.9	T.O	T.O	T.O
Parameterized	<0.1	0.26	<0.1	<0.1	0.1

PUG has checked more kernels than shown in above tables, some of which come from a GPGPU class recently taught in the University of Utah. Although they are small-medium size programs (typically 50-200 lines of code), it is nontrivial to verify these highly optimized parallel programs. Furthmore, even for a small kernel, loop unrolling often results in many CAs to be checked. Encouragingly, PUG is able to identify the bugs (if any) within few seconds. For kernels involving floating-point operations, we plan to extend PUG by incorporating SMT solver's support for real numbers.

7.5 Discussions

Our checker presented in this chapter is the first parameterized checker for GPGPU kernels. Particularly, the parameterized method is highly scalable for identifying the semantics discrepancy between kernels. In addition to finding better ways to handle quantified formulas, we plan to extend PUG to dealing with more complicated programs. For equivalence checking, we plan to deal with nontrivial loop transformations.

CHAPTER 8

SUMMARY AND FUTURE WORK

We have demonstrated how to model and formally validate programs with various computation and communication models. The techniques we use for formal analysis include theorem proving, model checking, and constraint solving.

8.1 Comparing Formal Analysis Techniques

In Table 8.1, we give a coarse comparison (which may be biased) of these techniques based on our experience so far.

Each method requires building a model of the program to be analyzed. Since we can define almost everything in the higher-order logic of HOL, theorem proving is the most general one. A model checker usually provides a specification language expressive enough to model a variety of systems; however, it often scarifies expressiveness for better performance. SMT solvers only support a few theories such as those for integers, arrays, and uninterpreted functions; and they have great difficulty in dealing with quantified formulas.

A theorem prover like HOL contains a tiny kernel consisting of only a few inference rules; this kernel is ensured to be sound with respect to the logic. When modeling a system, a user gives a small set of axioms, from which all formal statements are derived. To guarantee the accuracy of the model of the entire system, one just needs to examine these basic axioms. For example, in order to reason about ARM programs, we first define the operational semantics of the ARM assembly language, then derive from this semantics a large set of rules (*e.g.*

Table 8.1: A comparison of involved formal analysis techniques.

Method	Theorem Proving	Model Checking	Constraint Solving
Generality	Good	Fair	Poor
Rigidity	Good	poor	Fair
Productivity	Poor	Fair	Good
Automation	Poor	Good	Good

composition rules for control flow structures) to facilitate the reasoning.

Model checking and constraint solving, on the other hand, require defining the entire model explicitly. No derivation is needed to obtain higher level views of the programs. For example, we can encode all concurrent behaviors of a CUDA kernel into a single formula, then perform all subsequent analyses on this formula. Unfortunately, building the model for a large system is tedious and error prone. As a compensation, constraint solvers can check self-consistency properties by modeling all possible cases with a symbolic formula and solving this formula in one go, while (explicit) model checkers can investigate only a subset of test cases.

From a user's perspective, theorem proving is notoriously tedious and time consuming: one not only needs to derive a sufficient set of theorems / rules from the axioms, but also has to manually take care of the solving procedure. The decision procedure in a theorem prover is weak and slow because it relies mainly on term rewriting, *e.g.* reducing a predicate to true or false. What is worse, the prover has to pick the right rules and apply them in the right order to obtain meaningful results. Because of these limitations, our trusted compiler is incapable of handing large and complex programs. In contrast, model checkers and constraint solvers are designed for automated checking and reasoning, and they also allow the users to define simple heuristics to guide the solving procedure. Since model checking suffers more from the space space blow-up problem, it requires more user intervention in order to search the state space wisely. In fact, partial order reduction along with symmetric reduction is a must for scaling model checking to large systems.

8.2 Future Work

There are plenty of possibilities to augment our work on building trusted compilers. For instance, it is desirable for the front-end to accept more advanced language features such as modules and objects. As some transformations are verified in a *transition validation* style, it is better to prove them once and for all in a *verifying compiler* manner. Furthermore, we would like to move from handling sequential programs to concurrent programs, *e.g.* construct and verify parallelizing compilers for shared memory architectures.

As for MPI programs, while we have been relatively happy with TLA+ as a specification language, much of the value we derived from TLA+ is from the

accompanying model checker TLC which uses the explicit state enumeration technology to calculate reachable states. Such tools cannot be used to calculate the outcome of general scenarios such as these: "what will happen if we initialize an MPI runtime to a state satisfying a high-level predicate and some partially specified symbolic inputs are applied?" Therefore, it would also be of interest to explore the use of symbolic reasoning capabilities in conjunction with API specifications. For example, we may use theorem proving to reason about MPI programs in the Isabelle/TLA+ setting [114].

Our focus in the near future is to improve the symbolic checker for GPU programs. First of all, we need to overcome the limitations pertaining to the calling context of CUDA kernels. Any method for obtaining automatically the constraints on calling context can help improve the degree of automation. And, loop invariant discovery methods that determine the loop refinement annotations will also enhance the usability of PUG.

Although traditional testing methods are ineffective at locating CUDA bugs because they assume concrete input values, they are extremely valuable in measuring the performance (*e.g.* the exact measurement of bank conflicts and global memory coalesce) and helping users to debug the programs interactively. We are developing a dynamic symbolic executor which executes the code directly rather than applying static analysis. It contains light-weight runtime code to check properties and measure the performance. In contrast to traditional testing tools, it allows program inputs to have symbolic values, and extends the memory model and the execution model to support symbolic values. This symbolic executor is able to automatically generate test cases with high coverage guarantee.

REFERENCES

[1] Abadi, M., Lamport, L., and Merz, S. A TLA solution to the RPC-memory specification problem. In *Formal System Specification: The RPC-Memory Specification Case Study* (1996), M. Broy, S. Merz, and K. Spies, Eds., vol. 1169 of *LNCS*, Springer-Verlag, pp. 21–66.

[2] Aiken, A., and Gay, D. Barrier inference. In *Symposium on the Principles of Programming Languages (POPL)* (1998), pp. 342–354.

[3] Allen, R., and Kennedy, K. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[4] Appel, A. W. Foundational proof-carrying code. In *Proc. 16th IEEE Symposium on Logic in Computer Science (LICS)* (2001), IEEE Computer Society, p. 247.

[5] Arons, T., Pnueli, A., Ruah, S., Xu, J., and Zuck, L. D. Parameterized verification with automatically computed inductive assertions. In *Computer Aided Verification (CAV)* (2001), pp. 221–234.

[6] Abstract State Machines. `http://www.eecs.umich.edu/gasm/`.

[7] Augustsson, L. Compiling pattern matching. In *Conference on Functional Programming Languages and Computer Architecture* (1985), pp. 368–381.

[8] Barrett, C. W., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A., and Zuck, L. D. TVOC: A translation validator for optimizing compilers. In *Computer Aided Verification (CAV)* (2005), pp. 83 – 94.

[9] Batson, B., and Lamport, L. High-level specifications: Lessons from industry. In *Formal Methods for Components and Objects (FMCO)* (2002), pp. 242–261.

[10] Benton, N., and Hur, C.-K. Biorthogonality, step-indexing and compiler correctness. In *ACM SIGPLAN International Conference on Functional programming (ICFP)* (2009), pp. 97–108.

[11] Benton, N., and Zarfaty, U. Formalizing and verifying semantic type soundness of a simple compiler. In *9th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP)* (2007), pp. 1 – 12.

[12] Birkedal, L., Tofte, M., and Vejlstrup, M. From region inference to von neumann machines via region representation inference. In *Symposium on Principles of Programming Languages (POPL)* (1996), pp. 171 – 183.

[13] Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., and Wansbrough, K. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *SIGCOMM* (2005), pp. 265–276.

[14] Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., and Wansbrough, K. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *Symposium on the Principles of Programming Languages (POPL)* (2006), pp. 55–66.

[15] Blazy, S., Dargaye, Z., and Leroy, X. Formal verification of a C compiler front-end. In *14th International Symposium on Formal Methods (FM), Hamilton, Canada* (2006).

[16] Blazy, S., and Leroy, X. Formal verification of a memory model for C-like imperative languages. In *International Conference on Formal Engineering Methods (ICFEM), Manchester, UK* (2005).

[17] Boehm, H.-J. Threads cannot be implemented as a library. In *Programming language design and implementation (PLDI)* (2005), pp. 261 – 268.

[18] Broy, M., Hinkel, U., Nipkow, T., Prehofer, C., and Schieder, B. Interpreter verification for a functional language. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)* (1994), pp. 77–88.

[19] Chapman, B., Jost, G., and Pas, R. v. *Using OpenMP.* MIT Press, 2008.

[20] Chlipala, A. A certified type-preserving compiler from lambda calculus to assembly language. In *Conference on Programming Language Design and Implementation (PLDI)* (2007).

[21] Chlipala, A. A verified compiler for an impure functional language. In *Symposium on the Principles of Programming Languages (POPL)* (2010), pp. 93–106.

[22] Clarke, E. M., Talupur, M., and Veith, H. Proving ptolemy right: The environment abstraction framework for model checking concurrent systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2008), pp. 33–47.

[23] CUDA Zone. `http://www.nvidia.com/object/cuda\_home.html`.

[24] CUDA Programming Guide Version 1.1, `http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf`.

[25] Dave, M. A. Compiler verification: a bibliography. *ACM SIGSOFT Software Engineering Notes 28*, 6 (2003), 2.

[26] Dold, A., Gaul, T., Vialard, V., and Zimmermann, W. ASM-based mechanized verification of compiler back-ends. In *Workshop on Abstract State Machines* (1998), pp. 50–67.

[27] Eijk, P. V., and Diaz, M., Eds. *Formal Description Technique Lotos: Results of the Esprit Sedos Project*. Elsevier Science Inc., New York, NY, USA, 1989.

[28] Emerson, E. A., and Namjoshi, K. S. Reasoning about rings. In *Symposium on the Principles of Programming Languages (POPL)* (1995), pp. 85–94.

[29] Feng, M., and Leiserson, C. E. Efficient detection of determinacy races in Cilk programs. In *Parallel Algorithms and Architectures (SPAA)* (1997), pp. 1–11.

[30] Next Generation CUDA Architecture (Fermi), `http://www.nvidia.com/object/fermiarchitecture.html`.

[31] Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. The essence of compiling with continuations. In *Conference on Programming Language Design and Implementation (PLDI)* (1993), pp. 237 – 247.

[32] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting (PVM/MPI)* (2004), pp. 97–104.

[33] Geist, A., 2008. MPI Must Evolve or Die. Invited Talk Given at EuroPVM/MPI 2008.

[34] Georgelin, P., Pierre, L., and Nguyen, T. A formal specification of the MPI primitives and communication mechanisms. Tech. rep., LIM, 1999.

[35] Goerigk, W., Dold, A., Gaul, T., Goos, G., Heberle, A., von Henke F., U., H., Langmaack, H., Pfeifer, H., Ruess, H., and Zimmermann, W. Compiler correctness and implementation verification: The verifix approach. In *Poster Session of CC'96. IDA Technical Report LiTH-IDA-R-96-12, Linkoeping, Sweden* (1996).

[36] Gordon, M., Iyoda, J., Owens, S., and Slind, K. Automatic formal synthesis of hardware from higher order logic. In *Proceedings of Fifth International Workshop on Automated Verification of Critical Systems (AVoCS)* (2005), vol. 145 of *ENTCS*, pp. 27–43.

[37] Gropp, W., Lusk, E. L., Doss, N. E., and Skjellum, A. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing 22*, 6 (1996), 789–828.

[38] Gropp, W. D. Learning from the success of MPI. In *8th International Conference High Performance Computing (HiPC)* (2001), pp. 81–92.

[39] Hanna, Y., Basu, S., and Rajan, H. Behavioral automata composition for automatic topology independent verification of parameterized systems. In *7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)* (2009), pp. 325–334.

[40] Hannan, J., and Pfenning, F. Compiler verification in LF. In *Proceedings of the 7th Symposium on Logic in Computer Science (LICS)* (1992).

[41] Harrison, J. Formal verification of square root algorithms. *Formal Methods in System Design 22*, 2 (Mar. 2003), 143–154.

[42] Herlihy, M., and Shavit, N. *The Art of Multiprocessor Programming*. Morgan Kauffman, 2008.

[43] Hickey, J., and Nogin, A. Formal compiler construction in a logical framework. *Journal of Higher-Order and Symbolic Computation 19*, 2-3 (2006), 197–230.

[44] The HOL-4 Theorem Prover. `http://hol.sourceforge.net/`.

[45] Holzmann, G. The model checker SPIN. *IEEE Transactions on Software Engineering 23*, 5 (May 1997), 279–295.

[46] IEEE. IEEE Standard for Radix-independent Floating-point Arithmetic, ANSI/IEEE Std 854-1987, 1987.

[47] Ct: C for Throughput Computing. `http://techresearch.intel.com/articles/Tera-Scale/1514.htm`.

[48] J. Boyle, R. R., and Winter, K. Do you trust your compiler? applying formal methods to constructing high-assurance compilers. In *High-Assurance Systems Engineering Workshop* (1997).

[49] Jackson, D. Alloy: A new technology for software modeling. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2002), vol. 2280 of *LNCS*, pp. 175–192.

[50] Jackson, D., Schechter, I., and Shlyahter, H. Alcoa: the ALLOY constraint analyzer. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering* (2000), pp. 730–733.

[51] Jaffer, A., 2007. A Formal Semantics of Scheme. `http://swissnet.ai.mit.edu/~jaffer/r5rs-formal.pdf`.

[52] Jiuxing Liu, Jiesheng Wu, D. K. P. High performance RDMA-based MPI implementation over infiniband. *International Journal of Parallel Programming 32*, 3 (2004), 167–198.

[53] Kirk, D. B., and mei W. Hwu, W. *Programming Massively Parallel Processors*. Morgan Kauffman, 2010.

[54] Klein, G., and Nipkow, T. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS) 28*, 4 (2006), 619–695.

[55] Kuchera, W., and Wallace, C. Toward a programmer-friendly formal specification of the UPC memory model. Tech. Rep. 03-01, Michigan Technological University, 2003.

[56] Leijens, D., and Schulte, W., 2008. The Design of a Task Parallel Library. `http://research.microsoft.com/apps/pubs/default.aspx?id=77368`.

[57] Leinenbach, D., Paul, W., and Petrova, E. Towards the formal verification of a C0 compiler: Code generation and implementation correctnes. In *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM)* (2005), pp. 2 – 12.

[58] Leroy, X. Formal certification of a compiler backend, or: programming a compiler with a proof assistant. In *Symposium on the Principles of Programming Languages (POPL)* (2006), ACM Press, pp. 42 – 54.

[59] Li, G., DeLisi, M., Gopalakrishnan, G., and Kirby, R. M. Formal specification of the MPI-2.0 standard in TLA+. In *13th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)* (2008), pp. 283–284.

[60] Li, G., and Gopalakrishnan, G. Technical Report and PUG Tool Download: `http://www.cs.utah.edu/fv/PUG`.

[61] Li, G., and Gopalakrishnan, G. Scalable SMT-based verification of GPU kernel functions. In *18th ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT FSE)* (2010).

[62] Li, G., Gopalakrishnan, G., Kirby, R. M., and Quinlan, D. A symbolic verifier for CUDA programs. In *15th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)* (2010), pp. 357–358.

[63] Li, G., Owens, S., and Slind, K. Structure of a proof-producing compiler for a subset of higher order logic. In *16th European Symposium on Programming (ESOP)* (2007), pp. 205–219.

[64] Li, G., Palmer, R., DeLisi, M., Gopalakrishnan, G., and Kirby, R. M. Formal specification of MPI 2.0: Case study in specifying a practical concurrent programming API. Tech. Rep. UUCS-09-003, University of Utah, 2009. `http://www.cs.utah.edu/research/techreports/2009/pdf/UUCS-09-003.pdf`.

[65] Li, G., Palmer, R., DeLisi, M., Gopalakrishnan, G., and Kirby, R. M. Formal specification of MPI 2.0: Case study in specifying a practical concurrent programming API. *Science of Computer Programming 75* (2010).

[66] Li, G., and Slind, K. Compilation as rewriting in higher order logic. In *21th Conference on Automated Deduction (CADE-21)* (2007), pp. 19–34.

[67] Li, G., and Slind, K. Trusted source translation of a total function language. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2008), pp. 471–485.

[68] Liang, C. C. Compiler construction in higher order logic programming. In *4th International Symposium on Practical Aspects of Declarative Languages (PADL)* (2002), pp. 47 – 63.

[69] Lublinerman, R., and Tripakis, S. Checking equivalence of SPMD programs using non-interference. Tech. Rep. UCB/EECS-2009-42, EECS, Berkeley, Mar 2009.

[70] The Maude System. http://maude.cs.uiuc.edu/.

[71] Multicore Communications API. http://www.multicore-association.org.

[72] McCarthy, J., and Painter, J. Correctness of a compiler for arithmetic expressions. In *Symposium in Applied Mathematics* (1967), vol. 19.

[73] Meyer, T., and Wolff, B. Tactic-based optimized compilation of functional programs. In *Types for Proofs and Programs Workshop (TYPES)* (2004), pp. 201–214.

[74] Milner, R., Tofte, M., Harper, R., and MacQueen, D. *The Definition of Standard ML, Revised Edition*. MIT Press, 1997.

[75] The MLton Standard ML Compiler. http://mlton.org.

[76] Moore, J. S. Piton: A verified assembly level language. Tech. rep., CLI-22, CLInc, 1988.

[77] Moore, J. S. A grand challenge proposal for formal methods: A verified stack. In *10th Anniversary Colloquium of UNU/IIST* (2002), pp. 161–172.

[78] MPI: A Message-Passing Interface Standard Version 2.1 Up to date specifications are at http://www.mpi-forum.org.

[79] Myreen, M. O., Slind, K., and Gordon, M. J. C. Extensible proof-producing compilation. In *18th International Conference on Compiler Construction (CC)* (2009), pp. 2–16.

[80] Necula, G. C. Translation validation for an optimizing compiler. In *Conference on Programming Language Design and Implementation (PLDI)* (2000), pp. 83–94.

[81] Necula, G. C., and Rahul, S. P. Oracle-based checking of untrusted software. In *Symposium on the Principles of Programming Languages (POPL)* (2001), pp. 142–154.

[82] Nielson, F., Nielson, H. R., and Hankin, C. *Principles of Program Analysis*. Springer-Verlag, 1999.

[83] Nipkow, T., Paulson, L. C., and Wenzel, M. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, vol. 2283 of *LNCS*. Springer, 2002.

[84] Norrish, M. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998. http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-453.pdf.

[85] Owre, S., Rushby, J. M., and Shankar, N. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE-11)* (1992), vol. 607 of *Lecture Notes in Artificial Intelligence*, pp. 748–752.

[86] Palmer, R., Delisi, M., Gopalakrishnan, G., and Kirby, R. M. An approach to formalization and analysis of message passing libraries. In *Formal Methods for Industry Critical Systems (FMICS)* (2007), pp. 164–181.

[87] Palmer, R., Gopalakrishnan, G., and Kirby, R. M. Semantics Driven Dynamic Partial-order Reduction of MPI-based Parallel Programs. In *ACM workshop on Parallel and Distributed Dystems: Testing and Debugging (PADTAD)* (2007), pp. 43 – 53.

[88] Pervez, S., Gopalakrishnan, G., Kirby, R. M., Thakur, R., and Gropp, W. Formal methods applied to high-performance computing software design: a case study of mpi one-sided communication-based locking. *Softw., Pract. Exper. 40*, 1 (2010), 23–43.

[89] Pike, L. Real-time system verification by k-induction. Tech. Rep. TM-2005-213751, NASA Langley Research Center, May 2005. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/reint.html`.

[90] Pnueli, A., Ruah, S., and Zuck, L. D. Automatic deductive verification with invisible invariants. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2001), pp. 82–97.

[91] Pnueli, A., Siegel, M., and Singerman, E. Translation validation. In *4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)* (1998), pp. 151 – 166.

[92] Pnueli, A., Xu, J., and Zuck, L. D. Liveness with (0, 1, infty)-counter abstraction. In *Computer Aided Verification (CAV)* (2002), pp. 107–122.

[93] Pugh, W. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *ACM/IEEE conference on Supercomputing (SC)* (1991), pp. 4 – 13.

[94] Reinders, J., 2008. Intel Thread Building Blocks.

[95] Reynolds, J. C. Separation logic: A logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science (LICS)* (2002), pp. 55–74.

[96] Rinard, M., and Marinov, D. Credible compilation with pointers. In *Proc. FLoC Workshop on Run-Time Result Verification* (1999).

[97] Robert S. Boyer, Y. Y. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM (JACM) 43*, 1 (1996), 166 – 192.

[98] The ROSE Compiler. `http://www.rosecompiler.org/`.

[99] Saabas, A., and Uustalu, T. A compositional natural semantics and hoare logic for low-level languages. *Theoretical Computer Science 373*, 3 (2007), 273–302.

[100] Symbolic Analysis Laboratory (SAL), `http://sal.csl.sri.com/`.

[101] Sampaio, A. *An Algebraic Approach to Compiler Design, volume 4 of AMAST Series in Computing*. World Scientific, 1997.

[102] Shashidhar, K. C., Bruynooghe, M., Catthoor, F., and Janssens, G. Verification of source code transformations by program equivalence checking. In *14th Conference on Compiler Construction (CC)* (2005), pp. 221–236.

[103] Siegel, S. F. Model Checking Nonblocking MPI Programs. In *8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)* (2007), pp. 44–58.

[104] Siegel, S. F., and Avrunin, G. Analysis of MPI programs. Tech. Rep. UM-CS-2003-036, Department of Computer Science, University of Massachusetts Amherst, 2003.

[105] Siegel, S. F., and Avrunin, G. S. Modeling wildcard-free MPI programs for verification. In *Principles and Practices of Parallel Programming (PPoPP)* (2005), pp. 95–106.

[106] Slind, K. *Reasoning about Terminating Functional Programs.* PhD thesis, Institut für Informatik, Technische Universität München, 1999. `http://tumb1.biblio.tu-muenchen.de/publ/diss/in/1999/slind.html`.

[107] Satisfiability Modulo Theories Competition (SMT-COMP). `http://www.smtcomp.org/2009`.

[108] Squyres, J. M., and Lumsdaine, A. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting (PVM/MPI)* (2003), pp. 379–387.

[109] Strecker, M. Formal verification of a Java compiler in isabelle. In *18th International Conference on Automated Deduction (CADE-18)* (2002), pp. 63–77.

[110] Tan, G., and Appel, A. W. A compositional logic for control flow. In *17th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)* (2006), LNCS, pp. 80–94.

[111] The Message Passing Interface Forum, 1995. MPI: A Message-Passing Interface Standard. `http://www.mpi-forum.org/docs/`.

[112] TLA - The Temporal Logic of Actions. `http://research.microsoft.com/users/lamport/tla/tla.html`.

[113] Leslie Lamport, The Win32 Threads API Specification. `http://research.microsoft.com/users/lamport/tla/threads/threads.html`.

[114] TLA+ as an Isabelle object logic. `http://www.loria.fr/~merz/stages/InternshipIsabelleTLA.html`.

[115] Tolmach, A., and Oliva, D. P. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming 8*, 4 (1998), 367 – 412.

[116] Träff, J. L., Gropp, W., and Thakur, R. Self-consistent MPI performance requirements. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface (PVM/MPI)* (2007), pp. 36–45.

[117] Tristan, J.-B., and Leroy, X. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *Symposium on the Principles of Programming Languages (POPL)* (2008), pp. 17–27.

[118] Vakkalanka, S., Gopalakrishnan, G., and Kirby, R. M. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In *20th International Conference on Computer Aided Verification (CAV)* (2008), pp. 66–79.

[119] van den Brand, M., Heering, J., Klint, P., and Olivier, P. A. Compiling language definitions: The ASF+SDF compiler. *ACM Transactions of Programming Language Systems 24*, 4 (2003), 334–368.

[120] Verdoolaege, S., Janssens, G., and Bruynooghe, M. Equivalence checking of static affine programs using widening to handle recurrences. In *Computer Aided Verification (CAV)* (2009), pp. 599–613.

[121] Vo, A., Vakkalanka, S. S., Delisi, M., Gopalakrishnan, G., Kirby, R. M., and Thakur, R. Formal verification of practical mpi programs. In *14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2009), pp. 261–270.

[122] Wheeler, D., and Needham, R. TEA, a tiny encryption algorithm. In *Fast Software Encryption: Second International Workshop* (1999).

[123] Winter, V. L. Program transformation in hats. In *Proceedings of the Software Transformation Systems Workshop* (1999).

[124] Yices: An SMT Solver. `http://yices.csl.sri.com`.

[125] Young, W. D. Verified compilation in micro-Gypsy. In *International Symposium on Software Testing and Analysis (ISSTA)* (1989), Springer-Verlag, pp. 20 – 26.

[126] Z3: An SMT solver. `http://research.microsoft.com/en-us/um/redmond/projects/z3`.

[127] Zaks, A., and Pnueli, A. CoVaC: Compiler validation by program analysis of the cross-product. In *15th International Symposium on Formal Methods (FM)* (2008), pp. 35–51.

[128] Zimmermann, W., and Gaul, T. On the construction of correct compiler back-ends: An ASM-approach. *Journal of Universal Computer Science 3*, 5 (1997), 504–567.