Towards the Pervasive Formal Verification of Multi-Core Operating Systems and Hypervisors Implemented in C

Dissertation zur Erlangung des Grades des Doktors der Ingenieurswissenschaften (Dr.-Ing.) der Naturwissenschaftlich-Technischen Fakultäten der Universität des Saarlandes

vorgelegt von

Sabine Bettina Schmaltz

Saarbrücken, Dezember 2012

UNIVERSITÄT DES SAARLANDES

Institut für Rechnerarchitektur und Parallelrechner,
Universität des Saarlandes, 66041 Saarbrücken

Hiermit erkläre ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Saarbrücken, im Dezember 2012

Tag des Kolloquiums	10. Mai 2013
Dekan	Prof. Dr. Mark Groves
Prüfungsausschuss	
Vorsitz	Prof. Dr. Reinhard Wilhelm, Universität des Saarlandes
1. Gutachter	Prof. Dr. Wolfgang J. Paul, Universität des Saarlandes
2. Gutachter	Prof. Dr. Kurt Mehlhorn, Universität des Saarlandes
Akademischer Mitarbeiter	Dr. Mikhail Kovalev, Universität des Saarlandes

Abstract

Short Abstract

This thesis deals with a semantic model stack for verification of functional correctness of multi-core hypervisors or operating systems. In contrast to implementations based on single-core architectures, there are additional features and resulting challenges for verifying correctness properties in the multi-core case, e.g. weak memory models (store buffers), or an inter processor interrupt mechanism.

The Verisoft XT project had the goal of verifying correctness of the Microsoft Hyper-V hypervisor and achieved great code verification results using the concurrent C verification tool VCC developed by our project partners during the project. A sound mathematical theory to support code verification was not established.

To remedy this shortcoming, we sketch a model stack for a simplified multi-core architecture based on a simplified MIPS model for system programmers and illustrate on a high level of abstraction how to obtain a simulation between neighboring models. A hardware model for this architecture is formalized at a detailed level of abstraction of the model stack. In addition, this thesis provides operational semantics for a quite simple intermediate language for C as well as an extension of this semantics with specification (ghost) state and code which can serve as a basis for arguing the soundness of VCC. Due to the powerful nature of specification code, a simulation between annotated and original program is not trivial. Thus, we give a pencil and paper proof.

Kurzzusammenfassung

Diese Arbeit befasst sich mit einem semantischen Modell-Stack für die Verifikation der Korrektheit von Multi-Core Hypervisoren oder Betriebssystemen. Im Gegensatz zu auf Implementierungen auf Single-Core Architekturen stellen sich im Multi-Core Fall zusätzliche Herausforderungen für die Verifikation von Korrektheitseigenschaften, z.B. durch schwache Speichermodelle oder die Existenz eines Inter-Prozessor-Interrupt Mechanismus.

Das Verisoft XT Projekt, welches zum Ziel hatte die Korrektheit des Microsoft Hyper-V Hypervisors zu verifizieren, erreichte unter Benutzung des Verifikationstools VCC hervorragende Resultate im Bereich der Codeverifikation. Die Erstellung einer fundierten mathematischen Theorie um diese Resultate zu untermauern wurde vernachlässigt.

Um diesen Mangel zu beheben, skizzieren wir einen Modell-Stack für eine vereinfachte Multi-Core Architektur basierend auf einem vereinfachten MIPS-Modell für Systemprogrammierer und illustrieren wie eine Simulation zwischen benachbarten Modellen erreicht wird. Ein Hardwaremodell für die Architektur wird auf einer detaillierten Abstraktionsebene präsentiert. Zusätzlich enthält diese Arbeit die operationale Semantik einer Zwischensprache für C und deren Erweiterung um Spezifikationszustand und -code welche als Basis für einen Korrektheitsbeweis des Tools VCC dienen kann. Da aufgrund der mächtigen Spezifikationssprache eine Simulation zwischen annotiertem und originalem Programm nicht trivial ist, führen wir den Beweis auf Papier.

Acknowledgements

I would like to thank all those people without whom it would have been impossible to write this thesis. This includes all those involved in the Hyper-V verification group of the Verisoft XT project and, in particular, those who remained and worked on our shared vision of a multi-core model stack at the chair of Prof. Paul after the project ended. We had so many useful discussions both during and after the project – the feedback on my semantic models and the collaboration in our group were invaluable to developing the models to a point that it is quite certain that they are at least adequate to the tasks we intend to use them and have used them in.

I would like to specifically thank Prof. Paul for advising this thesis, my husband for supporting me during these years, and my baby son for first providing a firm – albeit not very predictable – deadline and for being a calm and content baby which allowed me to complete this thesis during maternity leave.

Saarbrücken, December 20th, 2012 Sabine Schmaltz

Contents

1 Introduction

Every time a new big software failure is added to the list of most catastrophic software failures, the public and experts alike require to learn what caused the failure, and, even more important, how it could have been prevented. In the worst cases, people died as a direct consequence of software failure (e.g. the incidents revolving around the radiation therapy machine Therac-25 [Lev93], or Multidata Systems International's radiation therapy software [Age01]), while, in other cases, companies or governments lost hundreds of millions of dollars (e.g. as the result of the explosion of Ariane 5 flight 501 [Lan97] or Intel's well-known pentium bug [Pra95]). Depending on the public impact, these failures may trigger political campaigns arguing for stricter safety standards, and/or force managers or politicians who were in charge to resign. These examples, however, appear insignificant compared to what indeed could happen when faulty soft- or hardware in highly critical systems like nuclear power plant control systems, or systems that control weapons of mass destruction leads to failure.

Considering the diversity of both application scenarios and systems used for implementing important systems, it is no surprise that the causes for failures are just as multifaceted as the engineering projects themselves. Often, these failing systems have been thoroughly tested or simulated. Even when performing tests under the right assumptions, the failure case might never occur during the testing phase of the development process. Many interesting systems that occur in practice are too complex to be tested exhaustively. Thus, often, when a system passes all tests, this provides no guarantee that it is indeed correct.

A solution frequently proposed by the academic community is to apply tools based on formal methods – formal verification tools. These tools can provide hard guarantees derived mathematically based on an abstract representation of the system. One necessary requirement on formal verification tools is that they must be sound, that is, if a property about the system is established by them, this property indeed holds for each execution of the system. The qualitative difference to testing lies in coverage: with most testing approaches, it can only be established that all runs that were considered during the testing phase are, in some sense, good runs. Using formal verification tools, it can be proven that all runs of a system obey the formal specification of the system.

Formal methods are now finding their way into the development cycles of industry projects [WLBF09]. However, very often, what is considered for formal verification does not comprise the whole system, and thus, results gained are inherently limited by this choice. After all, formal methods can merely establish correctness with respect to an underlying mathematical model. If that model is not correct, verification results are flawed. Code-verification tools, for example, generally assume the model of computation given by the programming language's semantics. This is not a bad choice, however: building a system that uses a faulty compiler to compile the verified code is likely to result in a faulty system.

To gain a correctness result over the whole system consisting of hard- and software, pervasive theory has to be constructed in such a way that correctness of implementation can be established with respect to some lowest-level underlying model of computation, e.g. the gate-level construction of hardware (or an even lower level model). This is what can earn formal methods the place they could rightfully take in certification. After all, just consider that testing of complete systems – comprising both hard- and software –, by its very nature, is a pervasive approach. Add to that that testing a system from a black-box perspective requires less knowledge about why the system works than does verifying it formally.

However, even if pervasive formal verification is more involved than testing, there are two main things that can be gained from the verification process: A formal specification of the system and the absolute confidence that this specification is obeyed by the system if the lowest-level underlying model of computation is correct and if the verification tool is sound. Essentially, this means, with trusted formal verification tools, testing can be reduced to testing the correctness of the gate-level hardware specification against the physical hardware. Performing those hardware-tests however, is still necessary, since, in current state-of-the-art hardware, it often is not even clear during development under what physical conditions the gate-model developed by the hardware designer provides a valid abstraction of the physical hardware. With circuits getting smaller and smaller, we enter the borderlands of particle physics where no applicable mathematical model describing reality to a sufficient degree of accuracy has been found, yet.

Constructing sound pervasive formal verification tools requires a pervasive formal theory of systems, including architecture, compilers, operating systems, and hypervisors. Considering that the components used in today's desktop computers are not even understood in all details by a significant part of their implementors, defining such a theory appears to be a nearly impossible task. Looking at the way how computer systems are built, it is obvious that we can think of them as being comprised of several layers: A hardware layer executes machine code, high-level languages provide more abstract programming models, and operating systems provide execution contexts for user processes and system calls. The connection between layers is given naturally by the way they are implemented: code of high-level languages is translated by compilers to machine code, and operating systems are implemented by giving the code that realizes them. With each layer, we associate a model of the system at a different degree of abstraction. A theory that makes use of these models to completely describe the system can be called a pervasive theory of systems. There have been several pervasive system verification efforts till today (e.g. [Moo89, Ver07, App11]) which build on the principle to establish a pervasive formal theory of the system in question.

Our work at the chair of computer architecture and parallel computing at Saarbrücken University over the last years has been driven by the goals of the Verisoft XT project [Ver10], which aimed at extending the pervasive theory developed in the course of its predecessor, the Verisoft project [Ver07], to the multi-core case. Verisoft XT is a three-year research project funded by the German Federal Ministry of Education and Research (BMBF) that ran from 2007 till 2010. As a case-study, we tried to verify formal correctness of the Microsoft Hyper-V hypervisor [LS09] using the verification tool VCC – a tool for verification of concurrent C code that makes use of specification state and code – developed by Microsoft Research [CDH$^+$09, DMS$^+$09] during the Verisoft XT project. The assumption was that the extension of the sequential theory of systems to the multi-core case would be reasonably straight forward, and thus, most resources were assigned to code verification. While code verification progressed, however, it became evident that extending the theory was highly nontrivial. There were certain parts of the system where, with existing theory, we could not argue that applying VCC in a certain way is actually sound: We were lacking a pervasive theory of multi-core systems. In that regard, our project failed to achieve all of its goals. On the other hand, we left this project with quite a few insights which – had we had them right at the beginning – could have made the project an even more noteworthy success.

Structure and content of this thesis The main goal of this thesis is to shed some light on the models that eventually need to be defined, applied, arranged, and integrated in a certain way such that a pervasive theory of modern multi-core architectures can be established by providing a model-stack of the system. It aims at providing the theory that is needed in order to verify pervasive formal correctness of operating systems and hypervisors, which, due to their potentially safety-critical nature, are interesting targets for formal methods. We provide semantic models to be used at different levels of abstraction and sketch how they are to be used in a pervasive model stack.

We start by defining the basic notation used in the remainder of this thesis in chapter 2.

Chapter 3 provides a high-level building plan of a model-stack for multi-core hypervisor verification – from detailed hardware models up to programming language semantics. Extending sequential hardware verification results to the concurrent case faces a number of interesting challenges. Multi-core architectures often offer features that are not meaningful in single core architectures, e.g. distributed caches maintained using cache-coherence protocols (e.g. MOESI) or advanced programmable interrupt controllers (APICs) for inter-processor-interrupt management. Hardware features that are invisible in the single-core case can suddenly become visible in the multi-core case: Consider the weak memory models that result from the use of store-buffers and memory systems that reorder accesses. Some architectures, like the x64-architectures offered by Intel and AMD considered in Verisoft XT, even provide built-in support for virtualization: e.g. tagged TLBs, nested paging, and an intercept mechanism that facilitates running guest operating systems natively on the machine.

In chapter 4, we provide a simple MIPS model extended with some features of the x64-architecture. We we call the resulting model MIPS-86. It was developed based on existing model fragments to better illustrate the nature of the reduction theorems presented in the previous chapter and, more importantly, to serve as an overall specification of the reverse-engineered gate-level hardware models of [Pau12]. Essentially, the MIPS-86 model is a single ”horizontal” slice of the model stack presented in chapter 3. It is designed in such a way that extension to higher or lower levels of the model stack can be achieved. Subsection 3.2.2 explains how the given MIPS-86 model fits into the overall theory.

Chapter 5 contains the definition of an intermediate-language for C and its formal operational semantics. Features of the language are a byte-addressable memory which allows pointer arithmetics in combination with a frame-based stack abstraction that describes local variable content and the control-flow state. We name the language *C-IL*, not to be confused with already existing intermediate-languages. The place of *C-IL* in the model stack is described in subsection 3.4.3.

In chapter 6, we define a model of specification state and code extending the C-intermediate-language introduced in chapter 5. Specification state – also called ghost state or auxiliary state – is commonly used in concurrent verification tools and helps to guide first-order provers to finding proofs. The VCC tool used in the Verisoft XT project makes use of ghost state and a first-order prover to discharge the generated verification conditions. Formally specifying the rich ghost state model offered by VCC and specifying formal operational semantics of it is a useful step towards a soundness proof for the VCC. We elaborate on this in section 3.7.

We propose future work and provide a conclusion of the thesis in chapter 7, followed by a list of reference work we cite.

Contributions The author can certainly claim both involvement in discussions concerning all the topics covered in this thesis and a significant impact on the formal development of the model stack presented in chapter 3 – which is, in fact, joint work over a period of five years with Prof. Wolfgang Paul and Ernie Cohen as well as other project members of Verisoft XT. The overall theory presented in this thesis cannot be attributed to the author alone: Some of the challenges described in chapter 3 will be or are already solved by the author's colleagues. Pointers to upcoming or already completed dissertations and papers from the Verisoft XT project that provide details on individual topics are given in chapter 3 and chapter 7. The main contributions of the author in the effort of developing this model stack lie in i) providing cornerstone semantic models which are both simple enough to use and detailed enough to argue about the multi-core system and in ii) challenging others to use and confirm or dispute these models – with legitimate disputes resulting in the necessary improvements being made. In fact, the understanding needed to devise the intricate model stack described in chapter 3 of this thesis could only be gained after we understood the nature of the semantic models we need in order to prove a multi-core hypervisor correct. This required defining formal semantics (as given by the models in the chapters 4, 5 and 6 which are provided by the author of this thesis) in the first place. Over the last years, the desire of the author to formally understand the vague ideas regarding individual issues already present during the VerisoftXT project has led to the creation of these models, which in turn led to better insight into the structure and major theorems of the overall theory described in chapter 3. The resulting models appear to provide a reasonable basis for formally establishing the missing parts of the overall theory. In this sense, work on this thesis can be considered a driving force behind the goal of achieving the pervasive model stack we need to justify code verification on a modern multi-core hypervisor.

2 Notation

no·ta·tion *noun*

1. annotation, note
2. **a** : the act, process, method, or an instance of representing by a system or set of marks, signs, figures, or characters
   b : a system of characters, symbols, or abbreviated expressions used in an art or science or in mathematics or logic to express technical facts or quantities

"notation." Merriam-Webster.com. Merriam-Webster, 2011. Web. 6 July 2011.

In the following, we introduce notation and fundamental definitions used in this thesis. This chapter should serve as a reference whenever things appear not to make sense.

2.1 Fundamental Notation

Natural Numbers

We use $\mathbb{N} = \{0, 1, 2, \ldots\}$ to denote the *set of* non-negative integers, also called the set of *natural numbers*.

Integers

$\mathbb{Z}$ denotes the *set of* integers.

Sets

Definition 2.1 (Hilbert-Choice-Operator) Given a set A, the Hilbert-choice-operator $\mathcal{E}$ chooses an element from A:

$$\mathcal{E}A \in A$$

This is particularly useful when the set consists of a single element, i.e.

$$\mathcal{E}\{x\} = x$$

or when a definition does not depend on the specific element chosen.

Definition 2.2 (Cardinality of a Finite Set) We use $\#A$ to denote the amount of elements of a finite set A.

Definition 2.3 (Power Set) Given a set A, we use

$$2^A \overset{def}{=} \{B \mid B \subseteq A\}$$

to denote the power set of A, i.e. the set of all subsets of A.

Functions

Definition 2.4 (Partial Function) We use

$$f : \mathcal{X} \rightharpoonup \mathcal{Y}$$

to denote that f is a partial function from set $\mathcal{X}$ to set $\mathcal{Y}$, i.e. a function $f : \mathcal{X}' \to \mathcal{Y}$ for some $\mathcal{X}' \subseteq \mathcal{X}$.

Definition 2.5 (Function Update) In order to be able to specify functions where a single mapping is redefined, we introduce the notation

$$f(x{:=}y) \overset{def}{=} f'$$

where

$$\forall z : f'(z) = \begin{cases} y & z = x \\ f(z) & \text{otherwise} \end{cases}$$

Definition 2.6 (Union of Functions) Given functions $f : \mathcal{X} \to \mathcal{Y}$ and $g : \mathcal{A} \to \mathcal{B}$ with $\mathcal{X} \cap \mathcal{A} = \emptyset$, we use

$$(f \cup g) : \mathcal{X} \cup \mathcal{A} \to \mathcal{Y} \cup \mathcal{B}$$

to describe the function with the following property:

$$\forall x \in \mathcal{X} \cup \mathcal{A} : (f \cup g)(x) = \begin{cases} f(x) & x \in \mathcal{X} \\ g(x) & \text{otherwise} \end{cases}$$

Record Notation

Throughout this thesis, we will often use tuples to represent, among other things, states of programs or hardware. In order to have a short notation to refer to individual elements of a tuple, we always name them in the same way:

Definition 2.7 (Record Notation for Tuples) Let A be a set which is the Cartesian product of sets $A_1, A_2, \ldots, A_k$ and let $n_1, n_2, \ldots, n_k$ be names for the individual tuple elements of A. Then, given a tuple

$$c \in A \overset{def}{\equiv} A_1 \times A_2 \times \ldots \times A_k$$

$$c = (a_1, a_2, \ldots, a_k)$$

we use $c.n_i$ to refer to a_i – the i-th name refers to the i-th *record field* of the tuple. We use the term *record* to refer to such a named tuple. We tend to introduce records $c \in A$ by defining

$$c = (c.n_1, c.n_2, \ldots, c.n_k)$$

followed by a definition of the types of record fields of c.

Definition 2.8 (Record Update) Sometimes it is convenient to have a short notation to describe that a single element of a record is updated. For this, given a record $c = (a_1, a_2, \dots, a_k) \in A$ with field names $n_1, n_2, \dots, n_k$, we define

$$c[n_i{:=}a] \overset{def}{=} c'$$

where $\forall j \neq i : c'.n_j = c.n_j$ and $c'.n_i = a$ to denote the record where $c.n_i$ is replaced by a while all other fields stay unchanged.

2.2 Sequences

Finite Sequences

Definition 2.9 (n-Tuples Over a Set $\mathcal{X}$) Given a set $\mathcal{X}$, we use

$$\mathcal{X}^n = \underbrace{\mathcal{X} \times \ldots \times \mathcal{X}}_{n \text{ times}} = \{(x_{n-1}, x_{n-2}, \dots, x_0) \mid \forall i \in \{0, \dots, n-1\} : \quad x_i \in \mathcal{X}\}$$

to denote the set of *n-tuples* over the set $\mathcal{X}$.

In this thesis, we will often need to talk about *strings* or *finite sequences* of certain elements. In a formal mathematical sense, these can all be represented with tuples over sets. However, in order to both make our notation easier to read and to keep it as short and flexible as possible, we introduce the following notation on tuples:

Definition 2.10 (Alternative Notation for Tuples) Given an n-tuple $x = (x_{n-1}, x_{n-2}, \dots, x_0)$, we define

$$x_{n-1}x_{n-2}\dots x_0 \overset{def}{=} x$$

$$x[n-1:0] \overset{def}{=} x$$

$$x[i:j] \overset{def}{=} x_i x_{i-1} \dots x_j = (x_i, x_{i-1}, \dots, x_j), \qquad n > i \geq j \geq 0$$

$$x[i] \overset{def}{=} x[i:i]$$

Note that, in order to avoid confusion, we will always number the elements of a tuple from right to left, starting from 0. Also note that we can only meaningfully write $x_{n-1}x_{n-2}\dots x_0$ when the elements from the underlying set $\mathcal{X}$ can be clearly separated from each other.

Definition 2.11 (Finite Sequences Over a Set $\mathcal{X}$) Given some set $\mathcal{X}$, we define the *set of finite sequences* $\mathcal{X}^*$ over this set as follows:

$$\mathcal{X}^* = \{\varepsilon\} \cup \bigcup_{i=1}^{\infty} \mathcal{X}^i$$

Here, ε denotes the *empty sequence*.

To denote the *set of non-empty finite sequences*, we use

$$\mathcal{X}^+ = \bigcup_{i=1}^{\infty} \mathcal{X}^i$$

Definition 2.12 (Length of a Finite Sequence) The *length of a finite sequence* $x \in \mathcal{X}^*$ is defined as follows:

$$|x| = \begin{cases} n & x = x_{n-1}x_{n-2}\dots x_0 \in \mathcal{X}^n \\ 0 & x = \varepsilon \end{cases}$$

Definition 2.13 (Concatenation of Finite Sequences) Sometimes we want to form the *concatenation of finite sequences* to combine two finite sequences into a single one. Given two finite sequences $a, b \in \mathcal{X}^+$ we obtain a new finite sequence

$$a \circ b \overset{def}{=} ab$$

We further have

$$\varepsilon \circ a = a \circ \varepsilon \overset{def}{=} a$$

Note that for any finite sequence $x \in \mathcal{X}^n$, we can now also write

$$x_{n-1} \circ \dots \circ x_0 = x[n-1:0]$$

since the x_i can be considered 1-tuples.

Definition 2.14 (Repeating an Element) For an element $x \in \mathcal{X}$ and a natural number $n \in \mathbb{N}$, we define

$$x^n \overset{def}{=} \begin{cases} x^{n-1} \circ x & n > 0 \\ \varepsilon & n = 0 \end{cases}$$

to denote the finite sequence which repeats x for n times.

Definition 2.15 (Head and Tail of a Finite Sequence) Often, when defining recursive functions, it is useful to talk about the first element (*head*) and the remainder (*tail*) of a non-empty finite sequence:

$$\mathbf{hd}(x[n-1:0]) = x_{n-1}$$

and

$$\mathbf{tl}(x[n-1:0]) = \begin{cases} x[n-2:0] & n > 1 \\ \varepsilon & n = 1 \end{cases}$$

It can easily be shown that, for all $x \in \mathcal{X}^+$,

$$x = \mathbf{hd}(x) \circ \mathbf{tl}(x)$$

holds.

Definition 2.16 (Applying a Function to Every Element of a Finite Sequence) Sometimes it is helpful to describe the application of a given function $f : \mathcal{X} \to \mathcal{Y}$ to every element of a finite sequence $x \in \mathcal{X}^n$, resulting in a finite sequence $y \in \mathcal{Y}^n$:

$$y = \mathbf{map}(f,x) \qquad \Leftrightarrow \qquad \forall i \in \{0,\dots,n-1\} : y[i] = f(x[i])$$

Definition 2.17 (Reverting a Finite Sequence) In order to revert the order of elements in a finite sequence, we define the function $\mathbf{rev} : \mathcal{X}^* \to \mathcal{X}^*$ as follows:

$$\mathbf{rev}(x) = \begin{cases} \mathbf{rev}(xs) \circ x' & x = x' \circ xs \\ \varepsilon & x = \varepsilon \end{cases}$$

2.3 Booleans and Logic Operators

Definition 2.18 (Set of Booleans) The *set of Booleans* $\mathbb{B} = \{0, 1\} \subset \mathbb{N}$ consists of the numbers 0 and 1.

Definition 2.19 (Logic Operators on Booleans) The logic operators $\wedge, \vee, \oplus : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$ (AND, OR, XOR) and $\neg : \mathbb{B} \rightarrow \mathbb{B}$ (NOT) are defined as usual and given by the following tables:

x	y	$x \wedge y$	$x \vee y$	$x \oplus y$	$\neg y$
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	-
1	1	1	1	0	-

Given an $x \in \mathbb{B}$, we use $\overline{x} \overset{def}{=} /x \overset{def}{=} \neg x$ as equivalent notation to refer to the NOT operator.

Definition 2.20 (Logic Operators on Bit-Strings) Given bit-strings $a, b \in \mathbb{B}^n$ and a binary logic operator $\circ \in \{\wedge, \vee, \oplus\}$, we define

$$a \circ_n b \overset{def}{=} ((a_{n-1} \circ_n b_{n-1}), \ldots, (a_0 \circ_n b_0))$$

to describe applying the logic operator bit-wise to the elements of the bit-strings of length n. When the length of bit-strings is clear from the context, we omit it and write $a \circ b$ instead.

Similarly, we define

$$/a \overset{def}{=} \overline{a} \overset{def}{=} \neg a \overset{def}{=} (\neg a_{n-1}, \ldots, \neg a_0)$$

Definition 2.21 (Ternary Operator) Given a Boolean value $A \in \mathbb{B}$ and values $x, y \in \mathcal{X}$, the value of the ternary operator is defined as follows:

$$(A?x : y) = \begin{cases} x & A = 1 \\ y & A = 0 \end{cases}$$

2.4 Representations of Numbers

Definition 2.22 (Number to Base *B*) Given $a \in \{0, \ldots, B-1\}^+$, we consider a a *number to base B*.

Definition 2.23 (Interpreting a Number to Base *B*) Given $a = a[n-1 : 0] \in \{0, \ldots, B-1\}^n$, the *value represented by a to base B* is given by

$$\langle a[n-1 : 0] \rangle_B = \sum_{i=0}^{n-1} a_i \cdot B^i$$

2.4.1 Binary Numbers and Bit-Strings

Definition 2.24 (Bit-String) A *bit-string* is a finite sequence over the set $\mathbb{B}$. Most of the time, we are interested in bit-strings $b \in \mathbb{B}^n$ of a specific length n.

In the following, we consider $\langle \cdot \rangle$ as a shorthand for $\langle \cdot \rangle_2$ which is the function that interprets a bit-string as a number to base 2, i.e. as a binary number.

Definition 2.25 (Two's-Complement Interpretation of a Bit-String) The *two's-complement value* of a bit-string $b \in \mathbb{B}^n$ is defined as follows:

$$[b] = -2^{n-1} \cdot \mathbf{hd}(b) + \langle \mathbf{tl}(b) \rangle$$

Definition 2.26 (Range of Values) We define the sets

$$B_n = \{\langle a \rangle \mid a \in \mathbb{B}^n\} = \{0, \dots, 2^n - 1\}$$

and

$$T_n = \{[a] \mid a \in \mathbb{B}^n\} = \{-2^{n-1}, \dots, 2^{n-1} - 1\}$$

which describe the *range of the interpretation functions* $\langle \cdot \rangle$ and $[\cdot]$, respectively.

Definition 2.27 (Binary Representations) We define for $x \in B_n$

$$bin_n(x) \overset{def}{=} \varepsilon\{a \mid a \in \mathbb{B}^n, \langle a \rangle = x\}$$

and for $x \in T_n$

$$twoc_n(x) \overset{def}{=} \varepsilon\{a \mid a \in \mathbb{B}^n, [a] = x\}$$

to get the binary or, respectively, two's-complement representation of a number. To get an even shorter notation, we tend to use $x_n \overset{def}{=} bin_n(x)$ as a shorthand for binary representation and $x_{\mathbf{t}n} \overset{def}{=} twoc_n(x)$ as a shorthand for two's-complement representation.

Definition 2.28 (Zero-Extension) We define for $a \in \mathbb{B}^n$ and $n, k \in \mathbb{N}, k > n$

$$zxt_k(a) \overset{def}{=} 0^{k-n}a$$

the *zero-extended* bit-string of length k for a.

Lemma 2.29 (Zero-Extension Preserves Binary Value) For $a \in \mathbb{B}^n$ and $n, k \in \mathbb{N}, k > n$, we have

$$\langle zxt_k(a) \rangle = \langle a \rangle$$

Definition 2.30 (Sign-Extension) For $a \in \mathbb{B}^n$ and $n, k \in \mathbb{N}, k > n$, we define

$$sxt_k(a) \overset{def}{=} a_{n-1}^{k-n}a$$

to mean the *sign-extended* bit-string of length k for a.

Lemma 2.31 (Sign-Extension Preserves Two's-Complement Value) For $a \in \mathbb{B}^n$ and $n, k \in \mathbb{N}, k > n$, we have

$$[sxt_k(a)] = [a]$$

Definition 2.32 (*i*-th Byte of a Bit-String) For a bit-string $a \in \mathbb{B}^{8k}, k \in \mathbb{N}$ and $0 \leq i < k$, we define

$$byte(i,a) \overset{def}{=} a[(i+1) \cdot 8 - 1 : i \cdot 8]$$

to denote the i-th byte in a.

2.4.2 Arithmetics on Binary Numbers

Definition 2.33 (Congruence Modulo) We use the equivalence relation $\equiv \mod k$ defined as follows for $a, b \in \mathbb{Z}, k \in \mathbb{N} \setminus \{0\}$:

$$a \equiv b \mod k \quad \Leftrightarrow \quad \exists z \in \mathbb{Z} : a - b = z \cdot k$$

Definition 2.34 (Modulo Operator) The modulo-operator is then defined by

$$a \,\mathbf{mod}\, k = \varepsilon\{x \mid a \equiv x \mod k \wedge x \in \{0, \ldots, k-1\}\}$$

Definition 2.35 (Two's-Complement Modulo Operator) For arithmetics on two's-complement numbers, we define an additional modulo operator for two's-complement numbers:

$$a \,\mathbf{modt}\, 2^k = \varepsilon\{x \mid a \equiv x \mod 2^k \wedge x \in T_k\}$$

Definition 2.36 (Arithmetic Operations: Addition and Subtraction) We define arithmetic operations on binary numbers:
Given $a, b \in \mathbb{B}^n$, we define

- Binary addition: $a +_n b \overset{def}{=} bin_n(\langle a \rangle + \langle b \rangle \,\mathbf{mod}\, 2^n)$
- Binary subtraction: $a -_n b \overset{def}{=} bin_n(\langle a \rangle - \langle b \rangle \,\mathbf{mod}\, 2^n)$

Lemma 2.37 (Binary Addition and Subtraction Apply to Two's-Complement Numbers) A nice observation here is that $+_n$ and $-_n$ also work for two's-complement numbers, i.e. it can be proven easily by applying previous definitions that

$$bin_n(\langle a \rangle + \langle b \rangle \,\mathbf{mod}\, 2^n) = twoc_n([a] + [b] \,\mathbf{modt}\, 2^n)$$

and

$$bin_n(\langle a \rangle - \langle b \rangle \,\mathbf{mod}\, 2^n) = twoc_n([a] - [b] \,\mathbf{modt}\, 2^n)$$

hold.

Definition 2.38 (Arithmetic Operations: Multiplication) Given $a, b \in \mathbb{B}^n$, we define

- $a \cdot_n b \overset{def}{=} bin_n(\langle a \rangle \cdot \langle b \rangle \,\mathbf{mod}\, 2^n)$

- $a \cdot_{\mathbf{t}n} b \overset{def}{=} twoc_n([a] \cdot [b] \,\mathbf{modt}\, 2^n)$

Definition 2.39 (Arithmetic Operations: Division) Given $a, b \in \mathbb{B}^n$, we define

- $a \div_n b \overset{def}{=} bin_n(\langle a \rangle \div \langle b \rangle \,\mathbf{mod}\, 2^n)$
- $a \div_{\mathbf{t}n} b \overset{def}{=} twoc_n([a] \div [b] \,\mathbf{modt}\, 2^n)$

Where $\div$ is the *division operator that rounds towards zero*: For $x, y \in \mathbb{Z}$:

$$x \div y \overset{def}{=} sign(x) \cdot sign(y) \cdot \lfloor |x| / |y| \rfloor$$

where for $z \in \mathbb{Z}$

$$sign(z) \overset{def}{=} \begin{cases} 1 & z \geq 0 \\ -1 & z < 0 \end{cases}$$

Note that for $x, y \in \mathbb{N}$,

$$x = (x \div y) \cdot y + (x \,\mathbf{mod}\, y)$$

holds.

2.5 Models of Computation

The term *computational model* or *model of computation* is widely used for all kinds of mathematical models that can be computed. Such models are often used to represent phenomena occurring in nature by computer simulation. Prominent examples include weather forecasting models, molecular protein models, traffic models, climate models, or models of social behavior.

In this thesis, we are particularly interested in computational models of computation performed by computers, e.g. gate-level hardware models, instruction set architecture models, or models of programming language execution (semantics of programming languages). Indeed, when we look at this from the generic perspective taken in the first paragraph, we can see these computational models of computation performed by a computer as abstract models of the natural phenomena that occur when we power on the computer's hardware.

One of the most common ways to model computation in computer science is by defining an appropriate *automaton*. Automata come in many different flavors – too many to discuss all of them meaningfully in this thesis. Thus, we give an explicit definition of the notion of automaton considered here in the following subsection. The formalism we chose is both generic and simple, and extends easily to other notions of automata.

2.5.1 Automata

An *automaton* is a mathematical representation of what can intuitively be considered a *transition system*. A transition system is characterized by states and by transitions that occur between these states. Starting from some *initial state*, transitions can be applied to reach other states.

Definition 2.40 (Automaton) Given

- a set S of *states*,
- a set Σ of *actions*, also called *alphabet*,
- a non-empty set $S_0 \subseteq S$ of *initial states*, and
- a *transition relation* $\delta \subseteq S \times \Sigma \times S$,

we consider the quadruple $A = (S, \Sigma, S_0, \delta)$ an *automaton*.

Starting the automaton, an arbitrary state $s_0 \in S_0$ is chosen as the current state. When an action and a pair of states is in the transition relation, $(s, a, s') \in \delta$, this means that a transition from state $s \in S$ to state $s' \in S$ is possible under action a. Actions represent interaction of the automaton with an external world which provides inputs to/reacts to outputs from the automaton. A step of the automaton applies an arbitrarily chosen possible transition from the current state, resulting in a new current state. The automaton repeatedly performs steps – this continues either infinitely often, or until there is no possible transition from the current state.

Note that when there are no inputs or outputs of interest, it is possible to simplify the definitions, eliminating the need for an alphabet. Automata formalizations without an alphabet can be easily embedded in this formalism by having an alphabet consisting of a single token that denotes the absence of input/output.

Definition 2.41 (Notation: Possible Step) In order to denote that a step from state $s \in S$ to state $s' \in S$ under action $a \in \Sigma$ is possible under a transition relation $\delta \subseteq S \times \Sigma \times S$, we use the notation

$$s \overset{a}{\underset{\delta}{\rightsquigarrow}} s' \quad \overset{def}{\Leftrightarrow} \quad (s, a, s') \in \delta$$

To say that there exists an action $a \in \Sigma$ such that a step from state s to state s' under a is possible under δ, we define

$$s \overset{\exists}{\underset{\delta}{\rightsquigarrow}} s' \quad \overset{def}{\Leftrightarrow} \quad \exists a \in \Sigma : s \overset{a}{\underset{\delta}{\rightsquigarrow}} s'$$

Note that $\overset{\exists}{\underset{\delta}{\rightsquigarrow}} \subseteq S \times S$ is a binary relation.

Definition 2.42 (Notation: No Possible Step) For a transition relation $\delta \subseteq S \times \Sigma \times S$, a state $s \in S$ and an action $a \in \Sigma$, we use

$$s \overset{a}{\underset{\delta}{\not\rightsquigarrow}} \quad \overset{def}{\Leftrightarrow} \quad \neg \exists s' \in S : s \overset{a}{\underset{\delta}{\rightsquigarrow}} s'$$

to express that under δ, there is *no possible step under action a* from state s. When there is no possible step under any action $a \in \Sigma$, we state that there is no possible step from s under δ:

$$s \overset{\forall}{\underset{\delta}{\not\rightsquigarrow}} \quad \overset{def}{\Leftrightarrow} \quad \forall a \in \Sigma : s \overset{a}{\underset{\delta}{\not\rightsquigarrow}}$$

Definition 2.43 (Notation: Deterministic Step) We state that under a transition relation $\delta \subseteq S \times \Sigma \times S$, a *deterministic step* from $s \in S$ under action $a \in \Sigma$ to state $s' \in S$ is possible by

$$s \underset{\delta}{\overset{a}{\rightarrow}} s' \quad \overset{def}{\Leftrightarrow} \quad s \underset{\delta}{\overset{a}{\rightsquigarrow}} s' \wedge (\forall s'' : s \underset{\delta}{\overset{a}{\rightsquigarrow}} s'' \Rightarrow s' = s'')$$

A deterministic step is characterized by the resulting state s' being *determined* by the previous state s and action a. In general, given a state and an action, several resulting states may be possible under a transition relation.

Definition 2.44 (Deterministic Transition Relation) A transition relation $\delta \subseteq S \times \Sigma \times S$ with

$$\forall s \in S, a \in \Sigma : (\exists s' \in S : s \underset{\delta}{\overset{a}{\rightarrow}} s') \vee s \underset{\delta}{\overset{a}{\not\rightsquigarrow}}$$

is called *deterministic transition relation*.

Definition 2.45 (Deterministic Automaton) An automaton $A = (S, \Sigma, S_0, \delta)$ with deterministic transition relation δ and a single initial state ($\#S_0 = 1$) is called *deterministic automaton*.

Note that, when an automaton is deterministic, we can use a partial function

$$\delta : S \times \Sigma \rightharpoonup S$$

to describe its transition relation. All automata occurring in this thesis are formalized in such a way that their transition relation is made deterministic by resolving all non-deterministic choice with corresponding input-symbols.

2.5.2 Reachability & Traces

Often, we are interested in whether (bad or good) states are reachable when the automaton performs steps starting from a given state s. To express reachability, we use the *reflexive transitive closure* of the binary transition relation.

Definition 2.46 (Reflexive Transitive Closure) The reflexive transitive closure R^* of a binary relation $R \subseteq \mathcal{X} \times \mathcal{X}$ can be defined as follows:

$$(a, b) \in R \quad \overset{def}{\Leftrightarrow} \quad a = b \vee \exists c : (a, c) \in R^* \wedge (c, b) \in R$$

For an automaton $A = (S, \Sigma, S_0, \delta)$, the relation $\underset{\delta}{\overset{\exists}{\rightsquigarrow}}{}^*$ describes reachability.

In theorems and lemmata, we sometimes need to argue about executions of an automaton. Formally, we model executions of an automaton as follows:

Definition 2.47 (Finite Execution Trace) Given a transition relation $\delta \subseteq S \times \Sigma \times S$, a *finite execution trace* (s, a) of δ is a pair of a finite sequence of states $s \in S^{n+1}$ and a finite sequence of actions $a \in \Sigma^n$ with

$$s_0 \underset{\delta}{\overset{a_0}{\rightsquigarrow}} s_1 \underset{\delta}{\overset{a_1}{\rightsquigarrow}} s_2 \underset{\delta}{\overset{a_2}{\rightsquigarrow}} \ldots \underset{\delta}{\overset{a_{n-1}}{\rightsquigarrow}} s_n$$

More formally,

$$\forall i \in \{0, \ldots, n-1\} : s_i \underset{\delta}{\overset{a_i}{\rightsquigarrow}} s_{i+1}$$

$$\begin{tikzcd}
s_B \arrow[r, "\overset{a}{\underset{\delta_B}{\rightsquigarrow}}"] \arrow[d, dotted, dash, "\sim"'] & s'_B \arrow[d, dotted, dash, "\sim"] \\
s_A \arrow[r, "\overset{a}{\underset{\delta_A}{\rightsquigarrow}}"'] & s'_A
\end{tikzcd}$$

Figure 2.1: Simulation between automata.

2.5.3 Simulation Theorems

Sometimes it is necessary to prove that two automata essentially behave in the same way. This can be done by proving a *simulation* between the two automata. When a simulation from automaton A to automaton B exists, it is proven that the structure of execution of automaton A (given by its transition relation) is mimicked by automaton B. This can be formulated by stating step-by-step simulation theorems like the following:

Definition 2.48 (Simulation) Given Automata $A = (S_A, \Sigma, S_{0A}, \delta_A)$ and $B = (S_B, \Sigma, S_{0B}, \delta_B)$, a relation $\sim \subseteq S_A \times S_B$ is a *simulation* from A to B iff

1. $\forall s_A \in S_{0A} : \exists s_B \in S_{0B} : \quad s_A \sim s_B$
2. $\forall s_A, s'_A \in S_A, s_B \in S_B, a \in \Sigma : \quad s_A \sim s_B \wedge s_A \overset{a}{\underset{\delta_A}{\rightsquigarrow}} s'_A \quad \Rightarrow \quad \exists s'_B : \quad s_B \overset{a}{\underset{\delta_B}{\rightsquigarrow}} s'_B \wedge s'_A \sim s'_B$

The above definition has a nice implication in terms of trace inclusion. However, it requires both automata to be formalized in such a way that they have the same alphabet Σ – which is only desirable when exactly the observable behavior of interest is represented by actions. Note that, when the input alphabet is used to resolve non-deterministic choice occurring in an automaton, there does not necessarily need to be a one-to-one correspondence between the input alphabets of the two automata.

Refinement Simulation A *refinement simulation* of automaton B (abstraction) by automaton A (refinement) is a special case of simulation where the simulation relation is given by an *abstraction function* $f : S_A \to S_B$ (also known as *refinement mapping* in [AL88]), such that $a \sim b \Leftrightarrow b = f(a)$. In this case, the definition of *simulation* simplifies to the following:

1. $\forall s_A \in S_{0A} : \quad f(s_A) \in S_{0B}$
2. $\forall s_A, s'_A \in S_A, a \in \Sigma : \quad s_A \overset{a}{\underset{\delta_A}{\rightsquigarrow}} s'_A \quad \Rightarrow \quad f(s_A) \overset{a}{\underset{\delta_B}{\rightsquigarrow}} f(s'_A)$

2.6 Semantics of Programming Languages

A program, in the context of most programming languages, can be considered a definition of a sequence of computations. In practice, such a program is often subdivided into functions that perform recurring tasks. As the study of *semantics* is a study of meaning, the study of semantics of programming languages is nothing other than the study of meaning of program execution. There are different established ways to mathematically define the meaning of a program: operational semantics, axiomatic semantics, and denotational semantics.

Denotational semantics formalizes the meaning of programs by providing mathematical objects that represent the behavior of the program, translating each program phrase to its denotation in some other (mathematical) language. This is a formalism in which is well suited to argue about program transformations. In this thesis, such a model does not occur.

Operational semantics are modeled in a way that focuses on the operational aspect of executing the program to define its semantics. This is done by defining the effect of each program phrase on the state of an abstract machine, e.g. an automaton. This is the kind of formalism we use in this thesis to describe the computational models we consider.

Axiomatic semantics states the effect of program execution on assertions about the program state. The most commonly cited example for axiomatic semantics is *Hoare logic* used in program verification.

2.6.1 Operational Semantics

> "The meaning of a program in the strict language is explained in terms of a hypothetical computer which performs the set of actions which constitute the elaboration of that program."
> *Algol68* [vW81], *Section 2*

Commonly, two styles of specifying operational semantics are distinguished, *structural operational semantics* and *natural semantics*. We briefly describe them in the following.

Structural Operational Semantics

Structural operational semantics of a programming language, also called *small-step semantics*, describes for every individual program action the state resulting by its execution. Essentially, we are specifying transitions on the states of the abstract machine which can be expressed by defining an appropriate automaton. The resulting formalism depends greatly on what kind of actions the programming language in question allows. For example, if there can be side-effects in expression-evaluation, expression evaluation can be considered a program action, whereas for languages without side-effects in expression we do not need to consider expression evaluation as a program action.

Natural Semantics

The distinguishing feature of small-step semantics is that in the resulting automaton, we have a transition for every program action. Natural semantics, also called *big-step semantics*, on the other hand, is defined in such a way that the overall effect of execution of the program is given. This can be done by applying small-step semantics until the program terminates, resulting in an automaton in which the transitions describe the effect of executing (terminating) programs, or functions. In a sequential setting, this notion of complete program execution tends to be applicable.

2.6.2 Formalizing Operational Semantics as an Automaton

For many programming languages (including the ones we will discuss in this thesis), we can consider the abstract machine to consist of two main parts:

- a static component $s \in \mathit{static}$ which never changes, and
- a dynamic component $d \in \mathit{dynamic}$ which is modified through execution of program actions.

Usually, one assumes that there is no self-modifying code, i.e. that execution of a program will never change the program being executed. Thus, the program that is executed is part of the static component s.

The dynamic component describes the configuration of the abstract machine, i.e. it contains memory state – which gives values for variables occurring in the program – and control state – which describes how program execution has to progress.

Operational Semantics Automata

When formalizing operational semantics using automata, there is a certain amount of choice in how exactly operational semantics can be formalized. We could have an automaton

$$A = (\mathit{static} \times \mathit{dynamic}, \Sigma, S_0, \delta)$$

where

$$((s, d), a, (s', d')) \in \delta \Rightarrow s = s'$$

while, on the other hand, we can achieve a very similar result by considering for each $s \in \mathit{static}$ an individual automaton

$$A_s = (\mathit{dynamic}, \Sigma, S_0, \delta_s)$$

with a transition relation δ_s that describes the possible transitions on the dynamic component under a given static component s:

$$((s, d), a, (s, d')) \in \delta \quad \Leftrightarrow \quad (d, a, d') \in \delta_s$$

In this thesis, we formalize operational semantics of programs in the latter way.

2.7 Concurrency

con·cur·rence *noun*

1. **a** : the simultaneous occurrence of events or circumstances
   b : the meeting of concurrent lines in a point
2. **a** : agreement or union in action : cooperation
   b (1) : agreement in opinion or design (2) : consent
3. a coincidence of equal powers in law

"concurrence." Merriam-Webster.com. Merriam-Webster, 2011. Web. 6 July 2011.

While the Merriam-Webster dictionary refers the reader to the definition of the word concurrence when looking for *concurrency*, the specific term concurrency is used widely in computer science to describe all kinds of computational models in which certain computations are happening, in some sense, at the same time. Often, such a concurrent computational model can easily be subdivided into disjoint parts which are considered to perform concurrent access to shared resources of some kind.

Structurally, there appear to be two major kinds of concurrent models: One where the components simultaneously perform steps (e.g. hardware which is clocked synchonously), and one where the components "take turns" performing their steps.

A system consisting of n components can usually be described by an automaton with the following state:

$$S \equiv S_1 \times S_2 \times \ldots \times S_n$$

where S_i is the set of states of component $i, 1 \leq i \leq n$, and where we assume for the sake of simplicity that we have transition functions

$$\delta_1 : S \to S_1, \quad \ldots \quad, \delta_n : S \to S_n$$

that model the steps performed by the components given the overall state of the system.

Definition 2.49 (Simultaneous/Parallel Concurrency) When all components perform a step during a transition of the overall system, we say that such a model exhibits *simultaneous concurrency*, or *parallel concurrency*, i.e.

$$\delta : S \to S, \qquad \delta(s) = (\delta_1(s), \delta_2(s), \ldots, \delta_n(s))$$

Definition 2.50 (Interleaved Concurrency) When every transition of the overall system corresponds to a single transition of some chosen subcomponent, we say that the model is based on *interleaved concurrency*, i.e.

$$\delta : S \times \underbrace{\{1, \ldots, n\}}_{=:\Sigma} \to S, \qquad \delta(s, i) = (s_1, \ldots, \delta_i(s), \ldots, s_n)$$

where $\Sigma = \{1, \ldots, n\}$ is called *scheduling alphabet* and describes which subcomponent performs a step.

Note that, assuming the behavior of components is deterministic, a model with parallel concurrency stays deterministic, while a model with interleaved concurrency can be seen as turning non-deterministic: A choice which subcomponent performs a step is introduced. Unless we know the exact order in which steps of components are interleaved, or any other restriction imposed on the scheduling of components, we have to consider all possible interleavings.

Scheduling alphabet In practice, defining a specific model that exhibits interleaved concurrency, instead of simply numbering subcomponents, we do give appropriate names to them. In such a case, we use descriptive names for the scheduling actions of the scheduling alphabet that refer to the corresponding components.

Definition 2.51 (Fairness) Given an infinite trace a of scheduling actions, we consider this trace *fair* if and only if every component of the corresponding concurrent system is scheduled infinitely often in a.

In this thesis, we are interested in models where the interleaved concurrency is at least restricted by this fairly weak fairness constraint.

3 Theory of Multi-Core Hypervisor Verification

In the following we present a roadmap for pervasive formal verification of operating systems and hypervisors on a multi-core architecture as it has evolved from the experiences made during and after the *Verisoft XT* project. It becomes obvious, that this is much more difficult than in the sequential case since, on the one hand, multi-core architectures often offer features that are not meaningful in single-core architectures, like, for example, advanced programmable interrupt controllers (APICs), or cache-protocols on distributed caches (e.g. the MOESI-protocol). On the other hand, hardware features that could be abstracted to clean and simple models without much effort in single-core architectures suddenly become visible for multi-core architectures, e.g., consider store-buffers and the resulting weak memory models.

This chapter is a simple reproduction – where references to this thesis are adjusted to refer to appropriate chapters of the thesis – of the SOFSEM2013 invited paper "*Theory of Multi Core Hypervisor Verification*" [CPS13] by Ernie Cohen, Wolfgang Paul and the author of this thesis.

3.1 Introduction and Overview

Low-level system software is an important target for formal verification; it represents a relatively small codebase that is widely used, of critical importance, and hard to get right. There have been a number of verification projects targetting such code, particularly *operating system* (OS) kernels. However, they are typically designed as providing a proof of concept, rather than a viable industrial process suitable for realistic code running on modern hardware. One of the goals of the Verisoft XT project [Ver10] was to deal with these issues. Its verification target, the hypervisor Hyper-V [LS09] was highly optimized, concurrent, shipping C/assembler code running on the most popular PC hardware platform (x64). The verification was done using VCC, a verifier for concurrent C code based on a methodology designed to maximize programmer productivity – instead of using a deep embedding of the language into a proof-checking tool where one can talk directly about the execution of the particular program on the particular hardware.

We were aware that taking this high-level view meant that we were creating a nontrivial gap between the abstractions we used in the software verification and the system on which the software was to execute. For example,

- VCC has an extension allowing it to verify x64 assembly code; why is its approach sound? For example, it would be unsound for the verifier to assume that hardware registers do not change when executing non-assembly code, even though they are not directly modified by the intervening C code.

- Concurrent C code (and to a lesser extent, the C compiler) tacitly assumes a strong memory model. What justifies executing it on a piece of hardware that provides only weak memory?

- The hypervisor has to manage threads (which involves setting up stacks and implementing thread switch) and memory (which includes managing its own page tables). But most of this management is done with C code, and the C runtime already assumes that this management is done correctly (to make memory behave like memory and threads behave like threads). Is this reasoning circular?

When we started the project, we had ideas of how to justify all of these pretenses, but had not worked out the details. Our purpose here is to i) outline the supporting theory, ii) review those parts of the theory that have already been worked out over the last few years, and iii) identify the parts of the theory that still have to be worked out.

3.1.1 Correctness of Operating System Kernels and Hypervisors

Hypervisors are, at their core, OS kernels, and every basic class about theoretical computer science presents something extremely close to the correctness proof of a kernel, namely the simulation of k one-tape *Turing machines* (TMs) by a single k-tape TM [HS65]. Turning that construction into a simulation of k one-tape TMs by a single one-tape TM (*virtualization* of k *guest* machines by one *host* machine) is a simple exercise. The standard solution is illustrated in figure 3.1. The tape of the host machine is subdivided into tracks, each representing the tape of one of the guest machines (*address translation*). Head position and state of the guest machines are stored on a dedicated field of the track of that machine (a kind of *process control block*). Steps of the guests are simulated by the host in a round robin way (a special way of *scheduling*). If we add an extra track for the data structures of the host and add some basic mechanisms for communications between guests (*inter process communication*) via *system calls*, we have nothing less than a one-tape TM kernel. Generalizing from TMs to an arbitrary computation model M (and adding I/O-devices), one can specify an M kernel as a program running on a machine of type M that provides

- virtualization: the simulation of k guest machines of type M on a single host machine of type M

- system calls: some basic communication mechanisms between guests, I/O devices, and the kernel

At least as far as the virtualization part is concerned, a kernel correctness theorem is essentially like the Turing machine simulation theorem, and can likewise be conveniently expressed as a forward simulation. For more realistic kernels, instead of TMs we have processors, described in dauntingly large manuals, like those for the MIPS32 [MIP05] (336 pages), PowerPC [Fre05] (640 pages), x86 or x64 [nxb10, Int10] (approx. 1500, resp. 3000 pages). The TM tape is replaced by RAM, and the tape head is replaced by a *memory management unit* (MMU), with address translation driven by in-memory *page tables*. Observe that a mathematical model of such machine is part of the definition of correctness for a 'real' kernel.

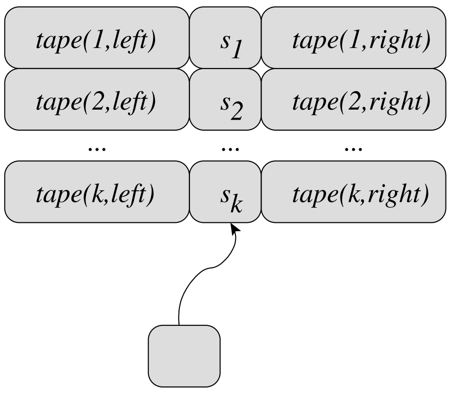


Figure 3.1: Simulating k Turing machines with 1 k-band Turing machine.

Hypervisors are kernels whose guests are themselves operating systems or kernels, i.e. each guest can run several *user processes*. In terms of the TM simulation, each guest track is subdivided into subtracks for each user, each subtrack having its own process control block; the actual tape address for the next operation of a process can be calculated from its tape address, the layout of the subtrack within the track, and the layout of the track itself. In a real kernel, the address of a memory access is calculated from a virtual address using two levels of address translation, the first level of traslation provided by the guest to users via the *guest page tables* (GPTs), and the second provided by the hypervisor to the guest. On many recent processors, this second level of address translation is provided in hardware by a separate set of *host page tables*. On processors providing only a single level of translation, it is possible to take advantage of the fact that the composition of two translations is again a translation, and so can be provided by a single set of page tables. Because these *shadow page tables* (SPTs) correspond to neither the guest nor the host page tables, they are constructed on the fly by the hypervisor from the GPTs, and the hypervisor must hide from the guest that translation goes through these tables rather than the GPTs. Thus, the combined efforts of the hypervisor and the MMU simulate a virtual MMU for each guest.

3.1.2 Overview

We discuss the following seven theories in the remainder of the paper:

Multi-Core ISA-sp We define a nondeterministic concurrent *instruction set architecture* (ISA) model, suitable for system programming. In addition to processor cores and main memory, it includes low-level (but architecturally visible) features such as store buffers, caches, and memory management units. Ideally, this would be given in (or at least derived from) the system programmer's manuals published by the chip manufacturers. In reality, many subtle (but essential) details are omitted from these manuals. Indeed, hardware manufacturers often deliberately avoid commiting themselves to architectural boundaries, to maximize their flexibility in optimizing the implementation, and many details leveraged by real operating systems (such as details concerning the walking of page tables) are shared only with their most important customers, under agreements of nondisclosure. Such fuzzy architectural boundaries are acceptable when clients are writing operating systems; a progammer can choose whether to program to a conservative model (e.g., by flushing translations after every change to the page tables) or program more aggressively to a model that takes advantage of architectural details of the current processor generation. But such a fuzzy model is fatal when trying to build an efficient hypervisor, because the architectural specification must both be strong enough for client operating systems to run correctly, yet weak enough that it can be implemented efficiently on top of the available hardware.

We provide evidence for the particular model we use and for the particular ways in which we resolved ambiguities of the manuals in the following way: i) we define a simplified ISA-sp that we call MIPS-86[1], which is simply MIPS processor cores extended with x86-64 like architecture features (in particular, memory system and interrupt controllers), ii) we reverse engineer the machine in a plausibly efficient way at the gate level, and iii) we prove that the construction meets the ISA-sp, and iv) we confirm with OS engineers that the model is sufficiently strong to support the memory management algorithms used in real operating systems. The correctness theorems in this theory deal with the correctness of hardware for multi-core processors at the gate level.

ISA Abstraction Multi-core machines are primarily optimized to efficiently run ordinary user code (as defined in the user programming manuals). In this simplified instruction set (ISA-u), architectural details like caches, page tables, MMUs, and store buffers should be transparent, and multithreaded programs should see sequentially consistent memory (assuming that they follow a suitable synchronization discipline). A naive discipline combines lock-protected data with shared variables, where writes to shared variables flush the store buffer. A slightly more sophisticated and efficient discipline requires a flush only when switching from writing to reading [CS10]. After proper configuration, a simulation between ISA-sp and ISA-u has to be shown in this theory for programs obeying such disciplines.

Serial Language Stack A realistic kernel is mostly written in a high-level language (typically C[2] or C++) with small parts written in macro assembler (which likewise provides the stack abstraction) and even smaller parts written in plain assembler (where the implementation of the stack using hardware registers is exposed, to allow operations like thread switch). The main definition of this theory is the formal semantics of this computational model. The main theorem is a combined correctness proof of optimizing compiler + macro assembler for this mixed language. Note that compilers translate from a source language to a clean assembly language, i.e. to ISA-u.

Adding Devices Formal models for at least two types of devices must be defined: regular devices and interrupt controllers (the APIC in x86/64). A particularly useful example device is a hard disk – which is needed for booting. Interrupt controllers are needed to handle both external interrupts and interrupt-driven interprocess communication (and must be

[1]See chapter 4.
[2]See chapter 5.

virtualized by the hypervisor since they belong to the architecture). Note that interrupt controllers are very particular kinds of devices in the sense that they are interconnected among each other and with processor cores in a way regular devices are not: They inject interrupts collected from regular devices and other interrupt controllers directly into the processor core. Thus, interrupt controllers must be considered specifically as part of an ISA-sp model with instantiable devices[3]. Crucial definitions in this theory are i) sequential models for the devices, ii) concurrent models for ISA-sp with devices, and iii) models for single core processors semantics of C with devices (accessed through *memory mapped I/O* (MMIO)). The crucial theorems of this theory show the correctness of drivers at the code level.

Extending the Serial Language Stack with Devices to Multi-Core Machines The crucial definition of this theory is the semantics of concurrent 'C + macro assembly + ISA-sp + devices'. Besides ISA-sp, this is *the* crucial definition of the overall theory, because it defines the language/computational model in which multi-core hypervisors are coded. Without this semantics, complete code level verification of a hypervisor is not meaningful. Essentially, the ownership discipline of the ISA abstraction theory is lifted to the C level; in order to enable the implementation of the ownership discipline, one has to extend serial C with volatile variables and a small number of compiler intrinsics (fences and atomic instructions). In this theory there are two types of major theorems. The first is compiler correctness: if the functions of a concurrent C program obeying the ownership discipline are compiled separately, then the resulting ISA-u code obeys the ownership discipline and the multi-core ISA-u code simulates the parallel C code. The second is a reduction theorem that allows us to pretend that a concurrent C program has scheduler boundaries only just before actions that race with other threads (I/O operations and accesses to volatile variables).

Soundness of VCC and its Use Programs in the concurrent C are verified using VCC. In order to argue that the formal proofs obtained in this way are meaningful, one has to prove the soundness of VCC for reasoning about concurrent C programs, and one has to show how to use VCC in a sound way to argue about programs in the richer models.

Obviously, for the first task, syntax and semantics of the annotation language of VCC has to be defined. VCC annotations consist essentially of "ghost" (a.k.a. "auxilliary" or "specification") state, ghost code (used to facilitate reasoning about the program, but not seen by the compiler) and annotations of the form "this is true here" (e.g. function pre/post-conditions, loop invariants, and data invariants). Then three kinds of results have to be proven. First, we must show that if a program (together with its ghost code) is certified by VCC, then the "this is true here" assertions do in fact hold for all executions. Second, we must show that the program with the ghost code simulates the program without the ghost code (which depends on VCC checking that there is no flow from ghost state to concrete state, and that all ghost code terminates[4]). Third, we must show that the

[3]MIPS-86 provides such an ISA-sp model with interrupt controllers and instantiable devices – albeit currently at a level where caches are already invisible.

[4]Chapter 6 of this thesis provides such a proof for the C intermediate language introduced in chapter 5.

verification implies that the program conforms to the Cohen/Schirmer [CS10] ownership discipline (to justify VCC's assumption of a sequentially consistent model of concurrent C).

To reason about richer programming models with VCC, we take advantage of the fact that the needed extensions can be encoded using C. In particular, one can add additional ghost data representing the states of processor registers, MMUs and devices to a C program; this state must be stored in "hybrid" memory that exists outside of the usual C address space but from which information can flow to C memory. We then represent assembly instructions as function calls, and represent active entities like MMUs and devices by concurrently running C threads.

Hypervisor Correctness The previous theories serve to provide a firm foundation for the real verification work, and to extend classical verification technology for serial programs to the rich computational models that are necessarily involved in (full) multi-core hypervisor verification. Verification of the hypervisor code itself involves several major components, including i) the implemention of a large numbers of 'C + macro assembly + assembly' threads on a multi-core processor with a fixed small number of cores, ii) for host hardware whose MMUs do not support two levels of translations, the correctness of a parallel shadow page table algorithm, iii) a TM-type simulation theorem showing virtualization of ISA-sp guests by the host, and iv) correct implementation of system calls.

3.2 ISA Specification and Processor Correctness

3.2.1 Related Work

For single core RISC (*reduced instruction set computer*) processors, it is well understood how to specify an ISA and how to formally prove hardware correctness. In the academic world, the papers [BJ01] and [BJK^{+}03] report the specification and formal verification of a MIPS-like processor with a pipelined core with forwarding and hardware interlock, internal interrupts, caches, a fully IEEE compliant pipelined floating point unit, a Tomasulo scheduler for out of order execution, and MMUs for single-level pages tables. In industry, the processor core of a high-end controller has been formally verified [Ver07]. To our knowledge, there is no complete formal model for any modern commercial CISC (*complex instruction set computer*); until recently, the best approximations to such a model were C simulators for large portions of the instruction set [Vir, Boc, QEM].

The classical memory model for multi-core processors is Lamport's sequentially consistent shared memory [Lam79]. However, most modern multi-core processors provide efficient implementations only of weaker memory models. The most accurate model of the memory system of modern x86/64 architectures, "x86-tso", is presented in [SSO^{+}10]. This model abstracts away caches and the memory modes specifying the cache coherence protocol to be used, and presents the memory system as a sequentially consistent shared memory, with a separate FIFO store buffer for each processor core. It is easy to show that the model collapses if one mixes in the same computation able and non cacheable memory modes on the same address (accesses in non cacheable memory modes bypass the cache; accesses in different non cacheable modes have

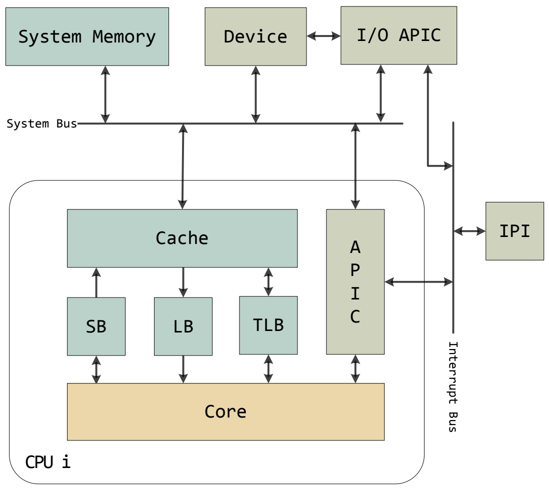


Figure 3.2: x86-64 processor model consisting of components whose steps are interleaved non-deterministically.

different side effects on the caches). That the view of a sequentially consistent shared memory can be maintained even if of one mixes in the same computation accesses to the same address with different "compatible" memory modes/coherence protocols is claimed in the classical paper introducing the MOESI protocol [SS86], but we are not aware of any proof of this fact.

Another surprising observation concerns correctness proofs for cache coherence protocols. The model checking literature abounds with papers showing that certain invariants of a cache system are maintained in an interleaving model where the individual cache steps are atomic. The most important of these invariants states that data for the same memory address present in two caches are identical and thus guarantees a consistent view on the memory at all caches. For a survey, see [CYGC10]. These results obviously provide an important step towards the provably correct construction of a sequentially consistent shared memory. Apart from our own results in [Pau12], we are not aware of a gate-level construction of hardware main memory, caches, cache controllers, and the busses connecting them for which it has been proved that parallel execution of hardware accesses to the caches simulates the high-level cache model.

3.2.2 Modeling an x86-64-like ISA-sp

A formal model of a very large subset of the x64 ISA-sp was constructed as part of the Hyper-V verification project, and is presented in [Deg11]. This 300 page model specifies 140 general purpose and system programming instructions. Due to time constraints, the model omits debug facilities, the alignment check exception, virtual-8086 mode, virtual interrupts, hardware task-switching, system management mode, and devices other than the local APICs. The MMX extension of the instruction set is formalized in the complementary thesis [Bau08]. The instruction set architecture is modeled by a set of communicating nondeterministic components as illustrated in figure 3.2. For each processor, there is a processor core, MMU, store buffer, caches (which become visible when accesses of non cacheable and cacheable memory modes to the same address are mixed in the same computation), and a local APIC for interrupt handling. The remaining components (shared between the cores) are main memory and other devices. Sizes of caches, buffers, and translation look aside buffers (TLBs) in the MMU are unbounded in the model, but the model is sufficiently nondeterministic to be implemented by an implementation using arbitrary specific sizes for each of these. In the same spirit, caches and MMUs nondeterministically load data within wide limits, allowing the model to be implemented using a variety of prefetch strategies. Nevertheless, the model is precise enough to permit proofs of program correctness.

As mentioned in the introduction, an accurate ISA-specification is more complex than meets the eye. Only if the executed code obeys a nontrivial set of software conditions, the hardware interprets instructions in the way specified in the manuals. In RISC machines, the alignment of accesses is a typical such condition. In pipelined machines, the effects of certain instructions only become visible at the ISA level after a certain number of instructions have been executed, or after an explicit pipeline flush. In the same spirit, a write to a page table becomes visible at the ISA level when the instruction has left the memory stages of the pipe, the write has left the store buffer, *and* previous translations effected by this write are flushed from the TLB by an INVLPG instruction (which in turn does only become visible when it has left the pipe). In a multi-core machine, things are even more complicated because a processor can change code and page tables of other processors. In the end, one also needs *some* specification of what the hardware does if the software violates the conditions, since the kernel generally cannot exclude their violation in guest code. In turn, one needs to guarantee that guest code violating software conditions does not violate the integrity of other user processes or the kernel itself. Each of these conditions exposes to the ISA programmer details of the hardware, in particular of the pipeline, in a limited way.

Obviously, if one wants to verify ISA programs, one has to check that they satisfy the software conditions. This raises the problem of how to identify a *complete* set of these conditions. In order to construct this set, we propose to reverse engineer the processor hardware, prove that it interprets the instructions set, and collect the software conditions we use in the correctness proof of the hardware. Reverse engineering a CISC machine as specified in [Deg11] is an extremely large project, but if we replace the CISC core by a MIPS core and restrict memory modes to 'write back' (WB) and 'uncacheable' (UC) (for device accesses), reverse engineering becomes feasible. A definition of the corresponding instruction set called 'MIPS-86' can be found in chapter 4 of this thesis.

3.2.3 Gate Level Correctness for Multi-Core Processors

A detailed correctness proof of a multi-core processor for an important subset of the MIPS-86 instruction set mentioned above can be found in the lecture notes [Pau12]. The processor cores have classical 5 stage pipelines, the memory system supports memory accesses by bytes, half words, and words, and the caches implement the MOESI protocol. Caches are connected to each other by an open collector bus and to main memory (realized by dynamic RAM) by a tri-state bus. There are no store buffers or MMUs, yet. Caches support only the 'write back' mode. The lecture notes contain a gate-level correctness proof for a sequentially consistent shared memory on 60 pages. Integrating the pipelined processor cores into this memory system is not completely trivial, and proving that this implements the MIPS-86 ISA takes another 50 pages. The present proof assumes the absence of self-modifying code.

Dealing with tri-state busses, open collector busses, and dynamic RAM involves design rules, which can be formulated but not motivated in a gate-level model. In analogy to data sheets of hardware components, [Pau12] therefore uses a detailed hardware model with minimal and maximal propagation delays, enable and disable times of drivers, and setup and hold times of registers as its basic model. This allows derivation of the design rules mentioned above. The usual digital hardware model is then derived as an abstraction of the detailed model.

3.2.4 Future Work

The correctness proof from [Pau12] has to be extended in the following ways to cover MIPS-86

- introducing a read-modify-write operation (easy),
- introducing memory fences (easy),
- extending memory modes to include an uncacheable mode UC (easy),
- extending the ISA-sp of MIPS-86 with more memory modes and providing an implementation with a coherence protocol that keeps the view of a single memory abstraction if only cacheable modes are used (easy)
- implementing interrupt handling and devices (subtle),
- implementing an MMU to perform address translation (subtle),
- adding store buffers (easy), and
- including a Tomasulo scheduler for out-of-order execution (hard).

3.3 Abstracting ISA-sp to ISA-u

One abstracts ISA-sp to ISA-u in three steps: i) eliminating the caches, ii) eliminating the store buffers, and iii) eliminating the MMUs. A complete reduction (for a naive store buffer elimination discipline and a simplified ownership discipline) is given in [Kov12].

3.3.1 Caches

To eliminate caches, an easy simulation shows that the system with caches S simulates a system without caches S'; the coupling invariant is that for every address, the S' value at that address is defined as the value cached at that address if it is cached (the value is unique, since all cache copies agree), and is the value stored in the S memory if the location is uncached. One gets the processor view of figure 3.3 (b).

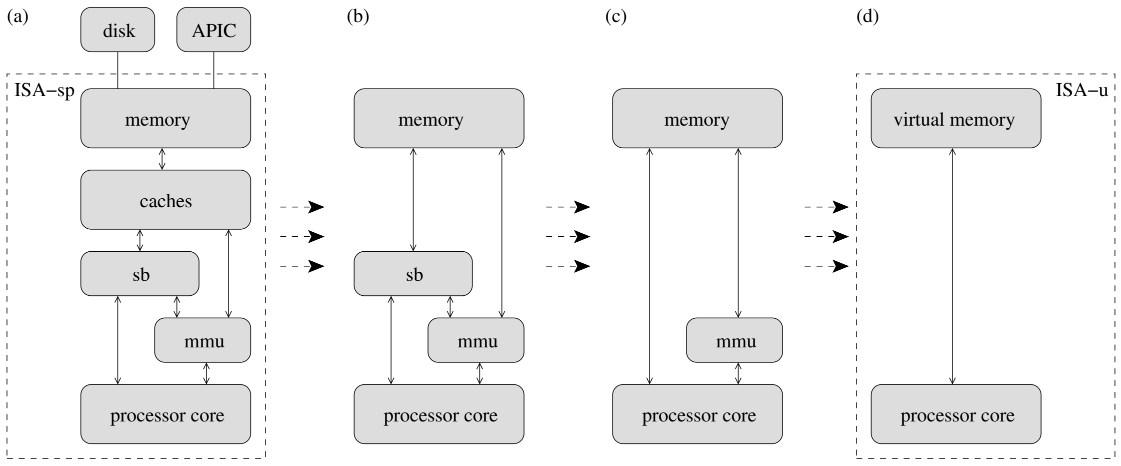


Figure 3.3: Abstracting ISA-sp to ISA-u.

3.3.2 Store Buffers and Ownership Disciplines

In single-core architectures, an easy simulation shows that the system with FIFO store buffers S simulates a system without store buffers S': the value stored at an address in S' is the value in the store buffer farthest away from memory (i.e., the last value saved) if the address appears in the store buffer, and is otherwise the value stored in the memory of S. For a single-core architecture, no extra software conditions are needed; a more careful proof can be found in [DPS09]. One gets the view of figure 3.3 (c). For the multi-core architecture, store buffers can only be made invisible if the executed code follows additional restrictions.

A trivial (but highly impractical) discipline is to use only flushing writes (which includes atomic read-modify-write operations); this has the effect of keeping the store buffers empty, thus rendering them invisible. A slightly more sophisticated discipline is to classify each address as either shared or owned by a single processor. Unshared locations can be accessed only by code in the owning processor; writes to shared addresses must flush the store buffer. The proof that this simulates a system without store buffers is almost the same as in the uniprocessor case: for each owned address, its value in the S' memory is the value in its owning processor according to the uniprocessor simulation, and for each shared address, the value is the value stored in memory.

A still more sophisticated discipline to use the same rule, but to require a flush only between a shared write and a subsequent share read on the same processor. In this case, a simple simulation via a coupling invariant is not possible, because the system, while sequentially consistent, is not linearizable. Instead, S' issues a write when the corresponding write in S actually emerges from the store buffer and hits the memory. S' issues a shared read at the same time as S; this is consistent with the writes because shared reads happen only when there are no shared writes in the store buffer. The unshared reads and writes are moved to fit with the shared writes[5]. It is

[5]Note that this means that in S', an unshared write from one processor might be reordered to happen before a shared write from another processor, even though the shared write hits memory first, so while the execution is

straightforward to extend this reduction to include shared “read-only” memory.

A final, rather surprising improvement to the last reduction discipline is to allow locations to change from one type to another programatically. For example, we would like to have a shared location representing a lock, where an ordinary operation on that lock (acquiring it) gives the thread performing that action ownership of some location protected by that lock. Moreover, we might want to allow the set of locations protected by that lock to change, perhaps determined by data values. [CS10] gives a very general reduction theorem for x86-TSO that allows these things to be done in the most flexible way possible, by allowing the program to take ownership of shared data, give up ownership of data, and change it between being read-only and read-write, in ordinary ghost code. This theorem says that if you can prove, *assuming sequential consistency*, that a concurrent program (which includes ghost code that might change memory types of locations) follows (a minor modification of) the flushing discipline above, then the program remains sequentially consistent when executed under x86-TSO. The proof of this reduction theorem is much more difficult than the previous ones, because reducing a single TSO history requires reasoning from the absence of certain races in related sequentially consistent histories.

3.3.3 Eliminating MMUs

Modern processors use page tables to control the mapping of virtual to physical addresses. However, page tables provide this translation only indirectly; the hardware has to walk these page tables, caching the walks (and even partial walks) in the hardware TLBs. x86/64 machines require the system programmer to manage the coherence of the TLBs in response to changes in the page tables. The simplest way to make MMUs invisible is to set up a page table tree that represents an injective translation (and does not map the page tables themselves), before switching on virtual memory. It is an easy folklore theorem that the resulting system simulates unvirtualized memory; a proof can be found in [DPS09]. One gets the view of figure 3.3 (d). However, this is not how real kernels manage memory; memory is constantly being mapped in and unmapped. The easiest way to do this is to map the page tables in at their physical addresses (since page table entries are based on physical, rather than virtual, page frame numbers). At the other extreme, one can model the TLBs explicitly, and keep track of those addresses that are guaranteed to be mapped to a particular address in all possible complete TLB walks (and to not have any walks that result in a page fault), and to keep track of a subset of these addresses, the “valid” addresses[6], such that the induced map on these addresses is injective. Only those addresses satisfying these criteria can be read or written. This flexibility is necessary for kernels that manage memory agressively, trying to minimize the number of TLB flushes. Essentially, this amounts to treating the TLB in the same way as a device, but with the additional proof obligation connecting memory management to reading and writing, through address validity. This, however, we currently consider future work.

sequentially consistent, it is not “memory sequential consistent” as defined in [Owe10], because it violates the triangle race condition. Allowing sequentially consistent executions with triangle races is an absolute requirement for practical reduction theorems for x86-TSO.

[6]Note that validity of an address is, in general, different for different processors in a given state, since they flush their TLB entries independently.

3.3.4 Mixed ISA-sp and ISA-u Computations

In a typical kernel, there is a stark contrast between the kernel code and the user programs running under the kernel. The kernel program needs a richer model that includes system instructions not accessible to user programs, but at the same time the kernel can be written using a programming discipline that eliminates many irrelevant and mathematically inconvenient details. For example, if the kernel is being proved memory-safe, the programming model in the kernel does not have to assign semantics to dereferencing of null pointers or overrunning array bounds, whereas the kernel must provide to user code a more complex semantics that takes such possibilities into account. Similarly, the kernel might obey a synchronization discipline that guarantees sequential consistency, but since user code cannot be constrained to follow such a discipline, the kernel must expose to user code the now architecturally-visible store buffers. In the case of a hypervisor, the guests are themselves operating systems, so the MMU, which is conveniently hidden from the hypervisor code (other than boot code and the memory manager), is exposed to guests.

3.3.5 Future Work

Extension of the work in [Kov12] to a full a proof of the naive store buffer reduction theorem should not be hard. In order to obtain the reduction theorem with dirty bits, it is clearly necessary to extend the store buffer reduction theorem of [CS10] to machines with MMUs. This extension is not completely trivial as MMUs directly access the caches without store buffers. Moreover MMUs do not only perform read accesses; they write to the 'accessed' and 'dirty' bits of page table entries. One way to treat MMUs and store buffers in a unified way is to treat the TLB as shared data (in a separate address space) and the MMU as a separate thread (with an always-empty store buffer). This does not quite work with the store buffer reduction theorem above; because the TLB is shared data, reading the TLB to obtain an address translation for memory access (which is done by the program thread) would have to flush the store buffer if it might contain a shared write, which is not what we want. However, the reduction theorem of [CS10] can be generalized so that a shared read does not require a flush as long as the same read can succeed when the read "emerges" from the store buffer; this condition is easily satisfied by the TLB, because a TLB of unbounded capacity can be assumed to grow monotonically between store buffer flushes.

3.4 Serial Language Stack

3.4.1 Using Consistency Relations to Switch Between Languages

As explained in the introduction, realistic kernel code consists mostly of high-level language code, with some assembler and possibly some macro assembler. Thus, complete verification requires semantics for programs composed of several languages L_k with $0 \leq k < n \in \mathbb{N}$. Since all these languages are, at some point, compiled to some machine code language L, we establish for each L_k that programs $p \in L_k$ are translated to programs $q \in L$ in such a way that *computations* (d^i) – i.e. sequences of configurations $d^i, i \in \mathbb{N}$ – of program q simulate computations (c^i) of program p via a consistency relation $consis(c,d)$ between high level configurations c and low level configurations d. Translation is done by compilers and macro assemblers. Translators can be optimizing or not. For non-optimizing translators, steps of language L_k are translated into one or more steps of language L. One shows that, for each computation (c^i) in the source language, there is a step function s such that one has

$$\forall i : consis(c^i, d^{s(i)})$$

If the translator is optimizing, consistency holds only at a subset of so called 'consistency points' of the source language. The translator does not optimize over these points. Let $CP(i)$ be a predicate indicating that configuration c^i is a consistency point. Then an optimizing translator satisfies

$$\forall i : CP(i) \rightarrow consis(c^i, d^{s(i)})$$

Note that the consistency relation and the consistency points together specify the compiler. The basic idea to formulate mixed language semantics is very simple. We explain it here only for two language levels (which can be thought of as machine code and high level abstract semantics, as in figure 3.4); extension to more levels is straightforward and occurs naturally when there is a model stack of intermediate languages for compilation[7]. Imagine the computations (c^i) of the source program and (q^j) of the target program as running in parallel from consistency point to consistency point. We assume the translator does not optimize over changes of language levels, so configurations c^i where the language level changes are consistency points of the high level language. Now there are two cases

- switching from L_k to L in configuration c^i of the high level language: we know $consis(c^i, d^{s(i)})$ and continue from $d^{s(i)}$ using the semantics of language L.
- switching from L to L_k in configuration d^j of the low level language: we try to find a configuration c' of the high level language such that $consis(c', d^j)$ holds. If we find it we continue from c' using the semantics of the high level langue. If we do not find a consistent high level language configuration, the low level portion of the program has messed up the simulation and the semantics of the mixed program switches to an error state.

In many cases, switching between high-level languages L_k and L_l by going from L_k to shared language L, and from the resulting configuration in L to L_l can be simplified to a direct transition from a configuration of L_k to a configuration of L_l by formalizing just the compiler calling convention and then proving that the resulting step is equivalent to applying the two involved consistency relations (e.g., see [SS12]). This explains why the specification of compilers necessarily enters into the verification of modern kernels.

3.4.2 Related Work

The formal verification of a non-optimizing compiler for the language C0, a type safe PASCAL-like subset of C, is reported in [Lei08]. The formal verification of an optimizing compiler for

[7]Note in particular, that, when two given high level language compilers have an intermediate language in common, we only need to switch downwards to the highest level shared intermediate language.

the intermediate language C-minor is reported in [Ler09]. Mixed language semantics for C0 + in line assembly is described in [GHLP05] as part of the Verisoft project [Ver07]. Semantics for C0 + external assembly functions is described in [Alk09]. Both mixed language semantics were used in sebsequent verification work of the Verisoft project. As the underlying C0 compiler was not optimizing, there was only a trivial calling convention. Note that the nontrivial calling conventions of optimizing compilers produce proof goals for external assembly functions: one has to show that the calling conventions are obeyed by these functions. Only if these proof goals are discharged, one can show that the combined C program with the external functions is compiled correctly.

3.4.3 A Serial Language Stack for Hypervisor Verification

Similar to [Ler09], we use an intermediate language *C-IL* with address arithmetic and function pointers. Chapter 5 of this thesis gives a formal definition of *C-IL*. The semantics of *C-ILtogether* with a macro assembler obeying the same calling conventions is described in [SS12]. Calls from *C-IL* to macro assembly and vice versa are allowed. To specify the combined semantics, one has to describe the ABI (i.e. the layout of stack frames and the calling convention used). In [Sha12], an optimizing compiler for *C-IL* is specified, a macro assembler is constructed and proven correct, and it is shown how to combine *C-IL* compiler + macro assembler to a translator for combined *C-IL* + macro assembly programs. As explained in the introduction, extension of this language stack to *C-IL* + macro assembly + assembly is necessary to argue about saving and restoring the base and stack pointers during a process switch or a task switch. This can be done using the construction explained in subsection 3.4.1.

3.4.4 Future Work

We believe that, for the consistency relations normally used for specifying compilers and macro assemblers, the mixed language semantics defined in subsection 3.4.1 is essentially deterministic in the following sense: if $consis(c, d)$ holds, then d is unique up to portions of c which will not affect the future I/O behavior of the program (e.g. non reachable portions of the heap). A proof of such a theorem should be written down.

3.5 Adding Devices

The obvious way to add devices is to represent them as concurrent threads, and to reason about the combined program in the usual way. This approach is justified only if the operational semantics of the language stack executing the program in parallel with the device models simulates the behavior of the compiled code running on the hardware in parallel with the devices. This is already nontrivial, but is further complicated by the addition of interrupts and interrupt handlers. It is obviously undesirable to introduce interrupt handling as a separate linguistic concept, so the natural way to model an interrupt handler is as a concurrent thread. However, the relationship between a program and an interrupting routine is somewhat closer than that between independent threads; for example, data that might be considered "thread local" in the context of a concurrent program might nevertheless be modified by an interrupt handler, which requires careful management of when interrupts are enabled and disabled. Another complication that arises with many kinds of devices is the need to capture and model real-time constraints.

3.5.1 Related Work

Formal methods have been extremely successful identifying large classes of frequent bugs in drivers [BLR11]. In contrast, *complete* formal verification results of even of the most simple drivers have only recently appeared. An obvious prerequisite is a formal model of devices that can be integrated with processor models both at the hardware and ISA level [HIdRP05]. At the hardware level, processor and devices work in parallel in the same clock domain. At the hardware level, the models of some devices are completely deterministic; an example is a dedicated device producing timer interrupts. But these models also can have nondeterministic portions, e.g. the response time (measured in hardware cycles) of a disk access. When we lift the hardware construction to the ISA model, one arrives in a natural way at a nondeterministic concurrent model of computation: processor and device steps are interleaved in an order not known to the programmer at the ISA level or above. This order observed at the ISA level can be constructed from the hardware construction and the nondeterminism stemming from the device models. A formal proof for the correctness of such a concurrent ISA model for a single core 'processor + devices' was given in [Tve09, HT09]. The hardware construction for catching the external interrupts and the corresponding correctness argument are somewhat tricky due to an - at first sight completely harmless - nonstandard specification in the instruction set of the underlying processor, which was taken from [MP00]. There, external interrupts are defined to be of type 'continue', i.e. the interrupted instruction is completed before the interrupt is handled. In the MIPS-86 instruction set, this definition was changed to reflect standard specification, where external interrupts are of type 'repeat', i.e. the interrupted instruction is not executed immediately, but is instead repeated after the run of the handler.

Now consider a system consisting of a (single core) processor and k devices as shown in figure 3.5, and consider a run of a driver for device i. Then one wants to specify the behavior of the driver by pre and post conditions. For example, if the driver writes a page from the processor to the disk, the precondition would state that the page is at a certain place in processor memory and the post condition would specify that it is stored at a certain place on the memory of the disk. To prove this one has to show that the other devices do not interfere with the driver run. Indeed one can show an order reduction theorem showing that if during the driver run i) interrupts of other devices are disabled and ii) the processor does not poll the devices, then in a driver run with arbitrary interleaving all steps of devices $\neq i$ can be reordered such that they occur after the driver run without affecting the result of the computation. A formal proof of this result is given in [ASS08, Alk09]. At the same place and in [AHL$^+$09] the integration of devices into the serial model stack of the Verisoft project (resulting in C0 + assembly + devices) and the formal verification of disk drivers is reported.

Note that the above reorder theorem for device steps has the nontrivial hypothesis that there are no side channels via the environment, i.e. the outside world between the devices. This is not explicitely stated; instead it it is implicitly assumed by formalizing figure 3.5 in the obvious way. For example, if device 2 is a timer triggering a gun aimed at device 1 during the driver run of device 1, the post condition is false after the run because the device is not there any more. Side

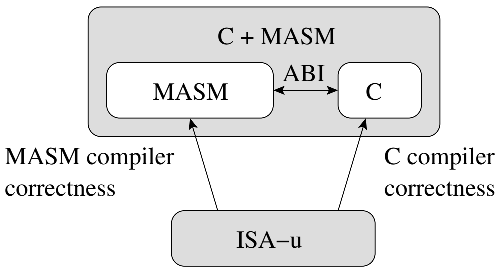


Figure 3.4: Combined semantics of C and Macro Assembler.

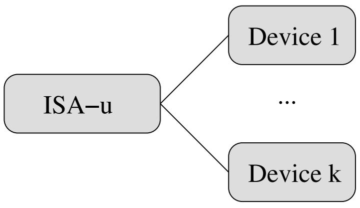


Figure 3.5: ISA model with devices.

channels abound of course, in particular in real time systems. If device 1 is the motor control and device 2 the climate control in a car, then the devices *are* coupled in the environment via the power consumption.

3.5.2 Multi-Core Processors and Devices

Only some very first steps have been made towards justifying verification of multi-core processors along with devices. The current MIPS-86 instruction set contains a generic device model and a specification of a simplified APIC system consisting of an I/O APIC (a device shared between processors that distributes device interrupts) and local APICs (processor local devices for handling external and inter processor interrupts). Rudimentary models of interrupt controllers for various architectures have been built as parts of VCC verifications.

3.5.3 Future Work

Instantiating the generic device model of MIPS-86 with an existing formal disk model is straightforward. Justification of the concurrent 'multi-core processor + devices model' of the MIPS-86 ISA requires of course the following steps

- extending the MIPS-ISA hardware from [Pau12] with the pipelined interrupt mechanism from [BJK$^+$03]. The catching and triggering of external interrupts needs to be modified to reflect that external interrupts are now of type 'repeat'. This should lead to a simplification of the construction.
- reverse engineering of hardware APICs and the mechanism for delivering inter processor interrupts (IPIs).
- showing that the hardware constructed in this ways interprets the MIPS-86 ISA. This proof should be simpler than the proof in [Tve09].

Three more tasks remain open: i) proving a reorder theorem for driver runs, ii) the reduction theorem from multi-core ISA-sp to ISA-u has to be generalized to the situation, where the hardware contains devices, and iii) devices must be integrated in the serial model stack (resulting in *C-IL*

+ macro assembly + assembly + devices) along the lines of [ASS08, Alk09, AHL$^+$09]. These results would justify language-level reasoning about device drivers in multi-core systems.

Formally verifying secure booting is another interesting research direction: 'secure boot' guarantees that the verified hypervisor (if we have it) is indeed loaded and run. This involves the use of verified hardware and libraries for crypto algorithms. Such libraries have already formally been verified in the Verisoft project [Ver07].

3.6 Extending the Serial Language Stack to Multi-Core Computations

Language stacks deal with languages at various levels and translations between them. A 'basic' language stack for multi-core computations consists simply of i) a specification of some version of 'structured parallel C', ii) a compiler from this version of parallel C to the ISA of a multi-core machine and iii) a correctness proof showing simulation between the source program and the target program. Definitions of versions of structured parallel C (or intermediate languages close to it) and correctnes proofs for their compilers proceed in the flowing way:

- one starts with a small-step semantics of the serial version of the high level language; configurations of such semantics have program rests/continuations, stacks, and global memory. Configurations for parallel C are easily defined: keep program rests and stack local for each thread; share the global variables among threads. Computations for unstructured parallel C are equally easy to define: interleave steps of the small steps semantics of the individual threads in an arbitrary order.

- compiling unstructured parallel C to multi-core machines tends to produce very inefficient code. Thus one structures the computation by restricting accesses to memory with an ownership discipline very similar to the one of [CS10]. Different versions of parallel C differ essentially by the ownership discipline used. As a directive for the compiler, variables which are allowed to be unowned and shared (such that they can e.g. be used for locks) are declared as volatile. Accesses to volatile variables constitute *I/O-points* of the computation.

- Compilers do not optimize over I/O-points, thus I/O-points are consistency points. Except for accesses to volatile variables, threads are simply compiled by serial compilers. Code produced for volatile accesses has two portions: i) possibly a fence instruction draining the local store buffer; clearly this is only necessary if the target machine has store buffers, and ii) an appropriate atomic ISA instruction.

- Compiler correctness is then argued in the following steps: i) The compiled code obeys the ownership discipline of the target language in such a way that volatile accesses of the compiled code correspond to volatile accesses of the source code, i.e. I/O points are preserved. Then one proves both for the source language and the target language that, due to the ownership discipline, memory accesses between I/O points are to local, owned, or shared-read-only addresses only. This implies at both language levels an order reduction theorem restricting interleavings to occur at I/O points only. We call such an interleaving an *I/O-block schedule*. iii) One concludes simulation between source code and target code using the fact that I/O points are compiler consistency points and thus in each thread compiler consistency is maintained by the serial (!) computations between I/O-points.

3.6.1 Related Work

The 'verified software toolchain' project [App11] presently deals with a 'basis' language stack. C minor is used as serial source language. The serial compiler is the formally verified optimizing compiler from the CompCert project [Ler09]. Permissions on memory are modified by operations on locks – this can be seen as a kind of ownersip discipline. The target machine has sequentially consistent shared memory in the present work; draining store buffers is identified as an issue for future work. Proofs are formalized in Coq. In the proofs the permission status of variables is maintained in the annotation language. We will return to this project in the next section.

3.6.2 Extending the Language Stack

A crucial result for the extension of a language stack for 'C + macro assembly + ISA-sp + devices' to the multi-core world is a general order reduction theorem that allows to restrict interleavings to I/O-block schedules for programs obeying the ownership discipline, even if the changes of language level occur in a single thread of a concurrent program. Besides the volatile memory accesses, this requires the introduction of additional I/O points: i) at the first step in hypervisor mode ($ASID = 0$) after a switch from guest mode ($ASID \neq 0$) because we need compiler consistency there, and ii) at any step in guest mode because guest computation is in ISA-sp and we no not want to restrict interleavings there. An appropriate general reorder theorem is reported in [Bau12]. Application of the theorem to justify correctness of compilation across the language stack for a version of parallel *C-IL* without dirty bits and a corresponding simple handling of store buffers is reported in [Kov12].

3.6.3 Future Work

The same reorder theorem should allow to establish correctness of compilation across the language stack for a structured parallel *C-IL* with dirty bits down to ISA-u with dirty bits. However, in order to justify that the resulting program is simulated in ISA-sp with store buffers one would need a version of the Cohen-Schirmer theorem for machines with MMUs.

The language stack we have introduced so far appears to establish semantics and correctness of compilation for the complete code of modern hypervisors, provided shadow pages tables (which we introduced in the introduction) are not shared between processors. This restriction is not terribly severe, because modern processors tend more and more to provide hardware support for two levels of translations, which renders shadow page tables unnecessary in the first place. As translations used by different processors are often identical, one can save space for shadow page tables by sharing them among processors. This permits the implemention of larger shadow page tables leading to fewer page faults and hence to increased performance. We observe that this introduces an interesting situation in the combined language stack: the shadow page tables are now a C data structure that is accessed concurrently by C programs in hypervisor mode *and* the MMUs of other processors running in guest mode. Recall that MMUs set accessed and dirty bits; thus both MMU und C program can read *and* write. Now interleaving of MMU steps and hypervisor steps must be restricted. One makes shadow page tables volatile and reorders MMU accesses of other MMUs immediately after the volatile writes of the hypervisor. To justify this, one has to argue that the MMUs of other MMUs running in guest mode never access data structures other than shadow page tables; with the modelling of the MMU as an explicit piece of concurrent code, this proof becomes part of ordinary program verification.

3.7 Soundness of VCC and its Use

Formal verification with unsound tools and methods is meaningless. In the context of proving the correctness of a hypervisor using VCC as a proof tool, two soundness arguments are obviously called for: i) a proof that VCC is sound for arguing about pure structured parallel C. ii) a method to 'abuse' VCC to argue about machine components that are not visible in C together with a soundness proof for this method.

3.7.1 Related Work

In the Verisoft project, the basic tool for proving program code correct was a verification condition generator for C0 whose proof obligations were discharged using the interactive theorem prover Isabell-HOL. The soundness of the verification condition generator for C0 was established in a formal proof [Sch06]. The proof tool was extended to handle 'external variables' and 'external functions' manipulating these variables. Components of configurations not visible in the C0 configuration of kernels (processor registers, configurations of user processes, and device state) were coded in these external variables. The proof technology is described in great detail in [AHL$^+$09].

A formal soundness proof for a program analysis tool for structured parallel C is developed in the 'verified software toolchain' project [App11].

3.7.2 Soundness of VCC

An obvious prerequisite for a soundness proof of VCC is a complete specification of VCC's annotation language and its semantics. VCC's annotations have two parts: i) a very rich language extension for ghost code, where ghost instructions manipulate both ghost variables and ghost fields which are added to records of the original implementation language, and ii) a rich assertion language referring to both implementation and ghost data. A complete definition of 'C-IL + ghost' can be found in chapter 6 of this thesis together with a proof that 'C-IL + ghost' is simulated by *C-IL* provided ghost code always terminates.

The VCC assertion language is documented informally in a reference manual and a tutorial [Micb]; the reference manual also has some rudimentary mathematical underpinnings. More of these underpinnings are described in various articles [CDH$^+$09, CAB$^+$09, CMTS09]. However, there is currently no single complete mathematical definition. Thus establishing the soundness of VCC still requires considerable work (see subsection 3.7.5).

3.7.3 Using VCC for Languages Other Than C

As C is a universal programming language, one can use C verifiers to prove the correctness of programs in any other programming language L by i) writing in C a (sound) simulator for programs in L, followed by ii) arguing in VCC about the simulated programs, and iii) proving property transfer from VCC results to results over the original code given in language L. Extending 'C + macro assembly + assembly' programs with a simulator for program portions not written in C then allows to argue in VCC about such programs. A VCC extension for x64 assembly code is described in [Mau11, MMS08] and was used to verify the 14K lines of macro assembly code of the Hyper-V hypervisor. In the tool, processor registers were coded in a straightforward way in a struct, a so called *hybrid* variable which serves the same role as an external variable in the Verisoft tool chain mentioned above. Coding the effect of assembly or macro assembly instructions amounts to trivial reformulation of the semantics of the instructions as C functions. Calls and returns of macro assembly functions are coded in a naive way. The extension supports gotos within a routine and function calls, but does not support more extreme forms of control flow, e.g. it cannot be used to prove the correctness of thread switch via change to the stack pointer.

Currently, however, there is a slight technical problem: VCC does currently not support hybrid variables directly. We cannot place hybrid variables in ghost memory, because information clearly flows from hybrid variables to implementation variables, and this would violates a crucial hypothesis in the simulation theorem between original and annotated program. If we place it into implementation memory, we have to guarantee that it is not reachable by address arithmetic from other variables. Fortunately, there currently is a possible workaround: physical addresses of modern processors have at most 48 bits and VCC allows up to 64 bit addresses. Thus hybrid variables can be placed in memory at addresses larger than $2^{48} - 1$. Future versions of VCC are planned to support hybrid memory as a third kind of memory (next to implementation and ghost memory) on which the use of mathematical types is allowed; in turn, the formalization of 'C-IL + ghost' should be extended to include this special hybrid memory.

The papers [Sha12, PSS12] show the soundness of an assembler verification approach in the spirit of Vx86 relative to the mixed 'C-IL + macro assembly' language semantics of our language stack.

3.7.4 Verifying Device Drivers with VCC

One way to reason about device drivers is to use techniques from concurrent program reasoning. In a concurrent program, one can rarely specify a function on shared state via a pre and post condition on the state of the device, since other concurrent operations may overlap the execution of the function. Instead, one can specify the function as a linearizable operation that appears to take place atomically at some point between invocation of the function and its return. In VCC, the usual idiom for such verification is to introduce a ghost data object representing the abstract state provided by the device in combination with the driver; the state of this object is coupled to the concrete states of the driver and the device via a coupling invariant. Associated with a function call is a ghost object representing the operation being performed; this operation includes a boolean field indicating whether the operation has completed; an invariant of the operation is that in any step in which this field changes, it goes from false to true and the abstract state of the device changes according to the semantics of the operation. The abstract device has a field that indicates which operation (if any) is "executing" in any particular step, and has an invariant that the abstract state of the device changes only according to the invariant of the operation being performed (which might also be an operation external to the system).

However, another current limitation of VCC is that it allows ordinary C memory operations only on locations that act like memory. This means that it cannot directly encode devices where writing or reading a *memory mapped I/O* (MMIO) address has an immediate side effect on the device state; currently, the semantics of such operations have to be captured via intrinsics.

3.7.5 Future Work

A proof of the soundness of VCC apparently still requires the following three major steps:

- documenting the assertion language. This language is rich and comprises i) the usual assertions for serial code, ii) an ownership calculus for objects which is used in place of separation logic to establish frame properties, and iii) a nontrivial amount of constructs supporting arguments about concurrency.

- documenting the assertions which are automatically generated by VCC in order to i) guarantee the termination of ghost code, and ii) enforce an ownership discipline on the variables and a flushing strategy for store buffers.

- proving the soundness of VCC by showing i) ghost code of verified programs terminates; thus we have simulation between the annotated program and the implementation code (as in chapter 6 of this thesis), ii) assertions proven in VCC hold in the parallel *C-IL* semantics; this is the part of the soundness proof one expects from classical theory (this portion of the soundness proofs for VCC should work along the lines of soundness proofs for rely/guarantee logics – a proof outline is given in the VCC manual [Micb]), and iii) variables of verified programs obey ownership discipline and code of translated programs obeys a flushing discipline for store buffers; this guarantees correct translation to the ISA-sp level of the multi-core machine.

3.8 Hypervisor Correctness

Figure 3.7 gives a very high-level overview of the structure of the overall theory. After establishing the model stack and the soundness of proof tools and their application, what is left to do is the actual work of program verification. Thus we are left in this survey paper with the task to outline the proof of a hypervisor correctness theorem which expresses virtualization of several ISA-sp machines enriched with system calls stated on 'C + macro assembly + ISA-u + ISA-sp'. Fortunately we can build on substantial technology from other kernel verification projects.

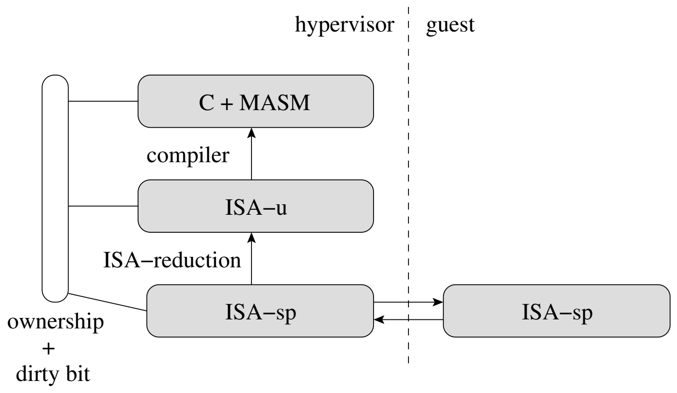


Figure 3.6: Using ownership and dirty bit conditions to achieve simulation in a hypervisor model stack by propagating ownership downwards.

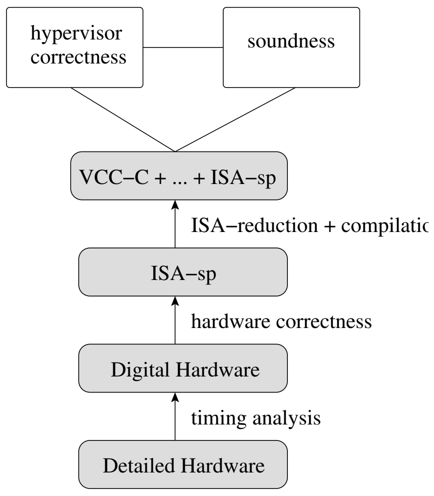


Figure 3.7: High-level overview of the model stack and main theorems.

3.8.1 Related Work

The well known 'seL4' project [KAE+10] succeeded in formally verifying the C portion of an industrial microkernel comprising about 9000 lines of code (LOC), although that verification ignored a number of important hardware issues such as the MMU and devices, and used a rather unrealistic approach to interrupt handling, largely because that verification was based entirely on sequential program reasoning. In the Verisoft project [Ver07] both the C portion and the assembly portion of two kernels was formally verified: i) the code of a small real real time kernel called OLOS comprising about 450 LOC [DSS09] and ii) the code of a general purpose kernel called VAMOS of about 2500 LOC [DDB08, Dör10]. In that project the verification of C portions and assembly portions was decoupled [GHLP05] in the following way: A generic concurrent model for kernels and their user processes called CVM (for 'communicating virtual machines') was introduced, where a so called 'abstract kernel' written in C communicates with a certain number of virtual machines $vm(u)$ (see figure 3.8) programmed in ISA. At any time either the abstract kernel or a user process $vm(u)$ is running. The abstract kernel uses a small number of external functions called 'CVM primitives' which realize communication between the kernel, user processes and devices. The semantics of these user processes is entirely specified in the concurrent CVM model. To obtain the complete kernel implementation, the abstract kernel is linked with a few new functions and data structures, essentially process control blocks, page tables and a page fault handler in case the kernel supports demand paging (e.g. like VAMOS does); CVM primitives are implemented in assembly language. The resulting kernel in called the 'concrete kernel'. Correctness theorems state that the CVM model is simulated in ISA by the compiled concrete kernel together with the user machines running in translated mode. Since the C portions of seL4 are already formally verified, one should be able to obtain a similar overall correctness result by declaring appropriate parts of seL4's C implementation as abstract kernel

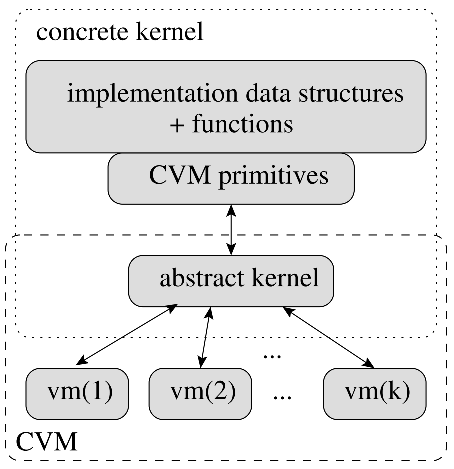


Figure 3.8: The CVM kernel model.

without too much extra effort.

3.8.2 Hypervisor Verification in VCC

That VCC allows to verify the implementations of locks has been demonstrated in [HL09]. Partial results concerning concurrent C programs and their interrupt handlers are reported in [ACHP10]. Program threads and their handlers are treated like different threads and only the C portions of the programs are considered; APICs and the mechanism for delivery of *inter processor interrupts* (IPIs) are not modeled. Thus the treatment of interrupts is still quite incomplete. The full formal verification of a small hypervisor written in 'C + macro assembly + assembly' in VCC using the serial language stack of section 3.4 (which is also illustrated in figure 3.6) and the proof technology described in subsection 3.7.3 is reported in [Sha12, PSS12]. The formal verification of shadow page table algorithms without sharing of shadow page tables between processors is reported in [Kov12, ACKP12].

3.8.3 Future Work

The following problems still have to be solved:

- Adding features to VCC that allow memory mapped devices to be triggered by reading or writing to an address that already has a value identical to the data written.

- Proving the correctness of a 'kernel layer' of a hypervisor. In order to provide guests with more virtual processors than the number *np* of physical processors of the host, one splits the hypervisor in a kernel layer and a virtualization layer. The kernel layer simulates large numbers *n* of 'C + macro assembly + ISA-sp' threads by *np* such threads. Implementation of thread switch is very similar to the switching of guests or of user processes. A data structure called thread control block (TCB) takes the role of process control block.

Correctness proofs should be analogous to kernel correctness proofs but hinge on the full power of the semantics stack.

- The theory of interrupt handling in concurrent C programs and its application in VCC has to be worked out. The conditions under which an interrupt handler can be treated as an extra thread needs to be worked out. This requires to refine ownership between program threads and their interrupt handlers. For reorder theorems, the start and return of handler threads has to become an I/O-point. Finally, for liveness proofs, the delivery of IPI's (and the liveness of this mechanism) has to be included in the concurrent language stack and the VCC proofs.
- Proving actual hypervisor correctness by showing that the virtualization layer (which possibly uses shadow page tables depending on the underlying processor) on top of the kernel layer simulates an abstract hypervisor together with a number of guest machines and their user processes. Large portions of this proof should work along the lines of the kernel correctness proofs of the Verisoft project. New proofs will be needed when one argues about the state of machine components that cannot explicitly be saved at a context switch. Store buffers of sleeping guests should be empty, but both caches and TLBs of sleeping processors may contain nontrivial data, some or all of which might be flushed during the run of other guests.

3.9 Conclusion

Looking at the last section, we see that i) the feasibility of formal correctness proofs for industrial kernels has already been demonstrated and that ii) correctness proofs for hypervisors are not that much more complex, provided an appropriate basis of mixed language semantics and proof technology has been established. It is true that we have spent 6 of the last 7 sections of this chapter for outlining a paper theory of this basis. But this basis seems to be general enough to work for a large variety of hypervisor constructions such that, for individual verification projects, 'only' the proofs outlined in section 3.8 need to be worked out.

4 MIPS-86 – a Formal Model of a Multi-Core MIPS Machine

This chapter provides a horizontal slice of the model stack presented in the last chapter in terms of a simple multi-core MIPS model we call MIPS-86. It aims at providing an overall specification for the reverse-engineered hardware models provided in [Pau12]. Essentially, we take the simple sequential MIPS processor model from [Pau12] and extend it with a memory and device model that resembles the one of modern x86 architectures. The model has the following features:

- Sequential processor core abstraction with atomic execute-and fetch transitions

  In order to justify modeling instruction execution by an atomic transition that combines fetching the instruction from memory and executing it, the absence of self-modifying code is a prerequisite. When instructions being fetched cannot be changed by the execution of other cores, these fetch cycles can be reordered to occur right before the corresponding execute cycle. This, in turn, means that the semantics of fetch and execute steps can be combined into single atomic steps.

- Memory-management unit (MMU) with translation-lookaside buffer (TLB)

  The memory-management unit considered performs a 2-level translation from virtual to physical addresses, caching partial and complete translations (which are called *walks*) in a *translation lookaside buffer* (TLB). The *page table origin*, i.e. the address of the first-level page table, is taken from the special purpose register *pto*. In order to allow an update of page-tables to be performed in a consistent manner, the machine is extended by two instructions: A *flush*-operation that empties the TLB of all walks, and an *invalidate-page*-operation that removes all walks to a certain virtual page address from the TLB.

- Store buffer (SB)

  In order to argue about store-buffer reduction, we provide a processor model with store-buffer. The store-buffer we consider is simple in the sense that it does not reorder or combine accesses but instead simply acts as a first-in-first-out queue for memory write operations to physical addresses. We provide two serializing instructions: A *fence*-operation that simply drains the store-buffer, and a *read-modify-write*-operation that atomically updates a memory word on a conditional basis while also draining the store-buffer.

- Processor-local advanced programmable interrupt controller (local APIC)

  In order to have a similar boot mechanism as the x86-architecture, we imitate the inter-processor-interrupt (IPI) mechanism of the x86-architecture. We extend our MIPS model by a strongly simplified local APIC for each processor. The local APIC provides interrupt signals to the processor and acts as a device for sending inter-processor-interrupts between processors. Local APIC ports are mapped into a processor's memory space by means of memory-mapped I/O.

- I/O APIC

  The I/O APIC is a component that is connected to the devices of the system and to the local APICs of the processors of the system. It provides the means to configure distribution of device interrupts to the processors of the multi-core system, i.e. whether a given device interrupt is masked and which processor will receive the interrupt. We do not consider edge-triggered interrupts, however, we do model the end-of-interrupt (EOI) protocol between local APIC and I/O APIC: After sending an interrupt signal to a local APIC, the I/O APIC will not sample a raised device interrupt again until the corresponding EOI message has been received from the local APIC.

- Devices

  We use a generic framework along the lines of the Verisoft device model [HRP05]: Device configurations are required to have a certain structure which can be instantiated, e.g. certain device transitions that specify side-effects associated with reading or writing device ports must be provided. Every device consists of ports and an interrupt line it may raise as well as some internal state that may be instantiated freely. Devices may interact with an external environment by receiving inputs and providing outputs.

In the following, we proceed by providing tables that give an overview over the instruction-set-architecture of MIPS-86, followed by operational semantics of the non-deterministic MIPS-86 model.

4.1 Instruction-Set-Architecture Overview and Tables

The instruction-set-architecture of MIPS-86 provides three different types of instructions: I-type instructions, J-type instructions and R-type instructions. I-type instructions are instructions that operate with two registers and a so-called *immediate constant*, J-type instructions are absolute jumps, and R-type instructions rely on three register operands.

4.1.1 Instruction Layout

The instruction-layout of MIPS-86 depends on the type of instruction. In the subsequent definition of the *MIPS-86* instruction layout, rs, rt and rd specify registers of the MIPS-86 machine.

I-Type Instruction Layout

Bits	31 ... 26	25 ... 21	20 ... 16	15 ... 0
Field Name	opcode	*rs*	*rt*	immediate constant *imm*

R-Type Instruction Layout

Bits	31 ... 26	25 ... 21	20 ... 16	15 ... 11	10 ... 6	5 ... 0
Field Name	opcode	*rs*	*rt*	*rd*	shift amount *sa*	function code *fun*

J-Type Instruction Layout

Bits	31 ... 26	25 ... 0
Field Name	opcode	instruction index *iindex*

Effect of Instructions

A quick overview of available instructions is given in tables 4.1 (for *I*-type), 4.2 (for *J*-type), and 4.3 (for *R*-type). Note that these tables – while giving a general idea what is available and what it approximately does – are not comprehensive. In particular, note that for all instructions whose mnemonic ends with ”u”, register values are interpreted as binary numbers whereas in all other cases they are interpreted as two's-complement numbers. Note also that MIPS-86 is still incomplete in the sense that in order to accommodate the distributed cache model of the hardware construction, the architecture needs to be extended to allow proper management of cache-bypassing memory access (e.g. to devices). The abstract model provided here is one where caches are already abstracted into a view of a single coherent memory. The exact semantics of all instructions present in the provided instruction-set architecture tables is given later in the transition function of the MIPS-86 processor core.

4.1.2 Coprocessor Instructions and Special-Purpose Registers

Note that in contrast to most MIPS-architectures, in MIPS-86 coprocessor-instructions are provided as *R*-type instructions. Coprocessor instructions in MIPS-86 deal with moving data between special-purpose register file and general-purpose register file and exception return. The available special purpose registers of MIPS-86 are listed in table 4.4.

4.1.3 Interrupts

Traditionally, hardware architectures provide an *interrupt* mechanism that allows the processor to react to events that require immediate attention. When an *interrupt signal* is raised, the hardware construction reacts by transferring control to an interrupt handler – on the level of hardware, this basically means that the program counter is set to the specific address where the hardware expects the interrupt handler code to be placed by the programmer and that information about the nature of the interrupt is provided in special registers. Since interrupts are mainly supposed to be handled by an operating system instead of by user processes, such a *jump to interrupt service routine* (JISR) step also tends to involve switching the processor to *system mode*.

Table 4.1: *I*-Type Instructions of MIPS-86.

opcode	Mnemonic	Assembler-Syntax	d	Effect
Data Transfer				
100 000	lb	lb *rt rs imm*	1	rt = sxt(m)
100 001	lh	lh *rt rs imm*	2	rt = sxt(m)
100 011	lw	lw *rt rs imm*	4	rt = m
100 100	lbu	lbu *rt rs imm*	1	rt = 0^{24}m
100 101	lhu	lhu *rt rs imm*	2	rt = 0^{16}m
101 000	sb	sb *rt rs imm*	1	m = rt[7:0]
101 001	sh	sh *rt rs imm*	2	m = rt[15:0]
101 011	sw	sw *rt rs imm*	4	m = rt
Arithmetic, Logical Operation, Test-and-Set				
001 000	addi	addi *rt rs imm*		rt = rs + sxt(imm)
001 001	addiu	addiu *rt rs imm*		rt = rs + sxt(imm)
001 010	slti	slti *rt rs imm*		rt = (rs < sxt(imm) ? 1 : 0)
001 011	sltui	sltui *rt rs imm*		rt = (rs < zxt(imm) ? 1 : 0)
001 100	andi	andi *rt rs imm*		rt = rs $\wedge$ zxt(imm)
001 101	ori	ori *rt rs imm*		rt = rs $\vee$ zxt(imm)
001 110	xori	xori *rt rs imm*		rt = rs $\oplus$ zxt(imm)
001 111	lui	lui *rt imm*		rt = imm0^{16}

opcode	rt	Mnemonic	Assembler-Syntax	Effect
Branch				
000 001	00000	bltz	bltz *rs imm*	pc = pc + (rs < 0 ? imm00 : 4)
000 001	00001	bgez	bgez *rs imm*	pc = pc + (rs ≥ 0 ? imm00 : 4)
000 100		beq	beq *rs rt imm*	pc = pc + (rs = rt ? imm00 : 4)
000 101		bne	bne *rs rt imm*	pc = pc + (rs ≠ rt ? imm00 : 4)
000 110	00000	blez	blez *rs imm*	pc = pc + (rs ≤ 0 ? imm00 : 4)
000 111	00000	bgtz	bgtz *rs imm*	pc = pc + (rs > 0 ? imm00 : 4)

Here, $m = m_d(ea(c, I))$.

Table 4.2: *J*-Type Instructions of MIPS-86

opcode	Mnemonic	Assembler-Syntax	Effect
Jumps			
000 010	j	j *iindex*	pc = bin_{32}(pc+4)[31:28]iindex00
000 011	jal	jal *iindex*	R31 = pc + 4 pc = bin_{32}(pc+4)[31:28]iindex00

Table 4.3: *R*-Type Instruction of MIPS-86.

opcode	fun	Mnemonic	Assembler-Syntax	Effect
Shift Operation				
000000	000 000	sll	sll *rd rt sa*	rd = sll(rt,sa)
000000	000 010	srl	srl *rd rt sa*	rd = srl(rt,sa)
000000	000 011	sra	sra *rd rt sa*	rd = sra(rt,sa)
000000	000 100	sllv	sllv *rd rt rs*	rd = sll(rt,rs)
000000	000 110	srlv	srlv *rd rt rs*	rd = srl(rt,rs)
000000	000 111	srav	srav *rd rt rs*	rd = sra(rt,rs)
Arithmetic, Logical Operation				
000000	100 000	add	add *rd rs rt*	rd = rs + rt
000000	100 001	addu	addu *rd rs rt*	rd = rs + rt
000000	100 010	sub	sub *rd rs rt*	rd = rs − rt
000000	100 011	subu	subu *rd rs rt*	rd = rs − rt
000000	100 100	and	and *rd rs rt*	rd = rs $\wedge$ rt
000000	100 101	or	or *rd rs rt*	rd = rs $\vee$ rt
000000	100 110	xor	xor *rd rs rt*	rd = rs $\oplus$ rt
000000	100 111	nor	nor *rd rs rt*	rd = rs $\overline{\vee}$ rt
Test Set Operation				
000000	101 010	slt	slt *rd rs rt*	rd = (rs < rt ? 1 : 0)
000000	101 011	sltu	sltu *rd rs rt*	rd = (rs < rt ? 1 : 0)
Jumps, System Call				
000000	001 000	jr	jr *rs*	pc = rs
000000	001 001	jalr	jalr *rd rs*	rd = pc + 4 pc = rs
000000	001 100	sysc	sysc	System Call
Synchronizing Memory Operations				
000000	111 111	rmw	rmw *rd rs rt*	rd’ = m m’ = (rd = m ? rt : m)
000000	111 110	mfence	mfence	
TLB Instructions				
000000	111 101	flush	flush	flushes TLB
000000	111 100	invlpg	invlpg *rd rs*	flushes TLB translations for addr. *rd* from ASID *rs*

Coprocessor Instructions					
opcode	rs	fun	Mnemonic	Assembler-Syntax	Effect
010000	10000	011 000	eret	eret	Exception Return
010000	00100		movg2s	movg2s *rd rt*	spr[rd] := gpr[rt]
010000	00000		movs2g	movs2g *rd rt*	gpr[rt] := spr[rd]

Table 4.4: MIPS-86 Special Purpose Registers.

i	synonym	
0	sr	status register (contains masks to enable/disable maskable interrupts)
1	esr	exception sr
2	eca	exception cause register
3	epc	exception pc (address to return to after interrupt handling)
4	edata	exception data (contains effective address on pfls)
5	pto	page table origin
6	asid	address space identifier
7	mode	mode register $\in \{0^{31}1, 0^{32}\}$

Table 4.5: MIPS-86 Interrupt Types and Priority.

interrupt level	shorthand	internal/external	type	maskable	
1	I/O	eev	repeat	1	devices
2	ill	iev	abort	0	illegal instruction
3	mal	iev	abort	0	misaligned
4	pff	iev	repeat	0	page fault fetch
5	pfls	iev	repeat	0	page fault load/store
6	sysc	iev	continue	0	system call
7	ovf	iev	continue	1	overflow

Interrupts come in two major flavors: There are *internal interrupts* that are triggered by executing instructions, e.g. an overflow occuring in an arithmetic operation, a system-call instruction being executed (which is supposed to have the semantics of returning control to the operating system in order to perform some task requested by a user processor), or a page fault interrupt due to a missing translation in the page tables. In contrast, there are *external interrupts* which are triggered by an external source, e.g. the reset signal or device interrupts. Interrupts of lesser importance tend to be *maskable*, i.e. there is a control register that allows the programmer to configure that certain kinds of interrupts shall be ignored by the hardware.

The possible interrupt sources and priorities of MIPS-86 are listed in table 4.5. Interrupts are either of type *repeat*, *abort*, or *continue*. Here *repeat* expresses that the interrupted instruction will be repeated after returning from the interrupt handler, *abort* means that the exception is usually so severe that the machine will be by default unable to return from the exception, and *continue* means that even though there is an interrupt, the execution of the interrupted execution will be completed before jumping to the interrupt-service-routine. In case of a *continue*-interrupt execution after exception return will proceed behind the interrupted execution. Note that the APIC mechanism is discussed in more detail in section 4.7.1.

4.2 Overview of the MIPS-86-Model

4.2.1 Configurations

We define the set K of configurations of an abstract simplified multi-core MIPS machine in a top-down way. I.e., we first give a definition of the overall concurrent machine configuration which is composed of several subcomponent configurations whose definitions follow.

Definition 4.1 (Configuration of MIPS-86) A configuration

$$c = (c.p, c.running, c.m, c.d, c.ioapic) \in K$$

of MIPS-86 is composed of the following components:

- a mapping from processor identifier to processor configuration, $c.p : [0 : np - 1] \rightarrow K_{\mathbf{p}}$,

  (np is a parameter that describes the number of processors of the multi-core machine)

- a mapping from processor identifier to a flag that describes whether the processor is running, $c.running : [0 : np - 1] \rightarrow \mathbb{B}$,

  (If $c.running(i) = 0$, this means that the processor is currently waiting for a startup-inter-processor-interrupt (SIPI))

- a shared global memory component $c.m \in K_{\mathbf{m}}$,

- a mapping from device identifiers to device configurations, $c.dev : [0 : nd - 1] \rightarrow K_{\mathbf{dev}}$, and

  (where $K_{\mathbf{dev}} = \bigcup_{i=0}^{nd-1} K_{\mathbf{dev(i)}}$ is the union of individual device configurations, and nd is a parameter that describes the number of devices considered)

- an I/O APIC, $c.ioapic \in K_{\mathbf{ioapic}}$.

Processor

Definition 4.2 (Processor Configuration of MIPS-86)

$$K_{\mathbf{p}} = K_{\mathbf{core}} \times K_{\mathbf{sb}} \times K_{\mathbf{tlb}} \times K_{\mathbf{apic}}$$

A processor $p = (p.core, p.sb, p.tlb, p.apic) \in K_{\mathbf{p}}$ is subdivided into the following components:

- a processor core $p.core \in K_{\mathbf{core}}$,

  The processor core executes instructions according to the Instruction-Set-Architecture (ISA).

- a store buffer $p.sb \in K_{\mathbf{sb}}$,

  A store-buffer buffers write accesses to the memory system local to the processor. If possible, read requests by the core are served by the store-buffer. Writes leave the store-buffer in the order they were placed (first-in-first-out).

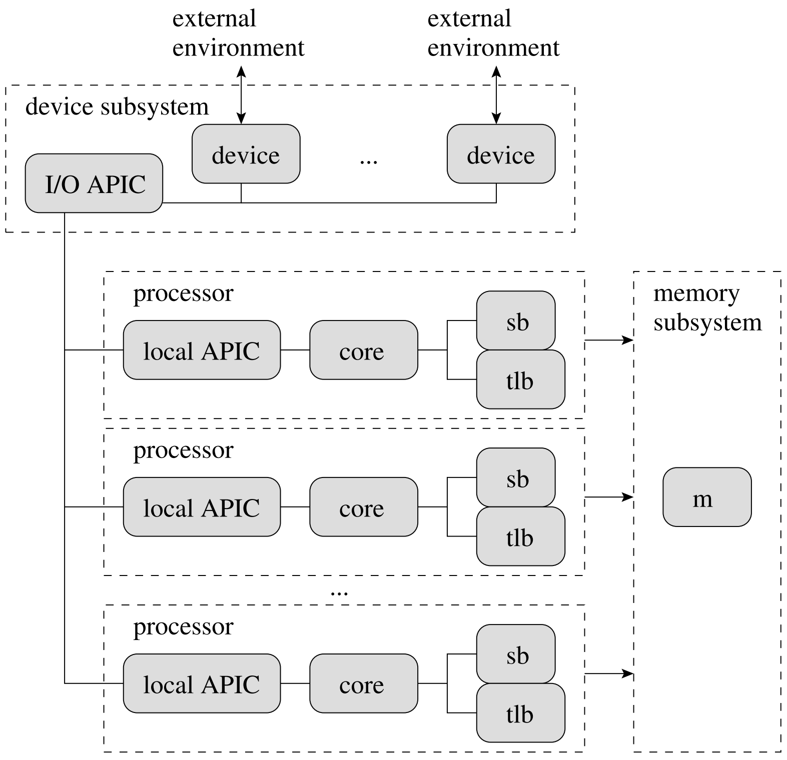


Figure 4.1: Overview of MIPS-86 Model Components.

- a TLB $p.tlb \in K_{\mathbf{tlb}}$

  A translation-lookaside buffer (TLB) performs and caches address translations to be used by the processor in order to establish a virtual memory abstraction.

- a local APIC $p.apic \in K_{\mathbf{apic}}$

  A local APIC receives interrupt signals from the I/O APIC and provides them to the processor. Additionally, it acts as a processor-local device that can send inter-processor-interrupts (IPI) to other processors of the system.

Definitions of $K_{\mathbf{core}}$, $K_{\mathbf{sb}}$, $K_{\mathbf{tlb}}$, and $K_{\mathbf{apic}}$ are each given in the section that defines the corresponding component in detail.

Memory

Definition 4.3 (Memory Configuration of MIPS-86) For this abstract machine, we consider a simple byte-addressable shared global memory component

$$K_{\mathbf{m}} \equiv \mathbb{B}^{32} \to \mathbb{B}^{8}$$

which is sequentially consistent.

Definition 4.4 (Reading Byte-Strings from Byte-Addressable Memory) For a memory $m \in K_{\mathbf{m}}$ and an address $a \in \mathbb{B}^{32}$ and a number $d \in \mathbb{N}$ of Bytes, we define

$$m_d(a) = \begin{cases} m_{d-1}(a +_{32} 1_{32}) \circ m(a) & d > 0 \\ \varepsilon & d = 0 \end{cases}$$

4.2.2 Transitions

We define the semantics of the concurrent MIPS machine MIPS-86 as an automaton with a partial transition function

$$\delta : K \times \Sigma \rightharpoonup K$$

and an output function

$$\lambda : K \times \Sigma \rightharpoonup \Omega$$

where Σ is the set of inputs to the automaton and Ω is the set of outputs of the automaton. In particular, in order to be able to define the semantics of our system as a deterministic automaton, these inputs do include scheduling information, i.e. they determine exactly which subcomponent makes what step. Note that, in the following sections, we will first define the semantics of all individual components before we give the definition of δ and λ in section 4.10.

Scheduling

We provide a model in which the execution order of individual components of the system is not known to the programmer. In order to prove correct execution of code, it is necessary to consider all possible execution orders given by the model. We model this non-deterministic behavior by deterministic automata which take, as part of their input, information about the execution order. This is done in such a way that, in every step, it is specified exactly which subcomponent of the overall system makes which particular step.

There are occasions where several components make a synchronous step. In such cases, our intuition is that one very specific subcomponent *actively* performs a step, while all other components make a *passive* step that merely responds to the active component in some way. The memory, in particular, is such a passive component. Also, devices can react passively to having their registers read by a processor, causing a side-effect on reading.

4.3 Memory

Definition 4.5 (Memory Transition Function) We define the memory transition function

$$\delta_{\mathbf{m}} : K_{\mathbf{m}} \times \Sigma_{\mathbf{m}} \rightharpoonup K_{\mathbf{m}}$$

where

$$\Sigma_{\mathbf{m}} = \mathbb{B}^{32} \times (\mathbb{B}^8)^* \cup \mathbb{B}^{32} \times \mathbb{B}^{32} \times \mathbb{B}^{32}$$

Here,

- $(a, v) \in \mathbb{B}^{32} \times (\mathbb{B}^8)^*$ – describes a write access to address a with value v, and
- $(c, a, v) \in (\mathbb{B}^{32})^3$ – describes a read-modify-write access to address a with compare-value c and value v to be written in case of success.

We have

$$\delta_{\mathbf{m}}(m, in)(x) = \begin{cases} byte(\langle x\rangle - \langle a\rangle, v) & in = (a, v) \wedge 0 \leq \langle x\rangle - \langle a\rangle < \mathbf{len}(v)/8 \\ byte(\langle x\rangle - \langle a\rangle, v) & in = (c, a, v) \wedge m_4(a) = c \wedge 0 \leq \langle x\rangle - \langle a\rangle < 4 \\ m(x) & otherwise \end{cases}$$

Note that, in this memory model, we assume that read accesses cannot change the memory state – thus, any result of a read operation can simply be computed directly from a given memory configuration. In a more concrete model where caches are still visible, we need to consider memory reads as explicit inputs to the cache system.

4.4 TLB

4.4.1 Address Translation

Most processors, including MIPS-86 and x86-64, provide *virtual memory* which is mostly used for implementing process separation in the context of an operating system. By performing *address translation* from virtual memory addresses to physical memory addresses (i.e. regular memory addresses of the machine's memory system), the notion of a virtual memory is established – if this translation is injective, virtual memory has regular memory semantics (i.e. writing to an address affects only this single address and values being written can be read again later). Mostly, this is used to establish several virtual address spaces that are mapped to disjoint address regions in the physical memory of the machine. User processes of the operating system can then each be run by the operating system in their respective address spaces without any risk of user processes affecting each other or the operating system.

Processors tend to provide a mechanism to activate and deactivate address translation – usually by writing some special control register. In the case of MIPS-86, a special-purpose-register *mode* is provided which decides whether the processor is running in *system mode*, i.e. without address translation, or in *user mode*, i.e. with address translation.

The translation from virtual addresses to physical addresses is usually given at the granularity of *memory pages* – in the case of MIPS-86, a *memory page* consists of 2^{12} consecutive bytes. Since MIPS-86 is a 32-bit architecture, a *page address* thus consists of 20 Bits. Defining a particular translation from virtual addresses to phyiscal addresses is done by establishing *page tables* that describe the translation. In the most simple case, a single-level translation can be given by a single page table – which is a sequence of page table entries that each describe how – and if – a given *virtual page address* is translated to a corresponding *physical page address*. The translation tends to be partial, i.e. not every virtual page address has a corresponding physical page address, which is reflected in the page table entry by the *present bit*. Trying to access a virtual address in user mode that does not have a translation set up in the page table according to the present bit then results in a *page-fault* interrupt which returns the processor to system mode – e.g. allowing the operating system to set up a translation for the faulting virtual page address.

MIPS-86, like x86-64, applies a *multi-level page table hierarchy*: Instead of translating using a single page table that describes the virtual page address to physical page address translation, there are several levels of page tables. One advantage of this is that multi-level page tables tend to require less memory space: Instead of providing a page table entry for every virtual page address, the page tables now form a graph in such a way that every level of page tables effectively describes a part of the page address translation by linking to page tables belonging to the next level of translation. Since only a part of the translation is provided, these page tables are much smaller than a single-level translation page table. MIPS-86 provides 2 levels of address translation (in comparison, x86-64 makes use of 4 levels). The first level page table – also called *page directory* – translates the first 10 Bits of a page address by describing where the page tables for translating the remaining 10 Bits can be found, i.e. they contain addresses of second level page tables. These *terminal page tables* then provide the physical page addresses. Note that, in defining a multi-level translation via page tables, page table entries can be marked as not present in early translation levels which essentially means that no further page tables need to be provided for the given virtual page address prefix – which effectively is the reason why multi-level translations tend to require less memory space.

The actual translation is performed in hardware by introducing a circuit called *memory management unit* (MMU) which serves translation requests from the processor by accessing the page tables residing in memory. In a naive hardware implementation of address translation, the processor running in user mode could simply issue translation requests to the MMU in order as needed for instruction execution and wait for the MMU circuit to respond with a translation. Such a synchronous implementation however, would mean that the processor is constantly waiting for the MMU to perform translations, limiting the speed of execution to that of the MMU performing as many memory accesses as needed to compute the translations needed for instruction fetch and execution. Fortunately, however, it can be observed that instruction fetch in user mode to a large degree tends to require translations for virtual addresses that lie in the same page (with an appropriate programming style this is also mostly true for memory accesses performed by instructions), thus, in order not to constantly have the MMU repeat a translation for the same virtual page address, it might be helpful to keep translations available to the processor in a special processor-local cache for translations. This cache is commonly called *translation lookaside buffer* (TLB) and is updated by the MMU whenever necessary in order to serve translation requests by the processor. Note that a hardware TLB may cache partial translations for virtual page address prefixes in order to reuse them later.

Since the operating system may modify the page tables, translations in the TLB may become outdated – removing or changing translations provided by the page tables can make the TLB inconsistent with the page tables. Thus, architectures with TLB tend to provide instructions that allow the processor to control the state of the TLB to some degree. The functionality needed in order to keep the TLB consistent with the page tables is in fact quite simple: In order to ensure that all translations present in the TLB are also given by the page tables, all we need is to be able to remove particular translations (or all translations) from the TLB. Both x86-64 and MIPS-86 provide such instructions – for MIPS-86, *invlpg* invalidates a single virtual page address, while $flush$ removes all translations from the TLB.

4.4.2 TLB Configuration

When the MIPS-86 processor is running in user mode, all memory accesses are subject to address translation according to page tables residing in memory. In order to perform address translation, the MMU operates on the page tables to create, extend, complete, and drop walks. A complete walk provides a translation from a virtual address to a physical address of the machine that can in turn be used by the processor core. Our TLB offers *address space identifier*s – a tag that can be used to associate translations with particular users – which reduces the need for TLB flushes when switching between users.

Definition 4.6 (TLB Configuration of MIPS-86) We define the set of configurations of a TLB as

$$K_{\mathbf{tlb}} = 2^{K_{\mathbf{walk}}}$$

where the set of walks $K_{\mathbf{walk}}$ is given by

$$K_{\mathbf{walk}} = \mathbb{B}^{20} \times \mathbb{B}^{6} \times \{0,1,2\} \times \mathbb{B}^{20} \times \mathbb{B}^{3} \times \mathbb{B}$$

The components of a walk $w = (w.va, w.asid, w.level, w.ba, w.r, w.fault) \in K_{\mathbf{walk}}$ are the following:

- $w.va \in \mathbb{B}^{20}$ – the virtual page address to be translated,
- $w.asid \in \mathbb{B}^{6}$ – the *address space identifier* (ASID) the translation belongs to,
- $w.level \in \{0,1,2\}$ – the current level of the walk, i.e. the number of remaining walk extensions needed to complete the walk,
- $w.ba \in \mathbb{B}^{20}$ – the physical page address of the page table to be accessed next, or, if the walk is complete, the result of the translation,
- $w.r \in \mathbb{B}^{3}$ – the accumulated access rights, and

  Here, $r[0]$ stands for write permission, $r[1]$ for user mode access, and $r[2]$ expresses execute permission.
- $w.fault \in \mathbb{B}$ – a page fault indicator.

4.4.3 TLB Definitions

In the following, we make definitions that describe the structure of page tables and the translation function specified by a given page table origin according to a memory configuration. Addresses are split in two page index components px_2, px_1 and a byte offset px_0 within a page:

$$a = a.px_2 \circ a.px_1 \circ a.px_0$$

Definition 4.7 (Page and Byte Index) Given an address $a \in \mathbb{B}^{32}$, we define

- the second-level page index $a.px_2 = a[31:22]$,

- the first-level page index $a.px_1 = a[21:12]$, and
- the byte offset $a.px_0 = a[11:0]$

of a.

Definition 4.8 (Base Address (Page Address)) The *base address* (also sometimes referred to as *page address*) of an address $a \in \mathbb{B}^{32}$ is then given by

$$a.ba = a.px_2 \circ a.px_1.$$

Definition 4.9 (Page Table Entry) A *page table entry pte* $\in \mathbb{B}^{32}$ consists of

- $pte.ba = pte[31:12]$ – the base address of the next page table or, if the page table is a terminal one, the resulting physical page address for a translation,
- $pte.p = pte[11]$ – the present bit,
- $pte.r = pte[10:8]$ – the access rights for pages accessed via a translation that involves the page table entry,
- $pte.a = pte[7]$ – the accessed flag that denotes whether the MMU has already used the page table entry for a translation, and
- $pte.d = pte[6]$ – the dirty flag that denotes whether the MMU has already used the page table entry for a translation that had write rights. This particular field is only used for terminal page tables.

Definition 4.10 (Page Table Entry Address) For a base address $ba \in \mathbb{B}^{20}$ and an index $i \in \mathbb{B}^{10}$, we define the corresponding *page table entry address* as

$$ptea(ba, i) = ba \circ 0^{12} +_{32} 0^{20} i 00$$

The page table entry address needed to extend a given walk $w \in K_{\mathbf{walk}}$ is then defined as

$$ptea(w) = ptea(w.ba, (w.va \circ 0^{12}).px_{w.level})$$

Definition 4.11 (Page Table Entry for a Walk) Given a memory $m \in K_{\mathbf{mem}}$ and a walk $w \in K_{\mathbf{walk}}$, we define the *page table entry needed to extend a walk* as

$$pte(m,w) = m_4(ptea(w))$$

Definition 4.12 (Walk Creation) We define the function

$$winit : \mathbb{B}^{20} \times \mathbb{B}^{20} \times \mathbb{B}^{6} \rightarrow K_{\mathbf{walk}}$$

which, given a virtual base address $va \in \mathbb{B}^{20}$, the base address $pto \in \mathbb{B}^{20}$ of the page table origin and an address space identifier $asid \in \mathbb{B}^6$, returns the *initial walk* for the translation of va.

$$winit(va,pto,asid) = w$$

is given by

$$w.va = va$$

$$w.asid = asid$$

$$w.level = 2$$

$$w.ba = pto$$

$$w.r = 111$$

$$w.fault = 0$$

Note that in our specification of the MMU, the initial walk always has full rights ($w.r = 111$). However, in every translation step, the rights associated with the walk can be restricted as needed by the translation request made by the processor core.

Definition 4.13 (Sufficient Access Rights) For a pair of access rights $r, r' \in \mathbb{B}^3$, we use

$$r \leq r' \quad \overset{def}{\Leftrightarrow} \quad \forall j \in [0:2] : r[j] \leq r'[j]$$

to describe that the access rights r are weaker than r', i.e. rights r' are sufficient to perform an access with rights r.

Definition 4.14 (Walk Extension) We define the function

$$wext : K_{\mathbf{walk}} \times \mathbb{B}^{32} \times \mathbb{B}^3 \to K_{\mathbf{walk}}$$

which extends a given walk $w \in K_{\mathbf{walk}}$ using a page table entry $pte \in \mathbb{B}^{32}$ and access rights $r \in \mathbb{B}^3$ in such a way that

$$wext(w,pte,r) = w'$$

is given by

$$w'.va = w.va$$

$$w'.asid = w.asid$$

$$w'.level = \begin{cases} w.level - 1 & pte.p \\ w.level & otherwise \end{cases}$$

$$w'.ba = \begin{cases} pte.ba & pte.p \\ w.ba & otherwise \end{cases}$$

$$w'.r = \begin{cases} w.r \wedge r & pte.p \\ w.r & otherwise \end{cases}$$

$$w'.fault = \neg pte.p \vee \neg r \leq pte.r$$

Note that, in addition to restricting the rights according to the rights set in the page table entry used to extend the walk, there is also a parameter r to restrict the rights even further – to account for translations performed for accesses with restricted rights. This is needed in order to allow the MMU to non-deterministically perform translation requests that do not need write rights, and thus, do not require the dirty flag to be set.

Definition 4.15 (Complete Walk) A walk $w \in K_{\textbf{walk}}$ with $w.level = 0$ is called a *complete walk*:

$$complete(w) \equiv w.level = 0$$

Definition 4.16 (Setting Accessed/Dirty Flags of a Page Table Entry) Given a page table entry $pte \in \mathbb{B}^{32}$ and a walk $w \in K_{\textbf{walk}}$, we define the function

$$set\text{-}ad(pte{,}w) = \begin{cases} pte[a := 1, d := 1] & w.r[0] \wedge w.level = 1 \wedge pte.r[0] \\ pte[a := 1] & otherwise \end{cases}$$

which returns an updated page table entry in which the accessed and dirty bits are updated when walk w is extended using *pte*. Extending a walk with write access right using a terminal page table results in the dirty flag being set for the page table entry. Otherwise, only the accessed flag is set.

Definition 4.17 (Translation Request) A translation request

$$trq = (trq.asid, trq.va, trq.r) \in \mathbb{B}^6 \times \mathbb{B}^{32} \times \mathbb{B}^3$$

is a triple of

- address space identifier $trq.asid \in \mathbb{B}^6$,
- virtual address $trq.va \in \mathbb{B}^{20}$, and
- access rights $trq.r \in \mathbb{B}^3$.

Definition 4.18 (TLB Hit) When a walk w matches a translation request trq in terms of virtual address, address space identifier and access rights, we call this a *TLB hit*:

$$hit(trq{,}w) \equiv w.va = trq.va[31 : 12] \wedge w.asid = trq.asid \wedge trq.r \leq w.r$$

Note, that a hit may be to an incomplete walk.

Definition 4.19 (Page-Faulting Walk Extension) A page fault for a given translation request can occur for a given walk when extending that walk would result in a fault: The page table entry needed to extend is not present or the translation would require more access rights than the page table entry provides. To denote this, we define the predicate

$$\begin{aligned} fault(m{,}trq{,}w) \equiv\ & /complete(w) \wedge hit(trq, w) \\ & \wedge wext(w, pte(m, w), trq.r).fault \end{aligned}$$

which, given a memory m, a translation request trq and a walk w, is fulfilled when walk extension for walk w under translation request trq in memory configuration m page-faults.

Note that a page fault may occur at any translation level. However, the TLB will only store non-faulting walks (this is an invariant of the TLB) – page faults are always triggered by considering a faulting extension of a walk in the TLB.

How page faults are triggered is defined in the top-level transition function of MIPS-86 as follows: the processor core always chooses walks from the TLB non-deterministically to either obtain a translation, or, to get a page-fault when the chosen walk has a page faulting walk extension. Note that, when a page-fault for a given pair of virtual address and address space identifier occurs, MIPS-86 flushes all corresponding walks from the TLB. Another side-effect of page-faults in the pipelined hardware implementation is that the pipeline is drained. Since the MIPS-86 model provides a model of sequential instruction execution, draining the pipeline cannot be expressed on this level, however, this behavior is needed in order to be able to prove that the pipelined implementation indeed behaves as specified by MIPS-86.

Definition 4.20 (Transition Function of the TLB) We define the transition function of the TLB that states the passive transitions of the TLB

$$\delta_{\mathbf{tlb}} : K_{\mathbf{tlb}} \times \Sigma_{\mathbf{tlb}} \rightarrow K_{\mathbf{tlb}}$$

where

$$\Sigma_{\mathbf{tlb}} = \{\mathbf{flush}\} \times \mathbb{B}^6 \times \mathbb{B}^{20} \cup \{\mathbf{flush\text{-}incomplete}\} \cup \{\mathbf{add\text{-}walk}\} \times K_{\mathbf{walk}}$$

as a case distinction on the given input:

- flushing a virtual address for a given address space identifier:

$$\delta_{\mathbf{tlb}}(tlb, (\mathbf{flush}, asid, va)) = \{w \in tlb \mid \neg(w.asid = asid \wedge w.va = va)\}$$

- flushing all incomplete walks from the TLB:

$$\delta_{\mathbf{tlb}}(tlb, \mathbf{flush\text{-}incomplete}) = \{w \in tlb \mid complete(w)\}$$

- adding a walk:

$$\delta_{\mathbf{tlb}}(tlb, (\mathbf{add\text{-}walk}, w)) = tlb \cup \{w\}$$

4.5 Processor Core

Definition 4.21 (Processor Core Configuration of *MIPS-86*) A *MIPS-86* processor core configuration $c = (c.pc, c.gpr, c.spr) \in K_{\mathbf{core}}$ consists of

- a program counter: $c.pc \in \mathbb{B}^{32}$,
- a general purpose register file: $c.gpr : \mathbb{B}^5 \rightarrow \mathbb{B}^{32}$, and
- a special purpose register file: $c.spr : \mathbb{B}^5 \rightarrow \mathbb{B}^{32}$.

Definition 4.22 (Processor Core Transition Function of *MIPS-86*) We define the processor core transition function

$$\delta_{\mathbf{core}} : K_{\mathbf{core}} \times \Sigma_{\mathbf{core}} \rightharpoonup K_{\mathbf{core}}$$

which takes a processor core input from

$$\Sigma_{\mathbf{core}} = \Sigma_{\mathbf{instr}} \times \Sigma_{\mathbf{eev}} \times \mathbb{B} \times \mathbb{B}$$

where

$$\Sigma_{\mathbf{instr}} = \mathbb{B}^{32} \times (\mathbb{B}^{8} \cup \mathbb{B}^{16} \cup \mathbb{B}^{32} \cup \{\bot\})$$

is the set of inputs required for instruction execution, i.e. a pair of instruction word $I \in \mathbb{B}^{32}$ and value $R \in \mathbb{B}^{8} \cup \mathbb{B}^{16} \cup \mathbb{B}^{32} \cup \{\bot\}$ read from memory (which is only needed for *read* or *rmw* instructions), and

$$\Sigma_{\mathbf{eev}} = \mathbb{B}^{256}$$

is used to represent a vector $eev \in \mathbb{B}^{256}$ of interrupt signals provided by the local APIC. Also, we explicitly pass the page fault fetch and page fault load/store signals $\mathit{pff}, \mathit{pfls} \in \mathbb{B}$.

We define the processor core transition function

$$\delta_{\mathbf{core}}(c, I, R, eev, \mathit{pff}, \mathit{pfls}) = \begin{cases} \delta_{\mathbf{jisr}}(c, I, R, eev, \mathit{pff}, \mathit{pfls}) & jisr(c, I, eev, \mathit{pff}, \mathit{pfls}) \\ \delta_{\mathbf{instr}}(c, I, R) & \neg jisr(c, I, eev, \mathit{pff}, \mathit{pfls}) \end{cases}$$

as a case distinction on the *jump-interrupt-service-routine*-signal *jisr* (for definition, see 4.5.3) which formalizes whether an interrupt is triggered in the current step of the machine.

In the definition above, we use the auxiliary transition functions

$$\delta_{\mathbf{instr}} : K_{\mathbf{core}} \times \Sigma_{\mathbf{instr}} \rightharpoonup K_{\mathbf{core}}$$

which executes a non-interrupted instruction of the instruction set architecture (for definition, see section 4.5.2), and

$$\delta_{\mathbf{jisr}} : K_{\mathbf{core}} \times \Sigma_{\mathbf{core}} \to K_{\mathbf{core}}$$

which is used to specify the state the core reaches when an interrupt is triggered (for definition, see section 4.5.4).

4.5.1 Auxiliary Definitions for Instruction Execution

In the following, we make auxiliary definitions in order to define the processor core transitions that deal with instruction execution. In order to execute an instruction, the processor core needs to read values from the memory. Of relevance to instruction execution is the instruction word $I \in \mathbb{B}^{32}$ and, if the instruction I is a *read* or *rmw* instruction, we need the value $R \in \mathbb{B}^{8} \cup \mathbb{B}^{16} \cup \mathbb{B}^{32}$ read from memory.

Instruction Decoding

Definition 4.23 (Fields of the Instruction Layout) Formalizing the tables given in subsection 4.1.1, we define the following shorthands for the fields of the MIPS-86 instruction layout:

- *instruction opcode*
$$opc(I) = I[31:26]$$
- *instruction type*
$$rtype(I) \equiv opc(I) = 0^6 \vee opc(I) = 010^4$$
$$jtype(I) \equiv opc(I) = 0^4 10 \vee opc(I) = 0^4 11$$
$$itype(I) \equiv \overline{rtype(I) \vee jtype(I)}$$
- *register addresses*
$$rs(I) = I[25:21]$$
$$rt(I) = I[20:16]$$
$$rd(I) = I[15:11]$$
- *shift amount*
$$sa(I) = I[10:6]$$
- *function code* (used only for *R*-type instructions)
$$fun(I) = I[5:0]$$
- *immediate constants* (for *I*-type and *J*-type instructions, respectively)
$$imm(I) = I[15:0]$$
$$iindex(I) = I[25:0]$$

Definition 4.24 (Instruction-Decode Predicates) For every MIPS-Instruction, we define a predicate on the MIPS-configuration which is true iff the corresponding instruction is to be executed next. The name of such an instruction-decode predicate is always the instruction's mnemonic (see MIPS ISA-tables at the beginning). Formally, the predicates check for the corresponding opcode and function code. E.g.

$$lw(I) \equiv opc(I) = 100011$$

$$\ldots$$

$$add(I) \equiv rtype(I) \wedge fun(I) = 100000$$

The instruction-decode predicates are so trivial to formalize that we do not explicitly list all of them here.

Definition 4.25 (Illegal Opcode) Let

$$ill(I) = \neg(lw(I) \vee \ldots \vee add(I))$$

be the predicate that formalizes that the opcode of instruction I is illegal by negating the disjunction of all instruction-decode predicates.

Note that, encountering an illegal opcode during instruction execution, an illegal instruction interrupt will be triggered.

Arithmetic and Logic Operations

The *arithmetic logic unit* (ALU) of *MIPS-86* behaves according to the following table:

alucon[3:0]	i	alures	ovf
0 000	*	$a +_{32} b$	0
0 001	*	$a +_{32} b$	$[a] + [b] \notin T_{32}$
0 010	*	$a -_{32} b$	0
0 011	*	$a -_{32} b$	$[a] - [b] \notin T_{32}$
0 100	*	$a \wedge_{32} b$	0
0 101	*	$a \vee_{32} b$	0
0 110	*	$a \oplus_{32} b$	0
0 111	0	$\neg_{32}(a \vee_{32} b)$	0
0 111	1	$b[15:0]0^{16}$	0
1 010	*	$0^{31}([a] < [b] ? 1 : 0)$	0
1 011	*	$0^{31}(\langle a \rangle < \langle b \rangle ? 1 : 0)$	0

Based on inputs $a, b \in \mathbb{B}^{32}$, $alucon \in \mathbb{B}^4$ and $i \in \mathbb{B}$, this table defines $alures(a,b,alucon,i) \in \mathbb{B}^{32}$ and $ovf(a,b,alucon,i) \in \mathbb{B}$.

Definition 4.26 (ALU Instruction Predicates) To describe whether a given instruction $I \in \mathbb{B}^{32}$ performs an arithmetic or logic operation, we define the following predicates:

- I-type ALU instruction: $compi(I) \equiv itype(I) \wedge I[31:29] = 001$
- R-type ALU instruction: $compr(I) \equiv rtype(I) \wedge I[5:4] = 10$
- any ALU instruction: $alu(I) \equiv compi(I) \vee compr(I)$

Definition 4.27 (ALU Operands of an Instruction) Following the instruction set architecture tables, we formalize the right and left operand of an ALU instruction $I \in \mathbb{B}^{32}$ based on a given processor core configuration $c \in K_{\textbf{core}}$ as follows:

- left ALU operand: $lop(c,I) = c.gpr(rs(I))$
- right ALU operand: $rop(c,I) = \begin{cases} c.gpr(rt(I)) & rtype(I) \\ sxt_{32}(imm(I)) & /rtype(I) \wedge /I[28] \\ zxt_{32}(imm(I)) & otherwise \end{cases}$

Definition 4.28 (ALU Control Bits of an Instruction) We define the ALU control bits of an instruction $I \in \mathbb{B}^{32}$ as

$$alucon(I)[2:0] = \begin{cases} I[2:0] & rtype(I) \\ I[28:26] & otherwise \end{cases}$$

$$alucon(I)[3] \equiv rtype(I) \wedge I[3] \vee /I[28] \wedge I[27]$$

Definition 4.29 (ALU Compute Result) The ALU result of an instruction I executed in processor core configuration $c \in K_{\mathbf{core}}$ is then given by

$$compres(c,I) = alures(lop(c,I), rop(c,I), alucon(I), itype(I))$$

Jump and Branch Instructions

Jump and branch instructions affect the program counter of the machine. The difference between branch instructions and jump instructions is that branch instructions perform conditional jumps based on some condition expressed over general purpose register values. The following table defines the branch condition result $bcres(a,b,bcon) \in \mathbb{B}$, i.e. whether for the given parameters the branch will be performed or not, based on inputs $a, b \in \mathbb{B}^{32}$ and $bcon \in \mathbb{B}^4$:

bcon[3:0]	bcres(a, b, bcon)
001 0	$[a] < 0$
001 1	$[a] \geq 0$
100 *	$a = b$
101 *	$a \neq b$
110 *	$[a] \leq 0$
111 *	$[a] > 0$

Definition 4.30 (Branch Instruction Predicates) We define the following branch instruction predicates that denote whether a given instruction $I \in \mathbb{B}^{32}$ is a jump or successful branch instruction given configuration $c \in K_{\mathbf{core}}$:

- branch instruction: $b(I) \equiv opc(I)[5:3] = 0^3 \wedge itype(I)$
- jump instruction: $jump(I) \equiv j(I) \vee jal(I) \vee jr(I) \vee jalr(I)$
- jump or branch taken:

$$jbtaken(c,I) \equiv jump(I) \vee b(I) \wedge bcres(c.gpr(rs(I)), c.gpr(rt(I)), opc[2:0]rt(I)[0])$$

Definition 4.31 (Branch Target) We define the target address of a jump or successful branch instruction $I \in \mathbb{B}^{32}$ in a given configuration $c \in K_{\mathbf{core}}$ as

$$btarget(c,I) \equiv \begin{cases} c.pc +_{32} sxt_{30}(imm(I))00 & b(I) \\ c.gpr(rs(I)) & jr(I) \vee jalr(I) \\ (c.pc +_{32} 4_{32})[31:28]iindex(c)00 & j(I) \vee jal(I) \end{cases}$$

Shift Operations

Shift instructions perform shift operations on general purpose registers.

Definition 4.32 (Shift Results) For $a[n-1:0] \in \mathbb{B}^n$ and $i \in \{0,\dots,n-1\}$ we define the following shift results ($\in \mathbb{B}^n$):

- shift left logical: $sll(a,i) = a[n-i-1:0]0^i$
- shift right logical: $srl(a,i) = 0^i a[n-1:i]$
- shift right arithmetic: $sra(a,i) = a_{n-1}^i a[n-1:i]$

Note that, for MIPS-86, we will use the aforementioned definitions only for $n = 32$.

Definition 4.33 (Shift Unit Result) We define the result of a shift operation based on inputs $a \in \mathbb{B}^n, i \in \{0,\dots,n-1\}$, and $sf \in \mathbb{B}^2$ as follows:

$$sures(a,i,sf) = \begin{cases} sll(a,i) & sf = 00 \\ srl(a,i) & sf = 10 \\ sra(a,i) & sf = 11 \end{cases}$$

Definition 4.34 (Shift Instruction Predicate) We define a predicate that, given an instruction $I \in \mathbb{B}^{32}$, expresses whether the instruction is a shift instruction by a simple disjunction of shift instruction predicates:

$$su(I) \equiv sll(I) \vee srl(I) \vee sra(I) \vee sllv(I) \vee srlv(I) \vee srav(I)$$

Definition 4.35 (Shift Operands) Given a shift instruction $I \in \mathbb{B}^{32}$ and a processor core configuration $c \in K_{\mathbf{core}}$, we define the following shift operands:

- shift distance: $sdist(c,I) = \begin{cases} \langle sa(I) \rangle \,\mathbf{mod}\, 32 & fun(I)[3] = 0 \\ \langle c.gpr(rs(I))[4:0] \rangle \,\mathbf{mod}\, 32 & fun(I)[3] = 1 \end{cases}$
- shift left operand: $slop(c,I) = c.gpr(rt(I))$

Definition 4.36 (Shift Function) The shift function of a shift instruction $I \in \mathbb{B}^{32}$ is given by

$$sf(I) = I[1:0]$$

Memory Accesses

We define auxiliary functions that we need in order to define how values are read/written from/to the memory in the overall system's transition function.

Definition 4.37 (Effective Address and Access Width) Given an instruction $I \in \mathbb{B}^{32}$ and a processor core configuration $c \in K_{\mathbf{core}}$, we define the effective address and access width of a memory access:

- effective address: $ea(c,I) = \begin{cases} c.gpr(rs(I)) +_{32} sxt_{32}(imm(I)) & itype(I) \\ c.gpr(rs(I)) & rtype(I) \end{cases}$

- access width: $d(I) = \begin{cases} 1 & lb(I) \vee lbu(I) \vee sb(I) \\ 2 & lh(I) \vee lhu(I) \vee sh(I) \\ 4 & sw(I) \vee lw(I) \vee rmw(I) \end{cases}$

The effective address is the first byte address affected by the memory address and the access width is the number of bytes which are read, or, respectively, written.

Definition 4.38 (Misalignment Predicate) For an instruction $I \in \mathbb{B}^{32}$ and a processor core configuration $c \in K_{\mathbf{core}}$, we define the predicate

$$\begin{aligned} mal(c,I) \quad \equiv \quad & (lw(I) \vee sw(I) \vee rmw(I)) \wedge ea(c,I)[1:0] \neq 00 \\ & \vee (lhu(I) \vee lh(I) \vee sh(I)) \wedge ea(c,I)[0] \neq 0 \end{aligned}$$

that describes whether the memory access is misaligned. To be correctly aligned, the effective address of the memory access must be divisible by the access width.

Note that misaligned memory access triggers the corresponding interrupt.

Definition 4.39 (Load/Store Instruction Predicates) In order to denote whether a given instruction $I \in \mathbb{B}^{32}$ is a load or store instruction, we define the following predicates:

- load instruction: $load(I) \equiv lw(I) \vee lhu(I) \vee lh(I) \vee lbu(I) \vee lb(I)$
- store instruction: $store(I) \equiv sw(I) \vee sh(I) \vee sb(I)$

Definition 4.40 (Load Value) The value read from memory $R \in \mathbb{B}^8 \cup \mathbb{B}^{16} \cup \mathbb{B}^{32}$ is given as an input to the transition function of the processor core. In order to write this value to a general purpose register, depending on the memory instruction used, we either need to sign-extend or zero-extend this value:

$$lv(R) = \begin{cases} zxt_{32}(R) & lbu(I) \vee lhu(I) \\ sxt_{32}(R) & \end{cases}$$

Definition 4.41 (Store Value) Given an instruction $I \in \mathbb{B}^{32}$ and a processor core configuration $c \in K_{\mathbf{core}}$, the store value is given by the last $d(I)$ bytes taken from the general purpose register specified by $rt(I)$:

$$sv(c,I) = c.gpr(rt(I))[8 \cdot d(I) - 1 : 0]$$

General Purpose Register Updates

Definition 4.42 (General Purpose Register Write Predicate) The predicate

$$gprw(I) \equiv alu(I) \vee su(I) \vee lw(I) \vee rmw(I) \vee jal(I) \vee jalr(I) \vee movs2g(I)$$

describes whether a given instruction $I \in \mathbb{B}^{32}$ results in a write to some general purpose register.

Definition 4.43 (General Purpose Register Result Destination) We define the result destination of an ALU/shift/coprocessor/memory instruction $I \in \mathbb{B}^{32}$ as the following general purpose register address:

$$rdes(I) = \begin{cases} rd(I) & rtype(I) \wedge /movs2g(I) \\ rt(I) & otherwise \end{cases}$$

Definition 4.44 (Written General Purpose Register) For an instruction $I \in \mathbb{B}^{32}$, the address of the general purpose register which is actually written to is defined as

$$cad(I) = \begin{cases} 1^5 & jal(I) \vee jalr(I) \\ rdes(I) & alu(I) \vee load(I) \vee rmw(I) \end{cases}$$

Definition 4.45 (General Purpose Register Input) We define the value written to the general purpose register specified above based on the instruction $I \in \mathbb{B}^{32}$ and a given processor core configuration $c \in K_{\textbf{core}}$ as

$$gprdin(c,I,R) = \begin{cases} c.pc +_{32} 4_{32} & jal(I) \vee jalr(I) \\ lv(R) & load(I) \vee rmw(I) \\ c.spr(rd(I)) & movs2g(I) \\ alures(lop(c,I), rop(c,I), alucon(I)) & alu(I) \\ sures(slop(c,I), sdist(c,I), sf(I)) & su(I) \end{cases}$$

4.5.2 Definition of Instruction Execution

Based on the auxiliary functions defined in the last subsection, we give the definition of instruction execution in closed form:

Definition 4.46 (Non-Interrupted Instruction Execution) We define the transition function for non-interrupted instruction execution

$$\delta_{\textbf{instr}} : K_{\textbf{core}} \times \Sigma_{\textbf{instr}} \rightharpoonup K_{\textbf{core}}$$

where

$$\Sigma_{\textbf{instr}} = \mathbb{B}^{32} \times (\mathbb{B}^{8} \cup \mathbb{B}^{16} \cup \mathbb{B}^{32} \cup \{\perp\})$$

as

$$\delta_{\textbf{instr}}(c, I, R) = \begin{cases} undefined & (load(I) \vee rmw(I)) \wedge R \notin \mathbb{B}^{8 \cdot d(I)} \\ c' & otherwise \end{cases}$$

where

- $c'.pc = \begin{cases} btarget(c, I) & jbtaken(c, I) \\ c.spr(epc) & eret(I) \\ c.pc +_{32} 4_{32} & otherwise \end{cases}$

- $c'.gpr(x) = \begin{cases} gprdin(c, I, R) & x = cad(I) \wedge gprw(I) \\ c.gpr(x) & otherwise \end{cases}$

- $c'.spr(x) = \begin{cases} c.gpr(rt(I)) & rd(I) = x \wedge movg2s(I) \\ 0^{31}1 & x = mode \wedge eret(I) \\ c.spr(esr) & x = sr \wedge eret(I) \\ c.spr(x) & otherwise \end{cases}$

4.5.3 Auxiliary Definitions for Triggering of Interrupts

MIPS-86 provides the following interrupt types which are ordered by their priority (interrupt level):

interrupt level	shorthand	internal/external	type	maskable	
1	I/O	eev	repeat	1	devices
2	ill	iev	abort	0	illegal instruction
3	mal	iev	abort	0	misaligned
4	pff	iev	repeat	0	page fault fetch
5	pfls	iev	repeat	0	page fault load/store
6	sysc	iev	continue	0	system call
7	ovf	iev	continue	1	overflow

Note that the all continue interrupts are either triggered by execution of ALU operations with overflow or execution of the sysc-Instruction.

While external event signals are provided by the local APIC as input $eev \in \mathbb{B}^{256}$ to the processor core transition function, the internal event signals $iev(c,I,pff,pfls) \in \mathbb{B}^8$ are defined by the following table that uses the page-fault signals $pff, pfls \in \mathbb{B}$ which are provided by the MMU of the processor to the processor core transition function.

internal event signal	defined by
$iev(c, I, pff, pfls)[2]$	$\equiv ill(I) \vee c.mode[0] = 1 \wedge (movg2s(I) \vee movs2g(I))$
$iev(c, I, pff, pfls)[3]$	$\equiv mal(c, I)$
$iev(c, I, pff, pfls)[4]$	$\equiv pff$
$iev(c, I, pff, pfls)[5]$	$\equiv pfls$
$iev(c, I, pff, pfls)[6]$	$\equiv sysc(I)$
$iev(c, I, pff, pfls)[7]$	$\equiv ovf(lop(c, I), rop(c, I), alucon(I), itype(I))$

Note that even though, from the view of the processor core, the page-fault signals appear just as external as the external event vector provided by the local APIC, the difference is that the external interrupts provided by the local APIC originate from devices while the page-fault signals originate from the MMU belonging to processor itself. This justifies classifying them as internal event signals.

When an interrupt occurs, information about the type of interrupt is stored in a special purpose register to allow the programmer to discover the reason, i.e. the cause, for the interrupt.

Definition 4.47 (Cause and Masked Cause of an Interrupt) We define the cause $ca \in \mathbb{B}^8$ of an interrupt and masked cause $mca \in \mathbb{B}^8$ of an interrupt based on the current processor core configuration $c \in K_{\mathbf{core}}$, the instruction $I \in \mathbb{B}^{32}$ to be executed, the external event vector $eev \in \mathbb{B}^{256}$ and the page-fault signals $pff, pfls \in \mathbb{B}$ as follows:

- cause of interrupt:

$$ca(c,I,eev,pff,pfls)[j] = \begin{cases} iev(c,I,pff,pfls)[j] & j \in [2:7] \\ \bigvee_{i=0}^{255} eev[i] & j = 1 \\ 0 & otherwise \end{cases}$$

- masked cause:

$$mca(c,I,eev,pff,pfls)[j] = \begin{cases} ca(c,I,eev,pff,pfls)[j] & j \notin \{1,7\} \\ ca(c,I,eev,pff,pfls)[j] \wedge c.spr(sr)[j] & j \in \{1,7\} \end{cases}$$

Only interrupt levels 1 and 7 are maskable; the corresponding mask can be found in special purpose register *sr* (status register) and is applied to the cause of interrupt to obtain the masked cause.

Definition 4.48 (Jump-to-Interrupt-Service-Routine Predicate) To denote that in a given configuration $c \in K_{\mathbf{core}}$ for a given instruction $I \in \mathbb{B}^{32}$, external event signals $eev \in \mathbb{B}^{256}$, and page-fault signals $pff, pfls \in \mathbb{B}$ an interrupt is triggered, we define the predicate

$$jisr(c,I,eev,pff,pfls) \equiv \bigvee_j mca(c,I,eev,pff,pfls)[j]$$

Definition 4.49 (Interrupt Level of the Triggered Interrupt) To determine the interrupt level of the triggered interrupt, we define the function

$$il(c,I,eev,pff,pfls) = min\{j \mid mca(c,I,eev,pff,pfls)[j] = 1\}$$

Definition 4.50 (Continue-Type Interrupt Predicate) The predicate

$$continue(c,I,eev,pff,pfls) \equiv il(c,I,R,eev) \in \{6,7\}$$

denotes whether the triggered interrupt is of continue type.

4.5.4 Definition of Interrupt Execution

Definition 4.51 (Interrupt Execution Transition Function) We define $\delta_{\mathbf{jisr}}(c,I,R,eev,pff,pfls) = c'$ where $I \in \mathbb{B}^{32}$ is the instruction to be executed and $eev \in \mathbb{B}^{256}$ are the event signals received from the local APIC and $pff, pfls \in \mathbb{B}$ are the page-fault signals provided by the processor's MMU.

Let $k = min\{j \mid eev[j] = 1\}$.

- $c'.pc = 0^{32}$

- $$c'.spr(x) = \begin{cases} 0^{32} & x = sr \\ 0^{32} & x = mode \\ c.sr & x = esr \\ zxt_{32}(mca(c, I, eev, pff, pfls)) & x = eca \\ c.pc & x = epc \wedge /continue(c, I, eev, pff, pfls) \\ \delta_{\mathbf{instr}}(c, I, R).pc & x = epc \wedge continue(c, I, eev, pff, pfls) \\ ea(c, I) & x = edata \wedge il(c, I, eev, pff, pfls) = 5 \\ bin_{32}(k) & x = edata \wedge il(c, I, eev, pff, pfls) = 1 \\ c.spr(x) & otherwise \end{cases}$$

- $$c'.gpr = \begin{cases} c.gpr & /continue(c, I, eev, pff, pfls) \\ \delta_{\mathbf{instr}}(c, I, R).gpr & otherwise \end{cases}$$

4.6 Store Buffer

Store buffers are, in their simplest form, first-in-first-out queues for write accesses that reside between processor core and memory. In a processor model with store-buffer, servicing memory reads is done by finding the newest store-buffer entry for the given address if one is available – otherwise the read is serviced by the memory subsystem. Essentially, this means that read accesses that rely on values from preceeding write accesses can be serviced even before they reach the caches. The benefit of store-buffers implemented in hardware is that instruction execution can proceed while the memory is still busy servicing previous write accesses.

In order to allow the programmer to obtain a sequentially consistent view of memory in the presence of store-buffers, architectures whose abstract model contains store-buffers tend to provide instructions that have an explicit effect on the store-buffer, e.g. by draining the pipeline. *MIPS-86* offers a memory fence instruction $fence$ that simply drains the store buffer and a read-modify-write operation *rmw* that performs an atomic conditional memory update with the side-effect of draining the store-buffer.

Note that, even in a machine that has no store-buffer in hardware, pipelining of instruction execution may introduce a store-buffer to the abstract machine model. We discuss this in the next subsection before we give a definition of the store-buffer of MIPS-86.

4.6.1 Instruction Pipelining May Introduce a Store-Buffer

The term *pipelining* used in the context of gate-level circuit design can be used to describe splitting up a hardware construction (e.g. of a processor) that computes some result in a single hardware cycle (e.g. executes an instruction) into n smaller components which are called *pipeline stages* whose outputs are always inputs of the next one and which each computes a part of the final result in its own registers – in such a way that, initially, after n cycles, the first result is provided by the nth component and then, subsequently, every following cycle a computation finishes. The reason why this is efficient lies in the fact that, in terms of electrophysics, smaller circuits require less time for input signals to propagate and for output signals to stabilize, thus, smaller circuits can be clocked faster than larger ones. Note that the increase in delay for inserting additional registers in the subcomponents tends to be less than the delay saved by splitting the construction into pipeline stages, resulting in an overall faster computation due to the achieved parallelization.

A common feature to be found in processors is *instruction pipelining*. For a basic RISC machine (like *MIPS-86*), the common five-stage pipeline is given by the following five stages: IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, and WB = Register write back. Note that a naive hardware implementation where all that is changed from the one-cycle hardware construction is that additional registers are inserted will, in general, behave differently than the original construction: Execution of the next instruction may depend on results from the execution of previous instructions which are still in the instruction pipeline. The occurrence of such a dependency where the result of a computation in the naively pipelined machine does not match the result of the sequential machine is referred to as a *hazard*. One way to circumvent this is by software: If the programmer/compiler ensures that the machine code executed by the pipelined machine does not cause any hazard (e.g. by inserting NOOPs (no operations, i.e. instructions that do not have any effect other than incrementing the program counter) or by reordering instructions). This, however, by requiring a much more conservative style of programming, reduces the speedup gained by introducing pipelining in the first place.

In fact, instead of leaving hazard detection and handling exclusively to the programmer of the machine, modern architectures implement proper handling of most hazards in hardware. When a *data hazard* is detected (i.e. an instruction depends on some value computed by an earlier instruction that is still in the pipeline), the hardware *stalls* execution of the pipeline on its own until the required result has been computed. Additional hardware then *forwards* the result from the later pipeline stage directly to the waiting instruction that is stalled in an earlier pipeline stage. Note that even though many hazards can be detected and resolved efficiently in hardware, it is not necessarily the best thing to prevent all hazards in hardware – overall performance of a system may be better when minor parts of hazard handling are left to the programmer/compiler. In fact, for modern pipelined architectures, it is common practice to allow slight changes to the abstract hardware model at ISA level which allow for a less strict but more performant treatment of hazards.

When a memory write is forwarded to a subsequent memory read instruction to the same address (or to the instruction fetch stage, possibly), this can be modeled by introducing a *store-buffer* between processor and memory system – even when there is no physical store-buffer present in the hardware implementation [Hot12]. For a single-core architecture, it is not overly hard to prove that a processor model with store-buffer is actually equivalent to the processor model without store-buffer. For a multi-core architecture however, it is more involved to prove that store-buffers become invisible on higher layers of abstraction: Since every processor has its own store-buffer and the values from store-buffers are not forwarded to other processors, any two processors may have different values for the same address present in their store-buffers. For an in-detail treatment of hardware construction and correctness for a pipelined simple MIPS machine, see [Pau12].

4.6.2 Configuration

Definition 4.52 (Store Buffer Configuration) The set of store buffer entries is given by

$$K_{\mathbf{sbe}} \equiv \{(a, v) \mid a \in \mathbb{B}^{32} \wedge v \in \mathbb{B}^{8} \cup \mathbb{B}^{16} \cup \mathbb{B}^{32}\}$$

while the set of store buffer configurations is defined as follows:

$$K_{\mathbf{sb}} \equiv K^{*}_{\mathbf{sbe}}$$

We consider a store buffer modeled by a finite sequence of store buffer write accesses (a, v) where $a \in \mathbb{B}^{32}$ is an address and $v \in \mathbb{B}^{8} \cup \mathbb{B}^{16} \cup \mathbb{B}^{32}$ is the value to be written starting at memory address a.

4.6.3 Transitions

A step of the store buffer produces the memory write specified by the oldest store-buffer entry – in order to be sent to the memory subsystem. When the store buffer is empty ($c.sb = \varepsilon$), it cannot make a step. We always append at the front and remove at the end of the list. Transitions of the store buffer are formalized in the overall transition relation – we do not provide an individual transition relation for the store buffer.

4.6.4 Auxiliary Definitions

We define some auxiliary functions for use in the definition of the system's transition function.

Definition 4.53 (Store Buffer Entry Hit Predicate) Given a store buffer entry $(a, w) \in K_{\mathbf{sbe}}$ and a byte address $x \in \mathbb{B}^{32}$, we define the predicate

$$sbehit((a,w),x) \equiv \langle x -_{32} a \rangle < |w|/8$$

which denotes that there is a store buffer hit for the given entry and address, i.e. the address is written by the write access represented by the store buffer entry.

Definition 4.54 (Newest Store Buffer Hit Index) Given a store buffer configuration $sb \in K_{\mathbf{sb}}$ and a byte address $x \in \mathbb{B}^{32}$, we define the function

$$maxsbhit(sb,x) \equiv \max\{j \mid sbehit(sb[j], x)\}$$

which computes the index of the newest entry of the store buffer for which there is a hit.

Definition 4.55 (Store Buffer Hit Predicate) Given a store buffer configuration $sb \in K_{\mathbf{sb}}$ and a byte address $x \in \mathbb{B}^{32}$, the predicate

$$sbhit(sb,x) \equiv \exists j : sbehit(sb[j], x)$$

denotes whether there is a store buffer hit for the given address in the given store buffer.

Definition 4.56 (Store Buffer Value) We define the function

$$sbv : K_{\mathbf{sb}} \times \mathbb{B}^{32} \rightharpoonup \mathbb{B}^8$$

which, given a store buffer configuration *sb* and a byte-address *x* computes the value forwarded from the store buffer for the given address, if defined:

$$sbv(sb,x) = \begin{cases} byte(\langle x \rangle - \langle a \rangle, v) & sb[maxsbhit(sb,x)] = (a,v) \\ undefined & otherwise \end{cases}$$

4.7 Devices

4.7.1 Introduction to Devices, Interrupts and the APIC Mechanism

Interesting hardware tends to include devices of some kind, e.g. screens, input devices, timers, network adapters, storage devices, or devices that control factory robots, conveyor belts, nuclear power plants, radiation therapy units, etc. In order to trigger interrupts in a processor core, devices tend to provide *interrupt request signals*. Commonly, device interrupt request signals are distinguished in *edge-triggered* signals, i.e. the device signals an interrupt by switching the state of its interrupt request signals, and *level-triggered* signals, i.e. the interrupt request signal is raised when an interrupt is currently pending (i.e. the interrupt request signal has the digital value 1). In a simple hardware implementation of a single-core architecture, level-triggered interrupt request signals originating from devices basically just need to be connected to the processor as external interrupt signals while edge-triggered signals need to be sampled properly and then provided to the processor until it can accept them. Note that in MIPS-86, we restrict interrupt request signals to level-triggered signals.

The processor tends to communicate with devices either by means of *memory mapped input/output* (memory mapped I/O), i.e. device *ports* (which can essentially be considered user-visible registers of a device) are mapped into the memory space of the processor and accessed with regular memory instructions, or by using special I/O instructions that operate on a seperate port address space where device ports reside. Reading or writing device ports tends to have side-effects, e.g. such as disabling the interrupt request signal raised by the device, allowing the processor to acknowledge that it has received the interrupt, or causing the device to initiate some interaction with the external world.

For a multi-core architecture, device interrupts become more interesting: Is it desirable to interrupt all processors when a device raises an interrupt signal? In many cases, the answer is no: It is fully sufficient to let one processor handle the interrupt. The main question is mostly which device interrupt is supposed to go to which processor. In x86 architectures this is resolved in hardware by providing each processor with a *local advanced programmable interrupt controller* (local APIC) that receives *interrupt messages* from a global *input/output advanced programmable interrupt controller* (I/O APIC) which collects and distributes the interrupt request signals of devices. In order not to transmit a single interrupt to a processor more often than necessary, there is a protocol between I/O APIC and the processor (via the local APIC) in which the processor has to acknowledge the handling of the interrupt by sending an *end of* *interrupt* (EOI) message via the local APIC. Only after such an EOI message has been received will the I/O APIC sample the corresponding interrupt vector again. Essentially, in the abstract hardware model, both local APIC and I/O APIC can be seen as a special kind of device that does not raise interrupts on its own but can be accessed by the processor by means of memory mapped I/O just like a regular device. Since MIPS-86 implements a greatly simplified version of the x86 APIC mechanism, we will not discuss the detailed x86 APIC mechanism in the following and focus on MIPS-86 in the following.

How exactly the I/O APIC distributes device interrupt signals to the individual processor cores is specified by the *redirect table* – which can be accessed through the I/O APIC ports. This redirect table – which must be set up by the programmer of the machine – specifies the following for each device interrupt request signal: The destination processor, whether the interrupt signal is masked already at the I/O APIC, and the *interrupt vector* to be triggered. The interrupt vector of an interrupt is used to provide information about the cause of the interrupt to the processor. Device interrupt signals are sampled at the I/O APIC and subsequently sent to the destination processor's local APIC over a common bus that connects all local APICs and the I/O APIC. The local APIC associated with a processor core receives interrupt messages from the I/O APIC, collecting interrupt vectors which are then passed to the processor core by raising external interrupt signals at the processor core.

In addition to providing a means of distributing device interrupts, the APIC mechanism offers processor cores of the multi-core system the opportunity to send *inter-processor interrupts* (IPIs). This can, for example, be useful to implement communication between different processors in the context of an operating system. Sending of an inter-processor interrupt is triggered by writing a particular control register belonging to the ports of the local APIC of the processor. The content of this control register describes the destination processors, delivery status, delivery mode and interrupt vector of the inter-processor interrupt. The IPI mechanism is particularly important for booting the machine: Initially, after power on, only the *bootstrap processor* (BSP) is running while all other processors are in a halted state with their local APICs waiting for an initialization inter-processor interrupt (INIT-IPI) and a subsequent startup inter-processor interrupt (SIPI). Effectively, booting the multi-core system can be done by the bootstrap processor in a sequential fashion until it initializes and starts the other processor cores of the system via the IPI mechanism.

4.7.2 Configuration

Definition 4.57 (Device Port Address Length) We assume a function

$$dev_{\mathbf{palen}} : [0 : nd - 1] \to \mathbb{N}$$

to be given that specifies the address length of port addresses of all $nd \in \mathbb{N}$ devices of the system in bits.

Definition 4.58 (Device Configuration) The configuration $d \in K_{\mathbf{dev(i)}}$ of device i is given by

- I/O ports $d.ports : \mathbb{B}^{dev_{\mathbf{palen}}(i)} \to \mathbb{B}^8$,
- an interrupt request signal $d.irq \in \mathbb{B}$, and

- internal state $d.internal \in \mathcal{D}_i$

Note that the $\mathcal{D}_i$ have to be defined by users of our model to describe the devices they want to argue about.

4.7.3 Transitions

Devices react to external inputs provided to them and they have side-effects that occur when their ports are read, or, respectively, written. Note that we currently do not model read-modify-write accesses to devices and we only consider word-accesses on device ports.

Definition 4.59 (Device Transition Function) For every device, we assume a transition function to be given of the form

$$\delta_{\mathbf{dev(i)}} : K_{\mathbf{dev(i)}} \times \Sigma_{\mathbf{dev(i)}} \rightharpoonup K_{\mathbf{dev(i)}}$$

with

$$\Sigma_{\mathbf{dev(i)}} = \Sigma_{\mathbf{ext(i)}} \cup \mathbb{B}^{dev_{\mathbf{palen}}} \cup \mathbb{B}^{dev_{\mathbf{palen}}} \times (\mathbb{B}^8)^* \cup \mathbb{B}^{32} \times \mathbb{B}^{dev_{\mathbf{palen}}} \times \mathbb{B}^{32}$$

where the input to the transition function $in \in \Sigma_{\mathbf{dev(i)}}$ is either

- an external input for device i: $in = ext \in \Sigma_{\mathbf{ext(i)}}$,
- an external input for device i: $in = a \in \mathbb{B}^{dev_{\mathbf{palen}}(i)}$, or
- a word write-access to port address a with value v: $in = (a, v) \in \mathbb{B}^{dev_{\mathbf{palen}}(i)} \times \mathbb{B}^{32}$.

Note that all active steps of a device are modeled via external inputs, i.e. every active step of the device should be modeled by an input from $\Sigma_{\mathbf{ext(i)}}$ that triggers the corresponding step. Further, $\Sigma_{\mathbf{ext(i)}}$ can be used to model how the device reacts to the external world.

Depending on the device in question, reading or writing port addresses may have side-effects – for example, deactivating the interrupt request when a specific port is read. This needs to be specified individually for the given device in its transition function. One restriction we make in this model is that even though reading ports may have side-effects, the value being read is always the one that is given in the I/O ports component of the device. This is reflected in the next section when an overall view of memory with device I/O ports mapped into the physical address space of the machine is defined.

4.7.4 Device Outputs

We allow devices to provide an output function

$$\lambda_{\mathbf{dev(i)}} : K_{\mathbf{ext(i)}} \times \Sigma_{\mathbf{dev(i)}} \rightharpoonup \Omega_{\mathbf{dev(i)}}$$

in order to allow interaction with some external world. This is a partial function, since a device does not need to produce an output for every given external input in a given configuration.

4.7.5 Device Initial State

To define a set of acceptable initial states of a device after reset, the predicate

$$initial_{\mathbf{state(i)}} : K_{\mathbf{dev(i)}} \to \mathbb{B}$$

shall be defined.

4.7.6 Specifying a Device

To specify a particular device of interest, we always need to define the following:

- $\mathcal{D}_i$ – the internal state of the device,
- $\Sigma_{\mathbf{ext(i)}}$ – the external inputs the device reacts to,
- $\Omega_{\mathbf{dev(i)}}$ – the possible outputs provided by the device,
- $dev_{\mathbf{palen}}(i)$ – the length of port addresses of the device,
- $\delta_{\mathbf{dev(i)}}$ – the transition function of the device,
- $\lambda_{\mathbf{dev(i)}}$ – the output function of the device, and
- $initial_{\mathbf{state(i)}}$ – the set of acceptable initial states of the device.

4.8 Local APIC

The local APIC receives device and inter-processor interrupts sent over the interrupt bus of the system and provides these interrupts to the processor core it is associated with. While the local APIC shares some behavior with devices (i.e. it is accessed by means of memory-mapped I/O) some of its behavior differs significantly from that of devices (i.e. communicating over the interrupt bus instead of raising an interrupt request signal, providing interrupt signals directly to the processor core).

The x86-64 model of [Deg11] provides a local APIC model that describes sending of inter-processor interrupts but ignores devices. While already simplified somewhat compared to the informal architecture definitions, this model is still quite complex. Thus, Hristo Pentchev provides a simplified version of the x86-64 local APIC model in his upcoming dissertation in order to prove formal correctness of an inter-processor interrupt protocol implementation [ACHP10]. On the one hand, the local APIC model we present in the following is even further simplified – mostly by expressing APIC system transitions atomically instead of in terms of many intermediate steps and by reducing the possible interrupt types and target modes. On the other hand, the model provided here is more powerful in the sense that device interrupts and I/O APIC are modeled.

We have the following simplifications over x86-64:

- We only consider level-triggered interrupts.

- We reduce IPI-delivery to Fixed, INIT and Startup interrupts. The I/O APIC only delivers Fixed interrupts.
- We only model physical destination mode where IPIs are addressed to a local APIC ID (or to a shorthand). We don't consider logical destination mode.
- We do not consider the error-status-register which keeps track of errors encountered when trying to deliver interrupts.

4.8.1 Configuration

Definition 4.60 (Local APIC Configuration) The configuration of a local APIC

$$apic = (apic.ports, apic.initrr, apic.sipirr, apic.sipivect, apic.eoipending) \in K_{\mathbf{apic}}$$

consists of

- I/O-ports $apic.ports : \mathbb{B}^7 \to \mathbb{B}^8$,
- INIT-request register $apic.initrr \in \mathbb{B}$,

  (a flag that denotes whether an INIT-request is pending to be delivered to the processor)
- SIPI-request register $apic.sipirr \in \mathbb{B}$,

  (a flag that denotes whether a SIPI-request is pending to be delivered to the processor)
- SIPI-vector register $apic.sipivect \in \mathbb{B}^8$,

  (the start address for the processor to execute code after receiving SIPI)
- EOI-pending register $apic.eoipending \in \mathbb{B}^{256}$

  (a register that keeps track of all interrupt vectors for which an EOI message is to be sent to the I/O APIC)

The I/O ports of the local apic can be accessed by the processor by means of memory mapped I/O. All other local APIC components cannot be accessed by other components. This is reflected in the overall transition relation of the system.

Local APIC ports

Let us define a few shorthands for specific regions in the local APIC ports:

- **APIC ID Register**

$$apic.\mathbf{APIC\_ID} = apic.ports_4(0_7)$$

Bits	description
31-28	reserved
27-24	local APIC ID
23-0	reserved

This register contains the local APIC ID of the local APIC. This ID is used when addressing inter-processor-interrupts to a specific local APIC.

- **Interrupt Command Register (ICR)**

$$apic.\mathbf{ICR} = apic.ports_8(4_7) \in \mathbb{B}^{64}$$

Bits	abbreviation	description
63-56	*dest*	destination field
55-20		reserved
19-18	*dsh*	destination shorthand
		00b = no shorthand, 01b = self 10b = all including self, 11b = all excluding self
17-13		reserved
12	*ds*	delivery status 0b = idle, 1b = send pending
11	*destmode*	destination mode 0b = physical
10-8	*dm*	delivery mode 000b = Fixed, 101b = INIT, 110b = Startup
7-0	*vect*	vector

This register is used to issue a request for sending an inter-processor interrupt to the local APIC.

- **End-Of-Interrupt Register**

$$apic.\mathbf{EOI} = apic.ports_4(12_7) \in \mathbb{B}^{32}$$

Writing to this register is used to signal to the local APIC that the interrupt-service-routine has finished. This has the effect that the local APIC will eventually send an end-of-interrupt acknowledgement to the I/O-APIC.

- **In-Service Register**

$$apic.\mathbf{ISR} = apic.ports_{32}(16_7) \in \mathbb{B}^{256}$$

This register keeps track of which interrupt vectors are currently being serviced by an interrupt-service-routine. For our simple processor, maskable interrupts (to which device interrupts belong) are by default masked in the processor core when an interrupt is triggered. However, when the programmer explicitly unmasks device interrupts during the interrupt handler run, it can happen that a higher-priority interrupt provided by the local APIC may trigger another interrupt, resulting in several interrupt vectors being in service at the same time.

- **Interrupt Request Register**

$$apic.\mathbf{IRR} = apic.ports_{32}(48_7) \in \mathbb{B}^{256}$$

This register keeps track for which interrupt vectors there is currently a request pending. These requests are provided to the processor as external event signals. In this process, all interrupt requests of lower priority than the ones currently in service are masked by the local APIC.

Definition 4.61 (Processor Core External Event Signals) We define the external event vector $eev \in \mathbb{B}^{256}$ provided by the local APIC $apic \in K_{\textbf{apic}}$ to the processor core as

$$eev(apic)[j] = \begin{cases} 0 & \exists k \leq j : apic.\textbf{ISR}[k] = 1 \\ apic.\textbf{IRR}[j] & otherwise \end{cases}$$

4.8.2 Transitions

We simplify device accesses in such a way that we expect only aligned word-accesses to occur on device ports, i.e. halfword and byte accesses on devices are not modeled.

For all passive steps of the local APIC, we define a transition function

$$\delta_{\textbf{apic}} : K_{\textbf{apic}} \times \Sigma_{\textbf{apic}} \to K_{\textbf{apic}}$$

where

$$\Sigma_{\textbf{apic}} \equiv \mathbb{B}^7 \times \mathbb{B}^{32} \cup \{\textbf{Fixed}, \textbf{INIT}, \textbf{SIPI}\} \times \mathbb{B}^8$$

A passive step of a local APIC is a write access to its ports or a receive-interrupt step. We define

$$\delta_{\textbf{apic}}(apic, in) = apic'$$

by a case-distinction:

- write without side-effects:

$$in = (a, v) \wedge a \neq 12_7$$

$$apic'.ports(x) = \begin{cases} byte(\langle x \rangle - \langle a \rangle, v) & in = (a, v) \wedge 0 \leq \langle x \rangle - \langle a \rangle < 4 \\ apic.ports(x) & otherwise \end{cases}$$

- write with side effects (to EOI-register):

$$in = (a, v) \wedge a = 12_7$$

$$apic'.\textbf{EOI} = v$$

$$apic'.\textbf{ISR}[j] = \begin{cases} 0 & j = min\{k \mid apic.\textbf{ISR}[k] = 1\} \\ apic.\textbf{ISR}[j] & otherwise \end{cases}$$

All other local APIC ports stay unchanged.

$$apic'.eoipending[j] = \begin{cases} 1 & j = min\{k \mid apic.\textbf{ISR}[k] = 1\} \\ apic.eoipending[j] & otherwise \end{cases}$$

Writing to the EOI-Register puts the local APIC in a state where it will send an EOI-message over the interrupt bus in order to acknowledge handling of the highest-priority interrupt to the I/O APIC.

- receiving an interrupt:
  - $in = (\textbf{Fixed}, vect)$

    $$apic'.\textbf{IRR}[j] = \begin{cases} 1 & j = \langle vect \rangle \\ apic.\textbf{IRR}[j] & otherwise \end{cases}$$

    All other ports unchanged.
  - $in = (\textbf{SIPI}, vect)$

    Receiving a startup-interrupt is successful when there is currently no startup-interrupt pending: If $apic.sipirr \neq 0$, $apic' = apic$ (the local APIC will discard the interrupt), otherwise

    $apic'.ports = apic.ports$

    $apic'.sipirr = 1$

    $apic'.sipivect = vect$

    This records a SIPI which can in turn be used by the local APIC to set the *running* flag of the corresponding processor, effectively starting it.
  - $in = (\textbf{INIT}, vect)$

    Receiving an INIT-interrupt is successful when there is currently no INIT-interrupt pending: If $apic.initrr \neq 0$, $apic' = apic$, otherwise

    $apic'.ports = apic.ports$

    $apic'.initrr = 1$

    When the local APIC received an INIT-IPI, it will force a reset on the corresponding processor.

All components not explicitly mentioned stay unchanged between *apic* and *apic'*.

The active steps of the local APIC (i.e. sending IPIs, sending EOI messages, applying SIPI and INIT-IPI to the processor) are treated in the overall transition relation of the system.

4.9 I/O APIC

The I/O APIC samples the interrupt request signals of devices and distributes the interrupts to the local APICs according to its redirect table by sending interrupt messages over the interrupt bus. Device interrupts can be masked directly at the I/O APIC.

4.9.1 Configuration

Definition 4.62 (I/O APIC Configuration) The configuration of an I/O APIC

$$ioapic = (ioapic.ports, ioapic.redirec) \in K_{\textbf{ioapic}}$$

is given by

- I/O-ports $ioapic.ports : \mathbb{B}^3 \to \mathbb{B}^8$, and
- a redirect table $ioapic.redirect : [0 : 23] \to \mathbb{B}^{32}$

I/O APIC Ports

Shorthands to the ports of the I/O APIC are

- select register $ioapic.\mathbf{IOREGSEL} = ioapic.ports_4(0_3)$
- data register $ioapic.\mathbf{IOWIN} = ports_4(4_3)$

Note a pecularity about the I/O APIC: instead of mapping the redirect table into the processor's memory, only a select and a data register are provided. Writing the select register has the side-effect of fetching the redirect table entry specified to the data register. Writing the data register has the side effect of also writing the redirect table entry specified by the select register. Reading from the I/O APIC ports does not have any side-effects.

Format of the Redirect Table

A redirect table entry $e \in \mathbb{B}^{32}$ has the following fields:

Bits	Short	Name	Description
24-31	*dest*	Destination	Local APIC ID of destination local APIC
17-24		reserved	
16	*mask*	Interrupt Mask	masked if set to 1
15		reserved	
14	*rirr*	Remote IRR	0b = EOI received 1b = interrupt was received by local APIC
13		reserved	
12	*ds*	Delivery Status	1b = Interrupt needs to be delivered to local APIC
11		reserved	
8-10	*dm*	Delivery Mode	000b = Fixed
0-7	*vect*	Interrupt Vector	

4.9.2 Transitions

We define a transition function for the passive steps of the I/O APIC

$$\delta_{\mathbf{ioapic}} : K_{\mathbf{ioapic}} \times \Sigma_{\mathbf{ioapic}} \rightharpoonup K_{\mathbf{ioapic}}$$

with

$$\Sigma_{\mathbf{ioapic}} = \mathbb{B}^3 \times \mathbb{B}^{32} \cup \mathbb{B}^8$$

where $in \in \Sigma$ is either

- $in = (a, v) \in \mathbb{B}^3 \times \mathbb{B}^{32}$ – a write access to port address a with value v,
- $in = vect \in \mathbb{B}^8$ – receiving an EOI message for interrupt vector $vect$

We define $\delta_{\mathbf{ioapic}}(ioapic, in) = ioapic'$ by case distinction on in:

- $in = (a, v) \in \mathbb{B}^3 \times \mathbb{B}^{32}$ – write access to the I/O APIC ports

  $\delta_{\mathbf{ioapic}}(ioapic, in)$ is undefined iff $a \notin \{0_3, 4_3\}$. Otherwise

  - **Case** $a = 0_3$:

    $ioapic'.\mathbf{IOREGSEL} = v$

    $ioapic.\mathbf{IOWIN} = ioapic.redirect(\langle v \rangle)$

  - **Case** $a = 4_3$:

    $ioapic'.\mathbf{IOWIN} = v$

    $$ioapic'.redirect(i) = \begin{cases} v & i = \langle ioapic.\mathbf{IOREGSEL} \rangle \\ ioapic.redirect(i) & otherwise \end{cases}$$

- $in = vect \in \mathbb{B}^8$ – receiving an EOI message for interrupt vector *vect*

  $$ioapic'.redirect(i).rirr = \begin{cases} 0 & ioapic.redirect(i).vect = vect \\ ioapic.redirect(i).rirr & otherwise \end{cases}$$

  Receiving an EOI message for interrupt vector *vect* resets the corresponding remote interrupt request signal associated with all redirect table entries associated with the interrupt vector. Note that it is adviseable to configure the system in such a way that interrupt vectors assigned to devices are unique.

All components not explicitly mentioned stay unchanged.

Active transitions of the I/O APIC can be found in the definition of the overall transition relation.

4.10 Multi-Core MIPS

Transitions of the abstract machine are defined as

$$\delta : K \times \Sigma \rightharpoonup K$$

Inputs of the system specify which processor component makes what kind of step and are defined below. On the level of abstraction provided, we assume that the memory subsystem does not make steps on its own, thus it may neither receive external inputs nor be scheduled to make an active step.

The transition functions of the subcomponents are given by

- memory transitions : $\delta_{\mathbf{m}} : K_{\mathbf{m}} \times \Sigma_{\mathbf{m}} \rightarrow K_{\mathbf{m}}$ (always passive, section 4.3),
- processor core transitions : $\delta_{\mathbf{core}} : K_{\mathbf{core}} \times \Sigma_{\mathbf{core}} \rightharpoonup K_{\mathbf{core}}$ (section 4.5),
- passive TLB transitions : $\delta_{\mathbf{tlb}} : K_{\mathbf{tlb}} \times \Sigma_{\mathbf{tlb}} \rightarrow K_{\mathbf{tlb}}$ (section 4.4, active transitions are given explicitly in the top level transition function),
- store-buffer transitions which are stated explicitly in the top level transition function,

- passive local APIC transitions : $\delta_{\mathbf{apic}} : K_{\mathbf{apic}} \times \Sigma_{\mathbf{apic}} \rightarrow K_{\mathbf{apic}}$ (see section 4.8),
- passive I/O APIC transitions: $\delta_{\mathbf{ioapic}} : K_{\mathbf{ioapic}} \times \Sigma_{\mathbf{ioapic}} \rightharpoonup K_{\mathbf{ioapic}}$ (section 4.9), and
- device transitions which are given by : $\delta_{\mathbf{dev(i)}} : K_{\mathbf{dev(i)}} \times \Sigma_{\mathbf{dev(i)}} \rightarrow K_{\mathbf{dev(i)}}$ (see section 4.7).

Additionally, we have an output-function

$$\lambda : K \times \Sigma \rightharpoonup \Omega$$

where

$$\Omega = \bigcup_{i=0}^{nd-1} \Omega_{\mathbf{dev(i)}}$$

that allows devices to interact in some way with the external world.

4.10.1 Inputs of the System

We define

$$\Sigma = \Sigma_{\mathbf{p}} \times [0 : np - 1] \cup \Sigma_{\mathbf{ioapic+dev}}$$

as the union of processor inputs and I/O APIC and device inputs. In the following, we define both of them.

Definition 4.63 (Processor Inputs) We have

$$\begin{aligned} \Sigma_{\mathbf{p}} \quad = \quad & \{\mathbf{core}\} \times K_{\mathbf{walk}} \times K_{\mathbf{walk}} \cup \{\mathbf{tlb\text{-}create}\} \times \mathbb{B}^{20} \cup \{\mathbf{tlb\text{-}extend}\} \times K_{\mathbf{walk}} \times \mathbb{B}^{3} \\ & \cup \{\mathbf{tlb\text{-}accessed\text{-}dirty}\} \times K_{\mathbf{walk}} \cup \{\mathbf{sb}\} \\ & \cup \{\mathbf{apic\text{-}sendIPI}, \mathbf{apic\text{-}sendEOI}, \mathbf{apic\text{-}initCore}, \mathbf{apic\text{-}startCore}\} \end{aligned}$$

Note that the processor and the store buffer are both deterministic, i.e. they have only one active step they can perform. In contrast, the TLB and the local APIC are non-deterministic, i.e. there are several steps that can be performed, thus, the scheduling part of the system's input specifies which step is made.

Definition 4.64 (I/O APIC and Device Inputs) We have

$$\begin{aligned} \Sigma_{\mathbf{ioapic+dev}} \quad = \quad & \{\mathbf{ioapic\text{-}sample}, \mathbf{ioapic\text{-}deliver}\} \times [0 : nd - 1] \\ & \cup \{\mathbf{device}\} \times [0 : nd - 1] \times \Sigma_{\mathbf{ext}} \end{aligned}$$

where

$$\Sigma_{\mathbf{ext}} = \bigcup_{i \in [0:nd-1]} \Sigma_{\mathbf{ext(i)}}$$

is the union of all external inputs to devices.

4.10.2 Auxiliary Definitions

In order to define the overall transition relation, we need a view of the memory that can serve read requests of the processor in the way we expect: depending on the address, a read request can go to a local apic, to the I/O-apic, to a device, to the store-buffer, or, if none of the aforementioned apply, to the memory. Depending on whether the machine is running in user mode or system mode, memory accesses are subject to address translation performed using the TLB component of the machine.

Definition 4.65 (Local APIC Base Address) The local APIC ports in this machine are mapped to address

$$apic_{\mathbf{base}} \equiv 1^{20}0^{12}$$

Definition 4.66 (Local APIC Addresses) The set of byte-addresses covered by local APIC ports is given by

$$A_{\mathbf{apic}} = \{a \in \mathbb{B}^{32} \mid 0 \leq \langle a \rangle - apic_{\mathbf{base}} < 128\}$$

Definition 4.67 (I/O APIC Base Address) The I/O APIC ports in this machine are always mapped to address

$$ioapic_{\mathbf{base}} \equiv 1^{19}0^{13}$$

Definition 4.68 (I/O APIC Addresses) The set of byte-addresses covered by the I/O APIC ports is

$$A_{\mathbf{ioapic}} = \{a \in \mathbb{B}^{32} \mid 0 \leq \langle a \rangle - ioapic_{\mathbf{base}} < 8\}$$

Definition 4.69 (Device Base Addresses) We assume a function

$$dev_{\mathbf{base}} : [0 : nd - 1] \rightarrow \mathbb{B}^{32}$$

to be given that specifies the base address of the ports region of all devices.

Definition 4.70 (Device Addresses) The set of addresses covered by device i's ports is given by

$$A_{\mathbf{dev(i)}} = \{a \in \mathbb{B}^{32} \mid 0 \leq \langle a \rangle - dev_{\mathbf{base}}(i) < 2^{dev_{\mathbf{palen}}}\}.$$

The set of all byte-addresses covered by device, local APIC and I/O APIC ports is defined as

$$A_{\mathbf{dev}} = \bigcup_{i=0}^{nd-1} A_{\mathbf{dev}}(i) \cup A_{\mathbf{apic}} \cup A_{\mathbf{ioapic}}$$

Definition 4.71 (Port Address) Given a memory address x, the corresponding port addresses of devices, local APIC and I/O APIC are computed as

$$dev_{\mathbf{adr}}(i{,}x)(i, x) = (x -_{32} dev_{\mathbf{base}}(i))[dev_{\mathbf{palen}}(i) - 1 : 0]$$

$$apic_{\mathbf{adr}}(x)(x) = (x -_{32} apic_{\mathbf{base}}(i))[6 : 0]$$

$$ioapic_{\mathbf{adr}}(x)(x) = (x -_{32} ioapic_{\mathbf{base}}(i))[2 : 0]$$

Definition 4.72 (Memory System) The results of read accesses performed by the processor core are described in terms of a memory system that takes into account device ports, I/O APIC ports, local APIC ports, the store buffer and the memory. We define a function *ms* that, given these components, returns the merged memory view seen by the processor core:

$$ms(dev,ioapic,apic,sb,m)(x) =$$

$$\begin{cases} sbv(sb,x) & sbhit(sb,x) \\ apic.ports(apic_{\mathbf{adr}}(x)) & \neg sbhit(sb,x) \land x \in A_{\mathbf{apic}} \\ ioapic.ports(ioapic_{\mathbf{adr}}(x)) & \neg sbhit(sb,x) \land x \in A_{\mathbf{ioapic}} \\ dev(i).ports(dev_{\mathbf{adr}}(i,x)) & \neg sbhit(sb,x) \land x \in A_{\mathbf{dev}}(i) \\ m(x) & otherwise \end{cases}$$

Note that, in order to have a meaningful memory system, the machine must be configured in such a way that address ranges of devices, I/O APIC and local APIC are pairwise disjoint.

Definition 4.73 (Current Address Space Identifier) The current address space identifier is given by the last 6 bits of the special purpose register *asid*:

$$asid(core) = core.spr(asid)[5:0]$$

4.10.3 Transitions of the Multi-Core MIPS

Let us define the transition function δ and the output function λ of MIPS-86 by a case distinction on the given input a:

$$\delta(c,a) = c'$$

Any subcomponent of configuration c' that is not listed explicitly in the following has the same value as in configuration c.

- $a = (\mathbf{core}, w_I, w_R, i)$ – processor core i performs a step (using walks w_I and w_R if running in translated mode; in system mode, w_I and w_R are ignored)

  In order to formalize a processor core step of processor i, we define the following shorthands:

  - $c_i = c.p(i).core$ – the processor core configuration of processor i,
  - $ms(i) = ms(c.dev, c.ioapic, c.p(i).apic, c.p(i).sb, m)$ – the memory view of processor i,
  - $mode_i = c_i.spr(mode)[0]$ – the execution mode of processor i,
  - $trqI = (asid(c_i), c_i.pc, 110)$ – the translation request for instruction execution if processor core i is running in user mode,
  - $pff \equiv mode_i = 1 \land fault(c.m, trqI, w_I)$ – signals whether there is a page-fault-on-fetch for the given walk w_I and the translation request $trqI$, and

- $pmaI = \begin{cases} w_I.pa \circ c_i.pc[11:0] & mode_i = 1 \\ c_i.pc & mode_i = 0 \end{cases}$

  – the physical memory address for instruction fetch of processor core i (which is only meaningful if no page-fault on instruction fetch occurs),
- $I = ms(i)_4(pmaI)$

  – the instruction fetched from memory for processor core i (in case of a page-fault-on-fetch the value of I has no further relevance),
- $trqEA = (asid(c_i), ea(c_i, I), 01 \circ (store(I) \vee rmw(I)))$ – the translation request for the effective address if processor core i is running in user mode,
- $pfls \equiv mode_i = 1 \wedge fault(c.m, trqEA, w_R) \wedge /pff \wedge (store(I) \vee load(I) \vee rmw(I))$

  – the page-fault-on-load-store signal for processor core i.
- $pmaEA = \begin{cases} w_R.pa \circ ea(c_i, I)[11:0] & mode_i = 1 \\ ea(c_i, I) & mode_i = 0 \end{cases}$

  – the physical memory address for the effective address of processor core i,
- $R = \begin{cases} \perp & pff \vee pfls \\ ms(i)_{d(I)}(pmaEA) & otherwise \end{cases}$

  – the value read from memory for a *read* or *rmw* instruction of processor core i,

$\delta(c, a)$ is defined iff all of the following hold:

- $mode_i = 1 \Rightarrow hit(w_I, trqI)$ – running in translated mode, the walk w_I must match the translation request for instruction fetch, and
- $mode_i = 1 \Rightarrow ((store(I) \vee load(I) \vee rmw(I)) \wedge \neg pff \Rightarrow (hit(w_R, trqEA)))$ – running in translated mode, if there is a read or write instruction and no page-fault on fetch has occurred, the walk w_R must match the translation request for the effective address, and
- $\neg pff \Rightarrow complete(w_I)$ – if there is no page-fault on fetch, walk w_I is complete, and thus, provides a translation from virtual to physical address, and
- $\neg pfls \Rightarrow complete(w_R)$ – if there is no page-fault on load/store, walk w_R is complete, and thus, provides a translation from virtual to physical address, and
- $(rmw(I) \vee mfence(I)) \Rightarrow c.sb = \varepsilon$ – a read-modify-write or a fence instruction can only be executed when the store-buffer is empty, and
- $pmaI \notin A_{\mathbf{dev}}$ – we do not fetch instructions from device ports, and
- $(rmw(I) \vee d(I) \neq 4 \wedge (load(I) \vee store(I))) \Rightarrow pmaEA \notin A_{\mathbf{dev}}$ – we exclude read-modify-write accesses and byte/halfword accesses to device ports, and
- $c.running(i)$ – only processors that are not waiting for a SIPI can execute.

Then,

$$c'.p(j).core = \begin{cases} \delta_{\mathbf{core}}(c_i, I, R, eev(c.p(i).apic), pff, pfls) & i = j \wedge (load(I) \vee rmw(I)) \\ \delta_{\mathbf{core}}(c_i, I, \perp, eev(c.p(i).apic), pff, pfls) & i = j \wedge (\neg load(I) \wedge \neg rmw(I)) \\ c.p(j).core & otherwise \end{cases}$$

$$c'.p(j).sb = \begin{cases} (pmaEA, sv(c_i, I)) \circ c.p(i).sb & i = j \wedge store(I) \\ c.p(j).sb & otherwise \end{cases}$$

$$c'.p(j).tlb = \begin{cases} \emptyset & i = j \wedge flush(I) \\ tlb' & i = j \wedge invlpg(I) \\ \delta_{\mathbf{tlb}}(c.p(i).tlb, (\mathbf{flush}, asid(c_i), c_i.pc.ba)) & i = j \wedge pff \\ \delta_{\mathbf{tlb}}(c.p(i).tlb, (\mathbf{flush}, asid(c_i), ea(c_i, I).ba)) & i = j \wedge /pff \wedge pfls \\ c.p(j).tlb & otherwise \end{cases}$$

where

$$tlb' = \delta_{\mathbf{tlb}}(\delta_{\mathbf{tlb}}(c.p(i).tlb, (\mathbf{flush}, c_i.gpr(rs(I))[5:0], c_i.gpr(rd(I)).ba)), \mathbf{flush\text{-}incomplete})$$

$$c'.m = \begin{cases} \delta_{\mathbf{m}}(c.m, (c_i.gpr(rd(I)), pmaEA, sv(c_i, I))) & rmw(I) \wedge pmaEA \notin A_{\mathbf{dev}} \\ c.m & otherwise \end{cases}$$

$$c'.dev(j) = \begin{cases} \delta_{\mathbf{dev(j)}}(c.dev(j), dev_{\mathbf{adr}}(j, pmaEA)) & lw(I) \wedge pmaEA \in A_{\mathbf{dev(j)}} \\ c.dev(j) & otherwise \end{cases}$$

The flag *running* cannot be modified by a processor core step; it can only be modified by the corresponding local APIC. Local APIC and I/O APIC configurations are never modified by a processor core step since neither local APICs nor I/O APIC have side-effects on reads and we do not allow read-modify-write accesses to devices – writes to devices always go through the store buffer, thus, any side-effects on device writes are triggered when the write leaves the store buffer.

Performing a processor core step of core i, we apply the processor core transition function to the current processor core configuration, providing the instruction word I read from memory, the read value R (if needed), the external event signals $eev(c.p(i).apic)$ provided by the local APIC belonging to processor i, and the page-fault signals *pff*, and *pfls* given above. In case of a *store*-instruction, the corresponding write access enters processor i's store buffer. If there is a page-fault, the TLB reacts by flushing all translations for the page-faulting address – this is necessary in our model in order to allow the MMU to rewalk the page-tables after interrupt handling without triggering the old page-fault. Only in case of a read-modify-write instruction, the memory component reacts directly to the read-modify-write access (since, in all other cases, the store-buffer receives any write requests). Last, in case there is a read-access to a device, the corresponding device transition function is triggered: It specifies how the device reacts to the read-access by specifying appropriate side-effects on reading for the device.

$\lambda(c, a)$ undefined: The processor core does not interact with the external environment, this is exclusive to devices.

Note that continue interrupts can only be caused by execution of instructions that affect only the processor core – thus, continue interrupts are covered in an adequate way in the definitions given here. That is, we do not need to consider changes to other components than the processor core in the case of a continue interrupt.

- $a = (\mathbf{sb}, i)$ – a memory write leaves store buffer i

  $\delta(c, a)$ is defined iff $c.p(i).sb \neq \varepsilon$ – the store buffer can only make a step when it is not empty. Then,

$$c'.p(j).sb = \begin{cases} c.p(i).sb[|c.p(i).sb| - 1 : 1] & i = j \\ c.p(j).sb & i \neq j \end{cases}$$

$$c'.p(j).apic = \begin{cases} \delta_{\mathbf{apic}}(c.p(i).apic, (apic_{\mathbf{adr}}(a), v)) & i = j \wedge c.p(i).sb[0] = (a, v) \wedge a \in A_{\mathbf{apic}} \\ c.p(j).apic & otherwise \end{cases}$$

$$c'.m = \begin{cases} \delta_{\mathbf{m}}(c.m, c.p(i).sb[0]) & c.p(i).sb[0] = (a, v) \wedge a \notin A_{\mathbf{dev}} \\ c.m & otherwise \end{cases}$$

$$c'.ioapic = \begin{cases} \delta_{\mathbf{ioapic}}(c.ioapic, (ioapic_{\mathbf{adr}}(a), v)) & c.p(i).sb[0] = (a, v) \wedge a \in A_{\mathbf{ioapic}} \\ c.ioapic & otherwise \end{cases}$$

$$c'.dev(j) = \begin{cases} \delta_{\mathbf{dev(j)}}(c.dev(j), (dev_{\mathbf{adr}}(j, a), v)) & c.p(i).sb[0] = (a, v) \wedge a \in A_{\mathbf{dev(j)}} \\ c.dev(j) & otherwise \end{cases}$$

  Store buffer steps never change processor core configurations, TLB configurations or the *running* flag. The oldest write in the store buffer is handled by the component the address belongs to. Note that here, we rely on the correct alignment of accesses, since otherwise, write accesses might partially cover ports and memory at the same time.

  $\lambda(c, a)$ undefined

- $a = (\textbf{tlb-create}, va, i)$ – a new walk for virtual address va is created in TLB i

  $\delta(c, a)$ is defined iff

  - $c.p(i).spr(mode)[0] = 1$ – the TLB will only create walks when the processor is running in user mode, and
  - $c.running(i)$ – the TLB will only create walks when the processor is not waiting for SIPI.

  Then,

$$c'.p(j).tlb = \begin{cases} c.p(i).tlb \cup winit(va, c_i.spr(pto).ba, asid(c.p(i)) & i = j \\ c.p(j).tlb & otherwise \end{cases}$$

  Creating a new walk in the TLB is a step that affects only the TLB.

  $\lambda(c, a)$ undefined

- $a = (\textbf{tlb-set-accessed-dirty}, w, i)$ – accessed and dirty bits of the page table entry needed to extend walk w in TLB i are set appropriately

  $\delta(c, a)$ is defined iff

  – $c.p(i).spr(mode)[0] = 1$ – page table entry flags can only be set in translated mode,
  – $w \in c.p(i).tlb \land \neg complete(w) \land w.asid = asid(c.p(i))$ – we only set dirty and accessed bits for incomplete walks of the current address space identifier, and
  – $pte(c.m, w).p = 1$ – the MMU can only set accessed/dirty flags for page table entries which are actually present.

  Then,

  $c'.m = \delta_{\mathbf{m}}(c.m, (ptea(w), set\text{-}ad(pte(c.m, w), w)))$

  Setting the page table entry flags only affects the corresponding page table entry in memory. In this model, the MMU non-deterministically sets accessed and dirty flags – enabling walk extension using the given page table entriy.

  $\lambda(c, a)$ undefined

- $a = (\textbf{tlb-extend}, w, i)$ – an existing walk in TLB i is extended

  $\delta(c, a)$ is defined iff

  – $w \in c.p(i).tlb$ – the walk is to be extended is contained in the TLB, and
  – $\neg complete(w)$ – the walk is not yet complete, and
  – $w.asid = asid(c.p(i))$ – the walk is for the current ASID, and
  – $pte(c.m, w).a \land (w.level = 1 \Rightarrow pte(c.m, w).d)$ – the accessed/dirty flags are set appropriately, and
  – $\neg wext(w, pte(c.m, w)).fault$ – the walk extension does not fault result in a faulty walk, and
  – $c.running(i)$ – the TLB will only extend walks when the processor is not waiting for SIPI.

  Then,

  $$c'.p(j).tlb = \begin{cases} \delta_{\mathbf{tlb}}(c.p(i).tlb, \textbf{add-walk}(wext(w, pte(c.m, w)))\} & i = j \\ c.p(j).tlb & otherwise \end{cases}$$

  Walk extension only affects the TLB, note however, that in order to perform walk extension, the corresponding page-table entry is read from memory.

  $\lambda(c, a)$ undefined

- $a = (\textbf{apic-sendIPI}, i)$ – local APIC i sends a pending inter-processor-interrupt to all target local APICs

  $\delta(c, a)$ is defined iff

  – $c.p(i).apic.\mathbf{ICR}.ds = 1$ – there is currently an inter-processor-interrupt to be delivered, and

- $c.p(i).apic.\mathbf{ICR}.destmode \neq 0$ – the destination mode is set to something other than 0

Then,

$$c'.p(j).apic = \begin{cases} \delta_{\mathbf{apic}}(apic', (type, vect)) & i = j \wedge \textit{self-target} \\ apic' & i = j \wedge \neg\textit{self-target} \\ \delta_{\mathbf{apic}}(c.p(j).apic, (type, vect)) & i \neq j \wedge (t = \mathbf{ID} \\ & \quad \wedge c.p(j).apic.\mathbf{APIC\_ID} = c.p(i).apic.\mathbf{ICR}.dest \\ & \vee t = \mathbf{ALL\text{-}BUT\text{-}SELF} \vee t = \mathbf{ALL}) \\ c.p(j).apic & otherwise \end{cases}$$

where

$$vect = c.p(i).apic.\mathbf{ICR}.vect[7:0]$$

is the interrupt vector that is sent over the interrupt bus,

$$type = \begin{cases} \mathbf{Fixed} & c.p(i).apic.\mathbf{ICR}.dm = 0^3 \\ \mathbf{INIT} & c.p(i).apic.\mathbf{ICR}.dm = 101 \\ \mathbf{SIPI} & c.p(i).apic.\mathbf{ICR}.dm = 110 \end{cases}$$

is the type of interrupt as specified by the command register of the sending local APIC,

$$t = \begin{cases} \mathbf{ALL} & c.p(i).apic.\mathbf{ICR}.dsh = 10 \\ \mathbf{ALL\text{-}BUT\text{-}SELF} & c.p(i).apic.\mathbf{ICR}.dsh = 11 \\ \mathbf{SELF} & c.p(i).apic.\mathbf{ICR}.dsh = 01 \\ \mathbf{ID} & c.p(i).apic.\mathbf{ICR}.dsh = 00 \end{cases}$$

describes the target mode of the requested inter-processor interrupt,

$$\textit{self-target} \equiv (t = \mathbf{SELF} \vee t = \mathbf{ALL} \vee (t = \mathbf{ID} \wedge c.p(i).apic.\mathbf{APIC\_ID} = c.p(i).apic.\mathbf{ICR}.dest))$$

expresses whether the sending local APIC is also a target of the inter-processor interrupt, and $apic'$ is identical to $c.p(i).apic$ everywhere except $apic'.\mathbf{ICR}.ds = 0$.

Sending an inter-processor-interrupt only affects local APIC configurations – both of the sending local APIC and the receiving ones. All receiving local APICs perform a passive receive-interrupt transition.

$\lambda(c, a)$ undefined

- $a = (\mathbf{apic\text{-}sendEOI}, i)$ – local APIC i sends an EOI message to the I/O APIC

  $\delta(c, a)$ is defined iff

  - $\bigvee_i c.p(i).apic.eoipending[i] = 1$ – there is currently an end-of-interrupt message pending

Then,

$$c'.p(j).apic = \begin{cases} apic' & i = j \\ c.p(j).apic & otherwise \end{cases}$$

where $apic'$ is identical to $c.p(i).apic$ everywhere except

$$apic'.eoipending[k] = \begin{cases} 0 & k = \langle vect \rangle \\ c.p(i).apic.eoipending[k] & otherwise \end{cases}$$

and $vect = min\{l \mid c.p(i).apic.eoipending[l] = 1\}_8$

$c'.ioapic = \delta_{\mathbf{ioapic}}(c.ioapic, vect)$

Transmitting an EOI message affects only the sending local APIC and the I/O APIC. When a local APIC sends an EOI message it is always for the smallest interrupt vector for which an EOI message is pending. The I/O APIC receives the EOI message and reacts with the passive transition given by $\delta_{\mathbf{ioapic}}$ that resets the remote interrupt request flag in its redirect table, re-enabling the I/O APIC to sample the corresponding device interrupt.

$\lambda(c, a)$ undefined

- $a =$ (**apic-initCore**, i) – local APIC i applies a pending INIT-IPI to processor core i

  $\delta(c, a)$ is defined iff $c.p(i).apic.initrr = 1$ – there is currently an INIT-IPI pending in the local APIC of processor i. Then,

  $$c'.p(j).core = \begin{cases} core' & i = j \\ c.p(j).core & otherwise \end{cases}$$

  where

  $core'$ is identical to $c.p(i).core$ except for $core'.pc = 0^{32}$, $core'.spr(mode) = 0^{32}$ and $core'.spr(eca) = 0^{32}$.

  $$c'.running(j) = \begin{cases} 0 & i = j \\ c.running(j) & otherwise \end{cases}$$

  $$c'.p(j).apic = \begin{cases} apic' & i = j \\ c.p(j).apic & otherwise \end{cases}$$

  where $apic'$ is identical to $c.p(i).apic$ everywhere except $apic'.initrr = 0$.

  When the local APIC applies an INIT-IPI to the corresponding processor core, the store buffer and TLB of that processor as well as the global memory and all other devices are not affected. The INIT-IPI effectively acts as a warm reset to the processor core.

  $\lambda(c, a)$ undefined

- $a =$ (**apic-startCore**, i) – local APIC i applies a pending SIPI interrupt to processor core i

  $\delta(c, a)$ is defined iff

  - $c.p(i).apic.sipirr = 1$ – there is currently a SIPI pending in the local APIC of processor i, and

- $c.running(i) = 0$ – the processor is currently waiting for SIPI.

Then,

$$c'.p(j).core = \begin{cases} core' & i = j \\ c.p(j).core & otherwise \end{cases}$$

where

$core'$ is identical to $c.p(i).core$ except for $core'.pc = c.p(i).apic.sipivect \circ 0^{24}$.

$$c'.running(j) = \begin{cases} 1 & i = j \\ c.running(j) & otherwise \end{cases}$$

$$c'.p(j).apic = \begin{cases} apic' & i = j \\ c.p(j).apic & otherwise \end{cases}$$

where $apic'$ is identical to $c.p(i).apic$ everywhere except $apic'.sipirr = 0$.

Similar to the INIT-IPI, the store buffer and TLB of that processor as well as the global memory and all other devices are not affected. The interrupt vector of the SIPI is used to initialize the program counter of the processor core and the flag $c.running(i)$ is set in order to allow the processor core to perform steps.

$\lambda(c, a)$ undefined

- $a = ($**ioapic-sample**$, k)$ – the I/O APIC samples the raised interrupt of device k

  $\delta(c, a)$ is defined iff

  - $c.dev(k).irq = 1$ – device k does currently have an interrupt signal activated, and
  - $c.ioapic.redirect(k).mask = 0$ – the interrupt of device k is not masked, and
  - $c.ioapic.redirect(k).rirr = 0$ – handling of any previous interrupt for device k has been acknowledged by an EOI message from the corresponding local APIC.

  Then,

  $c'.ioapic$ is identical to $c.ioapic$ except for

$$c'.ioapic.redirect(i).ds = \begin{cases} 1 & i = k \\ c.ioapic.redirect(i).ds & otherwise \end{cases}$$

  Sampling a device interrupt only affects the state of the I/O APIC. The delivery status bit of the corresponding redirect table entry is set in order to allow a subsequent **ioapic-deliver** transition.

  $\lambda(c, a)$ undefined

- $a = ($**ioapic-deliver**$, k)$ – the I/O APIC delivers a pending interrupt from device k to the target local APIC

  $\delta(c, a)$ is defined iff

– $c.ioapic.redirect(k).ds = 1$ – there is currently an interrupt pending to be delivered for device k

Then,

$$c'.p(j).apic = \begin{cases} \delta_{\mathbf{apic}}(c.p(j).apic, (\mathbf{Fixed}, v)) & c.p(j).apic.\mathbf{APIC\_ID} = c.ioapic.redirect(k).dest \\ c.p(j).apic & otherwise \end{cases}$$

where $v = c.ioapic.redirect(k).vect$

and $c'.ioapic$ is identical to $c.ioapic$ except for the following:

– $c'.ioapic.redirect(k).ds = 0$

– $c'.ioapic.redirect(k).rirr = 1$

Delivering a pending device interrupt only affects the I/O APIC and the target local APIC specified in the redirect table of the I/O APIC. The I/O APIC sets the remote interrupt request flag in order to prevent sampling the same device interrupt several times. Only after the corresponding local APIC acknowledges handling of the interrupt vector to the I/O APIC by sending an EOI message, sampling the device interrupt is possible again.

$\lambda(c, a)$ undefined

- $a = (\mathbf{device}, k, ext)$ – device k performs a step under given external input ext

  $\delta(c, a)$ is defined iff

  – $ext \in \Sigma_{\mathbf{dev(k)}}$ – the external input belongs to device k.

  Then,

$$c'.dev(j) = \begin{cases} \delta_{\mathbf{dev(k)}}(c.dev(k), ext) & j = k \\ c.dev(j) & otherwise \end{cases}$$

  Only device k is affected. Execution proceeds as specified by the device transition function.

  $\lambda(c, a) = \lambda_{\mathbf{dev(k)}}(c.dev(k), ext)$

  The device may provide an output to the external world as specified by its output function.

4.10.4 Multi-Core MIPS Computation

Given an input sequence $s : \mathbb{N} \to \Sigma$ of the Multi-Core MIPS and a sequence of Multi-Core MIPS configurations (c^i), these two define a Multi-Core MIPS computation if and only if

$$\forall i : \quad c^{i+1} = \delta(c^i, s(i))$$

4.11 Booting a MIPS-86 Machine

4.11.1 Initial Configuration after Reset

After reset, an arbitrary configuration c fulfilling the following restrictions is chosen non-deterministically:

- The program counter of all processors is initialized: $\forall i \in [0 : np-1] : \quad c.p(i).pc = 0^{32}$
- All processors except the boot-strap processor (BSP) are waiting for SIPI:

$$c.running(0) = 1, \qquad \forall i \in [1 : np-1] : \quad c.running(i) = 0$$

- All processors are in system mode:

$$\forall i \in [0 : np-1] : \quad c.p(i).mode = 0^{32}$$

- The store-buffers of all processors are empty:

$$\forall i \in [0 : np-1] : \quad c.p(i).sb = \varepsilon$$

- The TLBs of all processors do not contain translations:

$$\forall i \in [0 : np-1] : \quad c.p(i).tlb = \emptyset$$

- The I/O APIC does not have any interrupts to deliver and all device interrupt lines are masked:

$$\forall i : c.ioapic.redirect(i).rirr = 0$$

$$\forall i : c.ioapic.redirect(i).ds = 0$$

$$\forall i : c.ioapic.redirect(i).mask = 0$$

- The local APICs of all processors do not have pending interrupts and the APIC ID is initialized:

$$\forall i \in [0 : np-1] : \quad c.p(i).apic.\mathbf{IRR} = 0^{256}$$

$$\forall i \in [0 : np-1] : \quad c.p(i).apic.\mathbf{ISR} = 0^{256}$$

$$\forall i \in [0 : np-1] : \quad c.p(i).apic.\mathbf{ICR} = 0^{64}$$

$$\forall i \in [0 : np-1] : \quad c.p(i).apic.\mathbf{APIC\_ID} = i_8$$

- All devices are in an initial state:

$$\forall i \in [0 : nd-1] : \quad initial_{\mathbf{state(i)}}(c.dev(i))$$

4.11.2 Booting

Since initially, all processors except $c.p(0)$ – the boot-strap processor (BSP) – are waiting for a SIPI-interrupt, the machine executes sequentially until the BSP uses its local APIC to issue SIPIs (startup inter-processor-interrupts) to the other processors.

5 C-IL Semantics

c *noun, often capitalized often attributive*

1. **a** : the 3rd letter of the English alphabet

[...]

8. a structured programming language designed to be compact and efficient

"c." Merriam-Webster.com. Merriam-Webster, 2011. Web. 6 July 2011.

5.1 The C Programming Language

A treatment of the history of C, describing the origins of the programming language C as a successor of the languages BCPL and B, is given by Dennis M. Ritchie in *"The development of the C language"*[Rit93]. C is a programming language designed from the very beginning with system programming in mind: The development of the UNIX operating system together with Ken Thompson required a programming language adequate to the task. Dennis M. Ritchie characterizes the language as follows:

> "BCPL, B, and C all fit firmly in the traditional procedural family typified by Fortran and Algol 60. They are particularly oriented towards system programming, are small and compactly described, and are amenable to translation by simple compilers. They are 'close to the machine' in that the abstractions they introduce are readily grounded in the concrete data types and operations supplied by conventional computers, and they rely on library routines for input-output and other interactions with an operating system."
>
> *"The development of the C language"*[Rit93], Dennis M. Ritchie

From this, it is quite obvious, that C was never designed to be a "beautiful" abstract language, but instead, to serve the purpose of solving an engineering problem.

> "Two ideas are most characteristic of C among languages of its class: the relationship between arrays and pointers, and the way in which declaration syntax mimics expression syntax. They are also among its most frequently criticized features, and often serve as stumbling blocks to the beginner."
>
> *"The development of the C language"*[Rit93], Dennis M. Ritchie

In fact, as one of the more serious stumbling blocks to C, I would like to mention the significant amount of implementation-defined semantics, as given by the specific compiler implementation and architecture used. Over the years, there have been many standardization attempts for C, e.g. the book *"The C Programming Language"*[KR88] by Dennis M. Ritchie and Brian W.

Kernighan, the ANSI98 standard for C agreed on by the American National Standards Institute (ANSI) which was – with some minor adaptations – adopted by the International Organization for Standardization (ISO) to under the name C90 – which evolved to the C99 standard (ISO/IEC 9899) [ISO99]. Since an exact formalization of C is an arduous task without much gain (it appears to be much more useful to retreat to a well-defined subset of C) the author of this thesis does not even try to formalize standard-conforming C in all of its intricate details. Instead, we take a route that others have gone before: We introduce a simple intermediate language that can be used to implement C. Necula et al.[NMRW02] report considerable benefits gained from translating C code to a "clean" language subset of C they call C intermediate language (CIL). Xavier Leroy et al.[BL09] also used simplified and intermediate language versions of C – *Cminor* and *Clight* – for their compiler verification efforts. Their language, however, obviously aims at a somewhat higher level of abstraction by providing a memory model with an explicit notion of a heap that obscures the exact memory layout. Implementing an operating system or hypervisor, however, it appears desirable to have full control over the memory layout in order to implement low-level memory management explicitly.

The main advantage of using an intermediate language is that, instead of considering many different kinds of control structures and all the syntactic sugar C provides to the programmer, we simply have to consider one representative instance for every unique semantic construct. To argue that such a reduced intermediate language is adequate, it suffices to provide compilation rules from C to the intermediate language and check that the semantics we obtain by executing the compiled code matches what we expect.

What follows is a description of basic features of the intermediate-language *C-IL* proposed by the author and a discussion of why the style of modeling chosen is useful in the bigger context of pervasive system verification we aim at. A formal description of *C-IL* is given afterwards.

5.2 Basic Features & Design Decisions

Requirements imposed on *C-IL* stem from five different goals: i) the *C-IL*-model must be easy to integrate with hardware models, ii) it must be possible to implement a large subset of standard-compliant C using *C-IL*, iii) it must provide abstraction to allow verification tools to be efficient, iv) it must be useable in a concurrent context, and v) it must be possible to write low-level system code in it.

C-IL is a simple *goto*-language with very few different statements. At the same time, the language makes use of a rich type set, a large subset of C's types are provided. *C-IL* supports complex expressions, which, however, are side-effect free. Programs operate on local variables and on a byte-addressable global memory that closely corresponds to hardware memory. Pointer arithmetics is allowed to the full extent covered by the standard. We model *C-IL*-programs using mathematics – while the notation used is in many cases similar to the syntax of C, there are certain differences. Since *C-IL* is merely an intermediate language, the explicit modeling of C's syntax is left as future work.

C-IL is designed as an intermediate language for C that can be adapted to different architectures and compilers in a modular way. For this, we introduce what we call *environment parameters*: We model information about architecture, compiler, and the execution environment explicitly and define the semantics of *C-IL* based on these parameters.

In the following, we elaborate on the basic features of *C-IL* and why they are adequate to fulfill the requirements imposed on *C-IL*.

Reduced Number of Statements *C-IL* allows assignment, goto, if-not-goto, function call and return statements. Obviously, with so few different statements, we obtain a very simple language for which is it easy to correctly implement verification tools. With only three statements (assignment, function call and return) updating memory content, it is not overly difficult to extend the language to the concurrent case.

Unstructured Control-Flow Since we aim for the most simple low-level C language we can imagine, it is an obvious choice to provide only one way to express the control flow of a program. While high-level programming language constructs as *while*- or *for*-loops provide a nice abstraction, they cannot be used easily to implement all types of branching provided by the programming language C (e.g. *switch*, *break*). Thus, it seems a much more natural choice to use a *goto*-formalism for *C-IL*; modeling the program as a flat transition system is also much closer to how control flow is implemented in hardware. We implement this formally by defining a *location counter* that points to the next statement to be executed in the current function body. A function body is defined by a list of program statements.

Rich Type-System The choice of providing many C types and qualifiers (except bit-fields, floating-point types and the restrict-qualifier) is rather unusual for an intermediate-language. The modeling of qualifiers is owed directly to the fact that *C-IL* should be used with optimizing compilers, while the explicit modeling of types is helpful in setting up an object-based concurrent verification methodology (such as the one used in VCC) for *C-IL*.

Side-Effect-Free Expressions *C-IL* provides the following expressions: constant, variable, function name, unary operation, binary operation, ternary operation, type cast, dereferencing of pointer, taking address-of, field access, size of types/expressions. In combination with the rich type system, we obtain a rather high-level language – considering *C-IL* is just an intermediate language.

Byte-Addressable Memory *C-IL* operates on a byte-addressable global memory which, with exception of the stack-region, coincides with the memory of the hardware machine. An important software condition is that we never perform reads or writes to memory addresses belonging to the stack region since we provide an abstract model of the stack that is to be used instead. Intrinsically, *C-IL* has no notion of a heap or of memory allocation. This kind of abstraction must be implemented by memory allocation functions, e.g. the ones from the C standard library, or those implemented by an operating system or hypervisor.

Stack Abstraction The only memory region we do not consider as byte-addressable is the stack-region. Note that we could consider the explicit stack layout in the global byte-addressable memory. However, we do not want to expose this in our semantics. This is why we provide a stack abstraction in form of a list of abstract stack frames. A frame contains information about the values of local variables (modeled as byte-strings) and the control state.

Explicit Pointer Arithmetics On the global byte-addressable memory, any computation with pointers is permitted. On local variables and parameters, pointer arithmetics is restricted to calculating offsets that fall in the scope of the byte-representation of the corresponding local variable or parameter. Since we abstract from the concrete stack layout by providing an abstract stack, we do not know the base addresses of local variables and parameters in our semantics. The execution of programs that do not adhere to this pretty lenient pointer arithmetics restriction cannot be modeled with our semantics: Execution will get stuck. Since modern code verification tools tend to enforce even stricter pointer arithmetics restrictions, this, however, is not a problem in practice.

Environment Parameters These specify how the compiler and architecture behave and how the program is set up in memory. This includes (but is not limited to) the endianness (byte-order) of the architecture, the addresses of global variables in global memory, a type-cast function that matches the behavior of the compiler, the size of struct/union types in bytes, offsets of fields in struct/union types, the addresses of function pointers, and the type of the *sizeof*-operator.

What follows is a formal definition of the language *C-IL* and its operational semantics. After introducing some basic sets, we begin by describing the environment parameters in detail, followed by a definition of types, values, expressions, and statements of *C-IL*. We formally define *C-IL* programs and the possible resulting program configurations. Expression evaluation is defined in order to define operational semantics – which is defined in terms of a *C-IL* automaton performing steps from configuration to configuration under some given program. *C-IL* can be instantiated for given architectures and compilers. Thus, we give a definition of MIPS-86-*C-IL* to illustrate how exactly *C-IL* semantics is supposed to be used.

5.2.1 Basic Sets

We define *C-IL* semantics based on some given sets which can be assumed to be isomorphic to the natural numbers (i.e. infinite and enumerable). These sets are

- $\mathbb{V}$ – the *set of variable names*

  Contains all possible variable names.

- $\mathbb{F}$ – the *set of field names*

  Contains all possible names of fields of struct types.

- $\mathbb{F}_{name}$ – the *set of function names*

  Contains all possible function names.

- $\mathbb{T}_C$ – the *set of composite type names*

  Contains all possible names of struct types.

While, essentially, all these sets could be defined to be identical, distinguishing between them in the following definition of *C-IL* semantics should provide a better intuition in terms of what kinds of names may occur where.

5.3 Environment Parameters

C-IL is such a low-level language that its semantics cannot be described fully without knowledge about the environment it runs in. In particular, this involves specifying how the *C-IL* compiler behaves and specifying certain features of the architecture the code will run on. Among other things, we require that it is specified how structural types are laid out in memory and that we know the base addresses of global variables in the byte-addressable memory.

We use $\theta \in params_{C\text{-}IL}$ to denote a record of environment parameters for *C-IL*. It consists of the following components:

- $\theta.\mathbb{T}_P \subset \mathbb{T}_{PP}$ – the set of primitive types offered by the compiler

  This set is given as a subset of the *set of possible primitive types* $\mathbb{T}_{PP}$ of *C-IL* which is defined in the next section and includes all theoretically possible signed and unsigned integer types as well as the type void. We currently do not consider floating-point types. This, however, is a simple extension.

- $\theta.endianness \in \{\mathbf{little}, \mathbf{big}\}$ – the endianness of the underlying architecture

  The endianness of an architecture determines the order of individual bytes in a memory access that affects several bytes (e.g. a word access): Either, such data is transferred to/from memory starting with the least significant byte (little-endian), or it is transferred starting with the most significant byte (big-endian). To the programmer, this architecture feature becomes visible when memory accesses of different byte-width are used on the same memory region. While there are more exotic types of endianness, e.g. middle-endian, we do not consider them currently.

- $\theta.size_{ptr} \in \mathbb{N}$ – the size of pointer types in bytes

  For many C compilers, the size of pointer types depends on the size of a machine word of the underlying architecture. Thus, we make it a parameter of the semantics.

- $\theta.size\_t \in \theta.\mathbb{T}_P$ – the type of the value returned by the `sizeof`-operator

  Every C compiler must fix a type for the value returned by the `sizeof`-operator. The C99 Standard does not prescribe a specific type for this value. In practice, however, the unsigned integer type that exactly matches a machine word of the architecture is chosen most commonly.

- $\theta.cast : val \times \mathbb{T} \rightharpoonup val$ – a function that describes how the compiler handles type cast

  The function takes a pair of value and type and returns the resulting value after casting the given value to the given type. The C99 standard partially specifies the type casting function. The undefined type casts may be handled by the compiler implementation in any particular way. In order to allow stating semantics even for compiler-dependent type casts, we have the type cast function as a parameter to the semantics. While the function may be partial to account for undefined type casts, it must be defined for any type cast that occurs in the considered program. A more in-depth examination of the type cast function can be found in the section on values of *C-IL* (section 5.5.1).

- $\theta.op_1 : \mathbb{O}_1 \to (val \rightharpoonup val)$ – a function that describes the behavior of un ary operators

  For each unary operator from the set $\mathbb{O}_1$ of unary operators provided by *C-IL* (this is a specific set of operators defined in section 5.5.2), it is possible to define how that unary operator behaves. Actually, most of the operators provided are explicitly defined and do not allow for implementation-specific behavior (see section 5.5.2). However, in case the C code relies on implementation-defined behavior it can be useful to provide the semantics explicitly.

- $\theta.op_2 : \mathbb{O}_2 \to (val \times val \rightharpoonup val)$ – a function that describes the behavior of binary operators

  Similar to the unary operators, we explicitly provide semantics for binary operators from the set $\mathbb{O}_2$ of *C-IL*. Note that most operators are only defined for pairs of values of the same type; there are exceptions for adding integers to pointers, though.

- $\theta.intrinsics : \mathbb{F}_{name} \rightharpoonup FunT$ – a function table for compiler intrinsic functions

  C compilers in general provide access to architecture-specific features by means of *intrinsic* functions. Essentially, these intrinsic functions stand for special hardware instructions that are usually inlined (instead of compiled to actual function calls). $\theta.intrinsics$ provides a function table for these intrinsic functions; a function table entry represents a function declaration, i.e. how many parameters the function takes and whether it produces a return value. The function table should be defined for any compiler intrinsic function used by the program.

- $\theta.offset : \mathbb{T}_C \times \mathbb{F} \rightharpoonup \mathbb{N}$ – byte offsets of fields in struct or union types

  In order to express how field accesses on values of structural type operate on the byte-addressable memory, we need to know the byte offsets of fields. These byte offsets can then be used to calculate the base address of the sub-variable corresponding to the field access for the memory access. It needs to be defined only for those field accesses which do actually occur in the program. A restriction that must be fulfilled by the compiler is that the byte offsets specified result in a memory layout in which the byte-representations of struct fields do not overlap.

- $\theta.offset : \mathbb{T}_C \rightharpoonup \mathbb{N}$ – the size of struct or union types in bytes

  The order of fields in struct and union types can be optimized by the compiler, e.g. in order to save space while maintaining a correct alignment for all fields. Thus, it is not defined by the C99 Standard how many bytes a struct or union occupies in memory. In order to state the semantics of the `sizeof`-operator for struct and union types, we thus need this information given as a parameter to the semantics. This partial function needs to be defined for any composite type on which we use the `sizeof`-operator. The restriction here is that the size needs to be large enough to completely cover the memory range occupied by the byte-representations of all fields.

- $\theta.alloc_{gvar} : \mathbb{V} \rightharpoonup \mathbb{B}^{8\cdot\theta.size_{ptr}}$ – the base addresses of global variables

  Since the memory model chosen is simply the byte-addressable memory of the host machine, we can only express the effect of reads and writes to global variables when we know their base addresses in that byte-addressable memory. This is a partial function that needs to be defined for every global variable the program accesses. An imporant restriction is that the global variable base addresses specified here result in non-overlapping memory ranges for the declared global variables.

- $\theta.\mathcal{F}_{adr} : \mathbb{F}_{name} \rightharpoonup \mathbb{B}^{8\cdot\theta.size_{ptr}}$ – function pointer values

  We do want to argue about code that stores function pointers in memory – in order to set up interrupt descriptor tables to point to the begin of the respective compiled interrupt service routines. Thus, we have – as a parameter to the semantics – a function that returns, for a given function name, the corresponding address in the byte-addressable memory where the compiled code for that function begins. This is a partial function that only needs to be defined for any function we need to take a function pointer of.

- $\theta.R_{\textbf{extern}} : \mathbb{F}_{name} \rightharpoonup 2^{val^{*}\times conf_{C\text{-}IL}\times conf_{C\text{-}IL}}$ – transition relations for external procedures

  In order to define the effect of functions whose implementation is not given by the *C-IL*-program, we define corresponding transition relations. These transition relations describe the effect of the corresponding function under a given finite sequence of parameter values on a pair of *C-IL*-states: When

  $$((p_1,\ldots,p_k),c,c') \in \theta.R_{\textbf{extern}}(f)$$

  this denotes that, given parameter values $p_1,\ldots,p_k$ the function call to f in given state c may result in state c'. This is used to define the effects of compiler intrinsic functions or external function calls (e.g. to device drivers or functions implemented in different programming languages or external libraries). A transition relation should be defined for any function call which is not implemented by the *C-IL*-program; otherwise the semantics will get "stuck" at such a call.

  Note that, for the purpose of *C-IL* on its own, we only need to express the effect of the external function call on the *C-IL* state. When we are interested in bigger systems, we must additionally specify the effect of external function calls on external state – which shall be defined individually for such a system.

The components $\theta.offset$ and $\theta.size_{comp}$ depend on the program for which semantics are stated – while $\theta.alloc_{gvar}$ and $\theta.\mathcal{F}_{adr}$ additionally depend on where in the memory of the physical machine the program is loaded, i.e. where the global memory starts and where the code region starts. Given the composite type declarations, global variable declarations and function declarations as well as their implementations, we can then get the desired information from the compiler.

5.4 Types

C-IL provides a quite rich type system that provides primitive types (e.g., `int`, `void`), composite types (`structs`), array types, pointer types, and function pointer types. Each of these can be accompanied by *type qualifiers* – for *C-IL*, we only model the type qualifiers `volatile` and `const` explicitly as type qualifiers (the type qualifier `unsigned` is instead modeled by defining a corresponding separate *C-IL* type). In the following, we give a formal model of the types of *C-IL*.

5.4.1 The Set of Types

Depending on the compiler and the underlying architecture, different primitive types may be provided. The environment parameters θ include a *set of primitive types* $\theta.\mathbb{T}_P$ which must be given as a subset of the *set of possible primitive types* $\mathbb{T}_{PP}$.

Definition 5.1 (Set of Possible Primitive Types) We define the set of possible primitive types $\mathbb{T}_{PP}$ of *C-IL* as the smallest set that fulfills the following:

- It contains the void type: $\mathbf{void} \in \mathbb{T}_{PP}$
- It contains all possible integer types: $n \in \mathbb{N}_{>0} \wedge 8 \mid n \Rightarrow \mathbf{i}n \in \mathbb{T}_{PP}$
- It contains all possible unsigned integer types: $n \in \mathbb{N}_{>0} \wedge 8 \mid n \Rightarrow \mathbf{u}n \in \mathbb{T}_{PP}$

Note that we do not provide the type `_Bool` specified in the C99 standard. Obviously, this type can be implemented on a higher level of abstraction using one of the primitive unsigned or signed integer types.

Most commonly, compilers tend to provide only such integer types $\mathtt{i}n, \mathtt{u}n$ for which $\exists k : n = 2^k \cdot 8$ holds – due to alignment restrictions of the underlying architecture.

Example 1 (Set of Primitive *MIPS-86-C-IL* Types) For *MIPS-86*, an appropriate set of primitive *C-IL* types is the following:

$$\mathbb{T}_{PP} = \{\mathbf{void}, \mathbf{i8}, \mathbf{u8}, \mathbf{i16}, \mathbf{u16}, \mathbf{i32}, \mathbf{u32}\}$$

Given just the set of primitive types $\theta.\mathbb{T}_P$ and the set of composite type names $\mathbb{T}_C$, we can formally define the set of types for *C-IL*.

Definition 5.2 (Set of Types) We define the *set of C-IL types* $\mathbb{T}$ inductively as follows:

- Primitive types: $t \in \theta.\mathbb{T}_P \Rightarrow t \in \mathbb{T}$

  A primitive type $t \in \mathbb{T}$ is simply one of the primitive types offered by the compiler – which are specified by the environment parameter $\theta.\mathbb{T}_P$.

- Pointer types: $t \in \mathbb{T} \Rightarrow \mathbf{ptr}(t) \in \mathbb{T}$

  We distinguish between regular pointer types and function pointer types in *C-IL*. A regular pointer type $\mathbf{ptr}(t) \in \mathbb{T}$ is characterized by the type t of the value a pointer of this type points to.

- Array types: $t \in \mathbb{T}, n \in \mathbb{N}, n > 0 \Rightarrow \mathbf{array}(t,n) \in \mathbb{T}$

  In *C-IL*, we only consider arrays of fixed size – dynamic arrays can be modeled by simply using pointers. An array type $\mathbf{array}(t, n) \in \mathbb{T}$ is always characterized by the type of elements t and the number of elements n of the array.

- Function pointers: $t \in \mathbb{T}, T \in (\mathbb{T} \setminus \{\mathbf{void}\})^* \Rightarrow \mathbf{funptr}(t,T) \in \mathbb{T}$

  A function pointer type describes the signature of the function pointed to: Given the type $\mathbf{funptr}(t, T) \in \mathbb{T}$ t is the type of the return value and T is a list of parameter types of the function pointed to.

- Struct types: $t_C \in \mathbb{T}_C \Rightarrow \mathbf{struct}\ t_C \in \mathbb{T}$

  A struct type $\mathbf{struct}\ t_C \in \mathbb{T}$ is simply identified by its composite type name t_C.

5.4.2 Type Qualifiers & Qualified Types

Type qualifiers can be used to annotate type declaration in order to give additional information about the accessed values to the compiler.

The type qualifier `volatile` specifies that a variable of that type is subject to change by means outside the program. This, for example, includes other threads accessing that variable, memory-mapped I/O (device memory), or an interrupt handler accessing the variable. The consequence is that the compiler optimizes much less aggressively over memory accesses marked as volatile – e.g. it always writes/reads the value to/from memory instead of caching it in a register for faster access. However, in many real-world compilers, the compilation of volatile memory accesses does not conform to this notion of "being safe under changes from outside" – effectively resulting in a common perception that the volatile keyword is actually useless in many of the interesting situations (for an interesting discussion of miscompilation of volatiles, see [ER08]).

In contrast, the type qualifier `const` tells the compiler that the variable in question never changes (and, as a consequence, may never be written). This may enable the compiler to perform more aggressive optimization.

Definition 5.3 (Set of Type Qualifiers) We define the *set of type qualifiers* as

$$\mathbb{Q} = \{\mathbf{volatile}, \mathbf{const}\}$$

Any type can be annotated with any subset of the set of type qualifiers.

Definition 5.4 (Set of Qualified Types) We define the *set of qualified types* $\mathbb{T}_Q$, inductively as the smallest set containing the following:

- Qualified primitive types: $q \subseteq \mathbb{Q}, t \in \theta.\mathbb{T}_P \Rightarrow (q, t) \in \mathbb{T}_Q$

- Qualified pointer types: $q \subseteq \mathbb{Q}, x \in \mathbb{T}_Q \Rightarrow (q, \mathbf{ptr}(x)) \in \mathbb{T}_Q$

- Qualified array types: $q \subseteq \mathbb{Q}, x \in \mathbb{T}_Q, n \in \mathbb{N} \Rightarrow (q, \mathbf{array}(x, n)) \in \mathbb{T}_Q$

- Qualified function pointers: $q \subseteq \mathbb{Q}, x \in \mathbb{T}_Q, X \in \mathbb{T}_Q^* \Rightarrow (q, \mathbf{funptr}(x, X)) \in \mathbb{T}_Q$

- Qualified struct types: $q \subseteq \mathbb{Q}, t_C \in \mathbb{T}_C \Rightarrow (q, \textbf{struct}\ t_C) \in \mathbb{T}_Q$

On the level of *C-IL* semantics, type qualifiers do not affect execution at all. Thus, in order to shorten the definitions, we use unqualified types in the formal definitions wherever possible. In order to properly describe the formal relation between qualified types and unqualified types, we introduce a conversion function that maps qualified types to the corresponding unqualified ones.

Definition 5.5 (Converting from Qualified Type to Unqualified Type) We define the function $qt2t : \mathbb{T}_Q \to \mathbb{T}$ that converts qualified types to unqualified types as

$$qt2t(x) = \begin{cases} t & x = (q, t) \wedge t \in \mathbb{T}_P \\ \textbf{ptr}(qt2t(x')) & x = (q, \textbf{ptr}(x')) \\ \textbf{array}(qt2t(x'), n) & x = (q, \textbf{array}(x', n)) \\ \textbf{funptr}(qt2t(x'), \textbf{map}\ qt2t\ X) & x = (q, \textbf{funptr}(x', X)) \\ \textbf{struct}\ t_C & x = (q, \textbf{struct}\ t_C) \end{cases}$$

Definition 5.6 (Auxiliary Predicates on Types) In order to shorten notation, we define the following predicates on unqualified types to be used in the formal definition of *C-IL*:

- $isptr(t) \equiv (\exists t' : t = \textbf{ptr}(t'))$
- $isarray(t) \equiv (\exists t', n : t = \textbf{array}(t', n))$
- $isfunptr(t) \equiv (\exists t', T : t = \textbf{funptr}(t', T))$

In order to specify memory accesses to our byte-addressable memory, we need to know the sizes of values of given types in bytes.

Definition 5.7 (Sizes of Types) We define the function $size_\theta : \mathbb{T} \to \mathbb{N}$ as follows:

$$size_\theta(t) = \begin{cases} k/8 & t = \textbf{i}k \vee t = \textbf{u}k \\ \theta.size_{ptr} & isptr(t) \vee isfunptr(t) \\ n \cdot size_\theta(t') & t = \textbf{array}(t', n) \\ \theta.size_{comp}(t_C) & t = \textbf{struct}\ t_C \end{cases}$$

A Note on Recursive Composite Types

In order to represent the structure of recursive composite type declarations like, for example,

```
struct A {struct A* x};
```

we assume a *composite type declaration function*

$$T_F : \mathbb{T}_C \to (\mathbb{F} \times \mathbb{T}_Q)^*$$

that maps a given *composite type name* $t_C \in \mathbb{T}_C$ to a list of its field declarations (specifying pairs of fields $f \in \mathbb{F}$ and qualified types $t \in \mathbb{T}_Q$). This function is always provided by the particular *C-IL* program considered (see section 5.6).

Example 2 (Qualified Members of a Struct Type) Let us consider the following struct type declaration:

```
struct A
{
  volatile int a;
  unsigned int b[10];
  const int *c;
  int *volatile d;
};
```

Translating this struct declaration of C to *C-IL* on a 64-bit architecture, the corresponding composite type declaration is:

$$\begin{aligned} T_F(A) \;=\; & [(a, (\{\textbf{volatile}\}, \textbf{i64})),\\ & (b, (\emptyset, \textbf{array}(\textbf{u64}, 10)),\\ & (c, (\emptyset, \textbf{ptr}(\{\textbf{const}\}, \textbf{i64}))),\\ & (d, (\{\textbf{volatile}\}, \textbf{ptr}(\textbf{i64})))] \end{aligned}$$

Note that the pointer c is not constant, however, the value it points to is. On the other hand, d is a volatile pointer to a non-volatile 64-bit integer value.

5.5 Values

Values in *C-IL* semantics are modeled in a quite different way than in many high-level programming language semantics. Instead of choosing an abstract representation for values, we simply model most values as pairs of a bit-string of appropriate length and a type. In the case of primitive values, the type denotes whether the bit-string component of the value has to be interpreted as a binary number or as a two's-complement-number.

Definition 5.8 (Set of C-IL Values) We define the *set of C-IL values val* as the union of the sets of values of a given type:

$$val \overset{def}{=} val_{\textbf{prim}} \cup val_{\textbf{ptr+array}} \cup val_{\textbf{funptr}}$$

In the following, we define and briefly discuss the sets of values of a given type.

Definition 5.9 (Set of Primitive Values) The *set of primitive values* $val_{\textbf{prim}}$ is defined as

$$val_{\textbf{prim}} \overset{def}{=} \bigcup_{t \in \mathbb{T}_{PP}} val_{\textbf{prim}(t)}$$

where

$$val_{\textbf{prim}(\textbf{u}k)} \overset{def}{=} \{\textbf{val}(n, \textbf{u}k) \mid n \in \mathbb{B}^k\}$$

and

$$val_{\textbf{prim}(\textbf{i}k)} \overset{def}{=} \{\textbf{val}(i, \textbf{i}k) \mid i \in \mathbb{B}^k\}$$

define the *unsigned values* and *signed values*, respectively.

Note that there is no value for the type **void** since the type void explicitly denotes the absence of a value. Expressions of type **void** thus cannot be evaluated in *C-IL* (even though some C compilers may effectively treat **void** as just another integer type). Also, struct types themselves do not have corresponding values. Structs in *C-IL* are always accessed through their specific primitive, pointer, or array type fields.

Since array values in C correspond directly to a pointer to the first element of the array, we simply model arrays as pointers with an array type. However, pointers come in two specific flavors: Pointers to the global memory and pointers to local variables (which reside on the stack). The latter ones we call *local reference*s while we call the former ones *pointer values* or *pointers to the global memory* in the following. A particular difference between them is that pointer arithmetics on local references is restricted while it is always possible on pointer values.

Definition 5.10 (Set of Pointer and Array Values) We define the *set of pointer and array values* as the union of the set of *pointer values* and the set of *local reference*s:

$$val_{\mathbf{ptr+array}} \overset{def}{=} val_{\mathbf{ptr}} \cup val_{\mathbf{lref}}$$

where a pointer value from the set

$$val_{\mathbf{ptr}} \overset{def}{=} \bigcup_{t \in \mathbb{T} \wedge (isptr(t) \vee isarray(t))} val_{\mathbf{ptr}(t)}$$

with

$$val_{\mathbf{ptr}(t)} \overset{def}{=} \{\mathbf{val}(a,t) \mid a \in \mathbb{B}^{8 \cdot \theta.size_{ptr}}\}$$

is a pair $\mathbf{val}(a,t)$ of a bit-string a representing the address in memory the pointer points to and a type t which is either a pointer or an array type, and where a local reference from the set

$$val_{\mathbf{lref}} \overset{def}{=} \bigcup_{t \in \mathbb{T} \wedge (isptr(t) \vee isarray(t))} val_{\mathbf{lref}(t)}$$

with

$$val_{\mathbf{lref}(t)} \overset{def}{=} \{\mathbf{lref}((v,o),i,t) \mid v \in \mathbb{V} \wedge o \in \mathbb{N} \wedge i \in \mathbb{N}\}$$

is a tuple $\mathbf{lref}((v,o),i,t)$ containing the name of the local variable v, a byte-offset o in the byte-representation of that variable, a stack frame number i that denotes the number of the stack frame the local variable can be found in, and a type t which is either a pointer or an array type.

In our semantics, we consider two different kinds of function pointer values: Those for which the function pointer address is given by the environment parameters and those, where it is not given. In the first case, we specify an actual function pointer value whereas in the second case, we simply introduce a symbolic value that represents the function pointer. The symbolic value can only be used to perform function calls – in particular, it can never be written to memory.

Definition 5.11 (Set of Function Pointer Values) The *set of function pointer values* $val_{\mathbf{funptr}}$ is defined as

$$val_{\mathbf{funptr}} \overset{def}{=} val_{\mathbf{fptr}} \cup val_{\mathbf{fun}}$$

where a function pointer from the set

$$val_{\mathbf{fptr}} \overset{def}{=} \bigcup_{t \in \mathbb{T} \wedge isfunptr(t)} val_{\mathbf{ptr}(t)}$$

is a pair $\mathbf{val}(a, t)$ of a bit-string a representing a memory address where the compiled code of the function starts and t is a function pointer type, and a symbolic function value from the set

$$val_{\mathbf{fun}} \overset{def}{=} \{\mathbf{fun}(f, t) \mid f \in \mathbb{F}_{name} \wedge isfunptr(t)\}$$

is characterized by a function name f and the corresponding function pointer type.

Operational semantics of *C-IL* function call is later defined both for explicit function pointer values and for symbolic function values. We provide both definitions in order to account for programs where we need to argue about function pointer addresses and for programs where we do not actually need to store function pointers in memory.

Definition 5.12 (Zero-Predicate for Values) We define a partial function

$$zero : params_{C\text{-}IL} \times val \rightharpoonup bool$$

that expresses whether a value is considered to be zero:

$$zero(\theta,x) \equiv \begin{cases} a = 0^{8 \cdot size_\theta(t)} & x = \mathbf{val}(a, t) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Definition 5.13 (Type of a Value) We define the function

$$\tau(\cdot) : val \to \mathbb{T}$$

that extracts the type information from a value as

$$\tau(x) \overset{def}{=} \begin{cases} t & x = \mathbf{val}(y, t) \\ t & x = \mathbf{fun}(f, t) \\ t & x = \mathbf{lref}((v, o), i, t) \end{cases}$$

5.5.1 Performing Type Cast on Values

The C99 standard requires certain properties to hold for the type cast function. In the following, we consider those relevant to *C-IL*.

Type Casting Integers

"6.3.1.3 Signed and unsigned integers

When a value with integer type is converted to another integer type other than `_Bool`, if the value can be represented by the new type, it is unchanged. Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type. Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised."

"The C99 Standard"[ISO99]

This gives us the following properties of integer type casting:

- The value can be represented by the new type:

  $cast(\mathbf{val}(b, \mathbf{i}i), \mathbf{i}j) = \mathbf{val}(b', \mathbf{i}j) \rightarrow [b]_i \in \{-2^{j-1}, \ldots, 2^{j-1} - 1\} \rightarrow [b]_i = [b']_j$

  $cast(\mathbf{val}(b, \mathbf{i}i), \mathbf{u}j) = \mathbf{val}(b', \mathbf{u}j) \rightarrow [b]_i \in \{0, \ldots, 2^j - 1\} \rightarrow [b]_i = \langle b' \rangle_j$

  $cast(\mathbf{val}(b, \mathbf{u}i), \mathbf{i}j) = \mathbf{val}(b', \mathbf{i}j) \rightarrow \langle b \rangle_i \in \{-2^{j-1}, \ldots, 2^{j-1} - 1\} \rightarrow \langle b \rangle_i = [b']_j$

  $cast(\mathbf{val}(b, \mathbf{u}i), \mathbf{u}j) = \mathbf{val}(b', \mathbf{u}j) \rightarrow \langle b \rangle_i \in \{0, \ldots, 2^j - 1\} \rightarrow \langle b \rangle_i = \langle b' \rangle_j$

- The value cannot be represented by the new unsigned integer type:

  $cast(\mathbf{val}(b, \mathbf{i}i), \mathbf{u}j) = \mathbf{val}(b', \mathbf{u}j) \rightarrow [b]_i \notin \{0, \ldots, 2^j - 1\} \rightarrow [b]_i \equiv_{\mathbf{mod}\ 2^j} \langle b' \rangle_j$

  $cast(\mathbf{val}(b, \mathbf{u}i), \mathbf{u}j) = \mathbf{val}(b', \mathbf{u}j) \rightarrow \langle b \rangle_i \notin \{0, \ldots, 2^j - 1\} \rightarrow \langle b \rangle_i \equiv_{\mathbf{mod}\ 2^j} \langle b' \rangle_j$

- The value cannot be represented by the new signed integer type:

  Implementation may define this in any way or may raise an exception.

Type Casting Pointers

"6.3.2.3 Pointers

A pointer to void may be converted to or from a pointer to any incomplete or object type. A pointer to any incomplete or object type may be converted to a pointer to void and back again; the result shall compare equal to the original pointer.

An integer may be converted to any pointer type. Except as previously specified, the result is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, and might be a trap representation.

Any pointer type may be converted to an integer type. Except as previously specified, the result is implementation-defined. If the result cannot be represented in the integer type, the behavior is undefined. The result need not be in the range of values of any integer type.

A pointer to a function of one type may be converted to a pointer to a function of another type and back again; the result shall compare equal to the original pointer.

If a converted pointer is used to call a function whose type is not compatible with the pointed-to type, the behavior is undefined."

"The C99 Standard"[ISO99]

Applied to *C-IL*, this can be expressed formally as follows:

- A pointer to a struct type can be cast to a void pointer and back again, resulting in the original pointer value:

  $cast(cast(\mathbf{val}(a, \mathbf{ptr}(\mathbf{struct}\ t_C)), \mathbf{ptr}(\mathbf{void})), \mathbf{ptr}(\mathbf{struct}\ t_C)) = \mathbf{val}(a, \mathbf{ptr}(\mathbf{struct}\ t_C))$

- Type casting integers to pointer types is implementation-defined.

- Type casting pointers to integers is implementation-defined.

- Casting a function pointer type to a different function pointer type and back again results in the original pointer value:

  $cast(cast(\mathbf{val}(a, \mathbf{funptr}(t, T)), \mathbf{funptr}(t', T')), \mathbf{funptr}(t, T)) = \mathbf{val}(a, \mathbf{funptr}(t, T))$

Note that, in practice, it is useful to define the type cast function implemented by the compiler in question explicitly at the level of detail needed. Working just with the requirements imposed by the C99 standard is often not feasible.

Example 3 (Type Cast Function for MIPS-86-*C-IL*) A definition of the type-cast function

$$\theta.cast : val \times \mathbb{T}_Q \rightharpoonup val$$

for MIPS-86-*C-IL* is given as

$$\theta.cast(x,t) \overset{def}{=} \begin{cases} \mathbf{val}(a,t) & x = \mathbf{val}(a,t') \wedge size_\theta(t') = size_\theta(t) \\ \mathbf{val}(sxt_k(i),t) & x = \mathbf{val}(i,\mathbf{i}j) \wedge t = \mathbf{i}k \wedge k > j \\ \mathbf{val}(i[k-1:0],t) & x = \mathbf{val}(i,\mathbf{i}j) \wedge t = \mathbf{i}k \wedge k < j \\ \mathbf{val}(0^{k-j} \circ n,t) & x = \mathbf{val}(n,\mathbf{u}j) \wedge t = \mathbf{u}k \wedge k > j \\ \mathbf{val}(n[k-1:0],t) & x = \mathbf{val}(n,\mathbf{u}j) \wedge t = \mathbf{u}k \wedge k < j \\ \mathbf{lref}((v,o),i,t) & x = \mathbf{lref}((v,o),i,t') \wedge (isptr(t) \vee isarray(t)) \\ \text{undefined} & \text{otherwise} \end{cases}$$

5.5.2 Operators

C-IL provides the following operators to occur in expressions:

- unary operators $\mathbb{O}_1 = \{-, \sim, !\}$

- binary operators $\mathbb{O}_2 = \{+, -, *, /, \%, <<, >>, <, >, <=, >=, ==, !=, \&, |, \hat{}, \&\&, ||\}$

Some more operators of C are provided (e.g. pointer-dereferencing, address-of, size-of). However, even though they are called operators, they are not mathematical operators in a classical sense. These language constructs are handled explicitly by providing them as *C-IL* expressions.

One of the simplifications over C that we make in *C-IL* is that we assume that the C program is compiled to type-correct *C-IL*, i.e. all necessary type casts for promotion of types are made explicit by using the corresponding type cast expression provided by *C-IL*. One exception we make is that we model array accesses as pointer addition with an integer value: When an integer typed value is added to a pointer, the added value is multiplied by the size of the type pointed to, respectively, the size of the type of array elements. Since we assume array access to be modeled as adding an integer type right operand to a pointer type left operand, we define just this kind of pointer addition in the definition of operators for *MIPS-86-C-IL*. Another exception is that we only allow shift amounts specified as unsigned integer values (since the effect of shifting by a negative amount is undefined).

For different compilers and architectures, operators may behave differently. We will not consider the restrictions of the C99 standard in the following. However, if one wants to implement a C99 conforming C on top of *C-IL*, it is necessary to do so. Let us instead give a brief definition of the individual operators for *MIPS-86-C-IL*:

- unary minus: $\theta.op_1(-)(v) = \begin{cases} \mathbf{val}(0_k -_k a, t) & v = \mathbf{val}(a,t) \wedge (t = \mathbf{i}k \vee t = \mathbf{u}k) \\ \text{undefined} & \text{otherwise} \end{cases}$

- bitwise negation: $\theta.op_1(\sim)(v) = \begin{cases} \mathbf{val}(\overline{a}, t) & v = \mathbf{val}(a,t) \\ \text{undefined} & \text{otherwise} \end{cases}$

- logical negation: $\theta.op_1(!)(v) = \begin{cases} \mathbf{val}(1_k, t) & v = \mathbf{val}(a,t) \wedge a = 0_k \wedge (t = \mathbf{i}k \vee t = \mathbf{u}k) \\ \mathbf{val}(0_k, t) & v = \mathbf{val}(a,t) \wedge a \neq 0_k \wedge (t = \mathbf{i}k \vee t = \mathbf{u}k) \\ \text{undefined} & \text{otherwise} \end{cases}$

- addition:

$$\theta.op_2(+)(v_1, v_2) = \begin{cases} \mathbf{val}(a_1 +_{8 \cdot size_\theta(t)} a_2, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \\ \mathbf{val}(a_1 +_{8 \cdot \theta.size_{ptr}} x, \mathbf{ptr}(t')) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, \mathbf{i}k) \\ & \wedge (t = \mathbf{ptr}(t') \vee t = \mathbf{array}(t', n)) \\ w & v_1 = \mathbf{lref}((v, o), i, t) \wedge v_2 = \mathbf{val}(a, \mathbf{i}k) \\ & \wedge (t = \mathbf{ptr}(t') \vee t = \mathbf{array}(t', n)) \\ & \wedge o + [a] \cdot size_\theta(t') \geq 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $x = ([a_2] \cdot size_\theta(t'))_{8 \cdot \theta.size_{ptr}}$ and $w = \mathbf{lref}((v, o + [a] \cdot size_\theta(t')), i, \mathbf{ptr}(t'))$. Note that, by performing pointer arithmetics on local references, we can obtain local references which are no longer meaningful since the offset o exceeds the boundary imposed by the size of the type of variable v. In such a case, *C-IL* semantics will get stuck when trying to read or write from/to such an illegal local reference. In practice, this problem can be avoided by enforcing a sufficiently strict programming discipline on the considered *C-IL* program.

- subtraction: $\theta.op_2(-)(v_1, v_2) = \begin{cases} \mathbf{val}(a_1 -_k a_2, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \\ \text{undefined} & \text{otherwise} \end{cases}$

- multiplication: $\theta.op_2(*)(v_1, v_2) = \begin{cases} \mathbf{val}(a_1 \cdot_k a_2, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \wedge t = \mathbf{u}k \\ \mathbf{val}(a_1 \cdot_{\mathrm{t}k} a_2, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \wedge t = \mathbf{i}k \\ \text{undefined} & \text{otherwise} \end{cases}$

- division: $\theta.op_2(/)(v_1, v_2) = \begin{cases} \mathbf{val}(a_1 \div_k a_2, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \wedge t = \mathbf{u}k \\ \mathbf{val}(a_1 \div_{\mathrm{t}k} a_2, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \wedge t = \mathbf{i}k \\ \text{undefined} & \text{otherwise} \end{cases}$

- modulo:

$$\theta.op_2(\%)(v_1, v_2) = \begin{cases} \mathbf{val}(a_1 -_k (a_1 \div_k a_2) \cdot_k a_2, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \wedge t = \mathbf{u}k \\ \mathbf{val}(a_1 -_k (a_1 \div_{\mathrm{t}k} a_2) \cdot_{\mathrm{t}k} a_2, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \wedge t = \mathbf{i}k \\ \text{undefined} & \text{otherwise} \end{cases}$$

- shift left:

$$\theta.op_2(<<)(v_1, v_2) = \begin{cases} \mathbf{val}(x, \mathbf{u}k) & v_1 = \mathbf{val}(a_1, \mathbf{u}k) \wedge v_2 = \mathbf{val}(a_2, \mathbf{u}l) \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $x = a_1[k - 1 - (\langle a_2 \rangle \,\mathbf{mod}\, k) : 0] \circ 0^{(\langle a_2 \rangle \,\mathbf{mod}\, k)}$

- shift right:

$$\theta.op_2(>>)(v_1, v_2) = \begin{cases} \mathbf{val}(x, \mathbf{u}k) & v_1 = \mathbf{val}(a_1, \mathbf{u}k) \wedge v_2 = \mathbf{val}(a_2, \mathbf{u}l) \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $x = 0^{(\langle a_2 \rangle \,\mathbf{mod}\, k)} \circ a_1[k - 1 : (\langle a_2 \rangle \,\mathbf{mod}\, k)]$

- less than:

$$\theta.op_2(<)(v_1, v_2) = \begin{cases} \mathbf{val}(1_k, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \\ & \wedge\, (t = \mathbf{u}k \wedge \langle a_1 \rangle < \langle a_2 \rangle \vee t = \mathbf{i}k \wedge [a_1] < [a_2]) \\ \mathbf{val}(0_k, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \\ & \wedge\, (t = \mathbf{u}k \wedge \langle a_1 \rangle \geq \langle a_2 \rangle \vee t = \mathbf{i}k \wedge [a_1] \geq [a_2]) \\ \text{undefined} & \text{otherwise} \end{cases}$$

- greater than:

$$\theta.op_2(>)(v_1, v_2) = \begin{cases} \mathbf{val}(1_k, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \\ & \wedge\, (t = \mathbf{u}k \wedge \langle a_1 \rangle > \langle a_2 \rangle \vee t = \mathbf{i}k \wedge [a_1] > [a_2]) \\ \mathbf{val}(0_k, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \\ & \wedge\, (t = \mathbf{u}k \wedge \langle a_1 \rangle \leq \langle a_2 \rangle \vee t = \mathbf{i}k \wedge [a_1] \leq [a_2]) \\ \text{undefined} & \text{otherwise} \end{cases}$$

- less than or equal:

$$\theta.op_2(<=)(v_1, v_2) = \begin{cases} \mathbf{val}(1_k, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \\ & \wedge\, (t = \mathbf{u}k \wedge \langle a_1 \rangle \leq \langle a_2 \rangle \vee t = \mathbf{i}k \wedge [a_1] \leq [a_2]) \\ \mathbf{val}(0_k, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \\ & \wedge\, (t = \mathbf{u}k \wedge \langle a_1 \rangle > \langle a_2 \rangle \vee t = \mathbf{i}k \wedge [a_1] > [a_2]) \\ \text{undefined} & \text{otherwise} \end{cases}$$

- greater than or equal:

$$\theta.op_2(>=)(v_1, v_2) = \begin{cases} \mathbf{val}(1_k, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \\ & \wedge\, (t = \mathbf{u}k \wedge \langle a_1 \rangle \geq \langle a_2 \rangle \vee t = \mathbf{i}k \wedge [a_1] \geq [a_2]) \\ \mathbf{val}(0_k, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \\ & \wedge\, (t = \mathbf{u}k \wedge \langle a_1 \rangle < \langle a_2 \rangle \vee t = \mathbf{i}k \wedge [a_1] < [a_2]) \\ \text{undefined} & \text{otherwise} \end{cases}$$

- equal:

$$\theta.op_2(==)(v_1, v_2) = \begin{cases} \mathbf{val}(1_k, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \wedge a_1 = a_2 \\ \mathbf{val}(0_k, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \wedge a_1 \neq a_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

- not equal:

$$\theta.op_2(!=)(v_1, v_2) = \begin{cases} \mathbf{val}(1_k, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \wedge a_1 \neq a_2 \\ \mathbf{val}(0_k, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \wedge a_1 = a_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

- logical AND:

$$\theta.op_2(\&\&)(v_1, v_2) = \begin{cases} \mathbf{val}(1_k, \tau(v_1)) & \tau(v_1) = \tau(v_2) \wedge \neg zero(\theta, v_1) \wedge \neg zero(\theta, v_2) \\ \mathbf{val}(0_k, \tau(v_1)) & \tau(v_1) = \tau(v_2) \wedge (zero(\theta, v_1) \vee zero(\theta, v_2)) \\ \text{undefined} & \text{otherwise} \end{cases}$$

- logical OR:

$$\theta.op_2(||)(v_1, v_2) = \begin{cases} \mathbf{val}(1_k, \tau(v_1)) & \tau(v_1) = \tau(v_2) \wedge (\neg zero(\theta, v_1) \vee \neg zero(\theta, v_2)) \\ \mathbf{val}(0_k, \tau(v_1)) & \tau(v_1) = \tau(v_2) \wedge zero(\theta, v_1) \wedge zero(\theta, v_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

- bitwise AND: $\theta.op_2(\&)(v_1, v_2) = \begin{cases} \mathbf{val}(a_1 \wedge a_2, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \\ \text{undefined} & \text{otherwise} \end{cases}$

- bitwise OR: $\theta.op_2(|)(v_1, v_2) = \begin{cases} \mathbf{val}(a_1 \vee a_2, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \\ \text{undefined} & \text{otherwise} \end{cases}$

- bitwise XOR: $\theta.op_2(\hat{})(v_1, v_2) = \begin{cases} \mathbf{val}(a_1 \oplus a_2, t) & v_1 = \mathbf{val}(a_1, t) \wedge v_2 = \mathbf{val}(a_2, t) \\ \text{undefined} & \text{otherwise} \end{cases}$

5.6 Programs

In order to define *C-IL* programs, we first define expressions and statements that can occur in *C-IL* programs.

5.6.1 Expressions

C-IL offers a decent subset of the expressions provided by C. However, some C expressions are treated as statements by *C-IL*, namely, function call and assignment.

Definition 5.14 (The Set of *C-IL* Expressions) We inductively define the *set of* C-IL *expressions* $\mathbb{E}$ as the smallest set that obeys the following rules:

- Constants: $c \in val \Rightarrow c \in \mathbb{E}$

  One base case of *C-IL* expressions are constants c given as *C-IL* value.

- Variable names: $v \in \mathbb{V} \Rightarrow v \in \mathbb{E}$

  Another basic expression is given by variable names v. Later, during expression evaluation, we will see how local variables occlude global variables when they have the same variable name.

- Function names: $f \in \mathbb{F}_{name} \Rightarrow f \in \mathbb{E}$

  The last base case of expressions is given by function names f. These are needed in order to call functions or to store function pointer values (for which a value is defined in the environment parameters) to memory.

- Unary operation on expression: $e \in \mathbb{E} \wedge \ominus \in \mathbb{O}_1 \Rightarrow \ominus e \in \mathbb{E}$

  Applying an unary operators to an expression results in an expression.

- Binary operation on expressions: $e_0, e_1 \in \mathbb{E} \wedge \oplus \in \mathbb{O}_2 \Rightarrow (e_0 \oplus e_1) \in \mathbb{E}$

  Application of binary operators to expressions is an expression.

- Conditional: $e, e_0, e_1 \in \mathbb{E} \Rightarrow (e\ ?\ e_0 : e_1) \in \mathbb{E}$

  The ternary (or conditional) operator is an expression based on sub-expressions e, e_0, and e_1. When expression e evaluates to a non-zero value, the resulting value is e_0, otherwise it is e_1.

- Type cast: $t \in \mathbb{T}_Q, e \in \mathbb{E} \Rightarrow (t)e \in \mathbb{E}$

  This expression describes type-casting expression e to type t.

- Dereferencing pointer: $e \in \mathbb{E} \Rightarrow *(e) \in \mathbb{E}$

  Dereferencing a pointer, the value pointed to is retrieved from the memory.

- Address of expression: $e \in \mathbb{E} \Rightarrow \&(e) \in \mathbb{E}$

  If applicable, evaluating this expression returns the pointer to the subvariable described by expression e. Note that taking address-of of an expression is not always meaningful: We can only take address of variables or subvariables, never of constants or values.

- Field access: $e \in \mathbb{E} \wedge f \in \mathbb{F} \Rightarrow (e).f \in \mathbb{E}$

  This expression describes accessing field f in the struct value described by expression e.

- Size of Type: $t \in \mathbb{T}_Q \Rightarrow \textbf{sizeof}(t) \in \mathbb{E}$

  Evaluating this expression returns the size in bytes of type t.

- Size of Expression: $e \in \mathbb{E} \Rightarrow \textbf{sizeof}(e) \in \mathbb{E}$

  Evaluating this expression returns the size in bytes of the type of expression e.

Note, that array access (e.g. `a[k]`) and pointer field access (e.g. `a->b`) are not expressions of *C-IL* since they can easily be translated as follows:

- `a[k]` $\mapsto$ `*((a + k))`

  In C, array access is essentially just a shorthand for adding the index to the array pointer.

- `a->b` $\mapsto$ `(*(a)).b`

  Accessing a field from a pointer is the same as performing a field access on the dereferences pointer value.

5.6.2 Statements

Since expressions are always evaluated in the context of a configuration of a *C-IL* program and under given *C-IL* environment parameters, the definition of expression evaluation is postponed until section 5.8 which deals with operational semantics of *C-IL*.

Definition 5.15 (The Set of *C-IL* Statements) We define the *set of* C-IL *statements* $\mathbb{S}$ inductively as follows:

- Assignment: $e_0, e_1 \in \mathbb{E} \Rightarrow (e_0{=}e_1) \in \mathbb{S}$

  The value of expression e_1 is written to the address of expression e_0.

- Goto: $l \in \mathbb{N} \Rightarrow$ **goto** $l \in \mathbb{S}$

  Control flow continues with the statement at label l.

- If-Not-Goto: $e \in \mathbb{E} \ \wedge \ l \in \mathbb{N} \Rightarrow$ **ifnot** e **goto** $l \in \mathbb{S}$

  If expression e evaluates non-zero, program execution continues from label l.

- Function Call: $e_0, e \in \mathbb{E}, E \in \mathbb{E}^* \Rightarrow (e_0{=}\textbf{call}\ e(E)) \in \mathbb{S}$

  If expression e evaluates to a function pointer value or a symbolic function value, a function call to the corresponding function is performed with parameters given by evaluating the parameter expression list E. The return value of the function is stored at the address of expression e_0.

- Procedure Call: $e \in \mathbb{E}, E \in \mathbb{E}^* \Rightarrow$ **call** $e(E) \in \mathbb{S}$

  A function call statement that has no return value.

- Return: $e \in \mathbb{E} \Rightarrow$ **return** $e \in \mathbb{S}$ and **return** $\in \mathbb{S}$

  Return from the current function with return value given by expression e. Functions with return type **void** use the variant without return expression e since they do not pass a return value.

With statements and expressions, we can finally define *C-IL* programs.

Definition 5.16 (*C-IL* Program) A *C-IL*-program

$$\pi = (\pi.\mathcal{V}_G, \pi.T_F, \pi.\mathcal{F}) \in prog_{C\text{-}IL}$$

is described by the following:

- $\pi.\mathcal{V}_G \in (\mathbb{V} \times \mathbb{T}_Q)^*$ – a list of global variable declarations

  This list declares all global variables of the *C-IL* program as pairs of variable name and qualified type.

- $\pi.T_F : \mathbb{T}_C \rightharpoonup (\mathbb{F} \times \mathbb{T}_Q)^*$ – a type table for struct types

  Given a struct name that is declared in the *C-IL* program, this function returns a field type declaration list, i.e. a list of pairs of field name and qualified type.

- $\pi..\mathcal{F} : \mathbb{F}_{name} \rightharpoonup FunT$ – a function table

  The function table maps all function names declared in the *C-IL* program to function table entries from the set $FunT$ – which we define now.

Definition 5.17 (*C-IL* Function Table Entry) A function table entry

$$fte = (fte.rettype, fte.npar, fte.\mathcal{V}, fte.\mathcal{P}) \in FunT$$

consists of the following:

- $fte.rettype \in \mathbb{T}_Q$ – type of the return value

  The type of the value returned by the function. Note that this type can be the `void` type that denotes the absence of a return value.

- $fte.npar \in \mathbb{N}$ – number of parameters

  How many parameters have to be passed to the function in a function call.

- $fte.\mathcal{V} : (\mathbb{V} \times \mathbb{T}_Q)^*$ – parameter and local variable declarations

  This list of pairs of variable name and qualified type declares the parameters and local variables of the function – the first $fte.npar$ entries describe the declared parameters.

- $fte.\mathcal{P} \in (\mathbb{S}^* \cup \{\textbf{extern}\})$ – function body

  This component describes either the program code of the function as a list of *C-IL* statements or expresses that the function in question is declared as an external function in the *C-IL* program.

5.7 Configurations

In order to define the semantics of *C-IL* programs, we proceed by defining the state of the abstract *C-IL* machine on which we state operational semantics in section 5.8.

Definition 5.18 (*C-IL* Stack Frame) A *C-IL* stack frame

$$s = (s.\mathcal{M}_\mathcal{E}, s.rds, s.f, s.loc) \in frame_{C\text{-}IL}$$

consists of

- $s.\mathcal{M}_\mathcal{E} : \mathbb{V} \rightharpoonup (\mathbb{B}^8)^*$ – local variables and parameters memory

  We formalize the memory for local variables and parameters that is part of a *C-IL* stack frame as a function that maps a declared variable name to a byte-string representation of the value of the respective variable. Note that, if t_v is the type of local variable v in stack frame s of a valid *C-IL* configuration that $|s.\mathcal{M}_\mathcal{E}(v)| = size_\theta(t_v)$. Note also that, in contrast to the global memory (which is a mapping of byte-addresses represented by bit-strings to bytes), we have lists of bytes here to represent the individual local variables' local memories.

- $s.rds \in val_{\textbf{ptr}} \cup val_{\textbf{lref}} \cup \{\bot\}$ – return value destination

  The return value destination rds describes where the return value of a function call has to be stored by the called function when it returns. It is a pointer, a local reference, or the value $\bot$ that denotes the absence of a return value destination.

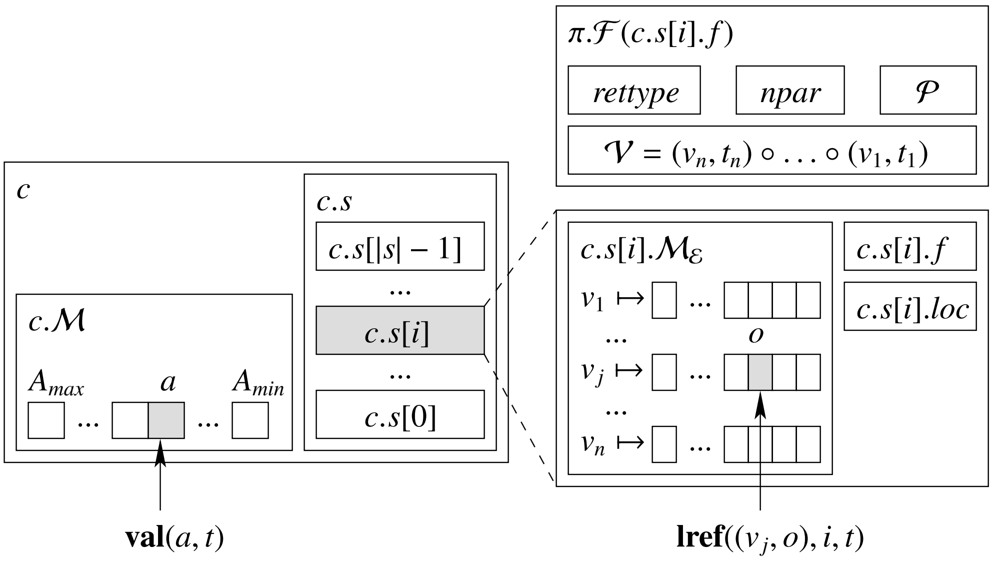


Figure 5.1: Where *C-IL* pointer $\mathbf{val}(a,t)$ and local reference $\mathbf{lref}((v_j,o),i,t)$ point to. Here, $A_{max} = 1^{\theta.size_{ptr}}$ and $A_{min} = 0^{\theta.size_{ptr}}$ are the maximal and, respectively, minimal address of the byte-addressable memory $c.\mathcal{M}$ and v_j is the j-th declared variable name of the function associated with stack frame i.

- $s.f \in \mathbb{F}_{name}$ – function name

  This is the name of the function the stack frame is associated with. We use this function name and the function table of the *C-IL* program to obtain the types of declared local variables and parameters, as well as the return value type and the function body.

- $s.loc \in \mathbb{N}$ – location counter

  The location counter loc is the index of the next statement to be executed in the function body of $s.f$.

Definition 5.19 (Sequential *C-IL*-Configuration) A *C-IL*-configuration

$$c = (c.\mathcal{M}, c.s) \in conf_{C\text{-}IL}$$

consists of a byte-addressable memory $c.\mathcal{M} : \mathbb{B}^{8 \cdot size_{ptr}} \to \mathbb{B}^8$ and a *C-IL* stack $c.s \in frame^*_{C\text{-}IL}$. A *C-IL* stack is simply a list of *C-IL* stack frames (see next definition).

5.7.1 Memory

Overall, *C-IL* operates on flat byte-addressable memories. On the one hand, we have the global memory, modeled as a mapping from addresses to bytes – similar to a hardware memory configuration. On the other hand, we have local memories in the stack frames of *C-IL*, which are modeled as mappings from variable names to byte-strings; the stack layout of the compiler is abstracted away, but the byte-string representation of individual variables is still exposed. To get an idea which memory locations *C-IL* pointers and local references point to, consider figure 5.1.

In some regards, the local memories of *C-IL* are similar to the memory model used in *Cminor* [BL09]: we provide memory blocks that model the byte-string representation of variables. Inside these memory blocks, pointer arithmetics is possible but we cannot argue about the addresses of local variables in relation to each other. References to subvariables of local variables are specified by giving an offset o (in our case modeled by a natural number) in a local variable v.

The global memory of *C-IL* is compatible with the memory model sketched for VCC in [CMTS09]. The difference is that on the level of *C-IL* semantics, we do not consider or enforce typedness of memory regions. Extension to the model sketched in [CMTS09] by overlaying a model of typed objects on the byte-addressable memory is trivial. In our global memory, the addresses of subvariables of global memory pointers are always computed explicitly according to the memory layout of the given type.

Byte-String Representation of Values

In order to access byte-addressable memories to read and write *C-IL* values, it is helpful to define functions that convert values between byte-string representation and *C-IL* representation. Depending on the endianness of the architecture, the order of bytes in the memory is considered from most-significant to least-significant byte or vice versa. This is reflected in the functions that convert between *C-IL*-values and byte-strings.

Definition 5.20 (Converting Value to Byte-String) We define the function

$$val2bytes_\theta : val \rightharpoonup (\mathbb{B}^8)^*$$

as

$$val2bytes_\theta(v) = \begin{cases} bits2bytes(b) & v = \mathbf{val}(b,t) \wedge \theta.endianness = \mathbf{little} \\ bits2bytes(\mathbf{rev}(b)) & v = \mathbf{val}(b,t) \wedge \theta.endianness = \mathbf{big} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Here, *bits2bytes* converts from bit-strings to byte-strings in the obvious way.

Definition 5.21 (Converting Byte-String to Value) The function

$$bytes2val_\theta : (\mathbb{B}^8)^* \times \mathbb{T} \rightharpoonup val$$

is given as

$$bytes2val_\theta(B,t) = \begin{cases} \mathbf{val}(bytes2bits(B), t) & t \neq \mathbf{struct}\ t_C \wedge \theta.endianness = \mathbf{little} \\ \mathbf{val}(bytes2bits(\mathbf{rev}(B)), t) & t \neq \mathbf{struct}\ t_C \wedge \theta.endianness = \mathbf{big} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Here, *bytes2bits* converts from byte-strings to bit-strings.

Reading and Writing Byte-Strings from/to Memory

We define functions that read and write byte-strings from/to the memory components of a *C-IL* configuration in order to express memory semantics of *C-IL*.

Definition 5.22 (Reading Byte-Strings from Global Memory) The function

$$read_{\mathcal{M}} : (\mathbb{B}^{8\cdot\theta.size_{ptr}} \to \mathbb{B}^8) \times \mathbb{B}^{8\cdot\theta.size_{ptr}} \times \mathbb{N} \to (\mathbb{B}^8)^*$$

given as

$$read_{\mathcal{M}}(\mathcal{M},a,s) = \begin{cases} read_{\mathcal{M}}(\mathcal{M}, a +_{8\cdot\theta.size_{ptr}} 1_{8\cdot\theta.size_{ptr}}, s-1) \circ \mathcal{M}(a) & s > 0 \\ \varepsilon & s = 0 \end{cases}$$

reads a byte-string of length s from a global memory $\mathcal{M}$ starting at address a.

Definition 5.23 (Writing Byte-Strings to Global Memory) We define the function

$$write_{\mathcal{M}} : (\mathbb{B}^{8\cdot\theta.size_{ptr}} \to \mathbb{B}^8) \times \mathbb{B}^{8\cdot\theta.size_{ptr}} \times (\mathbb{B}^8)^* \to (\mathbb{B}^{8\cdot\theta.size_{ptr}} \to \mathbb{B}^8)$$

that writes a byte-string B to a global memory $\mathcal{M}$ starting at address a such that

$$\forall x \in \mathbb{B}^{8\cdot\theta.size_{ptr}} : write_{\mathcal{M}}(\mathcal{M},a,B)(x) = \begin{cases} \mathcal{M}(x) & \langle x\rangle - \langle a\rangle \notin \{0,\dots,|B|-1\} \\ B[\langle x\rangle - \langle a\rangle] & \text{otherwise} \end{cases}$$

Definition 5.24 (Reading Byte-Strings from Local Memory) The function

$$read_{\mathcal{E}} : (\mathbb{V} \rightharpoonup (\mathbb{B}^8)^*) \times \mathbb{V} \times \mathbb{N} \times \mathbb{N} \rightharpoonup (\mathbb{B}^8)^*$$

with

$$read_{\mathcal{E}}(\mathcal{M}_{\mathcal{E}},v,o,s) = \mathcal{M}_{\mathcal{E}}(v)[o+s-1] \circ \ldots \circ \mathcal{M}_{\mathcal{E}}(v)[o]$$

reads a byte-string of length s from local memory $\mathcal{M}_{\mathcal{E}}$ for a local variable v starting at offset o. If $s + o > |\mathcal{M}_{\mathcal{E}}(v)|$ or $v \notin dom(\mathcal{M}_{\mathcal{E}})$, the function is undefined for the given parameters.

Definition 5.25 (Writing Byte-Strings to Local Memory) We define the function

$$write_{\mathcal{E}} : (\mathbb{V} \rightharpoonup (\mathbb{B}^8)^*) \times \mathbb{V} \times \mathbb{N} \times \mathbb{B}^8 \rightharpoonup (\mathbb{V} \rightharpoonup (\mathbb{B}^8)^*)$$

that writes a byte-string B to variable v of a local memory $\mathcal{M}_{\mathcal{E}}$ starting at offset o such that

$$\forall w \in \mathbb{V} : \forall i < |\mathcal{M}_{\mathcal{E}}(w)| : write_{\mathcal{E}}(\mathcal{M}_{\mathcal{E}},v,o,B)(w)[i] = \begin{cases} \mathcal{M}_{\mathcal{E}}(w)[i] & w \neq v \lor i \notin \{o,\dots,o+|B|-1\} \\ B[i-o] & \text{otherwise} \end{cases}$$

If, however, $|B| + o > |\mathcal{M}_{\mathcal{E}}(v)|$ or $v \notin dom(\mathcal{M}_{\mathcal{E}})$, the function is undefined for the given parameters.

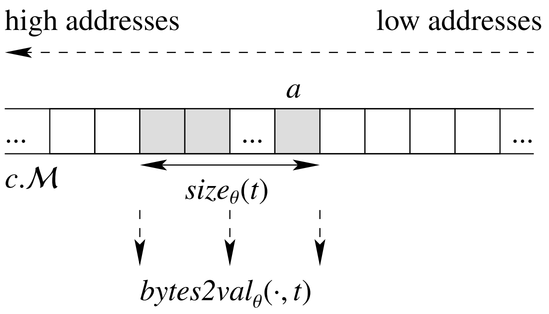


Figure 5.2: Dereferencing a pointer $\mathbf{val}(a, \mathbf{ptr}(t))$ to the global memory, i.e. reading the value pointed to from memory, by reading $size_\theta(t)$ bytes starting at address a of the global memory $c.\mathcal{M}$ of *C-IL* configuration c.

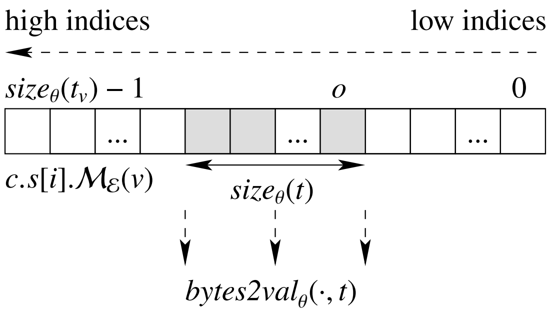


Figure 5.3: Dereferencing a local reference $\mathbf{lref}((v, o), i, \mathbf{ptr}(t))$ by reading $size_\theta(t)$ bytes starting at offset o of local memory $c.s[i].\mathcal{M}_\mathcal{E}(v)$ of local variable v of type t_v in stack frame i of *C-IL* configuration c.

Reading and Writing Memory Values of a Configuration

In order to have a short notation for specifying updates on the *C-IL*-state, we introduce functions that read and write *C-IL* values to/from *C-IL* configurations.

Definition 5.26 (Reading a Value from a *C-IL* Configuration) We define the function

$$read : params_{C\text{-}IL} \times conf_{C_{IL}} \times val \rightharpoonup val$$

as

$$read(\theta,c,x) = \begin{cases} bytes2val_\theta(read_\mathcal{M}(c.\mathcal{M}, a, size_\theta(t)), t) & x = \mathbf{val}(a, \mathbf{ptr}(t)) \\ bytes2val_\theta(read_\mathcal{E}(c.s[i].\mathcal{M}_\mathcal{E}(v), o, size_\theta(t)), t) & x = \mathbf{lref}((v, o), i, \mathbf{ptr}(t)) \\ read(\theta, c, \mathbf{val}(a, \mathbf{ptr}(t))) & x = \mathbf{val}(a, \mathbf{array}(t, n)) \\ read(\theta, c, \mathbf{lref}((v, o), i, \mathbf{ptr}(t))) & x = \mathbf{lref}((v, o), i, \mathbf{array}(t, n)) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Given a record θ of *C-IL* environment parameters, a *C-IL* configuration c and a pointer value x, it returns the *C-IL* value read from the memory pointed to. For an illustration of this definition, see figures 5.2 and 5.3.

Note that in C, reading or writing an array means that the array is promoted to a pointer. Thus, reading or writing of the first element of the array is performed. This is reflected in the read and write functions for *C-IL*.

Definition 5.27 (Writing a Value to a *C-IL* Configuration) We define the function

$$write : params_{C\text{-}IL} \times conf_{C_{IL}} \times val \times val \rightharpoonup conf_{C_{IL}}$$

that writes a given *C-IL* value y to a *C-IL* configuration c at the memory pointed to by pointer x according to environment parameters θ as

$$write(\theta,c,x,y) = \begin{cases} c[\mathcal{M} := write_{\mathcal{M}}(c.\mathcal{M}, val2bytes_{\theta}(x), val2bytes_{\theta}(y))] & x = \mathbf{val}(a, \mathbf{ptr}(t)) \\ & \wedge\, y = \mathbf{val}(b,t) \\ c' & x = \mathbf{lref}((v,o),i,\mathbf{ptr}(t)) \\ & \wedge\, y = \mathbf{val}(b,t) \\ write(\theta, c, \mathbf{val}(a, \mathbf{ptr}(t)), y) & x = \mathbf{val}(a, \mathbf{array}(t,n)) \\ write(\theta, c, \mathbf{lref}((v,o),i,\mathbf{ptr}(t)), y) & x = \mathbf{lref}((v,o),i,\mathbf{array}(t,n)) \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $c'.s[i].\mathcal{M}_{\mathcal{E}} = write_{\mathcal{E}}(\mathcal{M}_{\mathcal{E}top}(c), v, o, val2bytes_{\theta}(y))$ and all other parts of c' are identical to c.

5.8 Operational Semantics

We define operational semantics for the execution of *C-IL* programs. For this, we first give some auxiliary definitions. Then we define how *C-IL* expressions are evaluated to *C-IL* values. Finally, we state the transitions between *C-IL* configurations with respect to a *C-IL* program under given *C-IL* environment parameters.

5.8.1 Auxiliary Definitions

For a shorter notation, we provide the following definitions:

Definition 5.28 (Combined Function Table) For a *C-IL* program π and a collection of *C-IL* environment parameters θ, we define the combined function table

$$\mathcal{F}_{\pi}^{\theta} = \pi.\mathcal{F} \cup \theta.intrinsics$$

Note that the names of functions used in the program must be different from the names of compiler intrinsic functions. This is a property that shall be checked by the compiler.

Definition 5.29 (Set of Declared Variables) For a variable declaration list, we define the function

$$decl : (\mathbb{V} \times \mathbb{T}_Q)^* \to 2^{\mathbb{V}}$$

that returns the set of declared variable names:

$$decl(\mathcal{V}) = \begin{cases} \{v\} \cup decl(\mathcal{V}') & \mathcal{V} = (v,t) \circ \mathcal{V}' \\ \emptyset & \mathcal{V} = \epsilon \end{cases}$$

Definition 5.30 (Type of a Variable in a Declaration List) We define the function

$$\tau_V : \mathbb{V} \times (\mathbb{V} \times \mathbb{T}_Q)^* \rightharpoonup \mathbb{T}_Q$$

that, given a variable and an appropriate declaration list, returns the type of the variable, as

$$\tau_V(v,\mathcal{V}) = \begin{cases} t & \mathcal{V} = (v,t) \circ \mathcal{V}' \\ \tau_V(v,\mathcal{V}') & \mathcal{V} = (v',t) \circ \mathcal{V}' \wedge v' \neq v \\ \text{undefined} & \mathcal{V} = \epsilon \end{cases}$$

Definition 5.31 (Type of a Field in a Declaration List) Analogously, we define

$$\tau_F : \mathbb{F} \times (\mathbb{F} \times \mathbb{T}_Q)^* \rightharpoonup \mathbb{T}_Q$$

which returns the type of a field as given by a field declaration list:

$$\tau_F(f,T) = \begin{cases} t & T = (f,t) \circ T' \\ \tau_F(f,T') & T = (f',t) \circ T' \wedge f' \neq f \\ \text{undefined} & T = \epsilon \end{cases}$$

Definition 5.32 (Type of a Function) For function pointer types, we extract the type information from the function table of the corresponding function. We define the function

$$\tau^{\mathcal{F}}_{fun} : \mathbb{F}_{name} \rightharpoonup \mathbb{T}_Q$$

as

$$\tau^{\mathcal{F}}_{fun}(fn) = (\emptyset, \mathbf{funptr}(\mathcal{F}(fn).rettype, [t_0, \ldots, t_{npar-1}]))$$

where $npar = \mathcal{F}(fn).npar$ and t_i is the type of the i-th declared variable: $\forall i < npar : \exists v_i : \mathcal{F}(fn).\mathcal{V}[i] = (v_i, t_i)$. The result of the function is undefined if and only if $fn \notin dom(\mathcal{F})$.

Definition 5.33 (Variable Declarations of the Top-Most Stack Frame) In evaluation, we will often need to know for a given configuration of a *C-IL*-program which variables are declared for the top-most stack frame. For this, we introduce the following shorthand notation:

$$\mathcal{V}_{top}(\pi,c) = \pi.\mathcal{F}(c.s[|c.s|-1].f).\mathcal{V}$$

Definition 5.34 (Qualified Type Evaluation) To talk about the optimizing compiler, we will need to talk about qualified types that occur in a program. For this, we introduce the qualified type evaluation function

$$\tau_{Q\cdot}^{\cdot,\cdot}(\cdot) : conf_{C\text{-}IL} \times prog_{C\text{-}IL} \times params_{C\text{-}IL} \times \mathbb{E} \to \mathbb{T}_Q$$

that, given a *C-IL* configuration, a *C-IL* program, *C-IL* environment parameters, and a *C-IL* expression, returns the qualified type of that expression in the given configuration of the program. We define the function by a case distinction over the expression whose type is to be evaluated.

- Constant: $x \in val \Rightarrow \tau_{Q_c}^{\pi,\theta}(x) = (\emptyset, \tau(x))$
- Variable Name: $v \in \mathbb{V} \Rightarrow \tau_{Q_c}^{\pi,\theta}(v) = \begin{cases} \tau_V(v, \mathcal{V}_{top}(\pi,c)) & v \in decl(\mathcal{V}_{top}(\pi,c)) \\ \tau_V(v, \pi.\mathcal{V}_G) & v \notin decl(\mathcal{V}_{top}(\pi,c)) \wedge v \in decl(\pi.\mathcal{V}_G) \\ (\emptyset, \mathbf{void}) & \text{otherwise} \end{cases}$

- Function Name: $fn \in \mathbb{F}_{name} \Rightarrow \tau Q_C^{\pi,\theta}(fn) = \tau_{fun}^{\mathcal{F}_\pi^\theta}(fn)$
- Unary Operator: $e \in \mathbb{E}, \ominus \in \mathbb{O}_1 \Rightarrow \tau Q_C^{\pi,\theta}(\ominus e) = \tau Q_C^{\pi,\theta}(e)$
- Binary Operator: $e_0, e_1 \in \mathbb{E}, \oplus \in \mathbb{O}_2 \Rightarrow \tau Q_C^{\pi,\theta}(e_0 \oplus e_1) = \tau Q_C^{\pi,\theta}(e_0)$
- Ternary Operator: $e, e_0, e_1 \in \mathbb{E} \Rightarrow \tau Q_C^{\pi,\theta}((e \mathbin{?} e_0 : e_1)) = \tau Q_C^{\pi,\theta}(e_0)$

  Note that the compiler should ensure that the types of e_0 and e_1 match.
- Type Cast: $t \in \mathbb{T}_Q, e \in \mathbb{E} \Rightarrow \tau Q_C^{\pi,\theta}((t)e) = t$
- Dereferencing a Pointer: $e \in \mathbb{E} \Rightarrow \tau Q_C^{\pi,\theta}(*(e)) = \begin{cases} t & \tau Q_C^{\pi,\theta}(e) = (q, \mathbf{ptr}(t)) \\ t & \tau Q_C^{\pi,\theta}(e) = (q, \mathbf{array}(t,n)) \\ (\emptyset, \mathbf{void}) & \text{otherwise} \end{cases}$
- Address of: $e \in \mathbb{E} \Rightarrow$

$$\tau Q_C^{\pi,\theta}(\&(e)) = \begin{cases} \tau Q_C^{\pi,\theta}(e') & e = *(e') \\ (\emptyset, \mathbf{ptr}(\tau Q_C^{\pi,\theta}(v)) & e = v \\ (\emptyset, \mathbf{ptr}(q' \cup q'', X)) & e = (e').f \wedge \tau Q_C^{\pi,\theta}(e') = (q', \mathbf{struct}\ t_C) \\ & \wedge\ \tau_F(f, \pi.T_F(t_C)) = (q'', X) \\ (\emptyset, \mathbf{void}) & \text{otherwise} \end{cases}$$

- Field Access: $e \in \mathbb{E}, f \in \mathbb{F} \Rightarrow \tau Q_C^{\pi,\theta}(e.f) = \tau Q_C^{\pi,\theta}(*(\&((e).f)))$
- Size of Type: $t \in \mathbb{T}_Q \Rightarrow \tau Q_C^{\pi,\theta}(\mathbf{sizeof}(t)) = (\emptyset, \theta.size\_t)$
- Size of Expression: $e \in \mathbb{E} \Rightarrow \tau Q_C^{\pi,\theta}(\mathbf{sizeof}(e)) = (\emptyset, \theta.size\_t)$

Note that this definition is actually not needed at all for *C-IL* semantics itself – the semantics of a *C-IL* program is not affected by type qualifiers, the correctness of its translation to machine code however may require a proper annotation of accesses with type qualifiers.

5.8.2 Expression Evaluation

Definition 5.35 (Field Reference Function) Given *C-IL* environment parameters θ and a *C-IL* program π, we define the function $\sigma_\theta^\pi : val \times \mathbb{F} \rightharpoonup val$ which takes a pointer or local reference to a struct type x and computes the pointer or local reference to a given field f in x:

$$\sigma_\theta^\pi(x,f) = \begin{cases} \mathbf{val}(a +_{8\cdot\theta.size_{ptr}} (\theta.offset(t_C, f))_{\mathbb{B}^{8\cdot\theta.size_{ptr}}}, t') & x = \mathbf{val}(a, \mathbf{ptr}(\mathbf{struct}\ t_C)) \\ \mathbf{lref}((v, o + \theta.offset(t_C, f)), i, t') & x = \mathbf{lref}((v, o), i, \mathbf{ptr}(\mathbf{struct}\ t_C)) \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $t' = \mathbf{ptr}(qt2t(\tau_F(f, \pi.T_F(t_C))))$.

Definition 5.36 (Expression Evaluation) We define expression evaluation as a function

$$[\![\cdot]\!]_{\cdot}^{\cdot,\cdot} : conf_{C\text{-}IL} \times prog_{C\text{-}IL} \times params_{C\text{-}IL} \times \mathbb{E} \rightharpoonup val$$

that takes a *C-IL* configuration c, a *C-IL* program π, *C-IL* environment parameters θ, and a *C-IL* expression as follows:

- Constant: $x \in val \Rightarrow [\![x]\!]_c^{\pi,\theta} = x$

  A constant x always evaluates to x.

- Variable Name: $v \in \mathbb{V} \Rightarrow [\![v]\!]_c^{\pi,\theta} = [\![*(\&(v))]\!]_c^{\pi,\theta}$

  Evaluating a variable name v is done by taking a pointer to v followed by dereferencing that pointer, effectively reading the value associated with v from the appropriate memory.

- Function Name: $fn \in \mathbb{F}_{name} \Rightarrow$

$$[\![fn]\!]_c^{\pi,\theta} = \begin{cases} \mathbf{val}(\theta.\mathcal{F}_{adr}(fn), qt2t(\tau_{fun}^{\mathcal{F}_\pi^\theta}(fn))) & fn \in dom(\mathcal{F}_\pi^\theta) \wedge fn \in dom(\theta.\mathcal{F}_{adr}) \\ \mathbf{fun}(fn, qt2t(\tau_{fun}^{\mathcal{F}_\pi^\theta}(fn))) & fn \in dom(\mathcal{F}_\pi^\theta) \wedge fn \notin dom(\theta.\mathcal{F}_{adr}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

  Evaluating a declared function name fn results either in an explicit function pointer (if an address for the function pointer is defined in the given *C-IL* environment θ) or in a symbolic function value (if no function pointer address is defined in θ).

- Unary Operator: $e \in \mathbb{E}, \ominus \in \mathbb{O}_1 \Rightarrow [\![\ominus e]\!]_c^{\pi,\theta} = \theta.op_1(\ominus)([\![e]\!]_c^{\pi,\theta})$

  Application of an unary operator $\ominus$ to *C-IL* expression e is evaluated by applying the operator function defined for $\ominus$ in θ to the value of expression e.

- Binary Operator: $e_0, e_1 \in \mathbb{E}, \oplus \in \mathbb{O}_2 \Rightarrow [\![e_0 \oplus e_1]\!]_c^{\pi,\theta} = \theta.op_2(\oplus)([\![e_0]\!]_c^{\pi,\theta}, [\![e_1]\!]_c^{\pi,\theta})$

  Analogous to evaluation of unary operators, we evaluate the sub-expressions and apply the operator function defined in θ.

- Ternary Operator: $e, e_0, e_1 \in \mathbb{E} \Rightarrow [\![(e\ ?\ e_0 : e_1)]\!]_c^{\pi,\theta} = \begin{cases} [\![e_0]\!]_c^{\pi,\theta} & zero(\theta, [\![e]\!]_c^{\pi,\theta}) \\ [\![e_1]\!]_c^{\pi,\theta} & \text{otherwise} \end{cases}$

  The ternary operator performs a case distinction on the value of expression e. If e evaluates to a zero-value, the ternary operator application evaluates to the value of e_0, otherwise we obtain the value of e_1.

- Type Cast: $t \in \mathbb{T}_Q, e \in \mathbb{E} \Rightarrow [\![(t)e]\!]_c^{\pi,\theta} = \theta.cast([\![e]\!]_c^{\pi,\theta}, qt2t(t))$

  Performing a type cast applies the type cast function given by θ to the value of expression e, resulting in a corresponding value of type t.

- Dereferencing a Pointer: $e \in \mathbb{E} \Rightarrow$

$$[\![*(e)]\!]_c^{\pi,\theta} = \begin{cases} read(\theta, c, [\![e]\!]_c^{\pi,\theta}) & (\tau([\![e]\!]_c^{\pi,\theta}) = \mathbf{ptr}(t) \wedge \neg isarray(t)) \\ & \vee\ \tau([\![e]\!]_c^{\pi,\theta}) = \mathbf{array}(t,n)) \\ \mathbf{val}(a, \mathbf{array}(t,n)) & [\![e]\!]_c^{\pi,\theta} = \mathbf{val}(a, \mathbf{ptr}(\mathbf{array}(t,n))) \\ \mathbf{lref}((v,o), i, \mathbf{array}(t,n)) & [\![e]\!]_c^{\pi,\theta} = \mathbf{lref}((v,o), i, \mathbf{ptr}(\mathbf{array}(t,n))) \\ \text{undefined} & \text{otherwise} \end{cases}$$

  Dereferencing a pointer type pointing to a non-array type or dereferencing an array type results in reading the corresponding value from the appropriate memory of configuration c. Dereferencing a pointer or local reference to an array type results in a corresponding array-type value (which is equivalent to a pointer to the array's first element). Note that function pointers cannot be dereferenced without a prior type-cast.

- Address of: $e \in \mathbb{E} \Rightarrow$

$$[\![\&(e)]\!]_c^{\pi,\theta} = \begin{cases} [\![e']\!]_c^{\pi,\theta} & e = *(e') \\ \mathbf{lref}((v,0), |c.s|-1, \mathbf{ptr}(qt2t(\tau_V(v, \mathcal{V}_{top}(\pi,c))))) & e = v \wedge v \in decl(\mathcal{V}_{top}(\pi,c)) \\ \mathbf{val}(\theta.alloc_{gvar}(v), \mathbf{ptr}(qt2t(\tau_V(v, \pi.\mathcal{V}_G)))) & e = v \wedge v \notin decl(\mathcal{V}_{top}(\pi,c)) \\ & \wedge\ v \in decl(\pi.\mathcal{V}_G) \\ \sigma_\theta^\pi([\![\&(e')]\!]_c^{\pi,\theta}, f) & e = (e').f \\ \text{undefined} & \text{otherwise} \end{cases}$$

  The address-of operation cancels out with the dereference-pointer operation. If the expression whose address is to be taken is a variable name v, the result depends on whether v is the name of a local variable declared in the top-most stack frame of configuration c. If yes, an appropriate local reference to v is returned, otherwise, if v is a declared global variable name, we return a pointer to the base address of the global variable. When the expression we take address-of is a field access, we compute the address of the resulting local reference or pointer by using the field reference function σ_θ^π.

- Field Access: $e \in \mathbb{E}, f \in \mathbb{F} \Rightarrow [\![(e).f]\!]_c^{\pi,\theta} = [\![*(\&((e).f))]\!]_c^{\pi,\theta}$

  Similar to evaluation of variable names, we compute the value of a field access by computing and dereferencing the address of the field access. That this definition is not circular can be checked by considering the definition of expression evaluation for the address-of and the pointer-dereference operation.

- Size of Type: $t \in \mathbb{T}_Q \Rightarrow [\![\mathbf{sizeof}(t)]\!]_c^{\pi,\theta} = \mathbf{val}(size_\theta(qt2t(t))_{8 \cdot size_\theta(\theta.size\_t)}, \theta.size\_t)$

  Evaluating the size-of-type expression amounts to constructing a *C-IL* value by converting the size of the type t in bytes to a byte-string of appropriate length for the type $\theta.size\_t$ which is the type of the size-of operator.

- Size of Expression: $e \in \mathbb{E} \Rightarrow [\![\mathbf{sizeof}(e)]\!]_c^{\pi,\theta} = [\![\mathbf{sizeof}(\tau_{Q_c}^{\pi,\theta}(e))]\!]_c^{\pi,\theta}$

  Size-of-expression is evaluated by evaluating the type of expression e and then using the previous definition.

5.8.3 Transition Function

For given *C-IL* program π and environment parameters θ, we define a partial transition function

$$\delta^{\pi,\theta}_{C\text{-}IL} : conf_{C\text{-}IL} \times \Sigma \rightharpoonup conf_{C\text{-}IL}$$

where Σ is an input alphabet used to resolve non-deterministic choice occurring in *C-IL* semantics. In fact, there are only two kinds of non-deterministic choice in *C-IL*: the first occurs in a function call step – the values of local variables of the new stack frame are not initialized, thus, they are chosen arbitrarily; the second is due to the possible non-deterministic nature of external function calls – here, one of the possible transitions specified by relation $\theta.R_{\mathbf{extern}}$ is chosen. To resolve this non-deterministic choice, our transition function gets as an input either the local memory configuration of all local variables, the resulting *C-IL* configuration for an external function call, or – in case of a deterministic step – the value $\perp$:

$$\Sigma = (\mathbb{V} \rightharpoonup (\mathbb{B}^8)^*) \cup conf_{C\text{-}IL} \cup \{\perp\}$$

Auxiliary Definitions

In defining the semantics of *C-IL* we will use the following shorthand notation to refer to information about the topmost stack frame in a *C-IL*-configuration:

- local memory of the topmost frame: $\mathcal{M}_{\mathcal{E}top}(c) = \mathbf{hd}(c.s).\mathcal{M}_{\mathcal{E}}$
- return destination of the topmost frame: $rds_{top}(c) = \mathbf{hd}(c.s).rds$
- function name of the topmost frame: $f_{top}(c) = \mathbf{hd}(c.s).f$
- location counter of the topmost frame: $loc_{top}(c) = \mathbf{hd}(c.s).loc$
- function body of the topmost frame: $\mathcal{P}_{top}(\pi,c) = \pi.\mathcal{F}(f_{top}(c)).\mathcal{P}$
- next statement to be executed: $stmt_{next}(\pi,c) = \mathcal{P}_{top}(\pi, c)[loc_{top}(c)]$

We define functions that perform specific updates on a *C-IL* configuration in the following.

Definition 5.37 (Increasing the Location Counter) We define the function

$$inc_{loc} : conf_{C\text{-}IL} \rightharpoonup conf_{C\text{-}IL}$$

that increases the location counter of the topmost stack frame of a *C-IL* configuration as follows:

$$inc_{loc}(c) = c[s := \mathbf{hd}(c.s)[loc := loc_{top}(c) + 1] \circ \mathbf{tl}(c.s)]$$

Definition 5.38 (Setting the Location Counter) The function

$$set_{loc} : conf_{C\text{-}IL} \times \mathbb{N} \rightharpoonup conf_{C\text{-}IL}$$

defined as

$$set_{loc}(c,l) = c[s := \mathbf{hd}(c.s)[loc := l] \circ \mathbf{tl}(c.s)]$$

sets the location counter of the top-most stack frame to location l.

Definition 5.39 (Removing the Topmost Frame) The function

$$drop_{frame} : conf_{C\text{-}IL} \rightharpoonup conf_{C\text{-}IL}$$

which removes the top-most stack frame from a *C-IL*-configuration is defined as:

$$drop_{frame}(c) = c[s := \mathbf{tl}(c.s)]$$

Definition 5.40 (Setting Return Destination) We define the function

$$set_{rds} : conf_{C\text{-}IL} \times (val_{\mathbf{lref}} \cup val_{\mathbf{ptr}} \cup \{\bot\}) \rightharpoonup conf_{C\text{-}IL}$$

that updates the return destination component of the top most stack frame as:

$$set_{rds}(c,v) = c[s := \mathbf{hd}(c.s)[rds := v] \circ \mathbf{tl}(c.s)]$$

Note that all of the functions defined above are only well-defined when the stack is not empty; this is why the latter functions have to be declared partial functions. In practice however, executing a *C-IL* program always requires a non-empty stack.

Operational Semantics

Definition 5.41 (*C-IL* Transition Function) We define the transition function

$$\delta^{\pi,\theta}_{C\text{-}IL} : conf_{C\text{-}IL} \times \Sigma \rightharpoonup conf_{C\text{-}IL}$$

by a case distinction on the given input:

- Deterministic step:

$$\delta^{\pi,\theta}_{C\text{-}IL}(c,\bot) = \begin{cases} inc_{loc}(write(\theta, c, [\![\&e_0]\!]^{\pi,\theta}_c, [\![e_1]\!]^{\pi,\theta}_c) & stmt_{next}(\pi, c) = (e_0 = e_1) \\ set_{loc}(c, l) & stmt_{next}(\pi, c) = \mathbf{goto}\ l \\ set_{loc}(c, l) & stmt_{next}(\pi, c) = \mathbf{ifnot}\ e\ \mathbf{goto}\ l \wedge zero(\theta, [\![e]\!]^{\pi,\theta}_c) \\ inc_{loc}(c) & stmt_{next}(\pi, c) = \mathbf{ifnot}\ e\ \mathbf{goto}\ l \wedge \neg zero(\theta, [\![e]\!]^{\pi,\theta}_c) \\ drop_{frame}(c) & stmt_{next}(\pi, c) = \mathbf{return} \\ drop_{frame}(c) & stmt_{next}(\pi, c) = \mathbf{return}\ e \wedge rds = \bot \\ write(\theta, c', rds, [\![e]\!]^{\pi,\theta}_c) & stmt_{next}(\pi, c) = \mathbf{return}\ e \wedge rds \neq \bot \\ \text{undefined} & \text{otherwise} \end{cases}$$

  where $c' = set_{rds}(drop_{frame}(c), \bot)$ and $rds = rds_{top}(drop_{frame}(c))$.

- Function call:

  $\delta^{\pi,\theta}_{C\text{-}IL}(c, \mathcal{M}_{lvar})$, where $\mathcal{M}_{lvar} \in \mathbb{V} \rightharpoonup (\mathbb{B}^8)^*$ provides initial values for all local variables of the called function, is defined if and only if all of the following hold:

  – $stmt_{next}(\pi, c) = \mathbf{call}\ e(E) \vee stmt_{next}(\pi, c) = (e_0 = \mathbf{call}\ e(E))$ – the next statement is a function call (without or with return value),

- $[\![e]\!]_c^{\pi,\theta} = \mathbf{val}(b, \mathbf{funptr}(t,T)) \wedge f = \theta.\mathcal{F}_{adr}{}^{-1}(b) \vee [\![e]\!]_c^{\pi,\theta} = \mathbf{fun}(f, \mathbf{funptr}(t,T))$ – expression e evaluates to some function f,
- $|E| = \mathcal{F}_\pi^\theta(f).npar \wedge \forall i \in \{0,\dots,|E|-1\} : \mathcal{F}_\pi^\theta(f).\mathcal{V}[i] = (v,t) \Rightarrow \tau([\![E[i]]\!]_c^{\pi,\theta}) = t$ – the types of all parameters passed match the declaration,
- $\mathcal{F}_\pi^\theta(f).\mathcal{P} \neq \mathbf{extern}$ – the function is not declared as extern in the function table, and
- $\forall i \geq \mathcal{F}_\pi^\theta(f).npar : \exists v,t : \mathcal{F}_\pi^\theta(f).\mathcal{V}[i] = (v,t) \Rightarrow |\mathcal{M}_{lvar}(v)| = size_\theta(t)$ – the byte-string memory provided for all local variables is of adequate length.

Then, we define

$$\delta_{C\text{-}IL}^{\pi,\theta}(c, \mathcal{M}_{lvar}) = c'$$

such that

$$c'.s = (\mathcal{M}'_{\mathcal{E}}, \bot, f, 0) \circ inc_{loc}(set_{rds}(c, rds)).s$$

$$c'.\mathcal{M} = c.\mathcal{M}$$

where

$$rds = \begin{cases} [\![\&e_0]\!]_c^{\pi,\theta} & stmt_{next}(\pi,c) = (e_0 = \mathbf{call}\ e(E)) \\ \bot & stmt_{next}(\pi,c) = \mathbf{call}\ e(E) \end{cases}$$

and

$$\mathcal{M}'_{\mathcal{E}}(v) = \begin{cases} val2bytes_\theta([\![E[i]]\!]_c^{\pi,\theta}) & \exists i : \mathcal{F}_\pi^\theta(f).\mathcal{V}[i] = (v,t) \wedge i < \mathcal{F}_\pi^\theta(f).npar \\ \mathcal{M}_{lvar}(v) & \exists i.\ \mathcal{F}_\pi^\theta(f).\mathcal{V}[i] = (v,t) \wedge i \geq \mathcal{F}_\pi^\theta(f).npar \\ \text{undefined} & \text{otherwise} \end{cases}$$

- External procedure call:

  $\delta_{C\text{-}IL}^{\pi,\theta}(c, c')$ is defined if and only if all of the following hold:

  - $stmt_{next}(\pi,c) = \mathbf{call}\ e(E)$ – the next statement is a function call without return value,
  - $[\![e]\!]_c^{\pi,\theta} = \mathbf{val}(b, \mathbf{funptr}(t,T)) \wedge f = \theta.\mathcal{F}_{adr}{}^{-1}(b) \vee [\![e]\!]_c^{\pi,\theta} = \mathbf{fun}(f, \mathbf{funptr}(t,T))$ – expression e evaluates to some function f,
  - $|E| = \mathcal{F}_\pi^\theta(f).npar \wedge \forall i \in \{0,\dots,|E|-1\} : \mathcal{F}_\pi^\theta(f).\mathcal{V}[i] = (v,t) \Rightarrow \tau([\![E[i]]\!]_c^{\pi,\theta}) = t$ – the types of all parameters passed match the declaration,
  - $\mathbf{tl}(c'.s) = \mathbf{tl}(inc_{loc}(c).s)$ – the external procedure call does not modify any stack frames other than the topmost frame,
  - $\mathbf{hd}(c'.s).loc = \mathbf{hd}(c.s).loc + 1 \wedge \mathbf{hd}(c'.s).f = \mathbf{hd}(c.s).f$ – the location counter of the topmost frame is incremented and the function is not changed,
  - $\mathcal{F}_\pi^\theta(f).\mathcal{P} = \mathbf{extern}$ – the function is declared as extern in the function table, and
  - $([\![E[0]]\!]_c^{\pi,\theta}, \dots, [\![E[|E|-1]]\!]_c^{\pi,\theta}, c, c') \in \theta.R_{\mathbf{extern}}$ – the transition relation for external functions given in the *C-IL* environment parameters allows a transition under given parameters E from c to c'.

Note that we restrict external function calls in such a way that they cannot be invoked with a return value. However, there is a simple way to allow an external function call to return a result: It is always possible to pass a pointer to some subvariable to which a return value from an external function call can be written.

Then,

$$\delta^{\pi,\theta}_{C\text{-}IL}(c, c') = c'$$

5.8.4 Concurrent *C-IL*

Since we want to argue about a multi-core machine that runs structured parallel C let us introduce concurrent *C-IL* (*CC-IL*) that offers different *C-IL* threads which execute on a shared memory interleaved at statement-level.

Definition 5.42 (Concurrent *C-IL* Configuration) A concurrent *C-IL* configuration

$$c = (c.\mathcal{M}, c.Th) \in conf_{CC\text{-}IL}$$

is a pair of a byte-addressable memory $c.\mathcal{M} : \mathbb{B}^{8\cdot size_{ptr}} \to \mathbb{B}^8$ and a partial function $c.Th : \mathbb{N} \rightharpoonup frame^*_{C\text{-}IL}$ that maps thread identifiers (given as natural numbers) to *C-IL* stacks.

Definition 5.43 (Transition Function of *CC-IL*) Semantics of *CC-IL* for a given *C-IL* program π and environment parameters θ is given by the transition function

$$\delta^{\pi,\theta}_{CC\text{-}IL} : conf_{CC\text{-}IL} \times \mathbb{N} \times \Sigma \rightharpoonup conf_{CC\text{-}IL}$$

where

$$\delta^{\pi,\theta}_{CC\text{-}IL}(c, t, in).\mathcal{M} = \delta^{\pi,\theta}_{C\text{-}IL}((c.\mathcal{M}, c.Th(t)), in).\mathcal{M}$$

and

$$\delta^{\pi,\theta}_{CC\text{-}IL}(c, t, in).Th(i) = \begin{cases} \delta^{\pi,\theta}_{C\text{-}IL}((c.\mathcal{M}, c.Th(t)), in).s & i = t \\ c.Th(i) & \text{otherwise} \end{cases}$$

Note that while we define concurrent *C-IL* semantics here by simply interleaving steps of *C-IL* threads non-deterministically at small-step semantics level, establishing this abstraction in a formal model stack is highly nontrivial, as described in chapter 3. Since *C-IL* statements tend to compile to several machine instructions, it is necessary to show that this model is a correct abstraction for a model that arbitrarily interleaves machine instructions. Applying an ownership discipline and appropriate order-reduction theory [Bau12], however, it can be shown that compiled *C-IL* code of programs that respect the ownership discipline can be reordered to *C-IL* instruction boundaries in such a way that the concurrent *C-IL* model provided here is a sound abstraction.

6 Specification (Ghost) State and Code

spec·i·fi·ca·tion *noun*

1. the act or process of specifying
2. **a** : a detailed precise presentation of something or of a plan or proposal for something – usually used in plural

   b : a statement of legal particulars (as of charges or of contract terms); also : a single item of such statement

   c : a written description of an invention for which a patent is sought

"specification." Merriam-Webster.com. Merriam-Webster, 2011. Web. 6 July 2011.

During the VerisoftXT project, we have used the tool VCC, which is a powerful automated verifier for concurrent C code. Code to be verified is annotated with specification of three kinds:

- specification (ghost) state,

  This includes both adding ghost variables as well as adding ghost fields to struct variables.

- specification (ghost) code,

  In order to update ghost state, we can use ghost code which is essentially ordinary C code operating on ghost state only. Ghost code is explicitly not restricted to assignments, e.g. it may include loops or function calls.

- assertions.

  VCC's assertion language allows to specify pre- and postconditions for functions and two-state invariants on data structures. In addition, assert and assume statements are provided in order to verify (or assume) specific assertions at specific points of program execution.

In order to argue soundness of VCC, it is necessary to prove that adding ghost state and code to a C program does in fact not change the semantics of the original C program. That ghost state and code where ghost code consists of simple assignment statements to ghost state can be eliminated has been proven before [HP08]. That more sophisticated ghost code which is powerful enough to implement control-flow altering or diverging program fragments does not change the semantics of the original program, however, is a folklore theorem whose proof so far tends to be ignored [CMST10]. A proof can be given by stating a simulation that expresses preservation of semantics between the original C program and the annotated 'C + ghost' program. In this chapter, we introduce a formal model of *C-IL*+$\mathcal{G}$ (*C-IL* extended with ghost state), and we provide such a proof. An important property of ghost code is that there is no information flow from ghost state to implementation state; this property is enforced by VCC. Another important requirement is termination of ghost code – since the provided ghost code is quite powerful, it is not hard to write ghost code that diverges, effectively allowing us to deduce whatever we want in the postcondition of a function with nonterminating ghost code; this property is to our knowledge not yet enforced by VCC although the authors of VCC are aware of it (VCC proves partial correctness only).

In the following, we define a formal model of *C-IL*+$\mathcal{G}$ stated in terms of operational semantics. It includes both ghost state and ghost code – but not assertions since these belong to the program logic applied to *C-IL*+$\mathcal{G}$. Given such a model, proving soundness of a logic for sequential program verification with respect to the given model is no longer a difficult task. In order to achieve a soundness proof of VCC's verification methodology, what remains to be proven is: i) that the concurrency verification methodology of VCC itself is sound (there are papers [CDH+09, CAB+09, CMTS09] that support such claims on a high level of abstraction) and ii) that the concurrency verification methodology is correctly implemented in the default ghost state (ownership) of VCC. The latter can be argued based on an explicit encoding of the ownership model of VCC in the ghost state of *C-IL*+$\mathcal{G}$ which is to be defined analogous to the ownership state and assertions VCC generates when it compiles annotated C programs to Boogie.

We give a description of *C-IL*+$\mathcal{G}$ by first providing all definitions needed to describe *ghost state* (i.e. *ghost types*, *ghost values*, *ghost memory*), followed by the definitions dealing with *ghost code* (i.e. ghost expressions, ghost statements). Then, we define operational semantics of *C-IL*+$\mathcal{G}$ (i.e. configurations, programs, and transition function) based on these definitions. Last, we give a paper-and-pencil proof of semantics-preserving ghost state and code elimination.

6.1 Ghost State

We extend *C-IL* with ghost state. Ghost state can occur in the following forms:

- Ghost fields attached to implementation structs,

  Any implementation struct may be enriched by adding ghost fields to it. A restriction is that ghost field names must be disjoint from implementation field names.

- Local ghost variables

  Functions may be annotated by declaring local ghost variables whose names shall be disjoint from the implementation local variable and parameter names.

- Ghost parameters in function calls

  Function calls and function declarations can be enriched by allowing the passing of ghost parameters. When a function declaration is extended by ghost parameters, all corresponding function calls must be type-correct with respect to the annotated declaration.

- Global ghost variables

  The annotated program may contain global ghost variables whose names are disjoint from the implementation global variable names.

- Ghost heap variables

  In order to allow for unlimited creation of ghost variables, we provide a ghost heap on which ghost variables may be allocated. The ghost heap is infinite: since we are not limited by physical constraints like a finite physical memory, we do not consider deallocation of ghost heap variables.

Similar to regular variables and struct fields, all ghost state components that are accessed by a program must be declared beforehand. This is done by annotating the program text with ghost declarations. These annotations are clearly separated from the original code by marking them as ghost in the formal definition of the *C-IL*+$\mathcal{G}$ program. In addition to the existing *C-IL*-types, the verification engineer may use special *ghost types*, e.g. $\mathbb{Z}$ (for a definition of ghost types, see section 6.1.2) for declaration of ghost state components.

Since we want to reuse as much of the already existing formalism as possible, we define ghost state to extend *C-IL* state (in which we have struct types, local variables and global variables already) in such a way that we reuse semantics wherever possible. However, since we do not support pointer arithmetics on ghost state – in particular defining a byte-string representation for ghost types would be quite pointless – we can use a much more structured memory model to store the content of ghost components.

To declare specification state, we extend a *C-IL* program to a *C-IL*+$\mathcal{G}$ program by adding local and global ghost variable declarations and a ghost field declaration function for struct types which may both use ghost types. To hold the values of these ghost variables and ghost fields, we add a global ghost memory to the *C-IL*+$\mathcal{G}$ configuration and local ghost memories to the stack frames of the *C-IL*+$\mathcal{G}$ configuration (see section 6.3.1). In the following, we provide definitions of ghost memory, ghost types and ghost values.

6.1.1 Ghost Memory

For our ghost state, we actually do not want a memory that is close to the hardware memory in terms of formalism. Instead, we can choose a much more abstract view. There is no point in supporting pointer arithmetics on ghost state. Thus, we can simply model ghost memory in a structured way in terms of structured ghost values.

Definition 6.1 (Structured Ghost Values) The set of *structured ghost values* $val_{\mathcal{M}_\mathcal{G}}$ is defined inductively as follows:

- Ordinary *C-IL* values, ghost values and no value: $val \cup val_\mathcal{G} \cup \{\bot\} \subseteq val_{\mathcal{M}_\mathcal{G}}$

  As a base case, a structured ghost value can be an ordinary *C-IL* value, a ghost value (for a definition of ghost values, see section 6.1.3) or the value $\bot$ that denotes an unknown value, e.g. the value of an uninitialized ghost field or variable.

- Struct and array values: $f : (\mathbb{F} \cup \mathbb{N}) \rightharpoonup val_{\mathcal{M}_\mathcal{G}} \Rightarrow f \in val_{\mathcal{M}_\mathcal{G}}$

  We model struct and array ghost memory values as functions from field names, or, respectively, array indices given as natural numbers to structured ghost values. Note that for a type-correct *C-IL*+$\mathcal{G}$ program, $dom(f)$ will always be appropriate for the type of the corresponding ghost subvariable.

```
struct A
{
  ... // implementation fields
  spec(t1 f1;) // ghost fields
  ...
  spec(int fi[m];)
  ...
  spec(tn fn;)
}
```

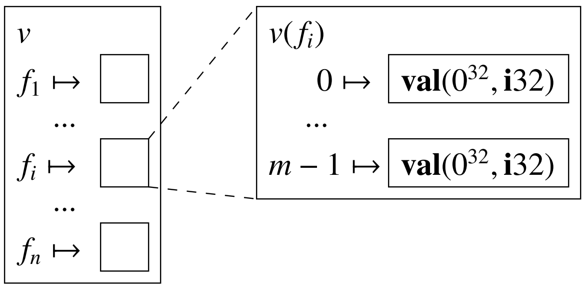

v
$f_1 \mapsto$
$f_i \mapsto$
$f_n \mapsto$
$v(f_i)$
$0 \mapsto$ $\mathbf{val}(0^{32}, \mathbf{i}32)$
$m-1 \mapsto$ $\mathbf{val}(0^{32}, \mathbf{i}32)$

Figure 6.1: Example structured ghost value v for struct A that provides values for the ghost fields $f_1, \ldots, f_n$ declared in A. In this example, field f_i is of type **array**($\mathbf{i}32, m$). The value of the array field f_i in structured ghost value v is given by structured ghost value $v(f_i)$ which, in this example, provides appropriately typed zero-values for all array elements.

For an example of a structured ghost value, consider figure 6.1. This way of modeling memory corresponds very closely to the memory model of C0 from the Verisoft project [Ver07]. Instead of considering a flat byte-addressable memory where all addresses of subvariables must be computed according to the concrete layout of structs and arrays (like in *C-IL*), here, we do not consider the concrete memory layout. Instead, when we need to talk about subvariables in *C-IL+$\mathcal{G}$* (e.g. when setting a ghost pointer to point to some ghost subvariable), we model this by a base variable (for which ghost memory provides a structured ghost value) in combination with a sequence of field and array accesses that describes a particular subvariable of the base variable. The intricate nature of the memory model of *C-IL+$\mathcal{G}$* results from the fact that we extend the byte-addressable memory of the implementation *C-IL* machine with an abstract memory model for the ghost state.

Definition 6.2 (Subvariable Selectors) In order to specify accessing a certain sub-value of a structured ghost value, we use *subvariable selectors*

$$s \in \mathbb{F} \cup \mathbb{N}$$

$s \in \mathbb{F}$ specifies a field access, while $s \in \mathbb{N}$ specifies an array access.

We use sequences of subvariable selectors to perform accesses to structured ghost values and to describe subvariables of ghost variables.

Reading and Writing Structured ghost values

We define auxiliary functions that read from and write to structured ghost values by descending into the structured ghost value by recursively applying subvariable selectors from a sequence of subvariable selectors.

Definition 6.3 (Reading a Subvalue of a Structured ghost value) We define the function

$$read_{val_{\mathcal{M}_\mathcal{G}}} : val_{\mathcal{M}_\mathcal{G}} \times (\mathbb{N} \cup \mathbb{F})^* \rightharpoonup val_{\mathcal{M}_\mathcal{G}}$$

that reads subvalues specified by a sequence of subvariable selectors from a structured ghost value as

$$read_{val_{\mathcal{M}_\mathcal{G}}}(v,S) = \begin{cases} v & S = \varepsilon \\ read_{val_{\mathcal{M}_\mathcal{G}}}(v(\mathbf{hd}(S)), \mathbf{tl}(S)) & S \neq \varepsilon \land \mathbf{hd}(S) \in dom(v) \\ undefined & \text{otherwise} \end{cases}$$

Definition 6.4 (Writing a Subvalue of a Structured ghost value) We define the function

$$write_{val_{\mathcal{M}_\mathcal{G}}} : val_{\mathcal{M}_\mathcal{G}} \times (\mathbb{N} \cup \mathbb{F})^* \times val_{\mathcal{M}_\mathcal{G}} \rightharpoonup val_{\mathcal{M}_\mathcal{G}}$$

that writes a given structured ghost value to replace a given subvalue of another structured ghost value as

$$write_{val_{\mathcal{M}_\mathcal{G}}}(v,S,y) = \begin{cases} y & S = \varepsilon \\ v(\mathbf{hd}(S) := write_{val_{\mathcal{M}_\mathcal{G}}}(v(\mathbf{hd}(S)), \mathbf{tl}(S), y)) & S \neq \varepsilon \land \land \mathbf{hd}(S) \in dom(v) \\ undefined & \text{otherwise} \end{cases}$$

Global Ghost Memory

Global ghost memory contains the following:

- ghost data (ghost fields) of global implementation structs,
- ghost heap variables, and
- global ghost variables.

Essentially, we have three different kinds of ghost variables here, i) ghost variables that contain the values of ghost fields of global implementation structs (which can be identified by a pair (a, t_C) of byte-address a and composite type name t_C), ii) ghost heap variables (which we identify by addressing them with natural numbers in the order of their allocation), and iii) global ghost variables (which are identified simply by their variable name).

Definition 6.5 (Global Ghost Memory) A *global ghost memory* is a partial function

$$\mathcal{M}_\mathcal{G} : (\mathbb{B}^{8 \cdot \theta.size_{ptr}} \times \mathbb{T}_C) \cup \mathbb{N} \cup \mathbb{V} \rightharpoonup val_{\mathcal{M}_\mathcal{G}}$$

that takes either a pair of implementation global memory address and composite type name, the ghost heap address of a ghost heap variable, or the name of a global ghost variable. It returns a structured ghost value that provides values for all subvariables of the given ghost variable.

Figure 6.2 illustrates this definition.

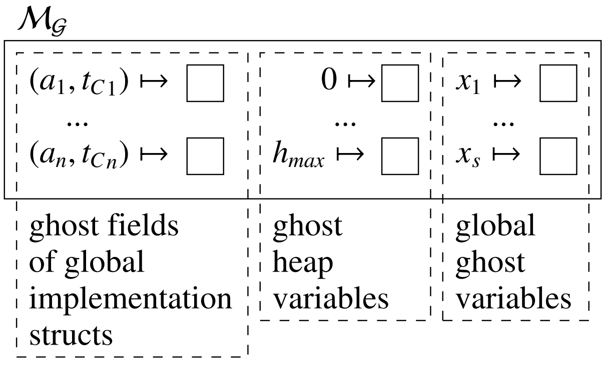


Figure 6.2: Contents of a global ghost memory $\mathcal{M}_{\mathcal{G}}$. Here, $(a_1, t_{C1}), \ldots, (a_n, t_{Cn})$ are pairs of implementation global memory address and composite type name for which the global ghost memory currently provides ghost field values, the current ghost heap variables are numbered from 0 to h_{max}, and $x_1, \ldots, x_s$ are the names of all declared global ghost variables.

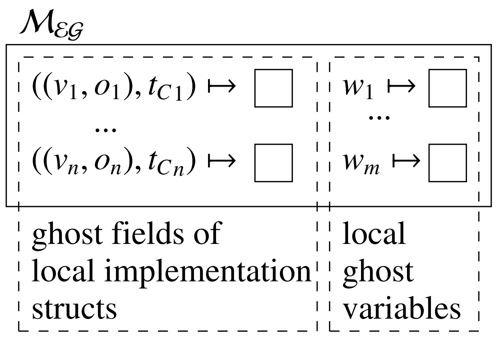

$\mathcal{M}_{\mathcal{E}\mathcal{G}}$
$((v_1, o_1), t_{C1}) \mapsto$
...
$((v_n, o_n), t_{Cn}) \mapsto$
$w_1 \mapsto$
...
$w_m \mapsto$
ghost fields of local implementation structs
local ghost variables

Figure 6.3: Contents of a local ghost memory $\mathcal{M}_{\mathcal{E}\mathcal{G}}$. Here, $((v_1, o_1), t_{C1}), \ldots, ((v_n, o_n), t_{Cn})$ are pairs of local implementation subvariable (specified by variable name and byte-offset) and composite type name for which the local ghost memory currently provides ghost field values, and $w_1, \ldots, w_m$ are the names of all local ghost variables stored in the local ghost memory.

Local Ghost Memory

Global ghost memory contains the following:

- ghost data (ghost fields) of local implementation struct-type subvariables, and
- local ghost variables (including ghost parameters).

Local ghost memory provides structured ghost values for two different kinds of ghost variables: i) local ghost variables used to hold the ghost fields of local implementation struct-type subvariables (which are identified by $((v, o), t_C)$ where v is a local implementation variable name, o is a byte-offset in the local implementation memory of v that specifies an implementation subvariable of v, and t_C is a composite type name), and ii) local ghost variables (which are identified by their variable name). Later, we will obtain *C-IL+$\mathcal{G}$* stack frames by extending the definition of *C-IL* stack frame with a local ghost memory.

Definition 6.6 (Local Ghost Memory) A *local ghost memory* is a partial function

$$\mathcal{M}_{\mathcal{E}\mathcal{G}} : (\mathbb{V} \times \mathbb{N}) \times \mathbb{T}_C \cup \mathbb{V} \rightharpoonup val_{\mathcal{M}_{\mathcal{G}}}$$

that maps local implementation struct-type subvariables and local ghost variables to their corresponding structured ghost values.

Figure 6.3 illustrates this definition.

6.1.2 Ghost Types

In addition to the regular types of *C-IL*, special ghost types can be used to declare ghost state. We define the set of all possible ghost types $\mathbb{T}_{\mathcal{G}}$. Note that both the set of *C-IL* types $\mathbb{T}$ and the set of ghost types $\mathbb{T}_{\mathcal{G}}$ will be used later in order to define *C-IL+$\mathcal{G}$* programs.

Definition 6.7 (Ghost Types) We define the set of *ghost types* $\mathbb{T}_{\mathcal{G}}$ inductively as as follows:

- Mathematical integers: **math_int** $\in \mathbb{T}_{\mathcal{G}}$

  A type for unbounded integers (from the set $\mathbb{Z}$).

- Generic pointers: **obj** $\in \mathbb{T}_{\mathcal{G}}$

  Not to be confused with **ptr**(t) which is a pointer type with a specific base type t. The type **obj** is a generic pointer type in the sense that values of this type can hold arbitrary pointers including their type information. In conjunction with maps (see below), this can be used to formalize pointer sets.

- State-snapshots: **state_t** $\in \mathbb{T}_{\mathcal{G}}$

  In VCC it is possible to take a snapshot of the current state of the *C-IL+$\mathcal{G}$* machine, storing this snapshot in ghost state. This can be useful in order to argue about a history of program states in invariants.

- Maps: $t, t' \in \mathbb{T}_{\mathcal{G}} \cup \mathbb{T} \Rightarrow$ **map**(t,t') $\in \mathbb{T}_{\mathcal{G}}$

  A function type with domain type t and range type t'. Note that we can define maps over both ghost types and regular *C-IL* types.

- Records: $t_C \in \mathbb{T}_C \Rightarrow$ **record** $t_C \in \mathbb{T}_{\mathcal{G}}$

  Records, like structs, are given in terms of a composite type name. Record values are functions from declared field names to correctly typed values. Note that we use the set of composite type names $\mathbb{T}_C$ known from *C-IL* also for ghost types; generally, a subset of names from $\mathbb{T}_C$ is used by the implementation *C-IL* program, while the remaining composite type names can be used in ghost type declarations. Note that the main difference between a ghost value of record type and a ghost value of struct type is that the fields of a struct-type ghost variable can be addressed by ghost subvariables while the fields of a record type cannot, i.e. a record is an atomic value.

- Pointers to ghost types: $t \in \mathbb{T}_{\mathcal{G}} \Rightarrow$ **ptr**(t) $\in \mathbb{T}_{\mathcal{G}}$

- Arrays over ghost types: $t \in \mathbb{T}_{\mathcal{G}} \wedge n \in \mathbb{N} \Rightarrow$ **array**(t,n) $\in \mathbb{T}_{\mathcal{G}}$

Definition 6.8 (Qualified ghost types) Analogously, we define the set of *qualified ghost types* $\mathbb{T}_{Q\mathcal{G}}$ inductively as the smallest set containing the following:

- Qualified mathematical integers: $q \subseteq \mathbb{Q} \Rightarrow (q,$ **math_int**$) \in \mathbb{T}_{Q\mathcal{G}}$

- Qualified generic pointers: $q \subseteq \mathbb{Q} \Rightarrow (q,$ **obj**$) \in \mathbb{T}_{Q\mathcal{G}}$

- Qualified state-snapshots: $q \subseteq \mathbb{Q} \Rightarrow (q, \textbf{state\_t}) \in \mathbb{T}_{Q\mathcal{G}}$
- Qualified maps: $q \subseteq \mathbb{Q} \wedge t, t' \in \mathbb{T}_{\mathcal{G}} \cup \mathbb{T} \Rightarrow (q, \textbf{map}(t, t')) \in \mathbb{T}_{Q\mathcal{G}}$
- Qualified records: $q \subseteq \mathbb{Q} \wedge t_C \in \mathbb{T}_C \Rightarrow (q, \textbf{record}\ t_C) \in \mathbb{T}_{Q\mathcal{G}}$
- Qualified pointers to ghost types: $q \subseteq \mathbb{Q} \wedge X \in \mathbb{T}_{Q\mathcal{G}} \Rightarrow (q, \textbf{ptr}(X)) \in \mathbb{T}_{Q\mathcal{G}}$
- Qualified arrays over ghost types: $q \subseteq \mathbb{Q} \wedge X \in \mathbb{T}_{Q\mathcal{G}} \wedge n \in \mathbb{N} \Rightarrow (q, \textbf{array}(X, n)) \in \mathbb{T}_{Q\mathcal{G}}$

Note that we will assume an extension of the function $qt2t : \mathbb{T}_Q \cup \mathbb{T}_{Q\mathcal{G}} \to \mathbb{T} \cup \mathbb{T}_{\mathcal{G}}$ defined in the obvious way – by stripping all type qualifiers from the qualified type recursively. We omit its definition here.

Definition 6.9 (Auxiliary Predicates on Types) For *C-IL+$\mathcal{G}$*, we extend the predicates unqualified types from *C-IL*:

- $isptr(t) \equiv (\exists t' : t = \textbf{ptr}(t'))$
- $isarray(t) \equiv (\exists t', n : t = \textbf{array}(t', n))$
- $isfunptr(t) \equiv (\exists t', T : t = \textbf{funptr}(t', T))$

Note that syntactically, the definition looks identical to the *C-IL* definition, the semantic difference here is that these definitions are for both *C-IL* and ghost types.

6.1.3 Ghost Values

C-IL+$\mathcal{G}$ makes use of ghost values and regular *C-IL* values. Thus, in the following we define ghost values.

Definition 6.10 (Set of ghost values) We define the set of *ghost values $val_{\mathcal{G}}$* as the union of the sets of ghost values of a given type:

$$val_{\mathcal{G}} \overset{def}{=} val_{\textbf{math\_int}} \cup val_{\textbf{obj}} \cup val_{\textbf{state\_t}} \cup val_{\textbf{map}} \cup val_{\textbf{record}} \cup val_{\textbf{ghost-ptr+array}} \cup val_{\textbf{gfun}}$$

In order to define range and domain of maps in a sufficiently strict way, we need a function that, given a regular *C-IL* or ghost type, returns a set of possible values for that type.

Definition 6.11 (Values of a Given Type) We define the function

$$val_{\mathcal{G}} : \mathbb{T} \cup \mathbb{T}_{\mathcal{G}} \to 2^{val \cup val_{\mathcal{G}}}$$

which returns the set of values that can occur in a map value for a given type as

$$val_{\mathcal{G}}(t) = \begin{cases} val_{\mathbf{prim}(t)} & t \in \mathbb{T} \wedge t \in \mathbb{T}_{PP} \\ val_{\mathbf{ptr}(t)} & t \in \mathbb{T} \wedge (isptr(t) \vee isarray(t) \vee isfunptr(t)) \\ val_{\mathbf{math\_int}} & t = \mathbf{math\_int} \\ val_{\mathbf{obj}} & t = \mathbf{obj} \\ val_{\mathbf{state\_t}} & t = \mathbf{state\_t} \\ val_{\mathbf{map(t',t'')}} & t = \mathbf{map}(t', t'') \\ val_{\mathbf{record}(t_C)} & t = \mathbf{record}\ t_C \\ val_{\mathbf{gref}(t)} & t \in \mathbb{T}_{\mathcal{G}} \wedge isptr(t) \vee isarray(t) \\ \emptyset & \text{otherwise} \end{cases}$$

Note that we do not support maps that contain references to local variables (we only allow pointers to the global memory) or symbolic function values. In this definition, we use sets of individual ghost types which are defined below.

Definition 6.12 (Set of Mathematical Integer Values) The set of ghost mathematical integer values is defined as

$$val_{\mathbf{math\_int}} \overset{def}{=} \{\mathbf{gval}(i, \mathbf{math\_int}) \mid i \in \mathbb{Z}\}$$

Definition 6.13 (Set of Generic Pointer Values) The set of *generic pointer values* is defined as

$$val_{\mathbf{obj}} \overset{def}{=} \{\mathbf{gval}(p, \mathbf{obj}) \mid p \in val_{\mathbf{ptr+array}} \cup val_{\mathbf{ghost\text{-}ptr+array}}\}$$

Note that values of generic pointer type can contain both implementation and ghost pointer values (including their type information).

Definition 6.14 (Set of State-Snapshots) We define the set of *state-snapshots* as

$$val_{\mathbf{state\_t}} \overset{def}{=} \{\mathbf{gval}(c, \mathbf{state\_t}) \mid c \in conf_{C\text{-}IL+\mathcal{G}}\}$$

Note that effectively, by using $val_{\mathcal{G}}$ in the definition of some components of $conf_{C\text{-}IL+\mathcal{G}}$ (defined in section 6.3.1), we obtain a mutually recursive definition. In practice, however, only configurations that have been encountered before can be stored in a state-snapshot value of a *C-IL+$\mathcal{G}$* configuration, in particular, we can only take a state-snapshot of the current state. Thus, we only use configurations and state-snapshots in a well-founded way.

Definition 6.15 (Set of Map Values) We define the set of *map values* as

$$val_{\mathbf{map}} = \bigcup_{\mathbf{map}(t,t') \in \mathbb{T}_{\mathcal{G}}} val_{\mathbf{map}(t,t')}$$

where

$$val_{\mathbf{map}(t,t')} = \{\mathbf{gval}(f, \mathbf{map}(t, t')) \mid f : val_{\mathcal{G}}(t) \to val_{\mathcal{G}}(t')\}$$

That is, map values of type $\mathbf{map}(t, t')$ are functions that map values of type t to values of type t'. Note that t or t' may be regular *C-IL* types or ghost types. This definition is mutually recursive with that of $val_{\mathcal{G}}(\cdot)$. However, since types are well-founded, this construction is also well-founded.

Definition 6.16 (Set of Record Values) We define the set of *record values* as

$$val_{\mathbf{record}} = \bigcup_{t_C \in \mathbb{T}_C} val_{\mathbf{record}(t_C)}$$

where

$$val_{\mathbf{record}(t_C)} = \{\mathbf{gval}(r, \mathbf{record}\ t_C) \mid r : \mathbb{F} \rightharpoonup (val \cup val_{\mathcal{G}})\}$$

A record value is a pair $\mathbf{gval}(r, \mathbf{record}\ t_C)$ consisting of a partial function r from field names to values that describes the content of the record and a record type $\mathbf{record}\ t_C$.

Definition 6.17 (Set of Ghost Pointers and Arrays) We define the set of *ghost pointer and array values* as the union of the set of *global ghost references* and the set of *local ghost references*:

$$val_{\textbf{ghost-ptr+array}} \overset{def}{=} val_{\mathbf{gref}} \cup val_{\mathbf{lref}_{\mathcal{G}}}$$

As in *C-IL*, we distinguish strictly between local and global references.

Definition 6.18 (Set of Global Ghost References) We define the set of *global ghost references* as

$$val_{\mathbf{gref}} \overset{def}{=} \bigcup_{t \in (\mathbb{T} \cup \mathbb{T}_{\mathcal{G}}) \wedge (isptr(t) \vee isarray(t))} val_{\mathbf{gref}(t)}$$

with

$$\begin{aligned} val_{\mathbf{gref}(t)} \overset{def}{=} \ & \{\mathbf{gref}(v, S, t) \mid v \in \mathbb{V} \wedge S \in (\mathbb{N} \cup \mathbb{F})^*\} \\ & \cup \{\mathbf{gref}(a, S, t) \mid a \in \mathbb{N} \wedge S \in (\mathbb{N} \cup \mathbb{F})^*\} \\ & \cup \{\mathbf{gref}((a, t_C), S, t) \mid \exists t_C : a \in \mathbb{B}^{8 \cdot \theta.size_{ptr}} \wedge t_C \in \mathbb{T}_C \wedge S \in (\mathbb{N} \cup \mathbb{F})^*\} \end{aligned}$$

A global ghost reference $\mathbf{gref}(x, S, t)$ where $S \in (\mathbb{N} \cup \mathbb{F})^*$ is a finite sequence of subvariable selectors, and t is a pointer or array type can be any of the following three:

- a reference to a subvariable of a global ghost variable, $x \in \mathbb{V}$

  Here, x is the name of a global ghost variable in which the ghost subvariable resides.

- a reference to a subvariable of a ghost object allocated on the ghost heap, $x \in \mathbb{N}$

  In this case, x is the ghost heap address of a ghost object which was allocated to the heap.

- a reference to a ghost subvariable of an implementation pointer, $x \in (\mathbb{B}^{8 \cdot \theta.size_{ptr}} \times \mathbb{T}_C)$

  A reference to a ghost subvariable of an implementation struct is given when x is a pair of implementation memory address and a composite type name.

Definition 6.19 (Set of Local Ghost References) We define the set of *local ghost references* as

$$val_{\mathbf{lref}_\mathcal{G}} \overset{def}{=} \bigcup_{t \in (\mathbb{T} \cup \mathbb{T}_\mathcal{G}) \wedge (isptr(t) \vee isarray(t))} val_{\mathbf{lref}_\mathcal{G}(t)}$$

where

$$\begin{aligned} val_{\mathbf{lref}_\mathcal{G}(t)} \overset{def}{=} & \{\mathbf{lref}_\mathcal{G}(v, i, S, t) \mid v \in \mathbb{V} \wedge S \in (\mathbb{N} \cup \mathbb{V})^*\} \\ & \cup \{\mathbf{lref}_\mathcal{G}(((v, o), t_C), i, S, t) \mid v \in \mathbb{V} \wedge o \in \mathbb{N} \wedge t_C \in \mathbb{T}_C \wedge S \in (\mathbb{N} \cup \mathbb{F})^*\} \end{aligned}$$

A local ghost reference $\mathbf{lref}_\mathcal{G}(x, i, S, t)$ where $i \in \mathbb{N}$ is a stack frame number, $S \in (\mathbb{N} \cup \mathbb{F})^*$ is a finite sequence of subvariable selectors, and t is a pointer or array type is either

- a reference to a subvariable of a local ghost variable, $x \in \mathbb{V}$,

  A local ghost variable can be identified uniquely by its name x and the stack frame number i it resides in. S then simply describes a ghost subvariable inside that local ghost variable.

- or a reference to a subvariable of a local implementation variable, $x = ((v, o), t_C) \in (\mathbb{V} \times \mathbb{N}) \times \mathbb{T}_C$

  The local implementation variable this local ghost reference refers to is identified by the corresponding local reference $\mathbf{lref}((v, o), i, \mathbf{struct}\ t_C)$, which is easily constructed from x and i.

The intuition behind these definitions is that, in order to refer to some ghost field, we only need to describe the base object in which the ghost field resides and then describe the sequence of struct and array accesses that lead to the ghost field.

Definition 6.20 (Symbolic Ghost Function Value) We define the set of *symbolic ghost function values* as

$$val_{\mathbf{gfun}} \overset{def}{=} \{\mathbf{gfun}(f) \mid f \in \mathbb{F}_{name}\}$$

Since we do not support function pointer addresses for ghost functions (which would indeed be quite pointless since they do not have compiled code that can reside anywhere in the memory) this symbolic function value definition is sufficient in order to allow us to evaluate function expressions in order to perform function calls to ghost functions.

Definition 6.21 (Type of Values and ghost values) We extend the function $\tau : val \to \mathbb{T}$ from C_{IL} to

$$\tau_\mathcal{G} : val \cup val_\mathcal{G} \rightharpoonup \mathbb{T} \cup \mathbb{T}_\mathcal{G}$$

where

$$\tau_\mathcal{G}(x) = \begin{cases} \tau(x) & x \in val \\ t & x = \mathbf{gval}(a, t) \\ t & x = \mathbf{gref}(a, t) \\ t & x = \mathbf{lref}_\mathcal{G}(a, i, S, t) \\ \text{undefined} & \text{otherwise} \end{cases}$$

This function takes a value or ghost value and returns the type of that value.

Definition 6.22 (Zero-Predicate for Values and ghost values) We extend the zero-predicate for values from *C-IL* as follows:

$$zero_{\mathcal{G}} : params_{C\text{-}IL} \times (val \cup val_{\mathcal{G}}) \rightharpoonup bool$$

where

$$zero_{\mathcal{G}}(\theta, v) \equiv \begin{cases} a = 0^{size_\theta(t)} & v = \mathbf{val}(a, t) \\ z = 0 & v = \mathbf{gval}(z, \mathbf{math\_int}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Operators on ghost values

For most of the considered ghost values, we do not need any operators, we simply want to use them to store auxiliary implementation variables or results that can be computed from implementation variables using the usual C operators. Mathematical integer values are an exception for which we provide some definitions (which can be extended in the obvious way to provide more operations if necessary).

Definition 6.23 (Mathematical Integer Operators of *C-IL+$\mathcal{G}$*) We define the four operators

$$+ : val_{\mathbf{math\_int}} \times val_{\mathbf{math\_int}} \rightarrow val_{\mathbf{math\_int}}$$

$$- : val_{\mathbf{math\_int}} \times val_{\mathbf{math\_int}} \rightarrow val_{\mathbf{math\_int}}$$

$$* : val_{\mathbf{math\_int}} \times val_{\mathbf{math\_int}} \rightarrow val_{\mathbf{math\_int}}$$

$$/ : val_{\mathbf{math\_int}} \times val_{\mathbf{math\_int}} \rightarrow val_{\mathbf{math\_int}}$$

where

$$\mathbf{gval}(x, \mathbf{math\_int}) + \mathbf{gval}(y, \mathbf{math\_int}) = \mathbf{gval}(x + y, \mathbf{math\_int})$$

and

$$\mathbf{gval}(x, \mathbf{math\_int}) - \mathbf{gval}(y, \mathbf{math\_int}) = \mathbf{gval}(x - y, \mathbf{math\_int})$$

and

$$\mathbf{gval}(x, \mathbf{math\_int}) * \mathbf{gval}(y, \mathbf{math\_int}) = \mathbf{gval}(x * y, \mathbf{math\_int})$$

and

$$\mathbf{gval}(x, \mathbf{math\_int}) / \mathbf{gval}(y, \mathbf{math\_int}) = \mathbf{gval}(\lfloor x/y \rfloor, \mathbf{math\_int})$$

In the remaining definitions, we will just assume partial functions $op_{1\mathcal{G}}$ and $op_{2\mathcal{G}}$ that map operator symbols to functions that compute results of applying these operators to ghost values. Since we have only defined a few mathematical operators on mathematical integer values, the reader may assume that $op_{1\mathcal{G}}$ is undefined and that the domain of $op_{2\mathcal{G}}$ is $\{+, -, *, /\} \subset \mathbb{O}_2$.

Type Cast for ghost values

For most ghost values, type casting does not make much sense (e.g. state snapshot, maps, records, ghost pointers, arrays). Exceptions are mathematical integers and generic pointers for which a type cast from appropriate values should be possible.

Definition 6.24 (Type Cast Function for *C-IL+$\mathcal{G}$*) We define the function

$$cast_{\mathcal{G}} : (val \cup val_{\mathcal{G}}) \times (\mathbb{T} \cup \mathbb{T}_{\mathcal{G}}) \rightharpoonup val \cup val_{\mathcal{G}}$$

as

$$cast_{\mathcal{G}}(v,t) = \begin{cases} \theta.cast(v,t) & v \in val \wedge t \in \mathbb{T} \\ \mathbf{gval}(\langle x \rangle, \mathbf{math\_int}) & v = \mathbf{val}(x, \mathbf{u}i) \in val_{\mathbf{prim}} \wedge t = \mathbf{math\_int} \\ \mathbf{gval}([x], \mathbf{math\_int}) & v = \mathbf{val}(x, \mathbf{i}i) \in val_{\mathbf{prim}} \wedge t = \mathbf{math\_int} \\ \mathbf{gval}(v, \mathbf{obj}) & v \in val_{\mathbf{ptr+array}} \cup val_{\mathbf{ghost\text{-}ptr+array}} \wedge t = \mathbf{obj} \\ p & v = \mathbf{gval}(p, \mathbf{obj}) \in val_{\mathbf{obj}} \wedge t = \tau_{\mathcal{G}}(p) \\ \text{undefined} & \text{otherwise} \end{cases}$$

It is currently unclear whether this type cast function is sufficient – if it turns out not to be, it should be extended to allow type casts between more types; usually in a straight-forward way.

6.2 Ghost Code

Similar to ghost state, ghost code comes in different flavors:

- regular *C-IL* statements operating on ghost state (i.e. assignment and gotos),
- calls to ghost functions,
- added ghost parameters on *C-IL* function calls, and
- allocation of ghost memory.

The goal is to define ghost code in such a way that it is not hard to show that a program annotated with ghost code simulates the same program without ghost code. What we would like to have is a clean separation between execution of ghost function calls and execution of ordinary function calls (possibly annotated with ghost code) – so that we can extract the implementation part of a *C-IL+$\mathcal{G}$* execution (which is a *C-IL* execution) easily. We restrict function calls in such a way that ghost functions cannot call implementation functions, whereas implementation functions can call both ghost and annotated implementation functions. This restriction occurs later in the definition of the transition function of *C-IL+$\mathcal{G}$* but may be helpful to keep in mind when looking at the following definitions. In the following, we formally introduce ghost code by defining *C-IL+$\mathcal{G}$* programs. This involves first defining ghost expressions followed by ghost statements and an extended version of *C-IL* statements with ghost parameters on function calls before we define *C-IL+$\mathcal{G}$* function tables and *C-IL+$\mathcal{G}$* programs.

6.2.1 Expressions

Definition 6.25 (Ghost Expressions) We inductively define the set of *ghost expressions* $\mathbb{E}_{\mathcal{G}}$ as follows:

- Constants: $c \in val \cup val_{\mathcal{G}} \Rightarrow c \in \mathbb{E}_{\mathcal{G}}$

  The difference to *C-IL* expressions is that constants may additionally be ghost values.

- Variable names: $v \in \mathbb{V} \Rightarrow v \in \mathbb{E}_{\mathcal{G}}$

  Identical to the same case of the *C-IL* expression definition.

- Function names: $f \in \mathbb{F}_{name} \Rightarrow f \in \mathbb{E}_{\mathcal{G}}$

  Identical to the same case of the *C-IL* expression definition.

- Unary operation on expression: $e \in \mathbb{E}_{\mathcal{G}} \wedge \ominus \in \mathbb{O}_1 \Rightarrow \ominus e \in \mathbb{E}_{\mathcal{G}}$

  Structurally identical to the corresponding *C-IL* expression, allows ghost subexpression.

- Binary operation on expressions: $e_0, e_1 \in \mathbb{E}_{\mathcal{G}} \wedge \oplus \in \mathbb{O}_2 \Rightarrow (e_0 \oplus e_1) \in \mathbb{E}_{\mathcal{G}}$

  Structurally identical to the corresponding *C-IL* expression, allows ghost subexpressions.

- Conditional: $e, e_0, e_1 \in \mathbb{E}_{\mathcal{G}} \Rightarrow (e \; ? \; e_0 : e_1) \in \mathbb{E}_{\mathcal{G}}$

  Structurally identical to the corresponding *C-IL* expression, allows ghost subexpressions.

- Type cast: $t \in (\mathbb{T}_Q \cup \mathbb{T}_{Q\mathcal{G}}), e \in \mathbb{E}_{\mathcal{G}} \Rightarrow (t)e \in \mathbb{E}_{\mathcal{G}}$

  Structurally identical to the corresponding *C-IL* expression, allows ghost subexpression and type.

- Dereferencing pointer: $e \in \mathbb{E}_{\mathcal{G}} \Rightarrow *(e) \in \mathbb{E}_{\mathcal{G}}$

  Structurally identical to the corresponding *C-IL* expression, allows ghost subexpression.

- Address of expression: $e \in \mathbb{E}_{\mathcal{G}} \Rightarrow \&(e) \in \mathbb{E}_{\mathcal{G}}$

  Structurally identical to the corresponding *C-IL* expression, allows ghost subexpression.

- Field access: $e \in \mathbb{E}_{\mathcal{G}} \wedge f \in \mathbb{F} \Rightarrow (e).f \in \mathbb{E}_{\mathcal{G}}$

  Structurally identical to the corresponding *C-IL* expression, allows ghost subexpression.

- Size of Type: $t \in \mathbb{T}_Q \Rightarrow \textbf{sizeof}(t) \in \mathbb{E}_{\mathcal{G}}$

  Identical to the same case of the *C-IL* expression definition. Note that we do not allow taking the size of a ghost type since that would be quite meaningless.

- Size of Expression: $e \in \mathbb{E} \Rightarrow \textbf{sizeof}(e) \in \mathbb{E}_{\mathcal{G}}$

  Identical to the same case of the *C-IL* expression definition. Note that we explicitly do not allow taking the size of a ghost expression.

- Map and ghost array access: $e, e' \in \mathbb{E}_{\mathcal{G}} \Rightarrow e[e'] \in \mathbb{E}_{\mathcal{G}}$

  Note that using brackets for array access is only defined for ghost arrays in the semantics of *C-IL+$\mathcal{G}$* since in *C-IL*, we model array access by means of pointer arithmetics. Map access means retrieving the value of the map described by e for a given element described by e' of its domain.

- Lambda expression: $t \in \mathbb{T}_Q \cup \mathbb{T}_{Q\mathcal{G}} \wedge v \in \mathbb{V} \wedge e \in \mathbb{E}_{\mathcal{G}} \Rightarrow \textbf{lambda}(t\ v; e) \in \mathbb{E}_{\mathcal{G}}$

  In order to assign map values, we provide a lambda expression that results in a map from elements of type t to elements of the type of e where the lambda parameter variable v may occur as a free variable in e.

- Record update: $e, e' \in \mathbb{E}_{\mathcal{G}} \wedge f \in \mathbb{F} \Rightarrow e\{f := e'\} \in \mathbb{E}_{\mathcal{G}}$

  C-IL+$\mathcal{G}$ provides an expression to update a field of a record. Note that in the semantics we do not provide an expression that assigns all fields of a record at once. This could, however, be added as syntactic sugar.

- State-snapshot: $\textbf{current\_state} \in \mathbb{E}_{\mathcal{G}}$

  In order to take a state-snapshot of the current state, *C-IL+$\mathcal{G}$* provides an expression. This is in fact the only way to obtain a state-snapshot.

- Use a state-snapshot to evaluate an expression: $e, e' \in \mathbb{E}_{\mathcal{G}} \Rightarrow \textbf{in\_state}(e,e') \in \mathbb{E}_{\mathcal{G}}$

  Given a *C-IL+$\mathcal{G}$* expression e that evaluates to a state-snapshot, e' is evaluated in the state specified by e'.

Note that, since the structure of the set of ghost expression encompasses that of the set of *C-IL* expressions, the proof that $\mathbb{E} \subset \mathbb{E}_{\mathcal{G}}$ is trivial.

6.2.2 Statements

Definition 6.26 (*C-IL* Statements Extended with Ghost Parameters on Function Call) We define the set $\mathbb{S}'$ which is the set of C-IL *statements extended with ghost parameters on function call*, or shorter *extended* C-IL *statements* as follows:

- Assignment: $e_0, e_1 \in \mathbb{E} \Rightarrow (e_0{=}e_1) \in \mathbb{S}'$
- Goto: $l \in \mathbb{N} \Rightarrow \textbf{goto}\ l \in \mathbb{S}''$
- If-Not-Goto: $e \in \mathbb{E}\ \wedge\ l \in \mathbb{N} \Rightarrow \textbf{ifnot}\ e\ \textbf{goto}\ l \in \mathbb{S}'$
- Function Call: $e_0, e \in \mathbb{E}, E \in \mathbb{E}^*, E' \in \mathbb{E}_{\mathcal{G}}^* \Rightarrow (e_0{=}\textbf{call}\ e(E,E')) \in \mathbb{S}'$
- Procedure Call: $e \in \mathbb{E}, E \in \mathbb{E}^*, E' \in \mathbb{E}_{\mathcal{G}}^* \Rightarrow \textbf{call}\ e(E,E') \in \mathbb{S}'$
- Return: $e \in \mathbb{E} \Rightarrow \textbf{return}\ e \in \mathbb{S}'$ and $\textbf{return} \in \mathbb{S}'$

Note that this definition is structurally identical to the definition of *C-IL* statements except for the fact that function calls now include a second list of parameters E' which is used for ghost parameter passing at function call. The reason for this is that these statements are used to represent the implementation *C-IL* program that is part of a *C-IL+$\mathcal{G}$* program. The definition for statements of ghost code follows just below.

Definition 6.27 (Ghost Statements) To represent – and visually distinguish – ghost code, we define the set of *ghost statements* $\mathbb{S}_{\mathcal{G}}$ as follows:

- Assignment: $e_0, e_1 \in \mathbb{E}_{\mathcal{G}} \Rightarrow \textbf{ghost}(e_0{=}e_1) \in \mathbb{S}_{\mathcal{G}}$
- Goto: $l \in \mathbb{N} \Rightarrow \textbf{ghost}(\textbf{goto}\ l) \in \mathbb{S}_{\mathcal{G}}$
- If-Not-Goto: $e \in \mathbb{E}_{\mathcal{G}}\ \wedge\ l \in \mathbb{N} \Rightarrow \textbf{ghost}(\textbf{ifnot}\ e\ \textbf{goto}\ l) \in \mathbb{S}_{\mathcal{G}}$
- Function Call: $e_0, e \in \mathbb{E}_{\mathcal{G}}, E \in \mathbb{E}_{\mathcal{G}}^* \Rightarrow \textbf{ghost}(e_0{=}\textbf{call}\ e(E)) \in \mathbb{S}_{\mathcal{G}}$
- Procedure Call: $e \in \mathbb{E}_{\mathcal{G}}, E \in \mathbb{E}_{\mathcal{G}}^* \Rightarrow \textbf{ghost}(\textbf{call}\ e(E)) \in \mathbb{S}_{\mathcal{G}}$
- Return: $e \in \mathbb{E}_{\mathcal{G}} \Rightarrow \textbf{ghost}(\textbf{return}\ e) \in \mathbb{S}_{\mathcal{G}}$ and $\textbf{ghost}(\textbf{return}) \in \mathbb{S}_{\mathcal{G}}$
- Ghost Allocation: $e \in \mathbb{E}_{\mathcal{G}} \wedge t \in \mathbb{T} \cup \mathbb{T}_{\mathcal{G}} \Rightarrow \textbf{ghost}(e{=}\textbf{allocghost}(t)) \in \mathbb{S}_{\mathcal{G}}$

Note that while we do not consider a heap abstraction for *C-IL*, we do consider an infinite heap for ghost objects. This is why we provide a ghost allocation statement that allocates a new ghost object of type t on the ghost heap. (For *C-IL*, such an abstraction can be implemented by *C-IL* code that explicitly performs memory management to establish the notion of a heap.)

6.2.3 Programs

Definition 6.28 (Annotated *C-IL* Function Table Entry) In order to describe ghost parameters annotated to *C-IL* functions, we use the following function table definition for implementation functions in *C-IL+$\mathcal{G}$*: An *annotated function table entry*

$$fte = (fte.rettype, fte.npar, fte.ngpar, fte.\mathcal{V}, fte.\mathcal{V}_{\mathcal{G}}, fte.\mathcal{P}) \in FunT'$$

consists of

- $fte.rettype \in \mathbb{T}_{Q}$ – type of the return value

  Identical to the *C-IL* definition.
- $fte.npar \in \mathbb{N}$ – number of implementation parameters

  Identical to the *C-IL* definition.
- $fte.ngpar \in \mathbb{N}$ – number of ghost parameters

  Describes how many ghost parameters must be passed to the function.

- $fte.\mathcal{V} \in (\mathbb{V} \times \mathbb{T}_Q)^*$ – implementation parameter and local variable declarations

  Identical to the *C-IL* definition.

- $fte.\mathcal{V}_\mathcal{G} \in (\mathbb{V} \times (\mathbb{T}_Q \cup \mathbb{T}_{Q\mathcal{G}}))^*$ – ghost parameter and local ghost variable declarations

  A list of pairs of variable names and qualified ghost or implementation types that declares ghost parameters and local ghost variables of the function. Analogous to the *C-IL* definition, the first $fte.ngpar$ entries describe the declared ghost parameters while the remaining entries declare local ghost variables. Note that ghost variables and parameters may use implementation types in addition to ghost types and that for a well-defined configuration, the declared ghost variable and parameter names should be disjoint from the implementation variable and parameter names.

- $fte.\mathcal{P} \in ((\mathbb{S}' \cup \mathbb{S}_\mathcal{G})^* \cup \{\textbf{extern}\})$ – function body

  The function body of an implementation function of *C-IL+$\mathcal{G}$* may contain both annotated *C-IL* statements and ghost statements.

Definition 6.29 (Ghost Function Table Entry) A *ghost function table entry*

$$fte = (fte.rettype, fte.ngpar, fte.\mathcal{V}_\mathcal{G}, fte.\mathcal{P}) \in FunT_\mathcal{G}$$

consists of the following:

- $fte.rettype \in \mathbb{T}_Q \cup \mathbb{T}_{Q\mathcal{G}}$ - type of the return value
- $fte.ngpar \in \mathbb{N}$ - number of ghost parameters
- $fte.\mathcal{V}_\mathcal{G} \in (\mathbb{V} \times (\mathbb{T}_Q \cup \mathbb{T}_{Q\mathcal{G}}))^*$ - ghost parameter and local ghost variable declarations (beginning with parameters)
- $fte.\mathcal{P} : (\mathbb{S}_\mathcal{G}^* \cup \{\textbf{extern}\})$ - function body

Note that a ghost function only has ghost parameters and uses only ghost statements but it may use implementation types for parameters or local variables.

With these definitions we are now able to define *C-IL+$\mathcal{G}$* programs.

Definition 6.30 (*C-IL+$\mathcal{G}$* Program) A *C-IL+$\mathcal{G}$* program

$$\pi = (\pi.\mathcal{V}_G, \pi.\mathcal{V}_{G\mathcal{G}}, \pi.T_F, \pi.T_{F\mathcal{G}}, \pi.\mathcal{F}, \pi.\mathcal{F}_\mathcal{G}) \in prog_{C\text{-}IL+\mathcal{G}}$$

is described by the following:

- $\pi.\mathcal{V}_G : (\mathbb{V} \times \mathbb{T}_Q)^*$ - a list of implementation global variable declarations

  Identical to the definition of *C-IL* programs.

- $.\mathcal{V}_{G\mathcal{G}} : (\mathbb{V} \times (\mathbb{T}_Q \cup \mathbb{T}_{Q\mathcal{G}}))^*$ - a list of global ghost variable declarations

  This list declares all global ghost variables of the *C-IL+$\mathcal{G}$* program. Declared ghost and implementation variable names must be disjoint.

- $\pi.T_F : \mathbb{T}_C \rightharpoonup (\mathbb{F} \times \mathbb{T}_Q)^*$ - a type table for implementation fields of struct types

  Identical to the definition of *C-IL* program. Note that pure ghost structs or record types will not have any implementation fields declared by this function. Thus, every composite type name in the domain of this function is an implementation struct type.

- $\pi.T_{F\mathcal{G}} : \mathbb{T}_C \rightharpoonup (\mathbb{F} \times (\mathbb{T}_Q \cup \mathbb{T}_{Q\mathcal{G}}))^*$ - a type table for ghost fields of struct types

  This function returns for a given composite type name all ghost field declarations. Note that in order to have a valid *C-IL+$\mathcal{G}$* program, we must require that for a given composite type t_C the declared implementation field names and declared ghost field names are disjoint.

- $\pi.\mathcal{F} : \mathbb{F}_{name} \rightharpoonup$ *FunT'* - a function table for annotated implementation functions

  The annotated function table for the implementation *C-IL* program.

- $\pi.\mathcal{F}_\mathcal{G} : \mathbb{F}_{name} \rightharpoonup FunT_\mathcal{G}$ - a function table for ghost functions

  The function table for ghost functions maps declared ghost function names to ghost function table entries. Similar to struct type name declarations, we must enforce $dom(\pi.\mathcal{F}) \cap dom(\pi.\mathcal{F}_\mathcal{G}) = \emptyset$ in order to have a meaningful *C-IL+$\mathcal{G}$* program.

Auxiliary Definitions

In the following, we present auxiliary definitions for *C-IL+$\mathcal{G}$* programs and parts thereof. Most of these definitions are very similar to auxiliary definitions from *C-IL* semantics and are just the old definitions extended to ghost programs in the obvious way.

Definition 6.31 (Implementation Function Table) For a *C-IL+$\mathcal{G}$* program π and *C-IL* environment parameters θ, we define the *implementation function table*

$$\mathcal{F}_\pi^\theta \overset{def}{=} \pi.\mathcal{F} \cup \theta.intrinsics$$

analogous to the *C-IL* definition.

Definition 6.32 (Combined *C-IL+$\mathcal{G}$* Function Table) For a *C-IL+$\mathcal{G}$* program π and *C-IL* environment parameters θ, we define the *combined* C-IL+$\mathcal{G}$ *function table* as union of the implementation function table and the ghost function table:

$$\mathcal{F}_{C\text{-}IL+\mathcal{G}} \overset{def}{=} \mathcal{F}_\pi^\theta \cup \pi.\mathcal{F}_\mathcal{G}$$

Definition 6.33 (Set of Declared Ghost Variables) We define the function

$$decl_\mathcal{G} : (\mathbb{V} \times (\mathbb{T}_Q \cup \mathbb{T}_{Q\mathcal{G}}))^* \to 2^\mathbb{V}$$

that, given a ghost variable declaration list, returns the set of declared variables as

$$decl_\mathcal{G}(\mathcal{V}) \overset{def}{=} \begin{cases} \{v\} \cup decl_\mathcal{G}(\mathcal{V}') & \mathcal{V} = \mathcal{V}' \circ (v, t) \\ \emptyset & \mathcal{V} = \varepsilon \end{cases}$$

Definition 6.34 (Set of Declared Ghost Field Names) We define the function

$$decl_{\mathcal{G}} : (\mathbb{F} \times (\mathbb{T}_Q \cup \mathbb{T}_{Q\mathcal{G}}))^* \to 2^{\mathbb{F}}$$

that, given a ghost field declaration list, returns the set of declared fields as

$$decl_{\mathcal{G}}(F) \overset{def}{=} \begin{cases} \{f\} \cup decl_{\mathcal{G}}(F') & F = F' \circ (f, t) \\ \emptyset & F = \varepsilon \end{cases}$$

Definition 6.35 (Type of Variable/Field in a Ghost Declaration List) We define the functions

$$\tau_{V\mathcal{G}} : \mathbb{V} \times (\mathbb{V} \times (\mathbb{T}_Q \cup \mathbb{T}_{Q\mathcal{G}}))^* \to \mathbb{T}_Q \cup \mathbb{T}_{Q\mathcal{G}}$$

and

$$\tau_{F\mathcal{G}} : \mathbb{F} \times (\mathbb{F} \times (\mathbb{T}_Q \cup \mathbb{T}_{Q\mathcal{G}}))^* \to \mathbb{T}_Q \cup \mathbb{T}_{Q\mathcal{G}}$$

that, given a variable (respectively, field) and an appropriate ghost declaration list, returns the type of the variable (field) in that declaration list.

$$\tau_{V\mathcal{G}}(v,\mathcal{V}) = \begin{cases} t & \mathcal{V} \overset{def}{=} (v, t) \circ \mathcal{V}' \\ \tau_{V\mathcal{G}}(v, \mathcal{V}') & \mathcal{V} = (v', t) \circ \mathcal{V}' \wedge v' \neq v \\ (\emptyset, \textbf{void}) & \mathcal{V} = \varepsilon \end{cases}$$

$$\tau_{F\mathcal{G}}(f,T) \overset{def}{=} \begin{cases} t & T = (f, t) \circ T' \\ \tau_{F\mathcal{G}}(f, T') & T = (f', t) \circ T' \wedge f' \neq f \\ (\emptyset, \textbf{void}) & T = \varepsilon \end{cases}$$

6.3 Operational Semantics

We proceed by defining the state of *C-IL+$\mathcal{G}$* in terms of *C-IL+$\mathcal{G}$* configurations and the transition function of *C-IL+$\mathcal{G}$* for which we define expression evaluation of *C-IL+$\mathcal{G}$* expressions. Note that *C-IL+$\mathcal{G}$* does not need any additional environment parameters and simply makes use of *C-IL*'s environment parameters to give semantics of the *C-IL* part of *C-IL+$\mathcal{G}$*.

6.3.1 Configurations

Definition 6.36 (*C-IL+$\mathcal{G}$* Stack Frame) A stack frame of *C-IL+$\mathcal{G}$*

$$s = (s.\mathcal{M}_{\mathcal{E}}, s.\mathcal{M}_{\mathcal{E}\mathcal{G}}, s.rds, s.f, s.loc) \in frame_{\textit{C-IL+}\mathcal{G}}$$

consists of

- $s.\mathcal{M}_{\mathcal{E}} : \mathbb{V} \to (\mathbb{B}^8)^*$ – local variables and parameters memory

  Identical to *C-IL*, used to model values of implementation local variables and parameters.

- $s.\mathcal{M}_{\mathcal{E}\mathcal{G}} : (\mathbb{V} \times \mathbb{N}) \times \mathbb{T}_C \cup \mathbb{V} \rightarrow val_{\mathcal{M}_\mathcal{G}}$ – local ghost variables and ghost parameters memory

  A local ghost memory that describes the content of local ghost variables and ghost parameters as well as the content of ghost fields of local implementation variables.

- $s.rds \in val_{\mathbf{ptr}} \cup val_{\mathbf{lref}} \cup val_{\mathbf{gref}} \cup val_{\mathbf{lref}_\mathcal{G}} \cup \{\perp\}$ – return value destination

  The return value destination which describes where the return value of the function call has to be stored when a called function returns to this stack frame. Note that ghost references will only occur in frames of ghost functions, for implementation functions we only use implementation pointers.

- $s.f \in \mathbb{F}_{name}$ – function name

  Identical to the *C-IL* definition. May, however, refer to declared ghost functions in addition to implementation functions.

- $s.loc \in \mathbb{N}$ – location counter

  Identical to the *C-IL* definition.

Definition 6.37 (Sequential *C-IL+$\mathcal{G}$*-Configuration) A *sequential* C-IL+$\mathcal{G}$ *configuration*

$$c = (c.\mathcal{M}, c.\mathcal{M}_\mathcal{G}, c.s, c.nf_\mathcal{G}) \in conf_{C\text{-}IL+\mathcal{G}}$$

consists of

- $c.\mathcal{M} : \mathbb{B}^{size_{ptr}} \rightharpoonup \mathbb{B}^8$ – global byte-addressable memory

  The same global byte-addressable memory that occurs in *C-IL* configurations.

- $c.\mathcal{M}_\mathcal{G} : (\mathbb{B}^{8 \cdot \theta.size_{ptr}} \times \mathbb{T}_C) \cup \mathbb{N} \cup \mathbb{V} \rightharpoonup val_{\mathcal{M}_\mathcal{G}}$ – global ghost memory of structured ghost values

  This memory maps implementation pointers, ghost heap addresses and global ghost variable names to structured ghost values which can be used to evaluate ghost subvariables of the aforementioned.

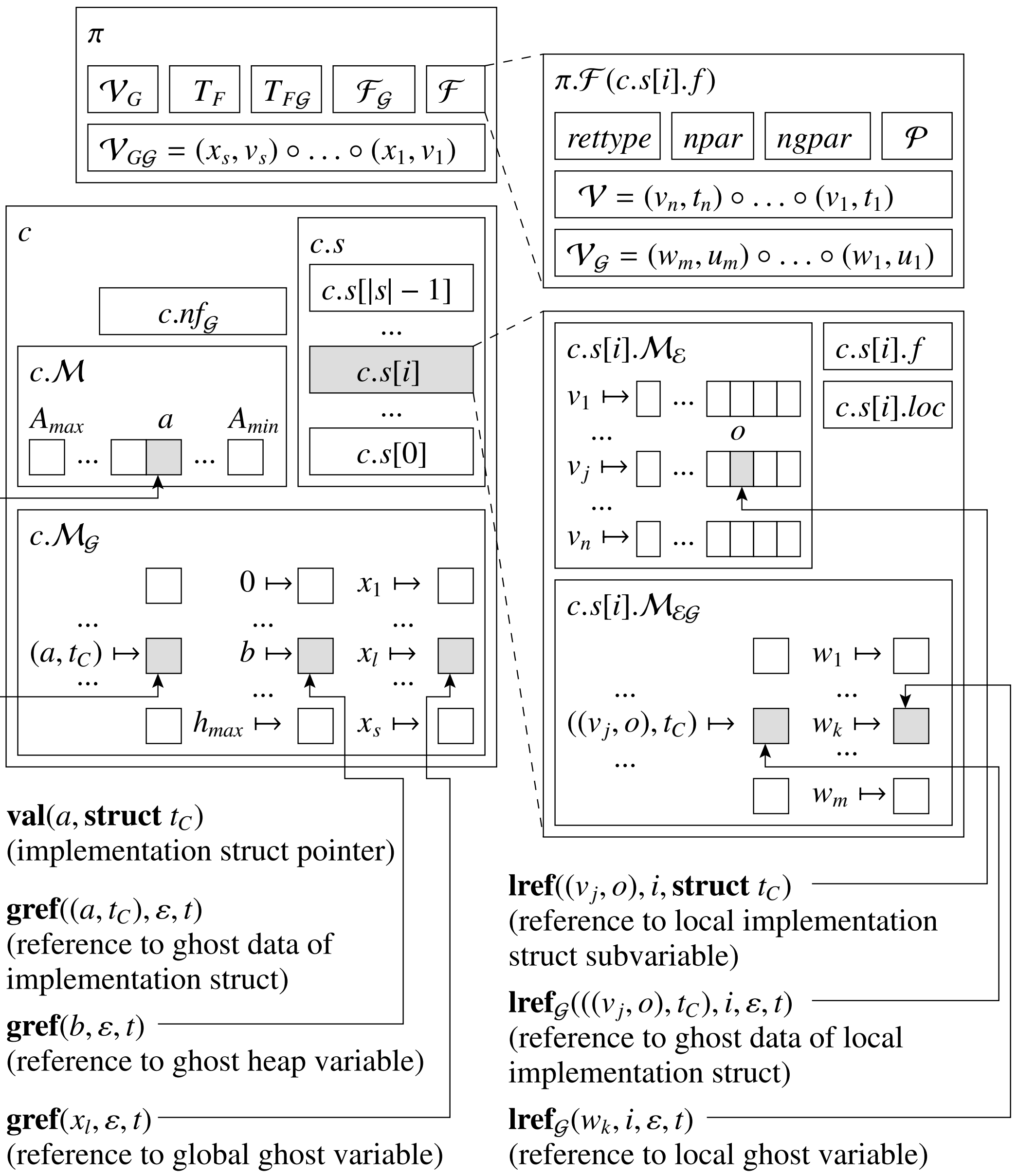
- $c.s : frame^*_{C\text{-}IL+\mathcal{G}}$ – stack

  The stack of a *C-IL+$\mathcal{G}$* configuration is composed of *C-IL+$\mathcal{G}$* stack frames.

- $c.nf_\mathcal{G} \in \mathbb{N}$ – the next free address on the ghost heap

  In order to maintain the ghost heap, we count the number of heap variables allocated on the ghost heap. Ghost heap variables are addressed by natural numbers in the order of their allocation.

Figure 6.4 illustrates where the values pointed to by different kinds of *C-IL+$\mathcal{G}$* pointers can be found in a *C-IL+$\mathcal{G}$* configuration.



π
$\mathcal{V}_G$
T_F
$T_{F\mathcal{G}}$
$\mathcal{F}_\mathcal{G}$
$\mathcal{F}$
$\mathcal{V}_{G\mathcal{G}} = (x_s, v_s) \circ \ldots \circ (x_1, v_1)$
$\pi.\mathcal{F}(c.s[i].f)$
$rettype$
$npar$
$ngpar$
$\mathcal{P}$
$\mathcal{V} = (v_n, t_n) \circ \ldots \circ (v_1, t_1)$
$\mathcal{V}_\mathcal{G} = (w_m, u_m) \circ \ldots \circ (w_1, u_1)$
c
$c.s$
$c.s[|s|-1]$
$c.s[i]$
$c.s[0]$
$c.nf_\mathcal{G}$
$c.\mathcal{M}$
A_{max}
a
A_{min}
$c.\mathcal{M}_\mathcal{G}$
$0 \mapsto$
$x_1 \mapsto$
$(a, t_C) \mapsto$
$b \mapsto$
$x_l \mapsto$
$h_{max} \mapsto$
$x_s \mapsto$
$c.s[i].\mathcal{M}_\mathcal{E}$
$v_1 \mapsto$
o
$v_j \mapsto$
$v_n \mapsto$
$c.s[i].f$
$c.s[i].loc$
$c.s[i].\mathcal{M}_{\mathcal{E}\mathcal{G}}$
$w_1 \mapsto$
$((v_j, o), t_C) \mapsto$
$w_k \mapsto$
$w_m \mapsto$
val$(a,$ **struct** $t_C)$
(implementation struct pointer)
gref$((a, t_C), \varepsilon, t)$
(reference to ghost data of implementation struct)
gref(b, ε, t)
(reference to ghost heap variable)
gref(x_l, ε, t)
(reference to global ghost variable)
lref$((v_j, o), i,$ **struct** $t_C)$
(reference to local implementation struct subvariable)
lref$_\mathcal{G}(((v_j, o), t_C), i, \varepsilon, t)$
(reference to ghost data of local implementation struct)
lref$_\mathcal{G}(w_k, i, \varepsilon, t)$
(reference to local ghost variable)

Figure 6.4: Where references and pointers of *C-IL+$\mathcal{G}$* point to. Here, A_{min} and A_{max} are the minimal and, respectively, maximal address of the implementation memory $c.\mathcal{M}$, $h_{max} = c.nf_\mathcal{G} - 1$ is the newest ghost heap variable, v_j is the j-th local variable declared in function $c.s[i].f$, w_k is the k-th local ghost variable declared in function $c.s[i].f$, b is the b-th ghost heap variable, and x_l is the l-the declared global ghost variable.

Definition 6.38 (Concurrent *C-IL+$\mathcal{G}$*-Configuration) A *concurrent* C-IL+$\mathcal{G}$ *configuration*

$$c = (c.\mathcal{M}, c.\mathcal{M}_{\mathcal{G}}, c.Th, c.nf_{\mathcal{G}}) \in conf_{CC\text{-}IL+\mathcal{G}}$$

consists of

- $c.\mathcal{M} : \mathbb{B}^{size_{ptr}} \to \mathbb{B}^8$ – shared global byte addressable memory
- $c.\mathcal{M}_{\mathcal{G}} : (\mathbb{B}^{8 \cdot \theta.size_{ptr}} \times \mathbb{T}_C) \cup \mathbb{N} \cup \mathbb{V} \to val_{\mathcal{M}_{\mathcal{G}}}$ – shared global ghost memory
- $c.Th : \mathbb{N} \rightharpoonup frame^*_{C\text{-}IL+\mathcal{G}}$ – mapping of thread identifiers to *C-IL+$\mathcal{G}$* stacks
- $c.nf_{\mathcal{G}} \in \mathbb{N}$ – the next free address on the shared ghost heap

Auxiliary Definitions

Concerning configurations, we make the following auxiliary definitions.

Definition 6.39 (Top-Most Stack Frame) We define the top-most stack frame of a *C-IL+$\mathcal{G}$* configuration

$$stack_{top} : conf_{C\text{-}IL+\mathcal{G}} \to frame_{C\text{-}IL+\mathcal{G}}$$

as

$$stack_{top}(c) = c.s[|c.s| - 1]$$

Definition 6.40 (Stack Index where Ghost Frames Begin) We define a function that calculates the first index on the stack where a ghost frame resides – if no such index exists the function returns the value $\bot$. We define

$$si_{\mathcal{G}} : prog_{C\text{-}IL+\mathcal{G}} \times conf_{C\text{-}IL+\mathcal{G}} \to \mathbb{N} \cup \{\bot\}$$

where

$$si_{\mathcal{G}}(\pi,c) = \begin{cases} \mathbf{min}\{i \in \{0, \ldots, |c.s| - 1\} \mid c.s[i].f \in \pi.\mathcal{F}_{\mathcal{G}}\} & \exists i : c.s[i].f \in \pi.\mathcal{F}_{\mathcal{G}} \\ \bot & \text{otherwise} \end{cases}$$

Note that, since ghost functions are restricted by the semantics to only be able to call other ghost functions, all subsequent frames will also be ghost frames for any *C-IL+$\mathcal{G}$* configuration that occurs in a valid *C-IL+$\mathcal{G}$* execution.

We extend the functions *read* and *write* from *C-IL* in order to have similar functions for *C-IL+$\mathcal{G}$*.

Definition 6.41 (Reading a Value from a *C-IL+$\mathcal{G}$* Configuration) We define the function

$$read_{\mathcal{G}} : params_{C\text{-}IL} \times conf_{C\text{-}IL+\mathcal{G}} \times (val \cup val_{\mathcal{G}}) \rightharpoonup val_{\mathcal{M}_{\mathcal{G}}}$$

where

$$read_{\mathcal{G}}(\theta,\pi,c,x) = \begin{cases} read(\theta, C\text{-}IL(\pi, c), x) & x \in val \\ read_{val_{\mathcal{M}_{\mathcal{G}}}}(c.\mathcal{M}_{\mathcal{G}}(y), S) & x = \mathbf{gref}(y, S, t) \\ read_{val_{\mathcal{M}_{\mathcal{G}}}}(c.s[i].\mathcal{M}_{\mathcal{E}\mathcal{G}}(y), S) & x = \mathbf{lref}_{\mathcal{G}}(y, i, S, t) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that this function returns structured ghost values which, however, include *C-IL* values ($val \subset val_{\mathcal{M}_{\mathcal{G}}}$). Thus, we can use the function *read* from *C-IL* to read from the implementation memories of the projected *C-IL* configuration $C\text{-}IL(\pi, c)$ that corresponds to the *C-IL*+$\mathcal{G}$ configuration c.

The formal definition of the projection function $C\text{-}IL : conf_{C\text{-}IL+\mathcal{G}} \rightarrow conf_{C\text{-}IL}$ which converts a *C-IL*+$\mathcal{G}$ configuration to a corresponding *C-IL* configuration by extracting everything related to ghost state and code is given in section 6.4.1 before we discuss simulation between *C-IL* and *C-IL*+$\mathcal{G}$. While this function could have been introduced here, it appears more sensible to give all definitions related to projecting *C-IL*+$\mathcal{G}$ to *C-IL* in a single place later.

Definition 6.42 (Writing a Value to a *C-IL*+$\mathcal{G}$ Configuration) We define the function

$$write_{\mathcal{G}} : params_{C\text{-}IL} \times conf_{C\text{-}IL+\mathcal{G}} \times (val \cup val_{\mathcal{G}}) \times val_{\mathcal{M}_{\mathcal{G}}} \rightharpoonup conf_{C\text{-}IL+\mathcal{G}}$$

as

$$write_{\mathcal{G}}(\theta,\pi,c,x,y) = \begin{cases} merge\text{-}impl\text{-}ghost(c, write(\theta, C\text{-}IL(\pi, c), x, y)) & x \in val \wedge y \in val \\ c[\mathcal{M}_{\mathcal{G}} := c.\mathcal{M}_{\mathcal{G}}[z := write_{val_{\mathcal{M}_{\mathcal{G}}}}(c.\mathcal{M}_{\mathcal{G}}(z), S, y)]] & x = \mathbf{gref}(z, S, t) \\ c' & x = \mathbf{lref}_{\mathcal{G}}(z, i, S, t) \\ \text{undefined} & \text{otherwise} \end{cases}$$

where *merge-impl-ghost* is a function that updates all implementation memories of a *C-IL*+$\mathcal{G}$ configuration based on a given *C-IL* configuration (formally defined below) and c' is a *C-IL*+$\mathcal{G}$ configuration which is identical to c except for $c'.s[i].\mathcal{M}_{\mathcal{E}\mathcal{G}}(z) = write_{val_{\mathcal{M}_{\mathcal{G}}}}(c.s[i].\mathcal{M}_{\mathcal{E}\mathcal{G}}(z), S, y)$.

In the previous definition, we have used a function $merge\text{-}impl\text{-}ghost(c, c')$ to merge the implementation memories of configuration c' to configuration c. The intuition behind this is that we would like to reuse the *C-IL* definition of the *write*-function in order to perform writes to implementation variables by converting the *C-IL*+$\mathcal{G}$ configuration to a *C-IL* configuration in which we perform the write. Now, to obtain a *C-IL*+$\mathcal{G}$ configuration in which the implementation variable was written, we need to take all memory components of the updated *C-IL* configuration and "merge" them into the original *C-IL*+$\mathcal{G}$ configuration.

Definition 6.43 (Merging a *C-IL*+$\mathcal{G}$ Configuration and a *C-IL* Configuration) The function

$$merge\text{-}impl\text{-}ghost : conf_{C\text{-}IL+\mathcal{G}} \times conf_{C\text{-}IL} \rightarrow conf_{C\text{-}IL+\mathcal{G}}$$

is defined as

$$merge\text{-}impl\text{-}ghost(c,c') = c''$$

where

$$c''.\mathcal{M} = c'.\mathcal{M}$$

$$c''.\mathcal{M}_{\mathcal{G}} = c.\mathcal{M}_{\mathcal{G}}$$

$$c''.s = c.s[|c.s|-1 : |c'.s|] \circ \mathit{merge\text{-}stacks}(c.s[|c'.s|-1:0], c'.s)$$

$$c''.nf_{\mathcal{G}} = c.nf_{\mathcal{G}}$$

Note that the stack of the *C-IL* configuration may that is merged may be shorter than the stack of the *C-IL+$\mathcal{G}$* configuration. This is to account for the fact that the *C-IL+$\mathcal{G}$* configuration may contain ghost frames that do not occur in a projected *C-IL* configuration.

Here,

$$\mathit{merge\text{-}stacks} : \mathit{frame}^*_{\mathit{C\text{-}IL}+\mathcal{G}} \times \mathit{frame}^*_{\mathit{C\text{-}IL}} \to \mathit{frame}^*_{\mathit{C\text{-}IL}+\mathcal{G}}$$

is defined as

$$\mathit{merge\text{-}stacks}(s, s') = \begin{cases} s & s' = \varepsilon \\ \mathbf{hd}(s)[\mathcal{M}_{\mathcal{E}} := \mathbf{hd}(s').\mathcal{M}_{\mathcal{E}}] \circ \mathit{merge\text{-}stacks}(\mathbf{tl}(s), \mathbf{tl}(s')) & s' \neq \varepsilon \end{cases}$$

6.3.2 Expression Evaluation

Analogous to the *C-IL* definitions, we define shorthand notation for a *C-IL+$\mathcal{G}$* to get the variable declarations for the top-most stack frame. For this, we introduce the following shorthand notation:

Definition 6.44 (Implementation Variables of the Top-Most Stack Frame) We denote the implementation variable declarations of the top-most stack frame of a *C-IL+$\mathcal{G}$* program π and a *C-IL+$\mathcal{G}$* configuration c by

$$\mathcal{V}_{top}(\pi,c) \overset{def}{=} \begin{cases} \pi.\mathcal{F}(stack_{top}(c).f).\mathcal{V} & si_{\mathcal{G}}(\pi, c) = \bot \\ \varepsilon & \text{otherwise} \end{cases}$$

Definition 6.45 (Ghost Variables of the Top-Most Stack Frame) We use

$$\mathcal{V}_{\mathcal{G}top}(\pi,c) \overset{def}{=} \mathcal{F}_{\mathit{C\text{-}IL}+\mathcal{G}}(stack_{top}(c).f).\mathcal{V}_{\mathcal{G}}$$

to refer to the ghost variable declarations of the top-most stack frame of a *C-IL+$\mathcal{G}$* configuration c of a given *C-IL+$\mathcal{G}$* program π.

Definition 6.46 (Field Reference Function for *C-IL+$\mathcal{G}$*) We provide an extension of the field reference function from *C-IL* to include ghost references. Given environment parameters θ and a *C-IL+$\mathcal{G}$* program π, we define the function

$$\sigma_{\mathcal{G}\theta}^{\pi} : (val \cup val_{\mathcal{G}}) \times \mathbb{F} \rightharpoonup val \cup val_{\mathcal{G}}$$

which takes a (ghost) pointer or local reference x and a field name f and computes the pointer or local reference for the field access of f in x.

$$\sigma\mathcal{G}_\theta^{\pi,c}(x,f) \overset{def}{=} \begin{cases} \sigma_\theta^\pi(x,f) & x \in val \wedge f \in \mathbb{F} \wedge f \notin decl_\mathcal{G}(\pi.T_{F\mathcal{G}}(t_C)) \\ \mathbf{lref}_\mathcal{G}(((v,o),t_C),i,[f],t) & x = \mathbf{lref}((v,o),i,\mathbf{ptr}(\mathbf{struct}\ t_C)) \wedge f \in decl_\mathcal{G}(\pi.T_{F\mathcal{G}}(t_C)) \\ \mathbf{gref}((a,t_C),[f],t) & x = \mathbf{val}(a,\mathbf{ptr}(\mathbf{struct}\ t_C)) \wedge f \in decl_\mathcal{G}(\pi.T_{F\mathcal{G}}(t_C)) \\ \mathbf{lref}_\mathcal{G}(y,i,S \circ f,t) & x = \mathbf{lref}_\mathcal{G}(y,i,S,\mathbf{ptr}(\mathbf{struct}\ t_C)) \wedge f \in decl_\mathcal{G}(\pi.T_{F\mathcal{G}}(t_C)) \\ \mathbf{gref}(y,S \circ f,t) & x = \mathbf{gref}(y,S,\mathbf{ptr}(\mathbf{struct}\ t_C)) \wedge f \in decl_\mathcal{G}(\pi.T_{F\mathcal{G}}(t_C)) \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $t = \mathbf{ptr}(\tau_F(t_C,\pi.T_{F\mathcal{G}}(t_C)))$. In case we access a field of a ghost structure, the corresponding field name is appended to the subvariable selector string of the ghost reference.

Note that when x is a pointer or local reference to an implementation variable of struct-pointer type, we explicitly construct an appropriate ghost reference to the ghost subvariable associated with accessing field f in the ghost data variable associated with x.

Definition 6.47 (Ghost Array Index Reference Function for *C-IL+$\mathcal{G}$*) In order to provide references to array subvariables of ghost variables at ghost array access, we define the function

$$\gamma\mathcal{G}_\theta^\pi : val_\mathcal{G} \times \mathbb{Z} \rightharpoonup val_\mathcal{G}$$

which takes a ghost pointer or local reference x and an index z and computes the ghost reference for the array access of z in x.

$$\gamma\mathcal{G}_\theta^{\pi,c}(x,z) \overset{def}{=} \begin{cases} \mathbf{lref}_\mathcal{G}(y,i,S \circ z,\mathbf{ptr}(t)) & x = \mathbf{lref}_\mathcal{G}(y,i,S,\mathbf{ptr}(\mathbf{array}(t,n))) \wedge z \in [0:n-1] \\ \mathbf{lref}_\mathcal{G}(y,i,S \circ z,\mathbf{ptr}(t)) & x = \mathbf{lref}_\mathcal{G}(y,i,S,\mathbf{array}(t,n)) \wedge z \in [0:n-1] \\ \mathbf{gref}(y,S \circ z,\mathbf{ptr}(t)) & x = \mathbf{gref}(y,S,\mathbf{ptr}(\mathbf{array}(t,n))) \wedge z \in [0:n-1] \\ \mathbf{gref}(y,S \circ z,\mathbf{ptr}(t)) & x = \mathbf{gref}(y,S,\mathbf{array}(t,n)) \wedge z \in [0:n-1] \\ \text{undefined} & \text{otherwise} \end{cases}$$

Definition 6.48 (Expression Evaluation of *C-IL+$\mathcal{G}$*) We define expression evaluation for *C-IL+$\mathcal{G}$* as a function that takes a *C-IL+$\mathcal{G}$*-configuration $c \in conf_{\text{C-IL+}\mathcal{G}}$, a partial function $p_\lambda : \mathbb{V} \rightharpoonup val_{\mathcal{M}_\mathcal{G}}$ providing lambda-parameter-values, a *C-IL+$\mathcal{G}$* program π, *C-IL* environment parameters $\theta \in params_{\text{C-IL}}$, and a ghost expression $e \in \mathbb{E}_\mathcal{G}$. The signature of expressions evaluation for *C-IL+$\mathcal{G}$* is the following:

$$^\mathcal{G}[\![\cdot]\!]^{\cdot,\cdot}_{\cdot,\cdot} : conf_{\text{C-IL+}\mathcal{G}} \times (\mathbb{V} \rightharpoonup val_{\mathcal{M}_\mathcal{G}}) \times program_{\text{C-IL+}\mathcal{G}} \times params_{\text{C-IL+}\mathcal{G}} \times \mathbb{E}_\mathcal{G} \rightharpoonup val_{\mathcal{M}_\mathcal{G}}$$

In the following we will use expression evaluation from *C-IL* where possible by using the projected *C-IL* configuration and the projected *C-IL* program which can be obtained from their *C-IL+$\mathcal{G}$* counterparts by removing everything related to ghost (definitions of the projection functions are provided later in section 6.4.1). We proceed by making a case distinction over the possible types of expressions:

- Constant: $x \in val \cup val_{\mathcal{G}} \Rightarrow {}^{\mathcal{G}}[\![x]\!]^{\pi,\theta}_{c,p_\lambda} = x$
- Variable Name: $v \in \mathbb{V} \Rightarrow$

$$ {}^{\mathcal{G}}[\![v]\!]^{\pi,\theta}_{c,p_\lambda} = \begin{cases} {}^{\mathcal{G}}[\![*\&v]\!]^{\pi,\theta}_{c,p_\lambda} & v \in decl_{\mathcal{G}}(\mathcal{V}_{top}(\pi,c) \circ \pi.\mathcal{V}_G \circ \mathcal{V}_{\mathcal{G}top}(\pi,c) \circ \pi.\mathcal{V}_{G\mathcal{G}}) \wedge v \notin dom(p_\lambda) \\ p_\lambda(v) & v \in dom(p_\lambda) \\ \text{undefined} & \text{otherwise} \end{cases} $$

- Function Name: $fn \in \mathbb{F}_{name} \Rightarrow$

$$ {}^{\mathcal{G}}[\![fn]\!]^{\pi,\theta}_{c,p_\lambda} = \begin{cases} [\![fn]\!]^{C\text{-}IL(\pi),\theta}_{C\text{-}IL(\pi,c)} & fn \in dom(\mathcal{F}^\theta_\pi) \\ \mathbf{gfun}(fn) & fn \in dom(\pi.\mathcal{F}_{\mathcal{G}}) \\ \text{undefined} & \text{otherwise} \end{cases} $$

- Unary Operator: $e \in \mathbb{E}_{\mathcal{G}}, \ominus \in \mathbb{O}_1 \Rightarrow {}^{\mathcal{G}}[\![\ominus e]\!]^{\pi,\theta}_{c,p_\lambda} = (\theta.op_1(\ominus) \cup op_{1\mathcal{G}}(\ominus))({}^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,p_\lambda})$
- Binary Operator: $e_0, e_1 \in \mathbb{E}_{\mathcal{G}}, \oplus \in \mathbb{O}_2 \Rightarrow$

$$ {}^{\mathcal{G}}[\![e_0 \oplus e_1]\!]^{\pi,\theta}_{c,p_\lambda} = (\theta.op_2(\oplus) \cup op_{2\mathcal{G}}(\oplus))({}^{\mathcal{G}}[\![e_0]\!]^{\pi,\theta}_{c,p_\lambda}, {}^{\mathcal{G}}[\![e_1]\!]^{\pi,\theta}_{c,p_\lambda}) $$

- Ternary Operator: $e, e_0, e_1 \in \mathbb{E}_{\mathcal{G}} \Rightarrow {}^{\mathcal{G}}[\![(e \mathbin{?} e_0 : e_1)]\!]^{\pi,\theta}_{c,p_\lambda} = \begin{cases} {}^{\mathcal{G}}[\![e_0]\!]^{\pi,\theta}_{c,p_\lambda} & zero_{\mathcal{G}}(\theta, {}^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,p_\lambda}) \\ {}^{\mathcal{G}}[\![e_1]\!]^{\pi,\theta}_{c,p_\lambda} & \text{otherwise} \end{cases}$
- Type Cast: $t \in \mathbb{T}_Q \cup \mathbb{T}_{Q\mathcal{G}}, e \in \mathbb{E}_{\mathcal{G}} \Rightarrow {}^{\mathcal{G}}[\![(t)e]\!]^{\pi,\theta}_{c,p_\lambda} = cast_{\mathcal{G}}({}^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,p_\lambda}, qt2t(t))$
- Dereferencing a Pointer: $e \in \mathbb{E}_{\mathcal{G}} \Rightarrow$

$$ {}^{\mathcal{G}}[\![*e]\!]^{\pi,\theta}_{c,p_\lambda} = \begin{cases} read_{\mathcal{G}}(\theta,\pi,c,x) & (\tau_{\mathcal{G}}(x) = \mathbf{ptr}(t) \wedge \neg isarray(t) \vee \tau(x) = \mathbf{array}(t,n)) \\ \mathbf{val}(a, \mathbf{array}(t,n)) & x = \mathbf{val}(a, \mathbf{ptr}(\mathbf{array}(t,n))) \\ \mathbf{lref}((v,o),i,\mathbf{array}(t,n)) & x = \mathbf{lref}((v,o),i,\mathbf{ptr}(\mathbf{array}(t,n))) \\ \mathbf{gref}(a,S,\mathbf{array}(t,n)) & x = \mathbf{gref}(a,S,\mathbf{ptr}(\mathbf{array}(t,n))) \\ \mathbf{lref}_{\mathcal{G}}(a,i,S,\mathbf{array}(t,n)) & x = \mathbf{lref}_{\mathcal{G}}(a,i,S,\mathbf{ptr}(\mathbf{array}(t,n))) \\ \text{undefined} & \text{otherwise} \end{cases} $$

  where $x = {}^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,p_\lambda}$.
- Address of: $e \in \mathbb{E}_{\mathcal{G}} \Rightarrow$

$$ {}^{\mathcal{G}}[\![\&e]\!]^{\pi,\theta}_{c,p_\lambda} = \begin{cases} {}^{\mathcal{G}}[\![e']\!]^{\pi,\theta}_{c,p_\lambda} & e = *e' \\ [\![\&e]\!]^{C\text{-}IL(\pi),\theta}_{C\text{-}IL(\pi,c)} & e = v \wedge (v \in decl(\mathcal{V}_{top}(\pi,c) \circ \pi.\mathcal{V}_G)) \\ x & e = v \wedge v \in decl_{\mathcal{G}}(\mathcal{V}_{\mathcal{G}top}(\pi,c)) \\ \mathbf{gref}(v,\varepsilon,\mathbf{ptr}(\tau_{V\mathcal{G}}(v,\pi.\mathcal{V}_{G\mathcal{G}}))) & e = v \wedge v \in decl_{\mathcal{G}}(\pi.\mathcal{V}_{G\mathcal{G}}) \setminus decl_{\mathcal{G}}(\mathcal{V}_{\mathcal{G}top}(\pi,c)) \\ \sigma_{\mathcal{G}^{\pi,c}_\theta}({}^{\mathcal{G}}[\![e']\!]^{\pi,\theta}_{c,p_\lambda}, f) & e = (e').f \\ \gamma_{\mathcal{G}^{\pi,c}_\theta}({}^{\mathcal{G}}[\![e']\!]^{\pi,\theta}_{c,p_\lambda}, z) & e = e'[e''] \wedge {}^{\mathcal{G}}[\![e'']\!]^{\pi,\theta}_{c,p_\lambda} = \mathbf{gval}(z, \mathbf{math\_int}) \\ \text{undefined} & \text{otherwise} \end{cases} $$

where $x = \mathbf{lref}_{\mathcal{G}}(v, |c.s| - 1, \varepsilon, \mathbf{ptr}(\tau_{V\mathcal{G}}(v, \mathcal{V}_{\mathcal{G}top}(\pi, c))))$.

- Field access: $e \in \mathbb{E}_{\mathcal{G}}, f \in \mathbb{F} \Rightarrow$

$$^{\mathcal{G}}[\![(e).f]\!]^{\pi,\theta}_{c,p_\lambda} = \begin{cases} ^{\mathcal{G}}[\![*\&e]\!]^{\pi,\theta}_{c,p_\lambda} & ^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,p_\lambda} \notin val_{\mathbf{record}} \\ r(f) & ^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,p_\lambda} = \mathbf{gval}(r, \mathbf{record}\ t_C) \wedge f \in decl_{\mathcal{G}}(\pi.T_{F\mathcal{G}}(t_C)) \\ \text{undefined} & \text{otherwise} \end{cases}$$

- Size of Type: $t \in \mathbb{T}_Q \Rightarrow {}^{\mathcal{G}}[\![\mathbf{sizeof}(t)]\!]^{\pi,\theta}_{c,p_\lambda} = [\![\mathbf{sizeof}(t)]\!]^{C\text{-}IL(\pi),\theta}_{C\text{-}IL(\pi,c)}$

- Size of Expression: $e \in \mathbb{E} \Rightarrow {}^{\mathcal{G}}[\![\mathbf{sizeof}(e)]\!]^{\pi,\theta}_{c,p_\lambda} = [\![\mathbf{sizeof}(e)]\!]^{C\text{-}IL(\pi),\theta}_{C\text{-}IL(\pi,c)}$

- Map or ghost array access: $e, e' \in \mathbb{E}_{\mathcal{G}} \Rightarrow$

$$^{\mathcal{G}}[\![e[e']]\!]^{\pi,\theta}_{c,p_\lambda} = \begin{cases} f(^{\mathcal{G}}[\![e']\!]^{\pi,\theta}_{c,p_\lambda}) & ^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,p_\lambda} = \mathbf{gval}(f, \mathbf{map}(t, t')) \wedge t = \tau_{\mathcal{G}}(^{\mathcal{G}}[\![e']\!]^{\pi,\theta}_{c,p_\lambda}) \\ *\&e[e'] & \text{otherwise} \end{cases}$$

- Lambda expression: $t \in \mathbb{T}_Q \cup \mathbb{T}_{Q\mathcal{G}}, v \in \mathbb{V}, e \in \mathbb{E}_{\mathcal{G}} \Rightarrow$

$$^{\mathcal{G}}[\![\mathbf{lambda}(t\ v; e)]\!]^{\pi,\theta}_{c,p_\lambda} = \mathbf{gval}(\lambda x \in val_{\mathcal{G}}(qt2t(t)).{}^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,p_\lambda(v:=x)}, \mathbf{map}(t, t'))$$

where $t' = \tau_{\mathcal{G}}(^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,p_\lambda(v:=\mathcal{E}(val_{\mathcal{G}}(qt2t(t))))})$. Note that, since for correctly typed programs, type evaluation does not depend on the particular value of a variable (i.e. it only depends on the type of the value), the particular chosen value used in evaluating the type here does not matter.

Note that, in the map value obtained, all occurrences of variable name v in expression e evaluate to the value passed to the function that defines the map. To achieve this, we use p_λ in expression evaluation to provide values for such bound subvariables in lambda-expressions.

- Record update: $e, e' \in \mathbb{E}_{\mathcal{G}}, f \in \mathbb{F} \Rightarrow$

$$^{\mathcal{G}}[\![e\{f := e'\}]\!]^{\pi,\theta}_{c,p_\lambda} = \begin{cases} \mathbf{gval}(r(f := {}^{\mathcal{G}}[\![e']\!]^{\pi,\theta}_{c,p_\lambda}), \mathbf{record}\ t_C) & ^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,p_\lambda} = \mathbf{gval}(r, \mathbf{record}\ t_C) \\ & \wedge\ \tau_{F\mathcal{G}}(f, \pi.T_{F\mathcal{G}}(t_C)) = \tau_{\mathcal{G}}(^{\mathcal{G}}[\![e']\!]^{\pi,\theta}_{c,p_\lambda}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

- State-snapshot: $^{\mathcal{G}}[\![\mathbf{current\_state}]\!]^{\pi,\theta}_{c,p_\lambda} = \mathbf{gval}(c, \mathbf{state\_t})$

- Use a state-snapshot to evaluate and expression: $e, e' \in \mathbb{E}_{\mathcal{G}} \Rightarrow$

$$^{\mathcal{G}}[\![\mathbf{in\_state}(e, e')]\!]^{\pi,\theta}_{c,p_\lambda} = \begin{cases} ^{\mathcal{G}}[\![e']\!]^{\pi,\theta}_{c',p_\lambda} & ^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,p_\lambda} = \mathbf{gval}(c', \mathbf{state\_t}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

6.3.3 Transition Function

We define the partial transition function

$$\delta^{\pi,\theta}_{C\text{-}IL+\mathcal{G}} : conf_{C\text{-}IL+\mathcal{G}} \times \Sigma \rightharpoonup conf_{C\text{-}IL+\mathcal{G}}$$

for a given *C-IL+$\mathcal{G}$* program π and *C-IL* environment parameters θ in a very similar way to that of *C-IL*. In fact, $\Sigma = (\mathbb{V} \rightharpoonup (\mathbb{B}^8)^*) \cup conf_{C\text{-}IL} \cup \{\bot\}$ is the same input alphabet used by *C-IL* to resolve nondeterminism. Note that ghost instructions are fully deterministic.

Auxiliary Definitions

We use the following shorthand notation that looks exactly identical to the corresponding *C-IL* definitions – the only difference is that these definitions apply to *C-IL+$\mathcal{G}$* configurations instead of *C-IL* configurations.

- function body of the topmost frame: $\mathcal{P}_{top\mathcal{G}}(\pi,c) = \mathcal{F}_{C\text{-}IL+\mathcal{G}}(stack_{top}(c).f).\mathcal{P}$
- location counter of the topmost frame: $loc_{top}(c) = stack_{top}(c).loc$
- next statement to be executed: $stmt_{next}(\pi,c) = \mathcal{P}_{top\mathcal{G}}(\pi, c)[loc_{top}(c)]$

Further, we define the following auxiliary functions:

Definition 6.49 (Increasing the Location Counter) We define

$$inc_{loc} : conf_{C\text{-}IL+\mathcal{G}} \rightharpoonup conf_{C\text{-}IL+\mathcal{G}}$$

which increments the location of the top-most stack frame of a *C-IL+$\mathcal{G}$*-configuration as

$$inc_{loc}(c) = c[s := \mathbf{hd}(c.s)[loc := c.loc_{top} + 1] \circ \mathbf{tl}(c.s)]$$

Definition 6.50 (Setting the Location Counter) The function

$$set_{loc} : conf_{C\text{-}IL+\mathcal{G}} \times \mathbb{N} \rightharpoonup conf_{C\text{-}IL+\mathcal{G}}$$

is defined as

$$set_{loc}(c,l) = c[s := \mathbf{hd}(c.s)[loc := l] \circ \mathbf{tl}(c.s)]$$

and sets the location of the top-most stack frame to location l.

Definition 6.51 (Removing the Topmost Frame) The function

$$drop_{frame} : conf_{C\text{-}IL+\mathcal{G}} \rightharpoonup conf_{C\text{-}IL+\mathcal{G}}$$

which removes the top-most stack frame from a *C-IL+$\mathcal{G}$*-configuration is defined as

$$drop_{frame}(c) = c[s := \mathbf{tl}(c.s)]$$

Definition 6.52 (Setting Return Destination) We define the function

$$set_{rds} : conf_{C\text{-}IL+\mathcal{G}} \times (val_{\mathbf{lref}} \cup val_{\mathbf{ptr}} \cup val_{\mathbf{lref}_{\mathcal{G}}} \cup val_{\mathbf{gref}} \cup \{\bot\}) \rightharpoonup conf_{C\text{-}IL+\mathcal{G}}$$

that updates the return destination component of the topmost stack frame as:

$$set_{rds}(c,v) = c[s := \mathbf{hd}(c.s)[rds := v] \circ \mathbf{tl}(c.s)]$$

Operational Semantics

Definition 6.53 (*C-IL+$\mathcal{G}$* Transition Function) We define the transition function

$$\delta^{\pi,\theta}_{\textit{C-IL+}\mathcal{G}} : \mathit{conf}_{\textit{C-IL+}\mathcal{G}} \times \Sigma \rightharpoonup \mathit{conf}_{\textit{C-IL+}\mathcal{G}}$$

by a case distinction on the given input:

- Deterministic step:

$$\delta^{\pi,\theta}_{\textit{C-IL+}\mathcal{G}}(c,\bot) = \begin{cases} c' & \mathit{stmt}_{next}(\pi,c) = (e_0 = e_1) \lor \mathit{stmt}_{next}(\pi,c) = \textbf{ghost}(e_0 = e_1) \\ \mathit{set}_{loc}(c,l) & \mathit{stmt}_{next}(\pi,c) = \textbf{goto}\ l \lor \mathit{stmt}_{next}(\pi,c) = \textbf{ghost}(\textbf{goto}\ l) \\ \mathit{set}_{loc}(c,l) & (\mathit{stmt}_{next}(\pi,c) = \textbf{ifnot}\ e\ \textbf{goto}\ l \lor \mathit{stmt}_{next}(\pi,c) = \textbf{ghost}(\textbf{ifnot}\ e\ \textbf{goto}\ l)) \\ & \land\ \mathit{zero}_{\mathcal{G}}(\theta, {}^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,\emptyset}) \\ \mathit{inc}_{loc}(c) & (\mathit{stmt}_{next}(\pi,c) = \textbf{ifnot}\ e\ \textbf{goto}\ l \lor \mathit{stmt}_{next}(\pi,c) = \textbf{ghost}(\textbf{ifnot}\ e\ \textbf{goto}\ l)) \\ & \land\ \neg\mathit{zero}_{\mathcal{G}}(\theta, {}^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,\emptyset}) \\ \mathit{set}_{rds}(\mathit{drop}_{frame}(c),\bot) & \mathit{stmt}_{next}(\pi,c) = \textbf{return} \lor \mathit{stmt}_{next}(\pi,c) = \textbf{ghost}(\textbf{return}) \\ \mathit{set}_{rds}(\mathit{drop}_{frame}(c),\bot) & (\mathit{stmt}_{next}(\pi,c) = \textbf{return}\ e \lor \mathit{stmt}_{next}(\pi,c) = \textbf{ghost}(\textbf{return}\ e)) \\ & \land\ \mathit{rds} = \bot \\ c'' & (\mathit{stmt}_{next}(\pi,c) = \textbf{return}\ e \lor \mathit{stmt}_{next}(\pi,c) = \textbf{ghost}(\textbf{return}\ e)) \\ & \land\ \mathit{rds} \neq \bot \\ c''' & \mathit{stmt}_{next}(\pi,c) = \textbf{ghost}(e = \textbf{allocghost}(t)) \\ c'''' & (\mathit{stmt}_{next}(\pi,c) = \textbf{ghost}(\textbf{call}\ e(E)) \lor \mathit{stmt}_{next} = \textbf{ghost}(e_0 = \textbf{call}\ e(E))) \\ & \land\ {}^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,\emptyset} = \textbf{gfun}(f) \land \mathcal{F}_{\textit{C-IL+}\mathcal{G}}(f).\mathcal{P} \neq \textbf{extern} \\ \text{undefined} & \text{otherwise} \end{cases}$$

where

$$\mathit{rds} = \mathit{rds}_{top}(\mathit{drop}_{frame}(c))$$

$$c' = \mathit{inc}_{loc}(\mathit{write}_{\mathcal{G}}(\theta,\pi,c,{}^{\mathcal{G}}[\![\&e_0]\!]^{\pi,\theta}_{c,\emptyset}, {}^{\mathcal{G}}[\![e_1]\!]^{\pi,\theta}_{c,\emptyset})$$

and

$$c'' = \mathit{write}_{\mathcal{G}}(\theta,\pi,\mathit{set}_{rds}(\mathit{drop}_{frame}(c),\bot),\mathit{rds}), {}^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,\emptyset})$$

and

$$c''' = \mathit{write}_{\mathcal{G}}(\theta,\pi,\mathit{inc}_{loc}(c), {}^{\mathcal{G}}[\![\&e]\!]^{\pi,\theta}_{c,\emptyset}, \textbf{gref}(c.\mathit{nf}_{\mathcal{G}},\varepsilon,\textbf{ptr}(t)))[\mathit{nf}_{\mathcal{G}} := c.\mathit{nf}_{\mathcal{G}} + 1]$$

and c'''' is the resulting configuration after calling a ghost function:

$$c''''.s = (\emptyset, \mathcal{M}'_{\mathcal{E}\mathcal{G}}, \bot, f, 0) \circ \mathit{inc}_{loc}(\mathit{set}_{rds}(c,\mathit{rds})).s$$

$$c''''.\mathcal{M} = c.\mathcal{M}, \qquad c''''.\mathcal{M}_{\mathcal{G}} = c.\mathcal{M}_{\mathcal{G}}, \qquad c''''.\mathit{nf}_{\mathcal{G}} = c.\mathit{nf}_{\mathcal{G}}$$

where

$$\mathit{rds} = \begin{cases} {}^{\mathcal{G}}[\![\&e_0]\!]^{\pi,\theta}_{c,\emptyset} & \mathit{stmt}_{next}(c) = \textbf{ghost}(e_0 = \textbf{call}\ e(E)) \\ \text{undefined} & \mathit{stmt}_{next}(c) = \textbf{ghost}(\textbf{call}\ e(E)) \end{cases}$$

and

$$\mathcal{M}'_{\mathcal{EG}}(v) = \begin{cases} {}^{\mathcal{G}}[\![E[i]]\!]^{\pi,\theta}_{c,\emptyset} & \exists i : \mathcal{F}_{C\text{-}IL+\mathcal{G}}(f).\mathcal{V}_{\mathcal{G}}[i] = (v,t) \wedge i < \mathcal{F}_{C\text{-}IL+\mathcal{G}}(f).ngpar \\ \text{undefined} & \text{otherwise} \end{cases}$$

- (Annotated) implementation function call:

  $\delta^{\pi,\theta}_{C\text{-}IL+\mathcal{G}}(c, \mathcal{M}_{lvar})$ is defined if and only if all of the following hold:
  - $stmt_{next} = \mathbf{call}\ e(E, E') \vee stmt_{next} = (e_0 = \mathbf{call}\ e(E, E'))$ – the next statement is an annotated function call,
  - ${}^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,\emptyset} = \mathbf{val}(b, \mathbf{funptr}(t, T)) \wedge f = \theta.\mathcal{F}_{adr}{}^{-1}(b) \vee {}^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,\emptyset} = \mathbf{fun}(f, \mathbf{funptr}(t, T))$ – expression e evaluates to some function f,
  - $|E| = \mathcal{F}_{C\text{-}IL+\mathcal{G}}(f).npar$
    $\wedge\ \forall i \in \{0, \ldots, |E| - 1\} : \mathcal{F}_{C\text{-}IL+\mathcal{G}}(f).\mathcal{V}[i] = (v,t) \Rightarrow \tau({}^{\mathcal{G}}[\![E[i]]\!]^{\pi,\theta}_{c,\emptyset}) = t$ – the types of all implementation parameters passed match the declaration,
  - $|E'| = \mathcal{F}_{C\text{-}IL+\mathcal{G}}(f).ngpar$
    $\wedge\ \forall i \in \{0, \ldots, |E'| - 1\} : \mathcal{F}_{C\text{-}IL+\mathcal{G}}(f).\mathcal{V}_{\mathcal{G}}[i] = (v,t) \Rightarrow \tau({}^{\mathcal{G}}[\![E'[i]]\!]^{\pi,\theta}_{c,\emptyset}) = t$ – the types of all ghost parameters passed match the declaration,
  - $\mathcal{F}_{C\text{-}IL+\mathcal{G}}(f).\mathcal{P} \neq \mathbf{extern}$ – the function is not declared as extern in the function table, and
  - $\forall i \geq \mathcal{F}_{C\text{-}IL+\mathcal{G}}(f).npar : \exists v, t : \mathcal{F}_{C\text{-}IL+\mathcal{G}}(f).\mathcal{V}[i] = (v,t) \Rightarrow |\mathcal{M}_{lvar}(v)| = size_\theta(t)$ – the byte-string memory provided for all local variables is of adequate length.

  Then, we define

  $$\delta^{\pi,\theta}_{C\text{-}IL+\mathcal{G}}(c, \mathcal{M}_{lvar}) = c'$$

  such that

  $$c'.s = (\mathcal{M}'_{\mathcal{E}}, \mathcal{M}'_{\mathcal{EG}}, \bot, f, 0) \circ inc_{loc}(set_{rds}(c, rds)).s$$

  $$c'.\mathcal{M} = c.\mathcal{M}, \qquad c'.\mathcal{M}_{\mathcal{G}} = c.\mathcal{M}_{\mathcal{G}}, \qquad c'.nf_{\mathcal{G}} = c.nf_{\mathcal{G}}$$

  where

  $$rds = \begin{cases} {}^{\mathcal{G}}[\![\&e_0]\!]^{\pi,\theta}_{c,\emptyset} & stmt_{next}(c) = (e_0 = \mathbf{call}\ e(E, E')) \\ \bot & stmt_{next}(c) = \mathbf{call}\ e(E, E') \end{cases}$$

  and

  $$\mathcal{M}'_{\mathcal{E}}(v) = \begin{cases} val2bytes_\theta({}^{\mathcal{G}}[\![E[i]]\!]^{\pi,\theta}_{c,\emptyset}) & \exists i : \mathcal{F}^{\theta}_{\pi}(f).\mathcal{V}[i] = (v,t) \wedge i < \mathcal{F}_{C\text{-}IL+\mathcal{G}}(f).npar \\ \mathcal{M}_{lvar}(v) & \exists i.\ \mathcal{F}_{C\text{-}IL+\mathcal{G}}(f).\mathcal{V}[i] = (v,t) \wedge i \geq \mathcal{F}_{C\text{-}IL+\mathcal{G}}(f).npar \\ \text{undefined} & \text{otherwise} \end{cases}$$

  and

  $$\mathcal{M}'_{\mathcal{EG}}(v) = \begin{cases} {}^{\mathcal{G}}[\![E'[i]]\!]^{\pi,\theta}_{c,\emptyset} & \exists i : \mathcal{F}_{C\text{-}IL+\mathcal{G}}(f).\mathcal{V}_{\mathcal{G}}[i] = (v,t) \wedge i < \mathcal{F}_{C\text{-}IL+\mathcal{G}}(f).ngpar \\ \text{undefined} & \text{otherwise} \end{cases}$$

- External procedure call:

  $\delta^{\pi,\theta}_{C\text{-}IL+\mathcal{G}}(c, c')$ is defined if and only if all of the following hold:

  - $stmt_{next} = \textbf{call } e(E, \varepsilon)$: the next statement is a function call without return value and without ghost parameters,
  - ${}^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,\emptyset} = \textbf{val}(b, \textbf{funptr}(t, T)) \wedge f = \theta.\mathcal{F}_{adr}{}^{-1}(b) \vee {}^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,\emptyset} = \textbf{fun}(f, \textbf{funptr}(t, T))$: expression e evaluates to some function f,
  - $|E| = \mathcal{F}^{\theta}_{\pi}(f).npar \wedge \forall i \in \{0, \ldots, |E| - 1\} : \mathcal{F}^{\theta}_{\pi}(f).\mathcal{V}[i] = (v, t) \Rightarrow \tau([\![E[i]]\!]^{\pi,\theta}_{c}) = t$ – the types of all parameters passed match the declaration,
  - $\mathcal{F}^{\theta}_{\pi}(f).\mathcal{P} = \textbf{extern}$: the function is declared as extern in the function table, and
  - $({}^{\mathcal{G}}[\![E[0]]\!]^{\pi,\theta}_{c,\emptyset}, \ldots, {}^{\mathcal{G}}[\![E[|E| - 1]]\!]^{\pi,\theta}_{c,\emptyset}, C\text{-}IL(\pi, c), c') \in \theta.R_{\textbf{extern}}$: the transition relation for external functions given in the *C-IL* environment parameters allows a transition under given parameters E from the projected *C-IL* configuration $C\text{-}IL(\pi, c)$ to c'.

  Then,

  $$\delta^{\pi,\theta}_{C\text{-}IL+\mathcal{G}}(c, c') = merge\text{-}impl\text{-}ghost(inc_{loc}(c), c')$$

  Note that we do not support annotating compiler intrinsics with ghost state updates directly. This, however, is not a problem since ghost steps in the concurrent machine will always be attached to the next implementation step – interleaving is performed only at implementation steps. Thus, we can effectively annotate any ghost state update on any given implementation step by simply adding ghost statements that perform the update.

Concurrent Operational Semantics

In contrast to the freely interleaved scheduling of concurrent *C-IL*, we consider a more restricted scheduling for *C-IL+$\mathcal{G}$*: Only implementation steps are interleaved, all ghost steps before an implementation step are attached to that implementation step.

Definition 6.54 (*C-IL+$\mathcal{G}$* Transition Function for Ghost Steps) We denote the transition function that executes ghost steps until it encounters an implementation step by:

$$^{\mathcal{G}}\delta^{\pi,\theta}_{C\text{-}IL+\mathcal{G}}(c) = \begin{cases} ^{\mathcal{G}}\delta^{\pi,\theta}_{C\text{-}IL+\mathcal{G}}(\delta^{\pi,\theta}_{C\text{-}IL+\mathcal{G}}(c, \bot)) & stmt_{next}(c) \in \mathbb{S}_{\mathcal{G}} \\ c & \text{otherwise} \end{cases}$$

Definition 6.55 (Transition Function of C*C-IL+$\mathcal{G}$*) We define the transition function for concurrent *C-IL+$\mathcal{G}$*

$$\delta^{\pi,\theta}_{CC\text{-}IL+\mathcal{G}} : conf_{CC\text{-}IL+\mathcal{G}} \times \mathbb{N} \times \Sigma \rightharpoonup conf_{CC\text{-}IL+\mathcal{G}}$$

as

$$\delta^{\pi,\theta}_{CC\text{-}IL+\mathcal{G}}(c, t, in) = c'$$

where

$$c'.\mathcal{M} = \delta^{\pi,\theta}_{C\text{-}IL+\mathcal{G}}({}^{\mathcal{G}}\delta^{\pi,\theta}_{C\text{-}IL+\mathcal{G}}(c_t), in).\mathcal{M}$$

$$c'.\mathcal{M}_{\mathcal{G}} = \delta^{\pi,\theta}_{C\text{-}IL+\mathcal{G}}({}^{\mathcal{G}}\delta^{\pi,\theta}_{C\text{-}IL+\mathcal{G}}(c_t), in).\mathcal{M}_{\mathcal{G}}$$

$$c'.Th = c.Th(t := \delta^{\pi,\theta}_{C\text{-}IL+\mathcal{G}}({}^{\mathcal{G}}\delta^{\pi,\theta}_{C\text{-}IL+\mathcal{G}}(c_t), in).s)$$

$$c'.nf_{\mathcal{G}} = \delta^{\pi,\theta}_{C\text{-}IL+\mathcal{G}}({}^{\mathcal{G}}\delta^{\pi,\theta}_{C\text{-}IL+\mathcal{G}}(c_t), in).nf_{\mathcal{G}}$$

and

$$c_t = (c.\mathcal{M}, c.\mathcal{M}_{\mathcal{G}}, c.Th(t).s, c.nf_{\mathcal{G}})$$

Thread t of the concurrent configuration makes as many ghost steps as necessary, followed by an implementation step on its own stack and the shared memories.

6.4 Simulation Between *C-IL* and *C-IL+$\mathcal{G}$*

In order to define a simulation relation between execution of a *C-IL* program annotated with ghost and the same program without all the ghost state and code, we begin by defining functions that project *C-IL+$\mathcal{G}$* components (i.e. program and configuration) to their underlying *C-IL* components by removing everything related to ghost state and code. These projection functions serve to define the simulation relation (and are also used in some parts of *C-IL+$\mathcal{G}$* semantics as defined in the last section). Afterwards, we state invariants that must be fulfilled in order to show the simulation and give a paper and pencil simulation proof.

6.4.1 Projecting *C-IL+$\mathcal{G}$* to *C-IL*

In the following, we introduce for every subcomponent of *C-IL+$\mathcal{G}$* programs a function *C-IL*$(\cdot)$ that converts to a corresponding subcomponent of *C-IL*. While these functions all share the same name, they can always be uniquely identified by their type.

Extracting the *C-IL* Program from a *C-IL+$\mathcal{G}$* Program

Definition 6.56 (Projecting *C-IL+$\mathcal{G}$* Program to *C-IL* Program) In every *C-IL+$\mathcal{G}$*-program, there is a *C-IL*-program we get by throwing away all the ghost code and declarations. For this, we define the function

$$C\text{-}IL : prog_{C\text{-}IL+\mathcal{G}} \rightarrow prog_{C\text{-}IL}$$

that takes a *C-IL+$\mathcal{G}$*-program and returns the corresponding *C-IL*-program as follows

$$C\text{-}IL(\pi).\mathcal{V}_G = \pi.\mathcal{V}_G$$

$$C\text{-}IL(\pi).T_F = \pi.T_F$$

$$C\text{-}IL(\pi).\mathcal{F} = (\lambda f. C\text{-}IL(\pi.\mathcal{F}(f)))$$

Note that we use the function *C-IL* : *FunT'* $\rightarrow$ *FunT* here which converts an annotated function table entry to a *C-IL* function table entry. The definition of this function can be found below.

Definition 6.57 (Projecting Annotated Function Table Entry to *C-IL* Function Table Entry) We define the function

$$C\text{-}IL : FunT' \rightarrow FunT$$

as

$$C\text{-}IL(fte).rettype = fte.rettype$$

$$C\text{-}IL(fte).npar = fte.npar$$

$$C\text{-}IL(fte).\mathcal{V} = fte.\mathcal{V}$$

$$C\text{-}IL(fte).\mathcal{P} = C\text{-}IL(fte.\mathcal{P})$$

In order to define the function *C-IL* which converts a *C-IL*+$\mathcal{G}$ function body to a *C-IL* function body, consider the following: We need to remove all ghost code, which, in particular, means that we need to recalculate the locations of all **goto** and **ifnotgoto** statements and we need to drop all annotation from implementation statements.

Definition 6.58 (Projecting Annotated *C-IL* Statement to *C-IL* Statement) The function

$$C\text{-}IL : \mathbb{S}' \to \mathbb{S}$$

which converts an annotated statement to its unannotated version by dropping ghost code is defined as

$$C\text{-}IL(s) = \begin{cases} \textbf{call } e(E) & s = \textbf{call } e(E, E') \\ (e_0 = \textbf{call } e(E)) & s = (e_0 = \textbf{call } e(E, E')) \\ s & \text{otherwise} \end{cases}$$

Dropping ghost code from *C-IL*+$\mathcal{G}$ implementation statements simply means removing ghost parameters on function calls.

Definition 6.59 (Projecting *C-IL*+$\mathcal{G}$ Function Body to *C-IL* Function Body) We define

$$C\text{-}IL : ((\mathbb{S}' \cup \mathbb{S}_{\mathcal{G}})^* \cup \{\textbf{extern}\}) \to (\mathbb{S}^* \cup \{\textbf{extern}\})$$

as

$$C\text{-}IL(\mathcal{P}) = drop_{\textbf{ghost}}(adjust_{\textbf{gotos}}(\mathcal{P}, 0))$$

where

$$drop_{\textbf{ghost}} : ((\mathbb{S}' \cup \mathbb{S}_{\mathcal{G}})^* \cup \{\textbf{extern}\}) \to (\mathbb{S}^* \cup \{\textbf{extern}\})$$

is defined as

$$drop_{\textbf{ghost}}(\mathcal{P}) = \begin{cases} C\text{-}IL(\textbf{hd}(\mathcal{P})) \circ drop_{\textbf{ghost}}(\textbf{tl}(\mathcal{P})) & \textbf{hd}(\mathcal{P}) \in \mathbb{S}' \\ drop_{\textbf{ghost}}(\textbf{tl}(\mathcal{P})) & \textbf{hd}(\mathcal{P}) \in \mathbb{S}_{\mathcal{G}} \\ \varepsilon & \mathcal{P} = \varepsilon \end{cases}$$

and drops all ghost code from the program while

$$adjust_{\textbf{gotos}} : ((\mathbb{S}' \cup \mathbb{S}_{\mathcal{G}})^* \times \mathbb{N} \to ((\mathbb{S}' \cup \mathbb{S}_{\mathcal{G}})^*$$

adjusts the target locations of all goto statements as follows:

$$adjust_{\mathbf{gotos}}(\mathcal{P},loc)$$

$$= \begin{cases} adjust_{\mathbf{gotos}}(\mathcal{P}[loc := \mathbf{goto}\ (count_{stmt}(\mathcal{P}, l))], loc+1) & \mathcal{P}[loc] = \mathbf{goto}\ l \\ adjust_{\mathbf{gotos}}(\mathcal{P}[loc := \mathbf{ifnot}\ e\ \mathbf{goto}\ (count_{stmt}(\mathcal{P}, l))], loc+1) & \mathcal{P}[loc] = \mathbf{ifnot}\ e\ \mathbf{goto}\ l \\ \mathcal{P} & \text{otherwise} \end{cases}$$

where

$$count_{stmt} : (\mathbb{S}' \cup \mathbb{S}_{\mathcal{G}})^* \times \mathbb{N} \to \mathbb{N}$$

counts the number of implementation statements in a function body $\mathcal{P}$ up to (but not including) a given location l:

$$count_{stmt}(\mathcal{P},l) = \begin{cases} 1 + count_{stmt}(\mathcal{P}, l-1) & \mathcal{P}[l-1] \in \mathbb{S}' \\ 0 + count_{stmt}(\mathcal{P}, l-1) & \mathcal{P}[l-1] \in \mathbb{S}_{\mathcal{G}} \\ 0 & \text{otherwise} \end{cases}$$

Extracting the *C-IL* Configuration from a *C-IL*+$\mathcal{G}$ Configuration

Definition 6.60 (Projecting *C-IL*+$\mathcal{G}$ Stack Frame to *C-IL* Stack Frame) We define the function

$$C\text{-}IL : frame_{C\text{-}IL+\mathcal{G}} \to frame_{C\text{-}IL}$$

which converts a single *C-IL*+$\mathcal{G}$ stack frame to a corresponding *C-IL* stack frame as follows:

$$C\text{-}IL(sf).\mathcal{M}_{\mathcal{E}} = sf.\mathcal{M}_{\mathcal{E}}$$

$$C\text{-}IL(sf).rds = \begin{cases} sf.rds & sf.rds \in val \\ \bot & \text{otherwise} \end{cases}$$

$$C\text{-}IL(sf).f = sf.f$$

$$C\text{-}IL(sf).loc = count_{stmt}(\pi.\mathcal{F}(sf.f).\mathcal{P}, sf.loc)$$

Definition 6.61 (Projecting *C-IL*+$\mathcal{G}$ Configuration to *C-IL* Configuration) To obtain the *C-IL* configuration associated with a *C-IL*+$\mathcal{G}$ configuration of a given *C-IL*+$\mathcal{G}$ program $\pi_{\mathcal{G}}$, we define

$$C\text{-}IL : prog_{C\text{-}IL+\mathcal{G}} \times conf_{C\text{-}IL+\mathcal{G}} \to conf_{C\text{-}IL}$$

as

$$C\text{-}IL(\pi_{\mathcal{G}},c).\mathcal{M} = c.\mathcal{M}$$

$$C\text{-}IL(\pi_{\mathcal{G}},c).s = \begin{cases} \mathbf{map}(C\text{-}IL, c.s[si_{\mathcal{G}}(\pi_{\mathcal{G}}, c_{\mathcal{G}}) - 1 : 0]) & si_{\mathcal{G}}(\pi_{\mathcal{G}}, c_{\mathcal{G}}) \in \mathbb{N} \\ \mathbf{map}(C\text{-}IL, c.s) & si_{\mathcal{G}}(\pi_{\mathcal{G}}, c_{\mathcal{G}}) = \bot \end{cases}$$

That is, we drop all pure ghost frames and convert the remaining *C-IL*+$\mathcal{G}$ stack frames to *C-IL* stack frames.

6.4.2 Software Conditions and Invariants for Simulation

In order to be able to prove simulation between an annotated program and its non-annotated version, we need several software conditions and we make use of a few invariants over program execution. These are described in the following.

Definition 6.62 (Software Condition 1 (Non-Ghost Expression does not use Ghost)) Given a *C-IL* expression $e \in \mathbb{E}$, a function name $f \in \mathbb{F}_{name}$ and a *C-IL+$\mathcal{G}$* program $\pi_{\mathcal{G}}$, we define the predicate

$$\begin{aligned}\mathbf{sw}_1(\pi_{\mathcal{G}}) \equiv \forall (e,f) \in \mathit{expr}(\pi_{\mathcal{G}}) : \quad & \mathit{vnames}(e) \subseteq \mathit{decl}(\pi_{\mathcal{G}}.\mathcal{V}_G) \cup \mathit{decl}(\pi_{\mathcal{G}}.\mathcal{F}(f).\mathcal{V}) \\ & \wedge \mathit{fnames}(e) \subseteq \mathit{dom}(\pi_{\mathcal{G}}.\mathcal{F})\end{aligned}$$

which states that all variable names and functions occurring in e are declared in *C-IL*($\pi_{\mathcal{G}}$) if expression e occurs in a given function f. Here, *expr* is a function that takes a *C-IL+$\mathcal{G}$* program and returns pairs of expression and function name in which the expression occurs by recursively collecting all subexpressions of implementation statements of the program, *vnames*(e) returns the set of all variable names occurring in e, and *fnames*(e) returns the set of all function names occurring in e. This property can be checked statically, i.e. by just considering the annotated program text. We need this software condition in order to show that expression evaluation in *C-IL+$\mathcal{G}$* and the corresponding projected *C-IL* configuration result in the same value.

Definition 6.63 (Software Condition 2 (Ghost Assignment has Ghost Left Value)) For a *C-IL+$\mathcal{G}$* configuration $c_{\mathcal{G}}$, a *C-IL+$\mathcal{G}$* program $\pi_{\mathcal{G}}$ and *C-IL* environment parameters θ, we define the predicate

$$\mathbf{sw}_2(c_{\mathcal{G}}, \pi_{\mathcal{G}}, \theta) \equiv \mathit{stmt}_{next}(c_{\mathcal{G}}) \in \{\mathbf{ghost}(e_0 = e_1), \mathbf{ghost}(e = \mathbf{allocghost}\ t)\} \Rightarrow {}^{\mathcal{G}}[\![\&e_0]\!]^{\pi,\theta}_{c,\emptyset} \in \mathit{val}_{\mathcal{G}}$$

which states that, if the next statement to execute in $c_{\mathcal{G}}$ is a ghost assignment or allocation statement, the address to be written will evaluate to a ghost left value. We need this condition in order to show that ghost statements never change implementation state.

Definition 6.64 (Software Condition 3 (Ghost Gotos Never Leave a Ghost Block)) We define the predicate

$$\begin{aligned}\mathbf{sw}_3(\pi_{\mathcal{G}}) \equiv \quad & \forall f \in \pi_{\mathcal{G}}.\mathcal{F} : \forall l < |\pi_{\mathcal{G}}.\mathcal{F}(f).\mathcal{P}| : \\ & (\pi_{\mathcal{G}}.\mathcal{F}(f).\mathcal{P}[l] = \mathbf{ghost}(\mathbf{goto}\ l') \vee \pi_{\mathcal{G}}.\mathcal{F}(f).\mathcal{P}[l] = \mathbf{ghost}(\mathbf{ifnot}\ e\ \mathbf{goto}\ l')) \Rightarrow \\ & \quad \mathit{count}_{\mathbf{stmt}}(\pi_{\mathcal{G}}.\mathcal{F}(f).\mathcal{P}, l') = \mathit{count}_{\mathbf{stmt}}(\pi_{\mathcal{G}}.\mathcal{F}(f).\mathcal{P}, l)\end{aligned}$$

which, given a *C-IL+$\mathcal{G}$* program $\pi_{\mathcal{G}}$ states that there are no ghost-gotos that leave a block of ghost instructions. This is expressed in terms of the number of implementation statements that can be counted up to the location l of the ghost goto statement and the target location l' of the ghost goto statement. We need this condition in order to show that annotating ghost code does not affect the control flow of the implementation program.

Definition 6.65 (Software Condition 4 (Return Destination of Ghost Function))
For a *C-IL+$\mathcal{G}$* configuration $c_{\mathcal{G}}$, a *C-IL+$\mathcal{G}$* program $\pi_{\mathcal{G}}$ and *C-IL* environment parameters θ, we define the predicate

$$\mathbf{sw}_4(c_{\mathcal{G}}, \pi_{\mathcal{G}}, \theta) \equiv stmt_{next}(c_{\mathcal{G}}) = \mathbf{ghost}(e_0 = \mathbf{call}\ e(E)) \Rightarrow {}^{\mathcal{G}}[\![\&e_0]\!]^{\pi,\theta}_{c,\emptyset} \in val_{\mathcal{G}}$$

which states that the left side of a ghost function call statement has a ghost lvalue. We need this in order to show that ghost code does not modify implementation state.

Definition 6.66 (Invariant 1 (Return Destination of Ghost Function has Ghost LValue))
By requiring software condition 4 for all reachable states, we can establish the following invariant on *C-IL+$\mathcal{G}$* configurations

$$\mathbf{Inv}_1(c_{\mathcal{G}}) \equiv (\forall i \in [si_{\mathcal{G}}(\pi_{\mathcal{G}}, c_{\mathcal{G}}) : |c_{\mathcal{G}}.s|] : \quad c_{\mathcal{G}}.s[i].rds \in val_{\mathcal{G}})$$

which allows to deduce that executing a ghost return statement in *C-IL+$\mathcal{G}$* does not modify implementation state.

Definition 6.67 (Software Condition 5 (Validity of Ghost Code)) We state a predicate for *C-IL+$\mathcal{G}$* programs $\pi_{\mathcal{G}}$ that expresses validity of the program as

$$\begin{aligned}\mathbf{sw}_5(\pi_{\mathcal{G}}) \equiv\ & (\forall (e,f) \in expr_{\mathcal{G}}(\pi_{\mathcal{G}}) :\\ &\quad vnames(e) \subseteq (decl(\pi_{\mathcal{G}}.\mathcal{V}_G) \cup decl_{\mathcal{G}}(\pi_{\mathcal{G}}.\mathcal{V}_{GG}) \cup decl(\pi_{\mathcal{G}}.\mathcal{F}(f).\mathcal{V})\\ &\qquad\quad \cup decl_{\mathcal{G}}(\pi_{\mathcal{G}}.\mathcal{F}(f).\mathcal{V}_{\mathcal{G}}) \cup decl_{\mathcal{G}}(\pi_{\mathcal{G}}.\mathcal{F}_{\mathcal{G}}(f).\mathcal{V}_{\mathcal{G}}))\\ &\quad \wedge fnames(e) \subseteq dom(\pi_{\mathcal{G}}.\mathcal{F}) \cup dom(\pi_{\mathcal{G}}.\mathcal{F}_{\mathcal{G}}))\\ &\wedge well\text{-}typed_{\mathcal{G}}(\pi_{\mathcal{G}})\\ &\wedge well\text{-}formed_{\mathcal{G}}(\pi_{\mathcal{G}})\end{aligned}$$

where $expr_{\mathcal{G}}$ is a function that takes a *C-IL+$\mathcal{G}$* program and returns a set of pairs of expression and function name in which the expression occurs that is obtained by recursively collecting all subexpressions of ghost statements. Validity of ghost code and state includes that all variables and function names in ghost expressions occurring in ghost code are properly declared in either the ghost or implementation part of the program, and that ghost code is well-typed. We omit the definition of $well\text{-}typed_{\mathcal{G}}$ here since it is quite obvious how to define well-typedness for *C-IL+$\mathcal{G}$* statements and *C-IL+$\mathcal{G}$* expressions recursively. The predicate $well\text{-}formed_{\mathcal{G}}(\pi_{\mathcal{G}})$ expresses certain well-formedness conditions over *C-IL+$\mathcal{G}$* programs:

- there are no ghost return statements in implementation functions, and
  $(\forall f \in dom(\pi_{\mathcal{G}}.\mathcal{F}) : \forall i < |\pi_{\mathcal{G}}.\mathcal{F}(f).\mathcal{P}| :$
  $\pi_{\mathcal{G}}.\mathcal{F}(f).\mathcal{P}[i] \notin \{\mathbf{ghost}(\mathbf{return}), \mathbf{ghost}(\mathbf{return}\ e)\})$

- ghost function calls always call ghost functions directly.
  $(\forall f \in dom(\pi_{\mathcal{G}}.\mathcal{F}_{\mathcal{G}}) : \forall i < |\pi_{\mathcal{G}}.\mathcal{F}_{\mathcal{G}}(f).\mathcal{P}| :$
  $\pi_{\mathcal{G}}.\mathcal{F}_{\mathcal{G}}(f).\mathcal{P}[i] \in \{\mathbf{ghost}(\mathbf{call}\ e(E)), \mathbf{ghost}(e_0 = \mathbf{call}\ e(E))\} \Rightarrow e \in dom(\pi_{\mathcal{G}}.\mathcal{F}_{\mathcal{G}}) \subset \mathbb{F}_{name})$

This property can be checked statically.

Definition 6.68 (Invariant 2 (Ghost Code Can Always Execute)) Using software condition 5 and the definition of the *C-IL+$\mathcal{G}$* transition function, we maintain the following invariant over *C-IL+$\mathcal{G}$* configurations:

$$\mathbf{Inv}_2(\pi_{\mathcal{G}}, \theta, c_{\mathcal{G}}) \equiv \mathit{stmt}_{next} \in \mathbb{S}_{\mathcal{G}} \Rightarrow \exists c'_{\mathcal{G}} : \delta^{\pi_{\mathcal{G}},\theta}_{\text{C-IL+}\mathcal{G}}(c_{\mathcal{G}}, \bot) = c'_{\mathcal{G}}$$

The machine of the operational semantics never gets stuck on ghost steps for valid ghost code.

Definition 6.69 (Invariant 3 (Return Destinations of Implementation Stack)) Also derived from software condition 5, we have for *C-IL+$\mathcal{G}$* configurations c and *C-IL+$\mathcal{G}$* programs π

$$\mathbf{Inv}_3(c, \pi) \equiv \mathit{rds}_{top}(\mathit{C\text{-}IL}(\pi, c)) = \bot \wedge (\forall i < |\mathit{C\text{-}IL}(\pi, c).s| : c_{\mathcal{G}}.s[i].\mathit{rds} \in \mathit{val} \cup \bot)$$

that the return destination of the projected top-most implementation frame is $\bot$ and that return destinations in the implementation part of the stack are all implementation values.

Definition 6.70 (Invariant 4 (State of the Ghost Stack)) For a *C-IL+$\mathcal{G}$* configuration $c_{\mathcal{G}}$ and a *C-IL+$\mathcal{G}$* program $\pi_{\mathcal{G}}$,

$$\begin{aligned}\mathbf{Inv}_4(c_{\mathcal{G}}, \pi_{\mathcal{G}}) \equiv\ & (\mathit{stmt}_{next}(c_{\mathcal{G}}) \in \mathbb{S}' \Rightarrow \mathit{si}_{\mathcal{G}}(\pi_{\mathcal{G}}, c_{\mathcal{G}}) = \bot) \\ & \wedge (\mathit{stmt}_{next}(c_{\mathcal{G}}) \in \{\mathbf{ghost}(\mathbf{return}\ e), \mathbf{ghost}(\mathbf{return})\} \Rightarrow \mathit{si}_{\mathcal{G}}(\pi_{\mathcal{G}}, c_{\mathcal{G}}) \neq \bot)\end{aligned}$$

when the next statement to execute is an implementation statement, there are no ghost frames on the stack, while, when the next statement to execute is a ghost return statement, there is at least one ghost frame on the stack.

Definition 6.71 (Software Condition 6 (Ghost Code Terminates)) Given a *C-IL+$\mathcal{G}$* configuration $c_{\mathcal{G}}$, a *C-IL+$\mathcal{G}$* program $\pi_{\mathcal{G}}$ and *C-IL* environment parameters θ, the predicate

$$\mathbf{sw}_6(c_{\mathcal{G}}, \pi_{\mathcal{G}}, \theta) \equiv \exists i, c'_{\mathcal{G}} : {\delta^{\pi_{\mathcal{G}},\theta}_{\text{C-IL+}\mathcal{G}}}^{i}(c_{\mathcal{G}}, \bot) = c'_{\mathcal{G}} \wedge \mathit{stmt}_{next}(c'_{\mathcal{G}}) \in \mathbb{S}'$$

states that we can always reach an implementation statement by executing a finite number of steps. This implies that ghost code must always terminate.

Definition 6.72 (All Software Conditions/Invariants in a Single Predicate) While many of the software conditions are expressed on particular configurations, we actually need to require them for all reachable configurations to prove simulation. Thus, we provide the predicate

$$\begin{aligned}P_{\mathbf{sw}}(c_{\mathcal{G}}, \pi_{\mathcal{G}}, \theta) \equiv\ & \mathbf{sw}_1(\pi_{\mathcal{G}}) \wedge \mathbf{sw}_3(\pi_{\mathcal{G}}) \wedge \mathbf{sw}_5(\pi_{\mathcal{G}}) \\ & \wedge \forall \beta, c'_{\mathcal{G}} : c_{\mathcal{G}} \underset{\delta^{\pi_{\mathcal{G}},\theta}_{\text{C-IL+}\mathcal{G}}}{\overset{\beta}{\Rightarrow}} c'_{\mathcal{G}} \Rightarrow \mathbf{sw}_2(c'_{\mathcal{G}}, \pi_{\mathcal{G}}, \theta) \wedge \mathbf{sw}_4(c'_{\mathcal{G}}, \pi_{\mathcal{G}}) \wedge \mathbf{sw}_6(c'_{\mathcal{G}}, \pi_{\mathcal{G}}, \theta) \\ & \wedge \mathbf{Inv}_1(c_{\mathcal{G}}) \wedge \mathbf{Inv}_2(c_{\mathcal{G}}, \pi_{\mathcal{G}}, \theta) \wedge \mathbf{Inv}_3(c_{\mathcal{G}}, \pi_{\mathcal{G}}) \wedge \mathbf{Inv}_4(c_{\mathcal{G}}, \pi_{\mathcal{G}})\end{aligned}$$

that collects all software conditions for all reachable states from given configuration $c_{\mathcal{G}}$ of a *C-IL+$\mathcal{G}$* program $\pi_{\mathcal{G}}$ under *C-IL* environment parameters θ. Here, $\underset{\delta^{\pi_{\mathcal{G}},\theta}_{\text{C-IL+}\mathcal{G}}}{\overset{\beta}{\Rightarrow}}$ denotes the relation that describes execution of several *C-IL+$\mathcal{G}$* steps under finite input sequence β.

6.4.3 Lemmas for Simulation

We present a collection of lemmas that we will need to prove a simulation between *C-IL+$\mathcal{G}$* and *C-IL* before we state the simulation theorem and give a paper and pencil proof of the theorem. The majority of these lemmas relies on the software conditions and invariants stated in the last subsection, thus, the reader should assume $P_{\mathbf{sw}}$ to hold for the configurations, programs and environment parameters occuring in the following lemmas.

Lemma 6.73 (Next Statement to Execute) Given a *C-IL+$\mathcal{G}$* configuration $c_{\mathcal{G}}$ and *C-IL+$\mathcal{G}$* program $\pi_{\mathcal{G}}$, the following holds: $stmt_{next}(c_{\mathcal{G}}) \notin \mathbb{S}_{\mathcal{G}} \Rightarrow$

$$stmt_{next}(C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})) = \begin{cases} \textbf{goto } (count_{\textbf{stmt}}(\mathcal{P}_{top}(c_{\mathcal{G}}), l)) & stmt_{next}(c_{\mathcal{G}}) = \textbf{goto } l \\ \textbf{ifnot } e \textbf{ goto } (count_{\textbf{stmt}}(\mathcal{P}_{top}(c_{\mathcal{G}}), l)) & stmt_{next}(c_{\mathcal{G}}) = \textbf{ifnot } e \textbf{ goto } l \\ C\text{-}IL(stmt_{next}(c_{\mathcal{G}})) & \text{otherwise} \end{cases}$$

That is, whenever the next statement to be executed in $c_{\mathcal{G}}$ is an implementation statement, the next statement to be executed in the projected *C-IL* configuration $C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})$ is an equivalent *C-IL* statement.

Lemma 6.74 (Expression Evaluation) Given a *C-IL+$\mathcal{G}$* configuration $c_{\mathcal{G}}$, a *C-IL+$\mathcal{G}$* program $\pi_{\mathcal{G}}$ and *C-IL* environment parameters θ, we have

$$\forall e \in \mathbb{E}, f \in \mathbb{F}_{name}: \quad (e, f) \in expr(\pi_{\mathcal{G}}) \Rightarrow {}^{\mathcal{G}}[\![e]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset} = [\![e]\!]^{C\text{-}IL(\pi_{\mathcal{G}}),\theta}_{C\text{-}IL(\pi_{\mathcal{G}},c_{\mathcal{G}})}$$

When software condition 1 is fulfilled for a *C-IL* expression e, we know that e will evaluate to the same value in the projected *C-IL* configuration as it does in *C-IL+$\mathcal{G}$*.

Lemma 6.75 (Write-Equivalence) Given a *C-IL+$\mathcal{G}$* configuration $c_{\mathcal{G}}$, *C-IL+$\mathcal{G}$* program $\pi_{\mathcal{G}}$ and *C-IL* environment parameters θ, the following holds:

$$\forall x, y \in val: \quad write(\theta, C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}), x, y) = C\text{-}IL(write_{\mathcal{G}}(\theta, \pi_{\mathcal{G}}, c_{\mathcal{G}}, x, y))$$

Writing some implementation value y at implementation address x in $c_{\mathcal{G}}$ has the same effect on the projected configuration as first projecting the configuration and then writing y to x.

Lemma 6.76 (Incrementing the Location Counter) Given a *C-IL+$\mathcal{G}$* configuration $c_{\mathcal{G}}$ and a *C-IL+$\mathcal{G}$* program $\pi_{\mathcal{G}}$, we have

$$stmt_{next} \in \mathbb{S}' \wedge inc_{loc}(C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})) = C\text{-}IL(inc_{loc}(c_{\mathcal{G}}))$$

$$\vee\; stmt_{next} \in \mathbb{S}_{\mathcal{G}} \wedge C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}) = C\text{-}IL(inc_{loc}(c_{\mathcal{G}}))$$

Either, the next statement to execute is a non-ghost statement and incrementing the location counter on the projected configuration yields the same as first increasing the location counter and then performing the projection, or, the next statement is a ghost statement and incrementing the location counter on the configuration with ghost state yields the same projected configuration as before.

Lemma 6.77 (Writing to Ghost State) For a *C-IL+$\mathcal{G}$* configuration $c_\mathcal{G}$, *C-IL+$\mathcal{G}$* program $\pi_\mathcal{G}$ and *C-IL* environment parameters θ,

$$\forall x, y \in val \cup val_\mathcal{G} : \quad x \in val_\mathcal{G} \Rightarrow \textit{C-IL}(\pi_\mathcal{G}, c_\mathcal{G}) = \textit{C-IL}(write_\mathcal{G}(\theta, \pi_\mathcal{G}, c_\mathcal{G}, x, y))$$

writing to ghost state does not change the projected configuration.

Lemma 6.78 (Setting the Location Counter) Given a *C-IL+$\mathcal{G}$* configuration $c_\mathcal{G}$, a *C-IL+$\mathcal{G}$* program $\pi_\mathcal{G}$ and a program location l,

$$\textit{C-IL}(set_{loc}(c_\mathcal{G}, l)) = set_{loc}(\textit{C-IL}(\pi_\mathcal{G}, c_\mathcal{G}), count_{\mathbf{stmt}}(\mathcal{P}_{top\mathcal{G}}(c_\mathcal{G}), l))$$

setting the location in $c_\mathcal{G}$ is the same as setting the location in the projected configuration to the number of implementation statements that can be counted up to location l in $\mathcal{P}_{top\mathcal{G}}(c_\mathcal{G})$.

Lemma 6.79 (Ghost-Goto Preserves Projected Location) For a given *C-IL+$\mathcal{G}$* configuration $c_\mathcal{G}$, *C-IL+$\mathcal{G}$* program $\pi_\mathcal{G}$, and program location l, we have

$$stmt_{next}(c_\mathcal{G}) \in \{\mathbf{ghost}(\mathbf{goto}\ l), \mathbf{ghost}(\mathbf{ifnot}\ e\ \mathbf{goto}\ l)\} \rightarrow \textit{C-IL}(set_{loc}(c_\mathcal{G}, l)) = \textit{C-IL}(\pi_\mathcal{G}, c_\mathcal{G})$$

that setting the location in $c_\mathcal{G}$ to some location l of a ghost-goto-statement does not change the location in the projected configuration due to software condition 3.

Lemma 6.80 (Updating the Stack in a Configuration) Given a *C-IL+$\mathcal{G}$* configuration $c_\mathcal{G}$, *C-IL+$\mathcal{G}$* program $\pi_\mathcal{G}$, and a new *C-IL+$\mathcal{G}$* stack s', we have

$$\textit{C-IL}(\pi_\mathcal{G}, c_\mathcal{G}[s := s']) = \begin{cases} \textit{C-IL}(\pi_\mathcal{G}, c_\mathcal{G})[s := \mathbf{map}(\textit{C-IL}, s'[si_\mathcal{G}(\pi_\mathcal{G}, c_\mathcal{G}) - 1 : 0])] & si_\mathcal{G}(\pi_\mathcal{G}, c_\mathcal{G}) \neq \bot \\ \textit{C-IL}(\pi_\mathcal{G}, c_\mathcal{G})[s := \mathbf{map}(\textit{C-IL}, s')] & \text{otherwise} \end{cases}$$

Projecting a configuration where the stack was updated to s' is the same as first projecting the configuration and then updating the projected configuration with the projected value.

Lemma 6.81 (Dropping Frame) For a *C-IL+$\mathcal{G}$* configuration $c_\mathcal{G}$ and a *C-IL+$\mathcal{G}$* program $\pi_\mathcal{G}$, we have

$$si_\mathcal{G}(\pi_\mathcal{G}, c_\mathcal{G}) = \bot \wedge drop_{frame}(\textit{C-IL}(\pi_\mathcal{G}, c_\mathcal{G})) = \textit{C-IL}(\pi_\mathcal{G}, drop_{frame}(c_\mathcal{G}))$$
$$\vee\ si_\mathcal{G}(\pi_\mathcal{G}, c_\mathcal{G}) \neq \bot \wedge \textit{C-IL}(\pi_\mathcal{G}, c_\mathcal{G}) = \textit{C-IL}(\pi_\mathcal{G}, drop_{frame}(c_\mathcal{G}))$$

If the topmost frame is a non-ghost frame, projecting the configuration after dropping the frame yields the same as projecting the configuration first, then dropping the frame, while dropping ghost frames does not change the projected configuration.

Lemma 6.82 (Return Destinations of Implementation Stack) We have for *C-IL+$\mathcal{G}$* configurations

$$\forall i < |\textit{C-IL}(\pi_\mathcal{G}, c_\mathcal{G}).s| - 1 : c_\mathcal{G}.s[i].rds = \textit{C-IL}(\pi_\mathcal{G}, c_\mathcal{G}).s[i].rds$$

that the return destinations of the implementation part of the *C-IL+$\mathcal{G}$* stack are identical to those of the projected configuration.

Lemma 6.83 (Updating the Ghost Heap Allocation Counter) For a *C-IL+$\mathcal{G}$* configuration $c_{\mathcal{G}}$ and a natural number x,

$$C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}[nf_{\mathcal{G}} := x]) = C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})$$

updating the $nf_{\mathcal{G}}$-component does not change the projected configuration.

Lemma 6.84 (Merging Implementation and Ghost) Given a *C-IL+$\mathcal{G}$* configuration $c_{\mathcal{G}}$, a *C-IL+$\mathcal{G}$* program $\pi_{\mathcal{G}}$ and a *C-IL* configuration c, we have

$$(\forall i \in [0 : |c.s| - 1] : c.s[i].f = c_{\mathcal{G}}.s[i].f \wedge c.s[i].loc = count_{\mathbf{stmt}}(\pi_{\mathcal{G}}.\mathcal{F}(c_{\mathcal{G}}.s[i].f).\mathcal{P}, c_{\mathcal{G}}.s[i].loc))$$

$$\Rightarrow C\text{-}IL(\pi_{\mathcal{G}}, merge\text{-}impl\text{-}ghost(c_{\mathcal{G}}, c)) = c.$$

Merging the implementation configuration with the *C-IL+$\mathcal{G}$* configuration and projecting afterwards results in the merged implementation configuration.

6.4.4 Simulation Proof

Ultimately, we aim at the simulation between concurrent *C-IL* and concurrent *C-IL+$\mathcal{G}$* as stated here:

Theorem 6.85 (Simulation between Concurrent *C-IL+$\mathcal{G}$* and Projected Concurrent *C-IL*) *Given a* C-IL+$\mathcal{G}$ *program $\pi_{\mathcal{G}}$ and* C-IL *environment parameters θ, as well as a concurrent* C-IL+$\mathcal{G}$ *configuration $c_{\mathcal{G}}$ and a concurrent* C-IL *configuration c,*

$$\begin{aligned} \forall c', t, in : \quad & P_{Csw}(c_{\mathcal{G}}, \pi_{\mathcal{G}}, \theta) \wedge c \sim c_{\mathcal{G}} \wedge c \underset{\delta_{\text{CC-IL}}^{\text{C-IL}(\pi_{\mathcal{G}}),\theta}}{\overset{t,in}{\rightsquigarrow}} c' \Rightarrow \\ & \exists c'_{\mathcal{G}} : c_{\mathcal{G}} \underset{\delta_{\text{CC-IL+}\mathcal{G}}^{\pi_{\mathcal{G}},\theta}}{\overset{t,in}{\rightsquigarrow}} c'_{\mathcal{G}} \wedge c' \sim c'_{\mathcal{G}} \end{aligned}$$

for every step of a thread t of a concurrent C-IL *program there exists a step of the corresponding concurrent* C-IL+$\mathcal{G}$ *program in such a way that the simulation relation given by the projection function is preserved for each thread. We use*

$$\begin{aligned} c \sim c_{\mathcal{G}} \quad \overset{def}{\Leftrightarrow} \quad & dom(c.Th) = dom(c_{\mathcal{G}}.Th) \\ & \wedge (\forall t : th(t, c) = \text{C-IL}(\pi_{\mathcal{G}}, th(t, c_{\mathcal{G}}))) \end{aligned}$$

and

$$P_{Csw}(c_{\mathcal{G}}, \pi_{\mathcal{G}}, \theta) \quad \overset{def}{\Leftrightarrow} \quad (\forall t : P_{sw}(th(t, c_{\mathcal{G}}), \pi_{\mathcal{G}}, \theta))$$

where $th(t, c)$ denotes the sequential configuration of the t-th thread of configuration c.

This, however can easily be proven by applying the sequential simulation theorem below:

Theorem 6.86 (Simulation between Sequential *C-IL+$\mathcal{G}$* and the Projected *C-IL*) *Given a* C-IL+$\mathcal{G}$ *program* $\pi_{\mathcal{G}}$ *and* C-IL *environment parameters* θ*, as well as a* C-IL+$\mathcal{G}$ *configuration* $c_{\mathcal{G}}$*, we show that*

$$\forall c', in : \quad P_{sw}(c_{\mathcal{G}}, \pi_{\mathcal{G}}, \theta) \wedge \text{C-IL}(\pi_{\mathcal{G}}, c_{\mathcal{G}}) \underset{\delta_{\text{C-IL}}^{\text{C-IL}(\pi_{\mathcal{G}}),\theta}}{\overset{in}{\rightsquigarrow}} c' \Rightarrow$$
$$\exists c'_{\mathcal{G}}, c''_{\mathcal{G}} : c_{\mathcal{G}} \underset{{}_{\mathcal{G}}\delta_{\text{C-IL+}\mathcal{G}}^{\pi_{\mathcal{G}},\theta}}{\rightsquigarrow} c''_{\mathcal{G}} \underset{\delta_{\text{C-IL+}\mathcal{G}}^{\pi_{\mathcal{G}},\theta}}{\overset{in}{\rightsquigarrow}} c'_{\mathcal{G}} \wedge c' = \text{C-IL}(\pi_{\mathcal{G}}, c'_{\mathcal{G}})$$

every C-IL *step of the projected configuration under the projected program is matched by executing an implementation step and its preceding ghost steps of the* C-IL+$\mathcal{G}$ *configuration under the* C-IL+$\mathcal{G}$ *program in such a way that the resulting* C-IL *configuration is the projection of the resulting* C-IL+$\mathcal{G}$ *configuration.*

This theorem can in turn be proven by using the following step-by-step simulation lemma:

Lemma 6.87 (Sequential Step-by-Step Simulation) Given a *C-IL+$\mathcal{G}$* program $\pi_{\mathcal{G}}$ and *C-IL* environment parameters θ, the following holds:

$$\begin{aligned}\forall c_{\mathcal{G}} : \quad & P_{\mathbf{sw}}(c_{\mathcal{G}}, \pi_{\mathcal{G}}, \theta) \Rightarrow \\ & (stmt_{next}(c_{\mathcal{G}}) \in \mathbb{S}_{\mathcal{G}} \wedge (\exists c'_{\mathcal{G}} : c_{\mathcal{G}} \underset{\delta_{\textit{C-IL+}\mathcal{G}}^{\pi_{\mathcal{G}},\theta}}{\overset{\perp}{\rightsquigarrow}} c'_{\mathcal{G}} \wedge \textit{C-IL}(\pi_{\mathcal{G}}, c_{\mathcal{G}}) = \textit{C-IL}(\pi_{\mathcal{G}}, c'_{\mathcal{G}})) \\ & \vee stmt_{next}(c_{\mathcal{G}}) \in \mathbb{S}' \wedge (\forall c', in : \textit{C-IL}(\pi_{\mathcal{G}}, c_{\mathcal{G}}) \underset{\delta_{\textit{C-IL}}^{\textit{C-IL}(\pi_{\mathcal{G}}),\theta}}{\overset{in}{\rightsquigarrow}} c' \\ & \qquad\qquad \Rightarrow (\exists c'_{\mathcal{G}} : c_{\mathcal{G}} \underset{\delta_{\textit{C-IL+}\mathcal{G}}^{\pi_{\mathcal{G}},\theta}}{\overset{in}{\rightsquigarrow}} c'_{\mathcal{G}} \wedge c' = \textit{C-IL}(\pi_{\mathcal{G}}, c'_{\mathcal{G}}))))\end{aligned}$$

The *C-IL+$\mathcal{G}$* program may either make a ghost step, preserving the projected configuration, or it must match any implementation step that is possible from the projected configuration.

Proof Let a *C-IL+$\mathcal{G}$* program $\pi_{\mathcal{G}}$ and *C-IL* environment parameters θ, as well as a *C-IL+$\mathcal{G}$* configuration $c_{\mathcal{G}}$ with

$$P_{\mathbf{sw}}(c_{\mathcal{G}}, \pi_{\mathcal{G}}, \theta)$$

be given. Depending on whether the next statement to be executed in $c_{\mathcal{G}}$ is a ghost or an implementation statement, we have to prove either that the *C-IL+$\mathcal{G}$* program makes a step without changing the projected configuration, or that, given a step of the projected program, the *C-IL+$\mathcal{G}$* program performs a matching implementation step.

In the following, we perform a case distinction on the next statement to be executed in $c_{\mathcal{G}}$:

1. **Case: Annotated Implementation Statement** $stmt_{next}(c_{\mathcal{G}}) \in \mathbb{S}'$

   We have to show

$$\forall c', in : \textit{C-IL}(\pi_{\mathcal{G}}, c_{\mathcal{G}}) \underset{\delta_{\textit{C-IL}}^{\textit{C-IL}(\pi_{\mathcal{G}}),\theta}}{\overset{in}{\rightsquigarrow}} c' \Rightarrow (\exists c'_{\mathcal{G}} : c_{\mathcal{G}} \underset{\delta_{\textit{C-IL+}\mathcal{G}}^{\pi_{\mathcal{G}},\theta}}{\overset{in}{\rightsquigarrow}} c'_{\mathcal{G}} \wedge c' = \textit{C-IL}(\pi_{\mathcal{G}}, c'_{\mathcal{G}}))$$

   Thus, let a *C-IL* configuration c' and an input to *C-IL* semantics $in \in \Sigma$ be given such that

$$\delta_{\textit{C-IL}}^{\textit{C-IL}(\pi_{\mathcal{G}}),\theta}(\textit{C-IL}(\pi_{\mathcal{G}}, c_{\mathcal{G}}), in) = c'$$

a) **Case: Assignment** $stmt_{next}(c_{\mathcal{G}}) = (e_0 = e_1)$:

We use lemma 6.73 to conclude that $stmt_{next}(C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})) = e_0 = e_1$, and thus, *C-IL*-semantics results in a step to

$$c' = inc_{loc}(write(\theta, C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}), [\![\&e_0]\!]^{C\text{-}IL(\pi_{\mathcal{G}}),\theta}_{C\text{-}IL(\pi_{\mathcal{G}},c_{\mathcal{G}})}, [\![e_1]\!]^{C\text{-}IL(\pi_{\mathcal{G}}),\theta}_{C\text{-}IL(\pi_{\mathcal{G}},c_{\mathcal{G}})}))$$

From the semantics of *C-IL+$\mathcal{G}$*, we can see that from $c_{\mathcal{G}}$ a step to

$$c'_{\mathcal{G}} := inc_{loc}(write_{\mathcal{G}}(\theta, \pi_{\mathcal{G}}, c_{\mathcal{G}}, {}^{\mathcal{G}}[\![\&e_0]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset}, {}^{\mathcal{G}}[\![e_1]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset}))$$

can be performed.

We merely have left to prove that $C\text{-}IL(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) = c'$ holds.

$$\begin{aligned}
&C\text{-}IL(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) \\
&= C\text{-}IL(\pi_{\mathcal{G}}, inc_{loc}(write_{\mathcal{G}}(\theta, \pi_{\mathcal{G}}, c_{\mathcal{G}}, {}^{\mathcal{G}}[\![\&e_0]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset}, {}^{\mathcal{G}}[\![e_1]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset}))) \\
&= C\text{-}IL(\pi_{\mathcal{G}}, inc_{loc}(write_{\mathcal{G}}(\theta, \pi_{\mathcal{G}}, c_{\mathcal{G}}, [\![\&e_0]\!]^{C\text{-}IL(\pi_{\mathcal{G}}),\theta}_{C\text{-}IL(\pi_{\mathcal{G}},c_{\mathcal{G}})}, [\![e_1]\!]^{C\text{-}IL(\pi_{\mathcal{G}}),\theta}_{C\text{-}IL(\pi_{\mathcal{G}},c_{\mathcal{G}})}))) && (6.74, \mathbf{sw}_1) \\
&= inc_{loc}(C\text{-}IL(\pi_{\mathcal{G}}, write_{\mathcal{G}}(\theta, \pi_{\mathcal{G}}, c_{\mathcal{G}}, [\![\&e_0]\!]^{C\text{-}IL(\pi_{\mathcal{G}}),\theta}_{C\text{-}IL(\pi_{\mathcal{G}},c_{\mathcal{G}})}, [\![e_1]\!]^{C\text{-}IL(\pi_{\mathcal{G}}),\theta}_{C\text{-}IL(\pi_{\mathcal{G}},c_{\mathcal{G}})}))) && (6.76) \\
&= inc_{loc}(write(\theta, C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}), [\![\&e_0]\!]^{C\text{-}IL(\pi_{\mathcal{G}}),\theta}_{C\text{-}IL(\pi_{\mathcal{G}},c_{\mathcal{G}})}, [\![e_1]\!]^{C\text{-}IL(\pi_{\mathcal{G}}),\theta}_{C\text{-}IL(\pi_{\mathcal{G}},c_{\mathcal{G}})})) && (6.75) \\
&= c' && \surd
\end{aligned}$$

b) **Case: Goto** $stmt_{next}(c_{\mathcal{G}}) = (\textbf{goto } l)$

$$\overset{(6.73)}{(\Rightarrow} stmt_{next}(C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})) = \textbf{goto } (count_{\mathbf{stmt}}(\mathcal{P}_{top\mathcal{G}}(c_{\mathcal{G}}), l)))$$

In this case, we have

$$c' = set_{loc}(c, count_{\mathbf{stmt}}(\mathcal{P}_{top\mathcal{G}}(c_{\mathcal{G}}), l))$$

From the semantics of *C-IL+$\mathcal{G}$*, we obtain

$$c'_{\mathcal{G}} = set_{loc}(c_{\mathcal{G}}, l)$$

Then,

$$\begin{aligned}
C\text{-}IL(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) &= C\text{-}IL(\pi_{\mathcal{G}}, set_{loc}(c_{\mathcal{G}}, l)) \\
&= set_{loc}(C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}), count_{\mathbf{stmt}}(\mathcal{P}_{top\mathcal{G}}(c_{\mathcal{G}}), l)) && (6.78) \\
&= set_{loc}(c, count_{\mathbf{stmt}}(\mathcal{P}_{top\mathcal{G}}(c_{\mathcal{G}}), l)) \\
&= c' && \surd
\end{aligned}$$

c) **Case: If-Not-Goto** $stmt_{next}(c_{\mathcal{G}}) = (\textbf{ifnot } e \textbf{ goto } l)$

$$\overset{(6.73)}{(\Rightarrow} stmt_{next}(C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})) = \textbf{ifnot } e \textbf{ goto } (count_{\mathbf{stmt}}(\mathcal{P}_{top\mathcal{G}}(c_{\mathcal{G}}), l)))$$

In this case, we have

$$c' = \begin{cases} set_{loc}(\textit{C-IL}(\pi_{\mathcal{G}}, c_{\mathcal{G}}), count_{\mathbf{stmt}}(\mathcal{P}_{top\mathcal{G}}(c_{\mathcal{G}}), l)) & zero(\theta, [\![e]\!]^{\textit{C-IL}(\pi_{\mathcal{G}}),\theta}_{\textit{C-IL}(\pi_{\mathcal{G}},c_{\mathcal{G}})}) \\ inc_{loc}(\textit{C-IL}(\pi_{\mathcal{G}}, c_{\mathcal{G}})) & \text{otherwise} \end{cases}$$

From the semantics of *C-IL+$\mathcal{G}$*:

$$c'_{\mathcal{G}} = \begin{cases} set_{loc}(c_{\mathcal{G}}, l) & zero_{\mathcal{G}}(\theta, {}^{\mathcal{G}}[\![e]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset}) \\ inc_{loc}(c_{\mathcal{G}}) & \text{otherwise} \end{cases}$$

Then,

$$\begin{aligned}
&\textit{C-IL}(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) \\
&= \begin{cases} \textit{C-IL}(\pi_{\mathcal{G}}, set_{loc}(c_{\mathcal{G}}, l)) & zero_{\mathcal{G}}(\theta, {}^{\mathcal{G}}[\![e]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset}) \\ \textit{C-IL}(\pi_{\mathcal{G}}, inc_{loc}(c_{\mathcal{G}})) & \text{otherwise} \end{cases} \\
&= \begin{cases} \textit{C-IL}(\pi_{\mathcal{G}}, set_{loc}(c_{\mathcal{G}}, l)) & zero(\theta, [\![e]\!]^{\textit{C-IL}(\pi_{\mathcal{G}}),\theta}_{\textit{C-IL}(\pi_{\mathcal{G}},c_{\mathcal{G}})}) \\ \textit{C-IL}(\pi_{\mathcal{G}}, inc_{loc}(c_{\mathcal{G}})) & \text{otherwise} \end{cases} && (\mathbf{sw}_1, 6.74, zero_{def}) \\
&= \begin{cases} set_{loc}(\textit{C-IL}(\pi_{\mathcal{G}}, c_{\mathcal{G}}), count_{\mathbf{stmt}}(\mathcal{P}_{top\mathcal{G}}(c_{\mathcal{G}}), l)) & zero(\theta, [\![e]\!]^{\textit{C-IL}(\pi_{\mathcal{G}}),\theta}_{\textit{C-IL}(\pi_{\mathcal{G}},c_{\mathcal{G}})}) \\ inc_{loc}(\textit{C-IL}(\pi_{\mathcal{G}}, c_{\mathcal{G}})) & \text{otherwise} \end{cases} && (6.78, 6.76) \\
&= c' && \surd
\end{aligned}$$

d) **Case: Regular Function/Procedure Call**
($stmt_{next}(c_{\mathcal{G}}) = \mathbf{call}\ e(E, E') \lor stmt_{next}(c_{\mathcal{G}}) = e_0 = \mathbf{call}\ e(E, E')$)

$(\overset{(6.73)}{\Rightarrow}\ stmt_{next}(\textit{C-IL}(\pi_{\mathcal{G}}, c_{\mathcal{G}})) = \mathbf{call}\ e(E) \lor stmt_{next}(\textit{C-IL}(\pi_{\mathcal{G}}, c_{\mathcal{G}})) = e_0 = \mathbf{call}\ e(E))$

From *C-IL* semantics we know that, in order to make this step, the following must already be fulfilled:

- $in \in \mathbb{V} \to (\mathbb{B}^8)^*$
- $[\![e]\!]^{\textit{C-IL}(\pi_{\mathcal{G}}),\theta}_{\textit{C-IL}(\pi_{\mathcal{G}},c_{\mathcal{G}})} = \mathbf{val}(b, \mathbf{funptr}(t, T)) \land f = \theta.\mathcal{F}_{adr}{}^{-1}(b)$
  $\lor [\![e]\!]^{\textit{C-IL}(\pi_{\mathcal{G}}),\theta}_{\textit{C-IL}(\pi_{\mathcal{G}},c_{\mathcal{G}})} = \mathbf{fun}(f, \mathbf{funptr}(t, T))$
- $\mathcal{F}^{\theta}_{\textit{C-IL}(\pi_{\mathcal{G}})}(f).\mathcal{P} \neq \mathbf{extern}$
- $\forall i \geq \mathcal{F}^{\theta}_{\textit{C-IL}(\pi_{\mathcal{G}})}(f).npar : \exists v, t : \mathcal{F}^{\theta}_{\textit{C-IL}(\pi_{\mathcal{G}})}(f).\mathcal{V}[i] = (v, t) \Rightarrow |in(v)| = size_{\theta}(t)$

According to *C-IL* semantics, we then have

$$c' = \textit{C-IL}(\pi_{\mathcal{G}}, c)[s := (\mathcal{M}'_{\mathcal{E}}, \bot, f, 0) \circ inc_{loc}(set_{rds}(\textit{C-IL}(\pi_{\mathcal{G}}, c_{\mathcal{G}}), rds)).s]$$

where

$$rds = \begin{cases} [\![\&e_0]\!]^{\textit{C-IL}(\pi_{\mathcal{G}}),\theta}_{\textit{C-IL}(\pi_{\mathcal{G}},c_{\mathcal{G}})} & stmt_{next}(\textit{C-IL}(\pi_{\mathcal{G}}, c_{\mathcal{G}})) = e_0 = \mathbf{call}\ e(E) \\ \text{undefined} & stmt_{next}(\textit{C-IL}(\pi_{\mathcal{G}}, c_{\mathcal{G}})) = \mathbf{call}\ e(E) \end{cases}$$

and

$$\mathcal{M}'_{\mathcal{E}}(v) = \begin{cases} \mathit{val2bytes}_\theta([\![E[i]]\!]^{\textit{C-IL}(\pi_{\mathcal{G}}),\theta}_{\textit{C-IL}(\pi_{\mathcal{G}},c_{\mathcal{G}})}) & \exists i : \mathcal{F}^\theta_{\textit{C-IL}(\pi_{\mathcal{G}})}(f).\mathcal{V}[i] = (v,t) \\ & \wedge\ i < \mathcal{F}^\theta_{\textit{C-IL}(\pi_{\mathcal{G}})}(f).\mathit{npar} \\ \mathit{in}(v) & \exists i.\ \mathcal{F}^\theta_{\textit{C-IL}(\pi_{\mathcal{G}})}(f).\mathcal{V}[i] = (v,t) \\ & \wedge\ i \geq \mathcal{F}^\theta_{\textit{C-IL}(\pi_{\mathcal{G}})}(f).\mathit{npar} \\ \text{undefined} & \text{otherwise} \end{cases}$$

We have to show that there exists a configuration $c'_{\mathcal{G}}$ with $\textit{C-IL}(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) = c'$ according to *C-IL*+$\mathcal{G}$ semantics. In order for the *C-IL*+$\mathcal{G}$ semantics to make a function or procedure call step, we need to show

- ${}^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,\emptyset} = \mathbf{val}(b, \mathbf{funptr}(t,T)) \wedge f = \theta.\mathcal{F}_{adr}{}^{-1}(b) \vee {}^{\mathcal{G}}[\![e]\!]^{\pi,\theta}_{c,\emptyset} = \mathbf{fun}(f, \mathbf{funptr}(t,T))$
- $\mathcal{F}_{\textit{C-IL+}\mathcal{G}}(f).\mathcal{P} \neq \mathbf{extern}$
- $\forall i \geq \mathcal{F}_{\textit{C-IL+}\mathcal{G}}(f).\mathit{npar} : \exists v,t : \mathcal{F}_{\textit{C-IL+}\mathcal{G}}(f).\mathcal{V}[i] = (v,t) \Rightarrow |\mathit{in}(v)| = \mathit{size}_\theta(t)$

These conditions can easily be shown using software condition 1 in conjunction with lemma 6.74 as well as the definition of the projection functions for *C-IL*+$\mathcal{G}$ function tables.

The step performed then results in configuration

$$c'_{\mathcal{G}} = c_{\mathcal{G}}[s := (\mathcal{M}''_{\mathcal{E}}, \mathcal{M}''_{\mathcal{E}\mathcal{G}}, \bot, f, 0) \circ \mathit{inc}_{loc}(\mathit{set}_{rds}(c_{\mathcal{G}}, \mathit{rds}_{\mathcal{G}})).s]$$

where

$$\mathit{rds}_{\mathcal{G}} = \begin{cases} {}^{\mathcal{G}}[\![\&e_0]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset} & \mathit{stmt}_{next}(c_{\mathcal{G}}) = (e_0 = \mathbf{call}\ e(E,E')) \\ \text{undefined} & \mathit{stmt}_{next}(c_{\mathcal{G}}) = \mathbf{call}\ e(E,E') \end{cases}$$

and

$$\mathcal{M}''_{\mathcal{E}}(v) = \begin{cases} \mathit{val2bytes}_\theta({}^{\mathcal{G}}[\![E[i]]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset}) & \exists i : \mathcal{F}^\theta_{\pi_{\mathcal{G}}}(f).\mathcal{V}[i] = (v,t) \\ & \wedge\ i < \mathcal{F}_{\textit{C-IL+}\mathcal{G}}(f).\mathit{npar} \\ \mathit{in}(v) & \exists i.\ \mathcal{F}_{\textit{C-IL+}\mathcal{G}}(f).\mathcal{V}[i] = (v,t) \\ & \wedge\ i \geq \mathcal{F}_{\textit{C-IL+}\mathcal{G}}(f).\mathit{npar} \\ \text{undefined} & \text{otherwise} \end{cases}$$

and

$$\mathcal{M}''_{\mathcal{E}\mathcal{G}}(v) = \begin{cases} {}^{\mathcal{G}}[\![E'[i]]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset} & \exists i : \mathcal{F}_{\textit{C-IL+}\mathcal{G}}(f).\mathcal{V}_{\mathcal{G}}[i] = (v,t) \wedge i < \mathcal{F}_{\textit{C-IL+}\mathcal{G}}(f).\mathit{ngpar} \\ \text{undefined} & \text{otherwise} \end{cases}$$

As always, we have to show $C\text{-}IL(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) = c'$:

$$\begin{aligned}
&C\text{-}IL(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) && \\
&= C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}[s := (\mathcal{M}''_{\mathcal{E}}, \mathcal{M}''_{\mathcal{E}\mathcal{G}}, \bot, f, 0) \circ inc_{loc}(set_{rds}(c_{\mathcal{G}}, rds_{\mathcal{G}})).s]) && \\
&= C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})[s := \mathbf{map}(C\text{-}IL, (\mathcal{M}''_{\mathcal{E}}, \mathcal{M}''_{\mathcal{E}\mathcal{G}}, \bot, f, 0) \circ inc_{loc}(set_{rds}(c_{\mathcal{G}}, rds_{\mathcal{G}})).s)] && (\mathbf{Inv}_4, 6.80) \\
&= C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})[s := C\text{-}IL((\mathcal{M}''_{\mathcal{E}}, \mathcal{M}''_{\mathcal{E}\mathcal{G}}, \bot, f, 0)) \circ \mathbf{map}(C\text{-}IL, inc_{loc}(c_{\mathcal{G}}).s)] && \text{Def. } \mathbf{map} \\
&= C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})[s := (\mathcal{M}''_{\mathcal{E}}, \bot, f, 0) \circ \mathbf{map}(C\text{-}IL, inc_{loc}(c_{\mathcal{G}}).s)] && \text{Def. } C\text{-}IL \\
&= C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})[s := (\mathcal{M}''_{\mathcal{E}}, \bot, f, 0) \circ C\text{-}IL(\pi_{\mathcal{G}}, inc_{loc}(c_{\mathcal{G}})).s] && (\mathbf{Inv}_4, 6.61) \\
&= C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})[s := (\mathcal{M}''_{\mathcal{E}}, \bot, f, 0) \circ inc_{loc}(C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})).s] && (6.76)
\end{aligned}$$

Using software condition 1 and lemma 6.74, we see that $\mathcal{M}'_{\mathcal{E}} = \mathcal{M}''_{\mathcal{E}}$ and, thus, $C\text{-}IL(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) = c'$ $\quad\surd$

e) **External Procedure Call** $stmt_{next}(c_{\mathcal{G}}) = \mathbf{call}\ e(E, \varepsilon)$

$$(\overset{(6.73)}{\Rightarrow} stmt_{next}(C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})) = \mathbf{call}\ e(E))$$

From $C\text{-}IL$ semantics, we know

- $c' = in$ (the resulting configuration is non-deterministically chosen and provided as input parameter to the $C\text{-}IL$ transition function, $\delta^{\pi,\theta}_{C\text{-}IL}(\pi_{\mathcal{G}}, c, in) = in$)
- $[\![e]\!]^{C\text{-}IL(\pi_{\mathcal{G}}),\theta}_{C\text{-}IL(\pi_{\mathcal{G}},c_{\mathcal{G}})} = \mathbf{val}(b, \mathbf{funptr}(t, T)) \wedge f = \theta.\mathcal{F}_{adr}{}^{-1}(b) \vee [\![e]\!]^{C\text{-}IL(\pi_{\mathcal{G}}),\theta}_{C\text{-}IL(\pi_{\mathcal{G}},c_{\mathcal{G}})} = \mathbf{fun}(f, \mathbf{funptr}(t, T))$
- $\mathbf{tl}(c'.s) = \mathbf{tl}(inc_{loc}(C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})).s)$
- $\mathbf{hd}(c'.s).loc = \mathbf{hd}(C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}).s).loc + 1 \wedge \mathbf{hd}(c'.s).f = \mathbf{hd}(C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}).s).f$
- $\mathcal{F}^{\theta}_{C\text{-}IL(\pi_{\mathcal{G}})}(f).\mathcal{P} = \mathbf{extern}$
- $([\![E[0]]\!]^{C\text{-}IL(\pi_{\mathcal{G}}),\theta}_{C\text{-}IL(\pi_{\mathcal{G}},c_{\mathcal{G}})}, \ldots, [\![E[|E|-1]]\!]^{C\text{-}IL(\pi_{\mathcal{G}}),\theta}_{C\text{-}IL(\pi_{\mathcal{G}},c_{\mathcal{G}})}, C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}), c') \in \theta.R_{\mathbf{extern}}$

Using these conditions, we prove the corresponding $C\text{-}IL\text{+}\mathcal{G}$ conditions for external procedure call applying software condition 1 and lemma 6.74.

According to the semantics of $C\text{-}IL\text{+}\mathcal{G}$, we have

$$c'_{\mathcal{G}} = merge\text{-}impl\text{-}ghost(inc_{loc}(c_{\mathcal{G}}), c')$$

for which we can show $C\text{-}IL(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) = c'$ by applying lemma 6.84.

f) **Case: Return** $stmt_{next}(c_{\mathcal{G}}) = \mathbf{return}\ e \vee stmt_{next}(c_{\mathcal{G}}) = \mathbf{return}$

$$(\overset{(6.73)}{\Rightarrow} stmt_{next}(C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})) = \mathbf{return}\ e \vee stmt_{next}(C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})) = \mathbf{return})$$

According to $C\text{-}IL$ semantics,

$$c' = \begin{cases} write(\theta, drop_{frame}(C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})), rds), [\![e]\!]^{C\text{-}IL(\pi_{\mathcal{G}}),\theta}_{C\text{-}IL(\pi_{\mathcal{G}},c_{\mathcal{G}})}) & rds \neq \bot \\ drop_{frame}(C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})) & \text{otherwise} \end{cases}$$

where $rds = rds_{top}(drop_{frame}(C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}))$

From *C-IL*+$\mathcal{G}$-semantics we have

$$c'_{\mathcal{G}} = \begin{cases} write_{\mathcal{G}}(\theta, \pi_{\mathcal{G}}, drop_{frame}(c_{\mathcal{G}}), rds_{\mathcal{G}}, {}^{\mathcal{G}}[\![e]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset}) & rds_{\mathcal{G}} \neq \bot \\ drop_{frame}(c_{\mathcal{G}}) & \text{otherwise} \end{cases}$$

where $rds_{\mathcal{G}} = rds_{top}(drop_{frame}(c_{\mathcal{G}}))$

Then,

$$\begin{aligned} & C\text{-}IL(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) \\ &= \begin{cases} C\text{-}IL(\pi_{\mathcal{G}}, write_{\mathcal{G}}(\theta, \pi_{\mathcal{G}}, drop_{frame}(c_{\mathcal{G}}), rds_{\mathcal{G}}, {}^{\mathcal{G}}[\![e]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset})) & rds_{\mathcal{G}} \neq \bot \\ C\text{-}IL(\pi_{\mathcal{G}}, drop_{frame}(c_{\mathcal{G}})) & \text{otherwise} \end{cases} \\ &= \begin{cases} C\text{-}IL(\pi_{\mathcal{G}}, write_{\mathcal{G}}(\theta, \pi_{\mathcal{G}}, drop_{frame}(c_{\mathcal{G}}), rds, [\![e]\!]^{C\text{-}IL(\pi_{\mathcal{G}}),\theta}_{C\text{-}IL(\pi_{\mathcal{G}},c_{\mathcal{G}})})) & rds \neq \bot \\ C\text{-}IL(\pi_{\mathcal{G}}, drop_{frame}(c_{\mathcal{G}})) & \text{otherwise} \end{cases} && (6.74, 6.82) \\ &= \begin{cases} write(\theta, C\text{-}IL(\pi_{\mathcal{G}}, drop_{frame}(c_{\mathcal{G}})), rds, [\![e]\!]^{C\text{-}IL(\pi_{\mathcal{G}}),\theta}_{C\text{-}IL(\pi_{\mathcal{G}},c_{\mathcal{G}})}) & rds \neq \bot \\ C\text{-}IL(\pi_{\mathcal{G}}, drop_{frame}(c_{\mathcal{G}})) & \text{otherwise} \end{cases} && (6.75) \\ &= \begin{cases} write(\theta, drop_{frame}(C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})), rds, [\![e]\!]^{C\text{-}IL(\pi_{\mathcal{G}}),\theta}_{C\text{-}IL(\pi_{\mathcal{G}},c_{\mathcal{G}})}) & rds \neq \bot \\ drop_{frame}(C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})) & \text{otherwise} \end{cases} && (\mathbf{Inv}_4, 6.81) \\ &= c' && \surd \end{aligned}$$

2. **Case: Ghost Statement** $stmt_{next}(c_{\mathcal{G}}) \in \mathbb{S}_{\mathcal{G}}$ We have to show

$$\exists c'_{\mathcal{G}} : c_{\mathcal{G}} \overset{\bot}{\underset{\delta^{\pi_{\mathcal{G}},\theta}_{C\text{-}IL+\mathcal{G}}}{\rightsquigarrow}} c'_{\mathcal{G}} \wedge C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}) = C\text{-}IL(\pi_{\mathcal{G}}, c'_{\mathcal{G}})$$

From invariant 2, we know that there exists a *C-IL*+$\mathcal{G}$ configuration $c'_{\mathcal{G}}$ such that

$$\delta^{\pi_{\mathcal{G}},\theta}_{C\text{-}IL+\mathcal{G}}(c_{\mathcal{G}}, \bot) = c'_{\mathcal{G}}$$

a) **Case: Assignment** $stmt_{next}(c_{\mathcal{G}}) = \mathbf{ghost}(e_0 = e_1)$

From the semantics of *C-IL*+$\mathcal{G}$, we see that a step from $c_{\mathcal{G}}$ to

$$c'_{\mathcal{G}} = inc_{loc}(write_{\mathcal{G}}(\theta, \pi_{\mathcal{G}}, c_{\mathcal{G}}, {}^{\mathcal{G}}[\![\&e_0]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset}, {}^{\mathcal{G}}[\![e_1]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset}))$$

is performed.

Then, we have

$$\begin{aligned} & C\text{-}IL(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) \\ &= C\text{-}IL(\pi_{\mathcal{G}}, inc_{loc}(write_{\mathcal{G}}(\theta, \pi_{\mathcal{G}}, c_{\mathcal{G}}, {}^{\mathcal{G}}[\![\&e_0]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset}, {}^{\mathcal{G}}[\![e_1]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset}))) \\ &= C\text{-}IL(\pi_{\mathcal{G}}, write_{\mathcal{G}}(\theta, \pi_{\mathcal{G}}, c_{\mathcal{G}}, {}^{\mathcal{G}}[\![\&e_0]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset}, {}^{\mathcal{G}}[\![e_1]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset})) && (6.76) \\ &= C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}) && (\mathbf{sw}_2, 6.77)\surd \end{aligned}$$

b) **Case: Goto** $stmt_{next}(c_{\mathcal{G}}) = \textbf{ghost}(\textbf{goto}\ l)$

We have

$$c'_{\mathcal{G}} = set_{loc}(c_{\mathcal{G}}, l)$$

Then,

$$\begin{aligned} C\text{-}IL(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) &= C\text{-}IL(\pi_{\mathcal{G}}, set_{loc}(c_{\mathcal{G}}, l)) \\ &= C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}) && (\mathbf{sw}_3, 6.79)\surd \end{aligned}$$

c) **Case: If-Not-Goto** $stmt_{next}(c_{\mathcal{G}}) = \textbf{ghost}(\textbf{ifnot}\ e\ \textbf{goto}\ l)$

We have

$$c'_{\mathcal{G}} = \begin{cases} set_{loc}(c_{\mathcal{G}}, l) & zero_{\mathcal{G}}(\theta, {}^{\mathcal{G}}[\![e]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset}) \\ inc_{loc}(c_{\mathcal{G}}) & \text{otherwise} \end{cases}$$

Then,

$$\begin{aligned} C\text{-}IL(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) &= \begin{cases} C\text{-}IL(\pi_{\mathcal{G}}, set_{loc}(c_{\mathcal{G}}, l)) & zero_{\mathcal{G}}(\theta, {}^{\mathcal{G}}[\![e]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset}) \\ C\text{-}IL(\pi_{\mathcal{G}}, inc_{loc}(c_{\mathcal{G}})) & \text{otherwise} \end{cases} \\ &= \begin{cases} C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}) & zero(\theta, [\![e]\!]^{C\text{-}IL(\pi_{\mathcal{G}}),\theta}_{C\text{-}IL(\pi_{\mathcal{G}},c_{\mathcal{G}})}) \\ C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}) & \text{otherwise} \end{cases} && (\mathbf{sw}_3, 6.79, 6.76) \\ &= C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}) && \surd \end{aligned}$$

d) **Case: Function/Procedure Call** $(stmt_{next}(c_{\mathcal{G}}) = \textbf{ghost}(\textbf{call}\ e(E)) \vee (stmt_{next}(c_{\mathcal{G}}) = \textbf{ghost}(e_0 = \textbf{call}\ e(E))))$

From $C\text{-}IL{+}\mathcal{G}$ semantics, we have

$$c'_{\mathcal{G}} = c_{\mathcal{G}}[s := (\emptyset, \mathcal{M}'_{\mathcal{EG}}, \bot, f, 0) \circ inc_{loc}(set_{rds}(c_{\mathcal{G}}, rds)).s]$$

where

$$rds = \begin{cases} {}^{\mathcal{G}}[\![\&e_0]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset} & stmt_{next}(c_{\mathcal{G}}) = \textbf{ghost}(e_0 = \textbf{call}\ e(E)) \\ \text{undefined} & stmt_{next}(c_{\mathcal{G}}) = \textbf{ghost}(\textbf{call}\ e(E)) \end{cases}$$

and

$$\mathcal{M}'_{\mathcal{EG}}(v) = \begin{cases} {}^{\mathcal{G}}[\![E[i]]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset} & \exists i : \mathcal{F}_{C\text{-}IL+\mathcal{G}}(f).\mathcal{V}_{\mathcal{G}}[i] = (v, t) \wedge i < \mathcal{F}_{C\text{-}IL+\mathcal{G}}(f).ngpar \\ \text{undefined} & \text{otherwise} \end{cases}$$

We only have to show $C\text{-}IL(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) = C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})$.

Since according to software condition 5, f is a ghost function, we know by definition of $si_{\mathcal{G}}$ that $si_{\mathcal{G}}(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) \neq \bot$ and that $si_{\mathcal{G}}(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) \leq |c_{\mathcal{G}}.s|$. Further, we can easily see by definition of $si_{\mathcal{G}}$ that

$$(*) \qquad si_{\mathcal{G}}(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) = \begin{cases} si_{\mathcal{G}}(\pi_{\mathcal{G}}, c_{\mathcal{G}}) & si_{\mathcal{G}}(\pi_{\mathcal{G}}, c_{\mathcal{G}}) \neq \bot \\ |c_{\mathcal{G}}.s| & \text{otherwise} \end{cases}$$

Let $c'' = inc_{loc}(set_{rds}(c_{\mathcal{G}}, rds))$ and

$$s' = \begin{cases} \mathbf{map}(C\text{-}IL, c''.s[sig_{\mathcal{G}}(\pi_{\mathcal{G}}, c_{\mathcal{G}}) - 1 : 0])] & sig_{\mathcal{G}}(\pi_{\mathcal{G}}, c_{\mathcal{G}}) \neq \bot \\ \mathbf{map}(C\text{-}IL, c''.s[|c_{\mathcal{G}}.s| - 1 : 0])] & \text{otherwise} \end{cases}$$

Thus, we have

$$\begin{aligned} & C\text{-}IL(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) \\ &= C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}[s := (\emptyset, \mathcal{M}'_{\mathcal{E}\mathcal{G}}, \bot, f, 0) \circ inc_{loc}(set_{rds}(c_{\mathcal{G}}, rds)).s]) \\ &= C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})[s := \mathbf{map}(C\text{-}IL, ((\emptyset, \mathcal{M}'_{\mathcal{E}\mathcal{G}}, \bot, f, 0) \circ \\ &\qquad\qquad c''.s)[sig_{\mathcal{G}}(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) - 1 : 0]] && (6.80) \\ &= C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})[s := \mathbf{map}(C\text{-}IL, c''.s[sig_{\mathcal{G}}(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) - 1 : 0])] && sig_{\mathcal{G}}(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) \\ & && \leq |c_{\mathcal{G}}.s| \\ &= C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})[s := s' && (*) \\ &= C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})[s := C\text{-}IL(\pi_{\mathcal{G}}, inc_{loc}(set_{rds}(c_{\mathcal{G}}, rds))).s] && (6.61) \\ &= C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})[s := C\text{-}IL(\pi_{\mathcal{G}}, set_{rds}(c_{\mathcal{G}}, rds)).s] && (6.76) \\ &= C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}})[s := C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}).s] && (\mathbf{sw}_4, \mathbf{Inv}_3, 6.60) \\ &= C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}) && \surd \end{aligned}$$

Using software condition 4 we maintain invariant 1.

e) **Case: Return** $stmt_{next}(c_{\mathcal{G}}) = \mathbf{ghost}(\mathbf{return}\ e) \vee stmt_{next}(c_{\mathcal{G}}) = \mathbf{ghost}(\mathbf{return})$

From $C\text{-}IL{+}\mathcal{G}$-semantics we have

$$c'_{\mathcal{G}} = \begin{cases} write_{\mathcal{G}}(\theta, \pi_{\mathcal{G}}, drop_{frame}(c_{\mathcal{G}}), rds_{\mathcal{G}}, {}^{\mathcal{G}}[\![e]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset}) & rds_{\mathcal{G}} \neq \bot \\ drop_{frame}(c_{\mathcal{G}}) & \text{otherwise} \end{cases}$$

where $rds_{\mathcal{G}} = rds_{top}(drop_{frame}(c_{\mathcal{G}}))$

Then,

$$\begin{aligned} C\text{-}IL(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) &= \begin{cases} C\text{-}IL(\pi_{\mathcal{G}}, drop_{frame}(c_{\mathcal{G}})) & rds_{\mathcal{G}} \neq \bot \\ C\text{-}IL(\pi_{\mathcal{G}}, drop_{frame}(c_{\mathcal{G}})) & \text{otherwise} \end{cases} && \mathbf{Inv}_1 + (6.77) \\ &= \begin{cases} C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}) & c_{\mathcal{G}}.rds_{top} \neq \bot \\ C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}) & \text{otherwise} \end{cases} && (\mathbf{Inv}_4, 6.81) \\ &= C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}) && \surd \end{aligned}$$

f) **Case: Ghost Allocation** $stmt_{next}(c_{\mathcal{G}}) = \mathbf{ghost}(e = \mathbf{allocghost}\ t)$

From $C\text{-}IL{+}\mathcal{G}$ semantics we have

$$c'_{\mathcal{G}} = write_{\mathcal{G}}(\theta, \pi_{\mathcal{G}}, inc_{loc}(c_{\mathcal{G}}), {}^{\mathcal{G}}[\![\&e]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset}, \mathbf{gref}(c_{\mathcal{G}}.nf_{\mathcal{G}}, \varepsilon, \mathbf{ptr}(t)))[nf_{\mathcal{G}} := c_{\mathcal{G}}.nf_{\mathcal{G}} + 1]$$

Thus,

$$\begin{aligned} & C\text{-}IL(\pi_{\mathcal{G}}, c'_{\mathcal{G}}) \\ &= C\text{-}IL(\pi_{\mathcal{G}}, write_{\mathcal{G}}(\theta, \pi_{\mathcal{G}}, inc_{loc}(c_{\mathcal{G}}), {}^{\mathcal{G}}[\![\&e]\!]^{\pi_{\mathcal{G}},\theta}_{c_{\mathcal{G}},\emptyset}, \mathbf{gref}(c_{\mathcal{G}}.nf_{\mathcal{G}}, \varepsilon, \mathbf{ptr}(t)))) && (6.83) \\ &= C\text{-}IL(\pi_{\mathcal{G}}, inc_{loc}(c_{\mathcal{G}})) && (\mathbf{sw}_2, 6.77) \\ &= C\text{-}IL(\pi_{\mathcal{G}}, c_{\mathcal{G}}) && (6.76)\surd \end{aligned}$$

Note that, in order to elaborate this to a complete proof, what is left to do is to prove all the lemmas and show that invariants are maintained for valid *C-IL*+$\mathcal{G}$ executions.

7 Future Work & Conclusion

In this thesis, we describe a model stack for pervasive formal verification of multi core operating systems and hypervisors. In particular, we defined cornerstone semantics of that model stack. This includes an abstract hardware model, a rather low-level intermediate language for C and an extension of that intermediate language with specification state and code which can be used to argue about the soundness of the verification tool VCC. We give the main lemmas and a pencil and paper proof of how to prove a simulation between program annotated with ghost code and the original program. This, however, is only a fraction of what needs to be done to achieve a pervasive theory of multi core systems and their verification with VCC. While a quite comprehensive list of future work and achieved results has been given already in chapter 3, the author would like to elaborate on a few specific topics related to the models given in this thesis in the following. A reference to where the corresponding item is mentioned in chapter 3 is given, when applicable. First, a few future work topics are discussed, followed by collaborations and subsequent work based on this dissertation.

Future Work

Extending and Improving MIPS-86 with Caches (Section 3.2.4) While MIPS-86 is a good start to serve as a top-level specification of the x86-64 inspired, reverse-engineered hardware models from [Pau12], it currently does not cover caches yet. Thus, a necessary extension is to introduce memory access modes (and a means to give control over memory access modes to the programmer) and replace the memory component by a memory system consisting of concurrent caches with visible cache states and a main memory on which memory accesses annotated with different access modes can be performed. In that model, reads to the memory system have side-effects: The cache state is modified according to the access mode. This necessarily results in a slight reformulation of the top-level transition function of MIPS-86: All reads from memory must be explicitly formalized as inputs to the memory system. Since all memory reads are given in the top-level transition function in order to pass them to the processor core's transition function (which performs instruction execution based on the values read from memory), performing this reformulation should be a simple task.

Completing the Reverse-Engineered Hardware Design for the MIPS-86 Model (Section 3.2.4) After caches have been introduced to the MIPS-86 system programmer specification, a main task is to give a complete the gate-level hardware design that actually implements MIPS-86. This involves integrating and extending existing gate-level designs, such as the pipelined processor core or the cache system from [Pau12], but also requires to construct gate-level designs for the APIC model and the MMU model of MIPS-86 in such a way that devices are accessible by memory mapped I/O.

Compiler Correctness One place where our formal theory is lacking is in the compiler correctness department. While a compiler correctness theorem (based on an explicit simulation relation) for *C-IL* has been stated and used in [Sha12], an actual compiler for *C-IL* has neither been defined nor implemented, and, as a consequence, we do not have a compiler correctness proof. In order, however, to transfer ownership safety from high levels of the model stack to the bottom layers, we do need to at least define a compiler, i.e. give a definition of the translation from ownership-annotated *C-IL* to ownership-annotated assembler code. Based on such a compiler definition, we can prove that safe *C-IL* code translates to safe assembler code – which in turn implies that concurrent *C-IL* is a sound abstraction of compiled machine code execution.

Since compilers are complex constructs when implemented in an efficient way, it is very tempting to resort to using the compiler correctness result of Xavier Leroy et. al's CompCert compiler [Ler09] – after all, their results are proven formally in Coq and describe quite an efficient optimizing C compiler. At the time of development of our theories, however, the compiler specification (which is given in terms of specific observable external events resulting from code execution, e.g. external function calls) of CompCert was too weak for our purposes. The collaboration between Appel and Leroy driven by Appel's verified software toolchain project [App11], however, has led to major improvements in the memory model of CompCert [LABS12] – including even a basic ownership discipline formulated in terms of permissions. The ownership discipline in question, however, is still simpler than the one we use in our model stack, thus, if we want to use the CompCert results, it has to be checked in what regards they are applicable and where extensions are needed. Another consequence of using the CompCert compiler in our model stack would be that we end up with a different (higher-level) C semantics for which the specification state and code extension would have to be redone. If we were to start this project anew at the time being however, evaluating the applicability of the CompCert results would be a high priority.

Documenting the Assertion Language of VCC (Section 3.7.5) *C-IL+$\mathcal{G}$* is defined with a soundness proof of VCC in mind. Using *C-IL+$\mathcal{G}$*, the assertion language and ownership discipline of VCC should be formalized in order to specify what exactly VCC proves about concurrent C programs. Except for some constructs of VCC, like claims, this should be quite simple since the memory model of VCC is compatible with the byte-addressable implementation memory of *C-IL+$\mathcal{G}$* and the control flow of *C-IL+$\mathcal{G}$* is very close to that of Boogie, the intermediate verification language VCC translates annotated C code to. Based on this semantics, the verification methodology of VCC could be proven sound as outlined in [Mica].

Collaborations, Subsequent Work and Work in Progress Based on this Thesis

Collaboration: *C-IL* Compiler Correctness Specification (Section 3.6) Andrey Shadrin formalized the compiler correctness theorem for *C-IL* in his dissertation [Sha12] in collaboration with the author of this thesis. A simulation relation that expresses when *C-IL* and baby

VAMP machine configurations are consistent is formalized explicitly and the simulation theorem states that the simulation relation is maintained at IO-points – which in the sequential case are essentially configurations before and after execution of an external function call.

Collaboration: Integrated Semantics of *C-IL* and Macro Assembler (Section 3.6) A formal model of *C-IL* and Macro Assembler code execution has been established in collaboration with Andrey Shadrin. The resulting integrated semantics which is based on the byte-addressable *C-IL* memory and stack abstraction can be found in the corresponding dissertation and conference paper [Sha12, SS12]. The main feature of the integrated semantics is the call stack between the two languages and the explicit modeling of compiler calling conventions without fully exposing the low-level stack layout. That the integrated semantics provides a sound abstraction of machine code execution is argued by applying the compiler correctness theorems for *C-IL* and MASM.

Subsequent Work: Simplified Model Stack for TLB Virtualization in the Context of Hypervisor Verification (Section 3.8.3) In his dissertation [Kov12], Mikhail Kovalev presents a detailed formulation of a simplified model stack for the x86-64 hardware model [Deg11] following the ideas presented in chapter 3 of this thesis. In order to prove TLB virtualization for a multi-core shadow page table algorithm, the focus is mainly on lifting the TLB model to *C-IL* level in order to justify the code verification done with VCC. Software conditions used in the given reduction theorems for other components of the multi-core architecture are more strict than we describe in chapter 3, resulting in a simplified but complete model stack.

Subsequent Work: Justifying the Approach of Macro Assembler Verification with a C Verifier (Section 3.7) Using the integrated semantics from [Sha12, SS12], Andrey Shadrin was able to state and prove correct a translation of Macro assembler code to *C-IL* code that implies a simulation between the original *C-IL+MASM* program and the resulting *C-IL* program in the spirit of the Vx86 assembler verification approach of Stefan Maus [MMS08].

Work in Progress: Concurrent Compiler Correctness (Section 3.6) Christoph Baumann uses his symbolic ownership-based order reduction theorem [Bau12] and the compiler correctness specification from [Sha12] to show that the concurrent *C-IL* model is a sound abstraction of concurrent machine code execution. The proof is based on using the ownership discipline to reorder the steps of individual *C-IL* threads to occur in sequential execution blocks between IO-points, which in the concurrent case also include volatile accesses.

Work in Progress: Proving Store Buffer Reduction on MIPS-86 (Section 3.5.5) As mentioned in chapter 3, in the presence of MMUs, the store buffer reduction theorem of [CS10] is not applicable: memory accesses of MMUs bypass store buffers. Thus, the model of the store buffer reduction theorem needs to be improved in order to allow an instantiation of the model – on which store buffer reduction is proven formally – with MIPS-86. There appear to be two particularly promising ways of how to do this: i) extend the model with components that access memory directly and allow communication between these components and designated program components, or ii) allow program components to issue store-buffer-bypassing memory accesses (which would allow instantiating processor core together with MMU as a program component). The main challenge here appears to lie in formulating an extended software condition – which is both simple and powerful – that restricts the store-buffer bypassing memory accesses in such a way that the store-buffers can still be reduced.

Work in Progress: Extending *C-IL+MASM* Semantics for a Thread-Switch Kernel (Section 3.8.3) One particularly interesting challenge encountered in the hypervisor verification project was to argue about the correctness of the kernel layer (which provides an abstraction of running many threads on individual processors by providing a cooperative thread switch routine and thread data structures) in a C verifier. C semantics tends to make use of some kind of stack abstraction. However, in implementing such a thread switch kernel, the explicit stack layout must be exploited to some degree in order to create threads, and the stack pointer registers of the machine must be accessed at thread switch – in order to store the stack pointers of the old thread and load the stack pointers of the new thread when performing the switch. The goal here is to extend *C-IL+MASM* semantics in such a way that a proper abstraction for threads and their stacks is provided on a level abstract enough to allow use of the C verifier in proving the correctness of the kernel implementation. Similar to the CVM model from the Verisoft project [Ver07] this can be done by providing appropriate *primitives* (i.e. external function calls that implement the desired functionality in pure assembler code; their effect being specified as an atomic step on the more abstract levels of the model stack) for thread creation and thread switch. The C verifier can then be used to argue about the kernel C code using the extended *C-IL+MASM* semantics with primitives – after we have proven that the implemented primitives indeed have the effect specified in the extended abstract semantics.

Conclusion

Establishing formal semantics for use in realistic systems is never a trivial task. Things get particularly interesting when the overall goal is to establish a pervasive theory in terms of a model stack with simulation theorems between neighboring models; now the task is not just to give an adequate model that describes machine or program execution, but, to give a model that is neither too specific nor too generic for the given abstraction level of the model stack. An additional design goal is that simulation between neighboring models should always be straightforward – avoiding unnecessary technical detours.

While an ultimate goal lies in proving symbolic theorems that can be instantiated for specific realistic languages and systems, getting to such a result depends largely on exploratory work as given in this thesis: In order to identify adequate symbolic models it is necessary to first solve interesting problems for specific machines and formalizations. In particular in the area of hardware verification and hardware modeling, there is still much work to be done in order to arrive at models that can be used in a pervasive theory. But also on higher levels of the model stack, it is often far from obvious what an ideal semantics would look like. In fact, which formulation is ideal depends quite obviously on the context in which the model is used: With application code verification in mind, a rather abstract programming language semantics with a high-level memory model is appropriate, while, for operating systems verification, dealing with a too high level of abstraction in programming language semantics can actually turn out to be more complicated than resorting to a lower-level formalization of programming language semantics in the first place. Indeed, in a model stack that goes from gate level up to application level, it even appears quite natural to have different levels of abstraction of the same programming language presented to the user of the pervasive theory (e.g. a low-level C semantics without the notion of a heap and a high-level C semantics with an abstract heap – which is obtained by implementing a library with memory management procedures in the low-level semantics, and, possibly, on the formalization side, by using a more abstract memory model). Unfortunately, it appears that, in computer science, the academic community is split into a (large) software community that overall prefers to deal with high-level abstract models and a (smaller) hardware community that appears not particularly interested in programming languages and compilers. This results in a quite large gap between the code-verification usually performed at a very high level of abstraction and the actual execution of machine code by an abstract hardware model. One of the main customers for models belonging into this gap is the (very small) operating systems verification community.

With this thesis, the author hopes to have brought some insight into what kinds of models can provide a solid basis for arguing about operating system and hypervisor implementations in VCC. We sketched a model stack for multi-core hypervisor and operating system verification and described important software conditions and simulation theorems needed to establish such a model stack. One of the main efforts was finding an appropriate order in which to apply reduction theorems that abstract away specific hardware features in the model stack – providing a roadmap to follow in constructing the desperately needed theory in chapter 3. In this process, we surveyed the current state of the theory and exposed gaps that need to be closed in order to achieve a sound verification with respect to a realistic underlying gate-level hardware construction.

While the models presented in this thesis are likely far from perfect, chapter 3 gives evidence that they at least appear to allow arguing about a pervasive model stack in a pretty straightforward way. The collaborations and subsequent work based on the models presented support this notion. The models so far proved to be particularly useful in providing a sanity check for theorems proven about symbolic models, e.g. the store-buffer reduction theorem of [CS10] or the general reorder theorem of [Bau12] – being able to instantiate a symbolic model to obtain a specific model of interest (e.g. such as MIPS-86) demonstrates the usefulness of the corresponding symbolic theory. The main advantage of the described MIPS-86 model over the one that was already established during the Verisoft-XT project, i.e. the x86-64 model defined by Ulan Degenbaev [Deg11], lies in its comparatively compact nature; the author made an effort to strip all unnecessary complexity while preserving the core features of the memory system which are reverse-engineered in the gate-level hardware designs provided in [Pau12]. A major problem with realistic models is, in fact, that they contain an abundance of little details which, by themselves, are not difficult to understand, but which nonetheless make it very difficult for a human to deal with these huge models; in particular, when the interesting parts of a correctness proof can also be expressed over a much simpler model and later be extended to a proof of the full detailed model with the help of computer-aided verification tools.

Indeed, the same is true for the programming language C: in order to implement and argue about an operating system or a hypervisor, only a tiny fragment of C is actually needed. Similar to how compilers use intermediate languages, verification tools like VCC use intermediate languages (e.g. VCC uses the intermediate verification language Boogie). In order to show soundness of such a verification tool, we can rely on the soundness of the verification condition generator for the intermediate verification language used. However, VCC is a verification tool that covers a very large subset of C code – mainly to provide a comfortable interface for the verification engineer – which means that the semantics of the generated intermediate verification language code and the original C code are quite different in structure. Thus, the task of proving VCC's soundness can be simplified by translating the C code to a C intermediate language whose semantics is structurally similar to the intermediate verification language (such as *C-IL*) and then proving a simulation between the resulting C intermediate language code and the generated intermediate verification language code, obtaining the soundness of verification condition generation for the C intermediate language; which can in turn be lifted to the C level. Since the VCC allows the verification engineer to use ghost state and quite powerful ghost code, it is necessary to prove the termination of ghost code – which in turn requires to have proper semantics for ghost code. These considerations resulted in the creation of *C-IL*+$\mathcal{G}$, a semantics of a C intermediate language combined with ghost code and state in the style provided by VCC. The goal of the language is to be both low-level enough to allow integration of low-level hardware features (e.g. such as MMUs setting accessed/dirty flags of page-tables, or allowing the semantics to be extended to properly model thread switch by writing stack pointers on an abstract level) and high-level enough to serve as a basis for arguing the soundness of the verification tool. These requirements naturally contradict each other: For proving soundness of a verification tool, it tends to be helpful to use a semantics as abstract as possible, while for arguing about code that exploits lowest-level hardware functionality, having a much more concrete semantics – closely matching the hardware machine model – is favored. The *C-IL* semantics given in this thesis results from a compromise between the two requirements which, so far, appears to work in an acceptable way for both applications: While the memory model of *C-IL* is as low-level as possible, the stack abstraction provided hides the concrete stack layout of the machine.

As has been argued, the main challenge in establishing a pervasive theory of multi-core systems lies in finding acceptable semantics for different layers of the model stack. This is why this thesis deals, to a large degree, with semantics. It is crucial to study semantics for realistic systems and, in a subsequent step, identify symbolic models of common interest; since systems are mostly built out of reuseable components, establishing symbolic theory for any such component is meaningful – albeit only after finding an appropriate abstract model that both captures exactly the relevant behavior and is applicable to a significant number of realistic implementations. It is crucial to identify how exactly the symbolic theory needs to be stated in order to allow different components and abstraction levels to be combined into models of real systems – symbolic models which can not be instantiated for realistic systems are effectively worthless. At the present time, while we are quite close to establishing a model stack for a specific multi-core architecture, we are still very far from establishing a symbolic model stack for multi-core architectures; such a symbolic model stack can be thought of as a patch-work of symbolic models and theorems which can be instantiated and arranged to obtain correctness results over different models that share particular features. While the store-buffer reduction theorem from [CS10] is one such symbolic theorem, it is too restricted to be applied to MIPS-86. However, realizing in which way the symbolic theorem is inadequate directs us towards a new and improved symbolic theorem which actually is flexible enough to be applied. Indeed, it would be desirable to have similar symbolic simulation theorems that abstract away other hardware features under appropriate software conditions, e.g. for MOESI-based cache systems or the virtual memory abstraction given by MMUs. To confirm that proposed symbolic models are indeed useful, MIPS-86 serves as an academic target architecture for which the proposed theory must be applicable.

Overall, it can be said that this thesis provides crucial semantic models and an outline for constructing a realistic multi-core model stack which can be used as a foundation in order to argue mathematically about the correctness of multi-core operating system and hypervisor verification.

Symbol Tables

In the following, we provide symbol tables for the basic notation used in this thesis (chapter 2), MIPS-86 (chapter 4), *C-IL* (chapter 5), and *C-IL*+$\mathcal{G}$ (chapter 6). Entries are ordered alphabetically, where possible. The page number given for a symbol table entry directs the reader to the definition of the corresponding symbol or function.

Basic Notation

$\mathbb{B}$	set of Booleans, 9
B_n	range of values of bit-strings interpreted as natural numbers, 10
$bin_n(x)$	binary representation of natural number x, 10
$byte(i,a)$	i-th byte of bit-string a, 11
$\mathcal{E}$	Hilbert-choice operator, 5
ε	empty finite sequence, 7
$\mathbf{hd}(x)$	head of a finite sequence x, 8
$\mathbf{map}(f,x)$	finite sequence obtained by applying function f to every element of finite sequence x, 8
$\equiv \mod k$	congruence modulo k, 11
mod	modulo operator, 11
modt	two's complement modulo operator, 11
$\mathbb{N}$	set of natural numbers (non-negative integers), 5
$\mathbf{rev}(x)$	finite sequence obtained by reverting the order of elements of finite sequence x, 8
$sxt_k(a)$	sign-extended version of bit-string a to length k, 10
T_n	range of values of bit-strings interpreted as two's-complement numbers, 10
$\mathbf{tl}(x)$	tail of a finite sequence x, 8
$twoc_n(x)$	two's-complement representation of integer x, 10
$\mathbb{Z}$	set of integers, 5
$zxt_k(a)$	zero-extended version of bit-string a to length k, 10
$\#A$	amount of elements of finite set A, 5

2^A	power ser of set A, 5
$f(x{:=}y)$	function update notation, replacing value of f for x by y, 6
$c[n_i{:=}a]$	record update notation, replacing value of field n_i of record c by a, 7
$\mathcal{X}^n$	set of n-tuples over the set $\mathcal{X}$, 7
$\mathcal{X}^*$	set of finite sequences over the set $\mathcal{X}$, 7
$\lvert x \rvert$	length of finite sequence x, 8
$a \circ b$	concatenation of finite sequences a and b, 8
x^n	finite sequence that repeats element x for n times, 8
$\wedge$	logic operator AND, 9
$\vee$	logic operator OR, 9
$\oplus$	logic operator XOR, 9
$(A?x : y)$	ternary operator that choses between value x and value y depending on statement A, 9
$\langle a[n-1:0]\rangle_B$	value of bit-string a interpreted as natural number to base B, 9
$[b]$	value of bit-string b interpreted as two's-complement number, 10
x_n	binary representation of natural number x (shorthand), 10
$x_{\mathbf{t}n}$	two's-complement representation of integer x (shorthand), 10
$+_n$	n-bit binary addition, 11
$-_n$	n-bit binary subtraction, 11
$\cdot_n$	n-bit binary multiplication, 11
$\cdot_{\mathbf{t}n}$	n-bit two's-complement multiplication, 12
$\div_n$	n-bit binary division, 12
$\div_{\mathbf{t}n}$	n-bit two's-complement division, 12

MIPS-86 Symbol Table

$.a$	accessed flag (of a page table entry), 57
$A_{\mathbf{apic}}$	set of byte-addresses covered by the local APIC I/O ports, 84
$A_{\mathbf{dev}}$	set of byte-addresses covered by device I/O ports, 84
$A_{\mathbf{dev(i)}}$	set of byte-addresses covered by device *i*'s I/O ports, 84
$A_{\mathbf{ioapic}}$	set of byte-addresses covered by the I/O APIC ports, 84
$alu(I)$	predicate that expresses whether the given instruction word *I* is an ALU instruction, 63
$alucon(I)$	ALU control bits of ALU instruction *I*, 64
$alures(a,b,alucon,i)$	ALU result for input values *a* and *b* given alu control bits *alucon* and *i*, 63
$.apic$	local APIC component (of a processor configuration), 52
$apic_{\mathbf{adr}}(x)$	function that computes the local APIC port address of a given memory address *x*, 84
$apic_{\mathbf{base}}$	base memory address of local APIC I/O ports, 84
.APIC_ID	shorthand for the APIC ID register (of the local APIC I/O ports), 77
$.asid$	address space identifier component (of a translation request), 59
$.asid$	address space identifier (ASID) component (of a TLB walk), 56
$asid(core)$	current address space identifier of processor core *core*, 85
$b(I)$	predicate that expresses whether instruction word *I* is a branch instruction, 64
$.ba$	base address of the next page table (of a page table entry), 57
$.ba$	physical page address component (of a TLB walk), 56
$.ba$	base address / page address (of an address), 57
$bcres(a,b,bcon)$	branch condition result for input values *a* and *b* given branch control bits *bcon*, 64
$btarget(c,I)$	function that computes the branch target of a jump of successful branch instruction *I* in processor core configuration *c*, 64
$ca(c,I,eev,pff,pfls)$	interrupt signals raised when instruction word *I* is executed in processor configuration *c* given external event signals *eev* and page-fault signals *pff* and *pfls*, 69
$cad(I)$	general purpose register address to which the result of an instruction *I* is to be written, 67

$compi(I)$	predicate that expresses whether the given instruction word I is an I-type ALU instruction, 63
$complete(w)$	predicate that expresses whether a TLB walk is complete, 59
$compr(I)$	predicate that expresses whether the given instruction word I is an R-type ALU instruction, 63
$compres(c,I)$	result value of ALU instruction I in processor core configuration c, 64
$continue(c,I,eev,pff,pfls)$	predicate that expresses whether the interrupt occuring during execution of instruction word I in processor configuration c given external event signals *eev* and page-fault signals *pff* and *pfls* is of type continue, 69
$.core$	processor core component (of a processor configuration), 51
$.d$	dirty flag (of a page table entry), 57
$d(I)$	access width in bytes of load/store instruction word I, 66
δ	MIPS-86 transition function, 82
$\delta_{\mathbf{apic}}$	transition function of the local APIC, 79
$\delta_{\mathbf{core}}$	transition function of the processor core, 61
$\delta_{\mathbf{dev(i)}}$	transition function of device i, 75
$\delta_{\mathbf{instr}}$	transition function for processor core instruction execution, 67
$\delta_{\mathbf{ioapic}}$	transition function of the I/O APIC, 81
$\delta_{\mathbf{jisr}}$	transition function for processor core interrupt execution, 69
$\delta_{\mathbf{m}}$	transition function of the memory, 53
$\delta_{\mathbf{tlb}}$	transition function of the TLB, 60
$.dev$	devices component (of a MIPS-86 configuration), 51
$dev_{\mathbf{adr}}(i,x)$	function that computes the device port address of device i given a memory address x, 84
$dev_{\mathbf{base}}$	base memory addresses of device I/O ports, 84
$dev_{\mathbf{palen}}$	function that specifies length of port addresses of devices in bits, 74
$ea(c,I)$	effective address of load/store instruction word I executed in processor core configuration c, 66
$eev(apic)$	external event vector provided by the local APIC *apic* to the processor core, 79
.EOI	shorthand for the end-of-interrupt register (of the local APIC I/O ports), 78
$.eoipending$	EOI-pending register component (of the local APIC), 77

$fault(m,trq,w)$	predicate that expresses whether extension of walk w using memory m page-faults for translation request trq, 59
$.fault$	page fault indicator component (of a TLB walk), 56
$fun(I)$	function that decodes the function code fun from instruction word I, 62
$.gpr$	general purpose register component (of a processor core), 60
$gprdin(c,I,R)$	value written to the gpr for instruction word I in processor core configuration c given value read from memory R, 67
$gprw(I)$	predicate that describes whether executing a given instruction word I involves writing to a general purpose register, 66
$hit(trq,w)$	predicate that expresses whether walk w is a TLB hit for translation request trq, 59
.ICR	shorthand for the interrupt command register (of the local APIC I/O ports), 78
$iev(c,I,pff,pfls)$	internal event signals that occur when instruction word I is executed in processor configuration c given page-fault signals pff and $pfls$, 68
$iindex(I)$	function that decodes the instruction index $iindex$ from instruction word I, 62
$il(c,I,eev,pff,pfls)$	function that computes the interrupt level of an interrupt occuring during execution of instruction word I in processor configuration c given external event signals eev and page-fault signals pff and $pfls$, 69
$ill(I)$	predicate that expresses whether instruction word I is an illegal instruction, 63
$imm(I)$	function that decodes the immediate constant imm from instruction word I, 62
$initial_{\mathbf{state(i)}}$	predicate that specifies a set of acceptable initial states for device i, 76
$.initrr$	INIT-request register component (of the local APIC), 77
$.internal$	internal state component (of a device), 75
$.ioapic$	I/O APIC component (of a MIPS-86 configuration), 51
$ioapic_{\mathbf{adr}}(x)$	function that computes the I/O APIC port address of a given memory address x, 84
$ioapic_{\mathbf{base}}$	base memory address of I/O APIC ports, 84
.IOREGSEL	shorthand for the select register (of the I/O APIC ports), 81
.IOWIN	shorthand for the data register (of the I/O APIC ports), 81
$.irq$	interrupt request signal component (of a device), 74

.**IRR**	shorthand for the interrupt request register (of the local APIC I/O ports), 78
.**ISR**	shorthand for the in-service register (of the local APIC I/O ports), 78
$itype(I)$	predicate that expresses whether instruction word I represents an I-type instruction, 62
$jbtaken(c,I)$	predicate that expresses whether execution of instruction word I in processor core configuration c results in a jump or branch being performed, 64
$jisr(c,I,eev,pff,pfls)$	predicate that expresses whether execution of instruction word I in processor configuration c given external event signals *eev* and page-fault signals *pff* and *pfls* triggers an interrupt, 69
$jtype(I)$	predicate that expresses whether instruction word I represents a J-type instruction, 62
$jump(I)$	predicate that expresses whether instruction word I is a jump instruction, 64
K	MIPS-86 configuration, 51
$K_{\mathbf{apic}}$	local APIC configuration, 77
$K_{\mathbf{core}}$	processor core configuration, 60
$K_{\mathbf{dev}}$	device configuration, 51
$K_{\mathbf{dev(i)}}$	configuration of device i, 74
$K_{\mathbf{ioapic}}$	I/O APIC configuration, 80
$K_{\mathbf{m}}$	memory configuration, 52
$K_{\mathbf{p}}$	processor configuration, 51
$K_{\mathbf{sb}}$	store buffer configuration, 72
$K_{\mathbf{sbe}}$	store buffer entry configuration, 72
$K_{\mathbf{tlb}}$	TLB configuration, 56
$K_{\mathbf{walk}}$	TLB walk, 56
λ	MIPS-86 output function, 83
$\lambda_{\mathbf{dev(i)}}$	output function of device i, 75
$.level$	level component (of a TLB walk), 56
$load(I)$	predicate that expresses whether instruction word I is a load instruction, 66
$lop(c,I)$	value of the left operand of ALU instruction I in processor core configuration c, 63
$lv(R)$	load value of the processor core given value read from memory R, 66
$.m$	shared global memory component (of a MIPS-86 configuration), 51

$m_d(a)$	reading d bytes starting from address a from memory m, 53
$mal(c,I)$	predicate that expresses whether the memory access of load/store instruction word I in processor configuration c is misaligned, 66
$maxsbhit(sb,x)$	index of the newest store buffer entry in store buffer sb for which there is a store buffer hit for address x, 72
$mca(c,I,eev,pff,pfls)$	interrupt signals raised after masking maskable interrupts when instruction word I is executed in processor configuration c given external event signals eev and page-fault signals pff and $pfls$, 69
$ms(dev,ioapic,apic,sb,m)$	merged memory view of MIPS-86 based on devices configuration dev, I/O apic configuration $ioapic$, local APIC configuration $apic$, store buffer configuration sb and memory configuration m, 85
nd	number of devices, 51
np	number of processors, 51
Ω	MIPS-86 output alphabet, 83
$\Omega_{\mathbf{dev(i)}}$	external output alphabet of device i, 75
$opc(I)$	opcode of an instruction word I, 62
$ovf(a,b,alucon,i)$	ALU overflow result for input values a and b given alu control bits $alucon$ and i, 63
$.p$	present bit (of a page table entry), 57
$.p$	processor component (of a MIPS-86 configuration), 51
$.pc$	program counter component (of a processor core), 60
$.ports$	I/O ports component (of the local APIC), 77
$.ports$	I/O ports component (of a device), 74
$.ports$	I/O ports component (of the I/O APIC), 80
$pte(m,w)$	page table entry of a given TLB walk w in memory m, 57
$ptea(w)$	page table entry address for a given TLB walk w, 57
$.px_0$	byte offset (of an address), 57
$.px_1$	first-level page index (of an address), 57
$.px_2$	second-level page index (of an address), 56
$.r$	access rights (of a page table entry), 57
$.r$	access rights component (of a translation request), 59
$.r$	accumulated rights component (of a TLB walk), 56
$rd(I)$	function that decodes register address rd from instruction word I, 62

$rdes(I)$	general purpose register address to which the result of an ALU/shift/coprocessor/memory instruction I is to be written, 67
$.redirect$	redirect table component (of the I/O APIC), 80
$rop(c,I)$	value of the right operand of ALU instruction I in processor core configuration c, 63
$rs(I)$	function that decodes register address rs from instruction word I, 62
$rt(I)$	function that decodes register address rt from instruction word I, 62
$rtype(I)$	predicate that expresses whether instruction word I represents an R-type instruction, 62
$.running$	processor running flags component (of a MIPS-86 configuration), 51
$sa(I)$	function that decodes the shift amount sa from instruction word I, 62
$.sb$	store buffer component (of a processor configuration), 51
$sbehit((a,w),x)$	predicate that denotes whether store buffer entry (a,w) is a store buffer hit for address x, 72
$sbhit(sb,x)$	predicate that denotes whether store buffer sb contains a store buffer hit for address x, 72
$sbv(sb,x)$	partial function that returns the value forwarded from store buffer sb for byte-address x, if it exists, 73
$sdist(c,I)$	value of the shift distance for shift instruction I in processor core configuration c, 65
$set\text{-}ad(pte,w)$	function that sets the accessed and dirty bits of a page-table entry pte depending on the access rights of TLB walk w, 59
$sf(I)$	shift function of a shift instruction word I, 65
Σ	MIPS-86 input alphabet, 83
$\Sigma_{\mathbf{apic}}$	input alphabet of the local APIC, 79
$\Sigma_{\mathbf{core}}$	input alphabet of the processor core, 61
$\Sigma_{\mathbf{dev(i)}}$	input alphabet of device i, 75
$\Sigma_{\mathbf{eev}}$	external event signals for the processor core, 61
$\Sigma_{\mathbf{ext}}$	external input alphabet for all devices, 83
$\Sigma_{\mathbf{ext(i)}}$	external input alphabet of device i, 75
$\Sigma_{\mathbf{instr}}$	input alphabet for processor core instruction execution, 61
$\Sigma_{\mathbf{ioapic}}$	input alphabet of the I/O APIC, 81
$\Sigma_{\mathbf{ioapic+dev}}$	input alphabet for I/O APIC and devices, 83
$\Sigma_{\mathbf{m}}$	input alphabet of the memory, 53
$\Sigma_{\mathbf{p}}$	input alphabet for the processor, 83
$\Sigma_{\mathbf{tlb}}$	input alphabet of the TLB, 60
$.sipirr$	SIPI-request register component (of the local APIC), 77

$.sipivect$	SIPI-vector register component (of the local APIC), 77
$sll(a,i)$	function that computes the shift left logical value for bit-string a shifted by natural number i, 65
$slop(c,I)$	value of the left shift operand for shift instruction I in processor core configuration c, 65
$.spr$	special purpose register component (of a processor core), 60
$sra(a,i)$	function that computes the shift right arithmetic value for bit-string a shifted by natural number i, 65
$srl(a,i)$	function that computes the shift right logical value for bit-string a shifted by natural number i, 65
$store(I)$	predicate that expresses whether instruction word I is a store instruction, 66
$su(I)$	predicate that expresses whether instruction word I represents a shift instruction, 65
$sures(a,i,sf)$	shift unit result for input values a and i given shift function bits sf, 65
$sv(c,I)$	store value for store instruction word I in processor core configuration c, 66
$.tlb$	TLB component (of a processor configuration), 52
$.va$	virtual address component (of a translation request), 59
$.va$	virtual page address component (of a TLB walk), 56
$wext(w,pte,r)$	function that extends TLB walk w using page table entry pte and access rights r, 58
$winit(va,pto,asid)$	intial TLB walk for given virtual address va, page table origin pto and address space identifier $asid$, 57
$r \leq r'$	access rights r' are sufficient to perform an access with access rights r, 58

C-IL Symbol Table

$.alloc_{gvar}$	base addresses of global variables (in environment parameters), 101
$\mathbf{array}(t,n) \in \mathbb{T}$	array type of length n with elements of type t (from the set $\mathbb{T}$), 103
$bytes2val_{\theta}(B,t)$	*C-IL* value of type t represented by byte-string B under environment parameters θ, 118
$c \in \mathbb{E}$	constant (in set of *C-IL* expressions), 113
$\mathbf{call}\ e(E) \in \mathbb{S}$	procedure call to function specified by expression e with parameters E (in set of *C-IL* statements), 115
$.cast$	type cast function (in environment parameters), 99
$conf_{CC\text{-}IL}$	concurrent *C-IL*-configuration, 129
$conf_{C\text{-}IL}$	sequential *C-IL*-configuration, 117
$decl(\mathcal{V})$	set of variable names declared by variable declaration list $\mathcal{V}$, 121
$\delta^{\pi,\theta}_{C\text{-}IL}$	*C-IL* semantics transition function, 127
$\delta^{\pi,\theta}_{CC\text{-}IL}$	concurrent *C-IL* semantics transition function, 129
$drop_{frame}(c)$	function that removes the topmost stack frame of a *C-IL* configuration c to l, 127
$\mathbb{E}$	set of expressions, 113
$.endianness$	endianness of the underlying architecture (in environment parameters), 99
$\mathbb{F}$	set of field names, 98
$.f$	function name component (in stack frame), 117
$.\mathcal{F}$	function table (in program), 115
$.\mathcal{F}_{adr}$	function pointer values (in environment parameters), 101
$f \in \mathbb{E}$	function name (in set of *C-IL* expressions), 113
$\mathbb{F}_{name}$	set of function names, 98
$\mathcal{F}^{\theta}_{\pi}$	combined function table for program π and environment parameters θ, 121
$f_{top}(c)$	function name of the topmost stack frame of configuration c, 126
$\mathbf{funptr}(t,T) \in \mathbb{T}$	function pointer type with return value type t and parameter type sequence T (from the set $\mathbb{T}$), 103
$FunT$	function table entry configuration, 116

goto $l \in \mathbb{S}$	jump to program location l (in set of *C-IL* statements), 115
ifnot e **goto** $l \in \mathbb{S}$	conditional jump to program location l based on expression e (in set of *C-IL* statements), 115
$inc_{loc}(c)$	function that increases the location counter of the topmost stack frame of a *C-IL* configuration c, 126
.intrinsics	function table for compiler intrinsic functions (in environment parameters), 100
$isarray(t)$	predicate that checks whether type t is an array type, 104
$isfunptr(t)$	predicate that checks whether type t is a function pointer type , 104
$isptr(t)$	predicate that checks whether type t is a pointer type, 104
.loc	location counter component (in stack frame), 117
$loc_{top}(c)$	location counter of the topmost stack frame of configuration c, 126
$.\mathcal{M}$	global memory component (in sequential *C-IL*-configuration), 117
$.\mathcal{M}$	global memory component (in concurrent *C-IL*-configuration), 129
$.\mathcal{M}_{\mathcal{E}}$	local memory component (in stack frame), 116
$\mathcal{M}_{\mathcal{E}top}(c)$	local memory of the topmost stack frame of configuration c, 126
.npar	number of function parameters (in function table entry), 116
$\mathbb{O}_1$	set of unary operator symbols, 109
$\mathbb{O}_2$	set of binary operator symbols, 109
.offset	byte offsets of fields in struct/union types (in environment parameters), 100
$.op_1$	unary operator semantics (in environment parameters), 100
$.op_2$	binary operator semantics (in environment parameters), 100
$.\mathcal{P}$	function body (in function table entry), 116
$\mathcal{P}_{top}(\pi,c)$	function body of the topmost stack frame of configuration c in program π, 126
$params_{C\text{-}IL}$	environment parameters configuration, 99
π	program, 115
$prog_{C\text{-}IL}$	program configuration, 115

$\mathbf{ptr}(t) \in \mathbb{T}$	pointer to type t (from the set $\mathbb{T}$), 102
$.size_{ptr}$	size of pointer types in bytes (in environment parameters), 99
$\mathbb{Q}$	set of type qualifiers, 103
$qt2t$	function that converts qualified type to unqualified type, 104
$.R_{\mathbf{extern}}$	function pointer values (in environment parameters), 101
$.rds$	return value destination component (in stack frame), 116
$rds_{top}(c)$	return destination of the topmost stack frame of configuration c, 126
$read(\theta,c,x)$	function that reads the memory value pointed to by pointer value x from *C-IL* configuration c under environment parameters θ, 120
$read_{\mathcal{E}}(\mathcal{M}_{\mathcal{E}},v,o,s)$	function that reads a byte-string of length s from local memory $\mathcal{M}_{\mathcal{E}}$ of variable v starting at offset o, 119
$read_{\mathcal{M}}(\mathcal{M},a,s)$	function that reads a byte-string of length s from global memory $\mathcal{M}$ starting at address a, 119
$.rettype$	return value type (in function table entry), 116
$\mathbf{return}\ e \in \mathbb{S}$	return the value of expression e (in set of *C-IL* statements), 115
$\mathbf{return} \in \mathbb{S}$	return without value (in set of *C-IL* statements), 115
$\mathbb{S}$	set of statements, 115
$.s$	stack component (in sequential *C-IL*-configuration), 117
$set_{loc}(c,l)$	function that sets the location counter of the topmost stack frame of a *C-IL* configuration c to l, 126
$set_{rds}(c,v)$	function that sets the return destination of the topmost stack frame of a *C-IL* configuration c to v, 127
$\sigma^{\pi}_{\theta}(x,f)$	field reference function, computes a pointer to the subvariable identified by field f in struct-type local reference or pointer x, 123
$.offset$	size of struct/union types in bytes (in environment parameters), 100
$size_{\theta}(t)$	size of values of type t in bytes, 104
$\mathbf{sizeof}(e) \in \mathbb{E}$	size of type (in set of *C-IL* expressions), 114
$\mathbf{sizeof}(t) \in \mathbb{E}$	size of type (in set of *C-IL* expressions), 114
$.size\_t$	type of the value returned by the `sizeof`-operator (in environment parameters), 99
$stmt_{next}(\pi,c)$	next statement to be executed in the function body of the topmost stack frame of configuration c in program π, 126
$\mathbf{struct}\ t_C \in \mathbb{T}$	struct type specified by composite type name t_C (from the set $\mathbb{T}$), 103

$\mathbb{T}$	set of *C-IL* types, 102
$\mathbb{T}_C$	set of composite type names, 98
$.T_F$	type table for struct field declarations (in program), 115
$t \in \mathbb{T}$	primitive type t (from the set $\mathbb{T}$), 102
$.\mathbb{T}_P$	set of primitive types offered by the compiler (in environment parameters), 99
$\mathbb{T}_{PP}$	set of possible primitive types, 102
$\mathbb{T}_Q$	set of qualified types, 103
$\tau(x)$	type of a value x, 107
$\tau_F(f,T)$	type of field f in field declaration list T, 122
$\tau^{\mathcal{F}}_{fun}(fn)$	type of function fn in function table $\mathcal{F}$, 122
$\tau_{Q_c}^{\pi,\theta}(x)$	function that returns the qualified type of an expression x in a given configuration c of a program π under environment parameters θ, 122
$\tau_V(v,\mathcal{V})$	type of variable v in variable declaration list $\mathcal{V}$, 122
$.Th$	thread component (in concurrent *C-IL*-configuration), 129
θ	environment parameters, 99
$\mathbb{V}$	set of variable names, 98
$.\mathcal{V}$	parameter and local variable declaration list (in function table entry), 116
$.\mathcal{V}_G$	global variable declaration list (in program), 115
$v \in \mathbb{E}$	variable name (in set of *C-IL* expressions), 113
$\mathcal{V}_{top}(\pi,c)$	local variable and parameter declaration list of the topmost stack frame of configuration c under program π, 122
val	set of values, 105
$val_{\textbf{fptr}}$	set of explicit function pointer values, 107
$val_{\textbf{fun}}$	set of symbolic function pointer values, 107
$val_{\textbf{funptr}}$	set of function pointer values, 107
$val_{\textbf{lref}}$	set of local references, 106
$val_{\textbf{prim}}$	set of primitive values, 105
$val_{\textbf{ptr}}$	set of global memory pointer values, 106
$val_{\textbf{ptr+array}}$	set of pointer and array values, 106
$val2bytes_\theta(v)$	byte-string representation of *C-IL* value v under environment parameters θ, 118
$write(\theta,c,x,y)$	function that writes the value y to the memory of *C-IL* configuration c at the location specified by pointer value x under environment parameters θ, 121
$write_{\mathcal{E}}(\mathcal{M}_{\mathcal{E}},v,o,B)$	function that writes byte-string B to local memory $\mathcal{M}_{\mathcal{E}}$ of variable v starting at offset o, 119

$write_{\mathcal{M}}(\mathcal{M},a,B)$	function that writes byte-string B to global memory $\mathcal{M}$ starting at address a, 119
$zero(\theta,x)$	predicate that expresses whether a given value x is a zero-value under environment parameters θ, 107
$[\![x]\!]_c^{\pi,\theta}$	function that evaluates an expression x in a given configuration c of a program π under environment parameters θ, returns a value, 124
$\ominus e \in \mathbb{E}$	unary operator application on expression e (in set of *C-IL* expressions), 113
$(e_0 \oplus e_1) \in \mathbb{E}$	binary operator application on expressions e_0 and e_1 (in set of *C-IL* expressions), 114
$(e\ ?\ e_0 : e_1) \in \mathbb{E}$	ternary operator on expressions e, e_0 and e_1 (in set of *C-IL* expressions), 114
$(t)e \in \mathbb{E}$	type cast (in set of *C-IL* expressions), 114
$*(e) \in \mathbb{E}$	dereferencing pointer operation (in set of *C-IL* expressions), 114
$\&(e) \in \mathbb{E}$	address of operation (in set of *C-IL* expressions), 114
$(e).f \in \mathbb{E}$	field access (in set of *C-IL* expressions), 114
$(e_0{=}e_1) \in \mathbb{S}$	assignment of value of expression e_1 to the memory location specified by the address of expression e_0 (in set of *C-IL* statements), 115
$(e_0{=}\textbf{call}\ e(E)) \in \mathbb{S}$	function call to function specified by expression e with parameters E that assigns the return value of the function to the memory location specified by the address of expression e_0 (in set of *C-IL* statements), 115

C-IL+$\mathcal{G}$ Symbol Table

$adjust_{\mathbf{gotos}}(\mathcal{P},loc)$	function that adjust the location counters of all implementation goto statements in function body $\mathcal{P}$, 164
$\mathbf{array}(t,n) \in \mathbb{T}_{\mathcal{G}}$	array ghost type with n elements of ghost type t (from the set of ghost types), 137
$c \in \mathbb{E}_{\mathcal{G}}$	constant (from the set of ghost expressions), 144
$\mathbf{call}\ e(E,E') \in \mathbb{S}'$	annotated procedure call to function specified by expression e with implementation parameters E and ghost parameters E' (from the set of annotated *C-IL* statements), 145
$cast_{\mathcal{G}}(v,t)$	function that performs type cast for implementation or ghost value v to implementation or ghost type t, 143
$C\text{-}IL(\pi)$	function that projects *C-IL+$\mathcal{G}$* program π to a *C-IL* program, 162
$C\text{-}IL(fte)$	function that projects annotated *C-IL+$\mathcal{G}$* function table entry fte to a *C-IL* function table entry, 163
$C\text{-}IL(s)$	function that projects annotated *C-IL+$\mathcal{G}$* statement s to a *C-IL* statement, 163
$C\text{-}IL(\mathcal{P})$	function that projects annotated *C-IL+$\mathcal{G}$* function body $\mathcal{P}$ to a *C-IL* function body, 163
$C\text{-}IL(sf)$	function that projects *C-IL+$\mathcal{G}$* stack frame sf to a *C-IL* stack frame, 164
$C\text{-}IL(\pi_{\mathcal{G}},c)$	function that projects *C-IL+$\mathcal{G}$* configuration c to a *C-IL* configuration given *C-IL+$\mathcal{G}$* program $\pi_{\mathcal{G}}$, 164
$conf_{CC\text{-}IL+\mathcal{G}}$	concurrent *C-IL+$\mathcal{G}$* configuration, 152
$conf_{C\text{-}IL+\mathcal{G}}$	sequential *C-IL+$\mathcal{G}$* configuration, 150
$count_{stmt}(\mathcal{P},l)$	function that counts the number of implementation statements in function body $\mathcal{P}$ up to (but not including) location l, 164
$\mathbf{current\_state} \in \mathbb{E}_{\mathcal{G}}$	state-snapshot of the current state (from the set of ghost expressions), 145
$decl_{\mathcal{G}}(\mathcal{V})$	declared variable names of a ghost variable declaration list $\mathcal{V}$, 148
$decl_{\mathcal{G}}(F)$	declared field names of a ghost field declaration list F, 149
$\delta^{\pi,\theta}_{CC\text{-}IL+\mathcal{G}}$	concurrent *C-IL+$\mathcal{G}$* transition function, 161
$\delta^{\pi,\theta}_{C\text{-}IL+\mathcal{G}}$	*C-IL+$\mathcal{G}$* transition function, 158
$drop_{frame}(c)$	function that removes the top-most stack frame of configuration c, 158

$drop_{\mathbf{ghost}}(\mathcal{P})$	function that removes all ghost statements from *C-IL+$\mathcal{G}$* function body $\mathcal{P}$, 163
$\mathbb{E}_{\mathcal{G}}$	set of ghost expressions, 144
$\mathcal{F}_{C\text{-}IL+\mathcal{G}}$	combined *C-IL+$\mathcal{G}$* function table, 148
$.\mathcal{F}$	function table of annotated implementation functions (in *C-IL+$\mathcal{G}$* program), 148
$.f$	function name (in *C-IL+$\mathcal{G}$* stack frame), 150
$.\mathcal{F}_{\mathcal{G}}$	function table of ghost functions (in *C-IL+$\mathcal{G}$* program), 148
$f \in \mathbb{E}_{\mathcal{G}}$	function name (from the set of ghost expressions), 144
$\mathcal{F}_{\pi}^{\theta}$	implementation function table, 148
$frame_{C\text{-}IL+\mathcal{G}}$	*C-IL+$\mathcal{G}$* stack frame configuration, 149
$FunT_{\mathcal{G}}$	set of ghost function table entries, 147
$FunT'$	set of annotated *C-IL* function table entries, 146
$\gamma_{\mathcal{G}_{\theta}}^{\pi,c}(x,z)$	array access reference function that computes a pointer value to the subvariable of array value x specified by array index z given *C-IL+$\mathcal{G}$* configuration c and *C-IL+$\mathcal{G}$* program π, 155
ghost(e=**allocghost**(t)) $\in \mathbb{S}_{\mathcal{G}}$	allocate a new ghost variable of implementation or ghost type t on the ghost heap, returning a pointer to the memory location specified by ghost expression e (from the set of ghost statements), 146
ghost(**call** $e(E)$) $\in \mathbb{S}_{\mathcal{G}}$	ghost procedure call to function specified by ghost expression e with ghost parameters E (from the set of ghostl statements), 146
ghost(**goto** l) $\in \mathbb{S}_{\mathcal{G}}$	ghost code jump to program location l (from the set of ghost statements), 146
ghost(**ifnot** e **goto** l) $\in \mathbb{S}_{\mathcal{G}}$	conditional ghost code jump based on the value of ghost expression e to program location l (from the set of ghost statements), 146
ghost(**return** e) $\in \mathbb{S}_{\mathcal{G}}$	return the value of ghost expression e (from the set of ghost statements), 146
ghost(**return**) $\in \mathbb{S}_{\mathcal{G}}$	return without value (from the set of ghost statements), 146
ghost(e_0=e_1) $\in \mathbb{S}_{\mathcal{G}}$	ghost assigment of value specified by expression e_1 to memory address specified by expression e_0 (from the set of ghost statements), 146
ghost(e_0=**call** $e(E)$) $\in \mathbb{S}_{\mathcal{G}}$	ghost function call to function specified by ghost expression e with ghost parameters E that writes the return value of the function to the memory location specified by ghost expression e_0 (from the set of ghost statements), 146

goto $l \in \mathbb{S}'$	jump to program location l (from the set of annotated *C-IL* statements), 145
ifnot e **goto** $l \in \mathbb{S}'$	conditional jump based on the value of expression e to program location l (from the set of annotated *C-IL* statements), 145
in_state$(e,e') \in \mathbb{E}_{\mathcal{G}}$	evaluating ghost expression e' in the state specified by state-snapshot given by expression e (from the set of ghost expressions), 145
$inc_{loc}(c)$	function that increases the location counter of the top-most stack frame of configuration c, 158
$\mathbf{Inv}_1$	invariant 1 (return destination of ghost functions are ghost left values, 166
$\mathbf{Inv}_2$	invariant 2 (ghost code can always execute), 167
$\mathbf{Inv}_3$	invariant 3 (return value destinations of implementation stack frames are all implementation values), 167
$\mathbf{Inv}_4$	invariant 4 (state of the ghost stack), 167
$isarray(t)$	predicate that checks whether implementation or ghost type t is an array type, 138
$isfunptr(t)$	predicate that checks whether implementation or ghost type t is a function pointer type, 138
$isptr(t)$	predicate that checks whether implementation or ghost type t is a pointer type, 138
lambda$(t\ v; e) \in \mathbb{E}_{\mathcal{G}}$	lambda expression with parameter v of type t specified by ghost expression e (from the set of ghost expressions), 145
$.loc$	location counter (in *C-IL*+$\mathcal{G}$ stack frame), 150
$.\mathcal{M}_{\mathcal{E}}$	local implementation memory (in *C-IL*+$\mathcal{G}$ stack frame), 149
$.\mathcal{M}_{\mathcal{E}\mathcal{G}}$	local ghost memory (in *C-IL*+$\mathcal{G}$ stack frame), 150
$\mathcal{M}_{\mathcal{E}\mathcal{G}}$	local ghost memory, 136
$\mathcal{M}_{\mathcal{G}}$	global ghost memory, 135
$.\mathcal{M}$	global implementation memory (in concurrent *C-IL*+$\mathcal{G}$ configuration), 152
$.\mathcal{M}$	global implementation memory (in sequential *C-IL*+$\mathcal{G}$ configuration), 150
map$(t,t') \in \mathbb{T}_{\mathcal{G}}$	map ghost type from type t to type t' (from the set of ghost types), 137
math_int $\in \mathbb{T}_{\mathcal{G}}$	mathematical integer ghost type (from the set of ghost types), 137
$merge\text{-}impl\text{-}ghost(c,c')$	function that merges the implementation memories of *C-IL* configuration c' to *C-IL*+$\mathcal{G}$ configuration c, 153

$\tau_{V\mathcal{G}}(v,\mathcal{V})$	declared type for variable name v in ghost variable declaration list $\mathcal{V}$, 149
$.Th$	thread component (in concurrent *C-IL+*$\mathcal{G}$ configuration), 152
$.\mathcal{V}_G$	implementation global variable declarations (in *C-IL+*$\mathcal{G}$ program), 147
$.\mathcal{V}_{G\mathcal{G}}$	global ghost variable declarations (in *C-IL+*$\mathcal{G}$ program), 147
$.\mathcal{V}$	implementation parameter and local variable declarations (in annotated *C-IL* function table entry), 147
$.\mathcal{V}_\mathcal{G}$	ghost parameter and local variable declarations (in annotated *C-IL* function table entry), 147
$.\mathcal{V}_\mathcal{G}$	ghost parameter and local variable declarations (in ghost function table entry), 147
$\mathcal{V}_{\mathcal{G}top}(\pi,c)$	function that returns the set of ghost local variable names declared in the top-most stack frame of *C-IL+*$\mathcal{G}$ configuration c given *C-IL+*$\mathcal{G}$ program π, 154
$v \in \mathbb{E}_\mathcal{G}$	variable name (from the set of ghost expressions), 144
$\mathcal{V}_{top}(\pi,c)$	function that returns the set of implementation local variable names declared in the top-most stack frame of *C-IL+*$\mathcal{G}$ configuration c given *C-IL+*$\mathcal{G}$ program π, 154
$val_{\mathbf{gfun}}$	set of symbolic ghost function values, 141
$val_\mathcal{G}$	set of ghost values, 138
$val_\mathcal{G}(t)$	set of values of implementation or ghost type t, 139
$val_{\mathbf{ghost\text{-}ptr+array}}$	set of ghost pointer and array values, 140
$val_{\mathbf{gref}}$	set of global ghost reference values, 140
$val_{\mathbf{lref}_\mathcal{G}}$	set of local ghost reference values, 141
$val_{\mathcal{M}_\mathcal{G}}$	set of structured ghost values, 133
$val_{\mathbf{map}}$	set of map values, 139
$val_{\mathbf{math\_int}}$	set of mathematical integer values, 139
$val_{\mathbf{obj}}$	set of generic pointer values, 139
$val_{\mathbf{record}}$	set of record values, 140
$val_{\mathbf{state\_t}}$	set of state-snapshot values, 139
$write_\mathcal{G}(\theta,\pi,c,x,y)$	function that writes implementation or ghost value y to *C-IL+*$\mathcal{G}$ configuration c to the memory location specified by *C-IL+*$\mathcal{G}$ pointer value x under environment parameters θ and *C-IL+*$\mathcal{G}$ program π, 153
$write_{val_{\mathcal{M}_\mathcal{G}}}(v,S,y)$	function that writes a structured ghost value y to a structured ghost value v as subvalue specified by subvariable selector string S, 135

$zero_{\mathcal{G}}(\theta,v)$	predicate that checks whether given implementation or ghost value x is a zero-value given *C-IL* environment parameters θ, 142
${}^{\mathcal{G}}[\![x]\!]^{\pi,\theta}_{c,p_\lambda}$	function that evaluates ghost expression x in a given *C-IL+$\mathcal{G}$* configuration c and *C-IL+$\mathcal{G}$* program π under environment parameters θ and lambda-parameter value function p_λ, 156
$\ominus e \in \mathbb{E}_{\mathcal{G}}$	unary operator applied to ghost expression e (from the set of ghost expressions), 144
$(e_0 \oplus e_1) \in \mathbb{E}_{\mathcal{G}}$	binary operator applied to ghost expressions e_0 and e_1 (from the set of ghost expressions), 144
$(e \mathbin{?} e_0 : e_1) \in \mathbb{E}_{\mathcal{G}}$	ternary operator applied to ghost expressions e, e_0 ane e_1 (from the set of ghost expressions), 144
$(t)e \in \mathbb{E}_{\mathcal{G}}$	type cast of ghost expression e to implementation or ghost type t (from the set of ghost expressions), 144
$*(e) \in \mathbb{E}_{\mathcal{G}}$	pointer dereferencing operation applied to ghost expression e (from the set of ghost expressions), 144
$\&(e) \in \mathbb{E}_{\mathcal{G}}$	address-of operation applied to ghost expression e (from the set of ghost expressions), 144
$(e).f \in \mathbb{E}_{\mathcal{G}}$	field access of field f in ghost expression e (from the set of ghost expressions), 144
$e[e'] \in \mathbb{E}_{\mathcal{G}}$	map or ghost array access of index or parameter specified by ghost expression e' in ghost expression e (from the set of ghost expressions), 145
$e\{f := e'\} \in \mathbb{E}_{\mathcal{G}}$	record update of field f with value specified by ghost expression e' to record specified by ghost expression e (from the set of ghost expressions), 145
$(e_0 = e_1) \in \mathbb{S}'$	implementation assigment of value specified by expression e_1 to memory address specified by expression e_0 (from the set of annotated *C-IL* statements), 145
$(e_0 = \textbf{call}\ e(E,E')) \in \mathbb{S}'$	annotated function call to function specified by expression e with implementation parameters E and ghost parameters E' that writes the return value of the function to the memory location specified by expression e_0 (from the set of annotated *C-IL* statements), 145

Bibliography

[ACHP10] E. Alkassar, E. Cohen, M. Hillebrand, and H. Pentchev. Modular specification and verification of interprocess communication. In *Formal Methods in Computer Aided Design (FMCAD) 2010*. IEEE, 2010.

[ACKP12] E. Alkassar, E. Cohen, M. Kovalev, and W. Paul. Verification of tlb virtualization implemented in c. In *4th International Conference on Verified Software: Theories, Tools, and Experiments, VSTTE'12*, Lecture Notes in Computer Science, Philadelphia, USA, 2012. Springer-Verlag.

[Age01] International Atomic Energy Agency. *Investigation of an accidental exposure of radiotherapy patients in Panama : report of a team of experts 26 May-1 June 2001*. IAEA, Vienna :, 2001.

[AHL$^+$09] E. Alkassar, M. A. Hillebrand, D. C. Leinenbach, N. W. Schirmer, A. Starostin, and A. Tsyban. Balancing the load: Leveraging semantics stack for systems verification. 42, Numbers 2-4:389–454, 2009.

[AL88] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1988.

[Alk09] Eyad Alkassar. *OS Verification Extended - On the Formal Verification of Device Drivers and the Correctness of Client/Server Software*. PhD thesis, University of Saarland, 2009.

[App11] Andrew W. Appel. Verified software toolchain. In *Proceedings of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software*, ESOP'11/ETAPS'11, pages 1–17, Berlin, Heidelberg, 2011. Springer-Verlag.

[ASS08] Eyad Alkassar, Norbert Schirmer, and Artem Starostin. Formal pervasive verification of a paging mechanism. In C. R. Ramakrishnan and Jakob Rehof, editors, *14th intl Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS08)*, volume 4963 of *LNCS*, pages 109–123. Springer, 2008.

[Bau08] Christoph Baumann. Formal specification of the x87 floating-point instruction set. Master's thesis, Saarland University, 2008.

[Bau12] Christoph Baumann. Reordering and simulation in concurrent systems. Technical report, Saarland University, Saarbrücken, 2012.

[BJ01] C. Berg and C. Jacobi. Formal verification of the VAMP floating point unit. In *Proc. 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *LNCS*, pages 325–339. Springer, 2001.

[BJK+03] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W.J. Paul. Instantiating uninterpreted functional units and memory system: functional verification of the vamp. In D. Geist and E. Tronci, editors, *CHARME 2003*, volume 2860 of *LNCS*, pages 51–65. Springer, 2003.

[BL09] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the clight subset of the c language. *Journal of Automated Reasoning*, 43:263–288, 2009.

[BLR11] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with slam. *Commun. ACM*, 54(7):68–76, July 2011.

[Boc] Bochs ia-32 emulator project. URL: `http://bochs.sourceforge.net`.

[CAB+09] E. Cohen, A. Alkassar, V. Boyarinov, M. Dahlweid, U. Degenbaev, M. Hillebrand, B. Langenstein, D. Leinenbach, M. Moskal, S. Obua, W. Paul, H. Pentchev, E. Petrova, T. Santen, N. Schirmer, S. Schmaltz, W. Schulte, A. Shadrin, S. Tobies, A. Tsyban, and S. Tverdyshev. Invariants, modularity, and rights. In Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics (PSI 2009)*, volume 5947 of *Lecture Notes in Computer Science*, pages 43–55. Springer, 2009.

[CDH+09] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Markus Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem proving in Higher-Order Logics (TPHOLs 2009)*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42, Munich, Germany, 2009. Springer.

[CMST10] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In *Proceedings of the 22nd international conference on Computer Aided Verification*, CAV'10, pages 480–494, Berlin, Heidelberg, 2010. Springer-Verlag.

[CMTS09] Ernie Cohen, Michał Moskal, Stephan Tobies, and Wolfram Schulte. A precise yet efficient memory model for C. *Electronic Notes in Theoretical Computer Science*, 254:85–103, 2009.

[CPS13] E. Cohen, W. Paul, and S. Schmaltz. Theory of multi core hypervisor verification. In *Proceedings of the 39th Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '13, Berlin, Heidelberg, 2013. Springer-Verlag.

[CS10] Ernie Cohen and Bert Schirmer. From total store order to sequential consistency: A practical reduction theorem. In Matt Kaufmann and Lawrence Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 403–418. Springer Berlin / Heidelberg, 2010.

[CYGC10] Xiaofang Chen, Yu Yang, Ganesh Gopalakrishnan, and Ching-Tsun Chou. Efficient methods for formally verifying safety properties of hierarchical cache coherence protocols. *Formal Methods in System Design*, 36:37–64, 2010. 10.1007/s10703-010-0092-y.

[DDB08] Matthias Daum, Jan Dörrenbächer, and Sebastian Bogan. Model stack for the pervasive verification of a microkernel-based operating system. In Bernhard Beckert and Gerwin Klein, editors, *5th International Verification Workshop (VERIFY'08)*, volume 372 of *CEUR Workshop Proceedings*, pages 56–70. CEUR-WS.org, 2008.

[Deg11] Ulan Degenbaev. *Formal Specification of the x86 Instruction Set Architecture*. PhD thesis, Saarland University, Saarbrücken, 2011.

[DMS$^+$09] Markus Dahlweid, Michał Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE Companion 2009: 31st International Conference on Software Engineering*, pages 429–430. IEEE, May 2009.

[Dör10] Jan Dörrenbächer. *Formal Specification and Verification of a Microkernel*. PhD thesis, Saarland University, Saarbrücken, 2010.

[DPS09] Ulan Degenbaev, Wolfgang J. Paul, and Norbert Schirmer. Pervasive theory of memory. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, *Efficient Algorithms – Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, volume 5760 of *Lecture Notes in Computer Science*, pages 74–98. Springer, 2009.

[DSS09] Matthias Daum, Norbert W. Schirmer, and Mareike Schmidt. Implementation correctness of a real-time operating system. In *7th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2009), 23–27 November 2009, Hanoi, Vietnam*, pages 23–32. IEEE, 2009.

[ER08] Eric Eide and John Regehr. Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th ACM international conference on Embedded software*, EMSOFT '08, pages 255–264, New York, NY, USA, 2008. ACM.

[Fre05] Freescale semiconductor. *Programming Environments Manual for 32-Bit Implementations of the PowerPC™Architecture*, September 2005.

[GHLP05] M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul. On the correctness of operating system kernels. In J. Hurd and T. Melham, editors, *Theorem Proving in High Order Logics (TPHOLs) 2005*, LNCS, Oxford, U.K., 2005. Springer.

[HIdRP05] M. Hillebrand, T. In der Rieden, and W.J. Paul. Dealing with I/O Devices in the Context of Pervasive System Verification. In *23nd IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD 2005), 2-5 October 2005, San Jose, CA, USA, Proceedings*, pages 309–316. IEEE, 2005.

[HL09] Mark A. Hillebrand and Dirk C. Leinenbach. Formal Verification of a Reader-Writer Lock Implementation in C. *Electron. Notes Theor. Comput. Sci.*, 254:123–141, October 2009.

[Hot12] Jenny Hotzkow. Store Buffers as an Alternative to Model Self Modifying Code. Bachelor's Thesis, Saarland University, Saarbrücken, 2012.

[HP08] Martin Hofmann and Mariela Pavlova. Elimination of ghost variables in program logics. In *In Trustworthy Global Computing: Revised Selected Papers from the Third Symposium TGC 2007, number 4912 in LNCS*, pages 1–20. Springer, 2008.

[HRP05] Mark A. Hillebrand, Thomas In der Rieden, and Wolfgang J. Paul. Dealing with I/O Devices in the Context of Pervasive System Verification. In *Proceedings of the 2005 International Conference on Computer Design*, ICCD '05, pages 309–316, Washington, DC, USA, 2005. IEEE Computer Society.

[HS65] Juris Hartmanis and Richard E. Stearns. On the Computational Complexity of Algorithms. *Transactions of the American Mathematical Society*, 117:285–306, May 1965.

[HT09] M. Hillebrand and S. Tverdyshev. Formal verification of gate-level computer systems. In A. Rybalchenko A. Morozov, K. Wagner and A. Frid., editors, *4th International Computer Science Symposium in Russia*, volume 5675 of *LNCS*, pages 322–333. Springer, 2009.

[Int10] Intel, Santa Clara, CA, USA. *Intel®64 and IA-32 Architectures Software Developer's Manual: Volumes 1-3b*, June 2010.

[ISO99] ISO 9899:1999 Programming Languages - C, 1999.

[KAE$^+$10] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an Operating System Kernel. *Communications of the ACM*, 53(6):107–115, Jun 2010.

[Kov12] Mikhail Kovalev. *TLB Virtualization in the Context of Hypervisor Verification*. PhD thesis, Saarland University, Saarbrücken, 2012.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[LABS12] Xavier Leroy, W. Appel, Andrew, Sandrine Blazy, and Gordon Stewart. The CompCert Memory Model, Version 2. Rapport de recherche RR-7987, INRIA, June 2012.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[Lan97] G. Le Lann. An analysis of the Ariane 5 flight 501 failure - a system engineering perspective. *Proceedings of the International Conference and Workshop on Engineering of Computer-Based Systems*, pages 339–346, 1997.

[Lei08] Dirk Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, Saarbrücken, 2008.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[Lev93] Nancy G. Leveson. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26:18–41, 1993.

[LS09] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *16th International Symposium on Formal Methods (FM 2009)*, volume 5850 of *Lecture Notes in Computer Science*, pages 806–809, Eindhoven, the Netherlands, 2009. Springer.

[Mau11] Stefan Maus. *Verification of Hypervisor Subroutines written in Assembler*. PhD thesis, Universität Freiburg, 2011.

[Mica] Microsoft Research. The VCC Manual. URL: `http://vcc.codeplex.com`.

[Micb] Microsoft Research. The VCC webpage. URL: `http://vcc.codeplex.com`.

[MIP05] MIPS Technologies, 1225 Charleston Road, Mountain View, CA. *MIPS32 Architecture for Programmers Volume II: The MIPS32 Instruction Set*, 2.5 edition, July 2005.

[MMS08] Stefan Maus, Michał Moskal, and Wolfram Schulte. Vx86: x86 assembler simulated in C powered by automated theorem proving. In José Meseguer and Grigore Roşu, editors, *Algebraic Methodology and Software Technology (AMAST 2008)*, volume 5140 of *Lecture Notes in Computer Science*, pages 284–298, Urbana, IL, USA, July 2008. Springer.

[Moo89] J Strother Moore. System verification. *Journal of Automated Reasoning*, 5:409–410, 1989. 10.1007/BF00243130.

[MP00] S.M. Müller and W.J. Paul. *Computer Architecture, Complexity and Correctness*. Springer, 2000.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, 2002. Springer-Verlag.

[nxb10] AMD64 Architecture Programmer's Manual: Volumes 1-3, 2010.

[Owe10] Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 478–503, Berlin, Heidelberg, 2010. Springer-Verlag.

[Pau12] Wolfgang Paul. A Pipelined Multi Core MIPS Machine - Hardware Implementation and Correctness Proof. URL: `http://www-wjp.cs.uni-saarland.de/lehre/vorlesung/rechnerarchitektur2/ws1112/layouts/multicorebook.pdf`, 2012.

[Pra95] Vaughan Pratt. Anatomy of the pentium bug. In *TAPSOFT '95: Theory and Practice of Software Development, volume 915 of Lecture Notes in Computer Science*, pages 97–107. Springer-Verlag, 1995.

[PSS12] Wolfgang J. Paul, Sabine Schmaltz, and Andrey Shadrin. Completing the automated verification of a small hypervisor - assembler code verification. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *SEFM*, volume 7504 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2012.

[QEM] Qemu processor emulator project. URL: `http://qemu.org`.

[Rit93] Dennis M. Ritchie. The development of the C language. In *The second ACM SIGPLAN conference on History of programming languages*, HOPL-II, pages 201–208, New York, NY, USA, 1993. ACM.

[Sch06] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[Sha12] Andrey Shadrin. *Mixed low- and high level programming language semantics and automated verification of a small hypervisor*. PhD thesis, Saarländische Universitäts- und Landesbibliothek, Postfach 151141, 66041 Saarbrücken, 2012.

[SS86] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. In *Proceedings of the 13th annual international symposium on Computer architecture*, ISCA '86, pages 414–423, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[SS12] S. Schmaltz and A. Shadrin. Integrated semantics of intermediate-language c and macro-assembler for pervasive formal verification of operating systems and hypervisors from verisoftxt. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *4th International Conference on Verified Software: Theories, Tools, and Experiments, VSTTE'12*, volume 7152 of *Lecture Notes in Computer Science*, Philadelphia, USA, 2012. Springer Berlin / Heidelberg.

[SSO$^+$10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010.

[Tve09] Sergey Tverdyshev. *Formal Verification of Gate-Level Computer Systems*. PhD thesis, Saarland University, Computer Science Department, 2009.

[Ver07] Verisoft Consortium. The Verisoft Project. URL: `http://www.verisoft.de/`, 2003-2007.

[Ver10] Verisoft Consortium. The Verisoft-XT Project. URL: `http://www.verisoftxt.de/`, 2007-2010.

[Vir] Virtualbox x86 virtualization project. URL: `http://www.virtualbox.org`.

[vW81] A. van Wijngaarden. Revised report of the algorithmic language algol 68. *ALGOL Bull.*, pages 1–119, August 1981.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41:19:1–19:36, October 2009.