# Verified Linking for Modular Kernel Verification


Dissertation

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

Thomas In der Rieden

idr@cs.uni-sb.de

Saarbrücken, September 2009

| | |
|---|---|
| Tag des Kolloqiums: | 20.11.2009 |
| Dekan: | Prof. Dr. Joachim Weickert |
| Vorsitzender des Prüfungsausschusses: | Prof. Dr. Philipp Slusallek |
| 1. Berichterstatter: | Prof. Dr. Wolfgang J. Paul |
| 2. Berichterstatter: | Prof. Dr. Michael Backes<br>(Vertreter für Prof. Dr. Manfred Broy) |
| Akademischer Mitarbeiter: | Dr. Dirk C. Leinenbach |

# Danksagung

> But the servant who did things that deserved a beating without knowing it will receive a light beating. Much will be required from everyone to whom much has been given. But even more will be demanded from the one to whom much has been entrusted.
>
> Luke 12:48

Ich möchte mich an dieser Stelle bei allen bedanken, die letztlich zum Gelingen dieser Arbeit beigetragen haben. Da ich mir meiner menschlichen Unzulänglichkeiten bewusst bin, entschuldige ich mich bei denjenigen, die ich vergessen habe, im Voraus.

Mein erstes Dankeschön geht an meine Eltern. Ein jeder, der mich kennt, kann sich eine Vorstellung davon machen, was es bedeutet, mich als Sohn zu haben—eine Vorstellung die ich sicherlich mühelos übertroffen habe. Danke für Eure nahezu grenzenlose Geduld und Euer zu oft ungerechtfertigtes Vertrauen in mich.

Normalerweise folgt hier der kanonische Dank an den betreuenden Professor und das tolle Thema, das er einem gegeben hat. Ich möchte dieser Tradition nicht folgen. Stattdessen möchte ich ein Dankeschön an einen ausgesprochen mutigen Philanthropen richten (auch wenn er das jetzt sicherlich weit von sich weist), der aus Prinzip nicht den einfachen Weg geht, eine Eigenschaft, die mich mit ihm verbindet. Danke Dir, Wolfgang, dass Du etwas gesehen hast, was den meisten anderen verborgen geblieben ist, und dass Du mir über all die Jahre ein guter (väterlicher ;)) Freund geworden bist.

Zwei Menschen in meiner Nähe verdanke ich sehr viel, ich weiß nicht, ob ich ohne sie diese Arbeit zu Ende gebracht hätte. Mark und Dirk, ich danke Euch für alle Unterstützung, die Ihr mir reichlich zu Teil habt werden lassen, technisch und moralisch. Abgesehen davon seid Ihr zwei erstklassige Kollegen und Freunde, ich bin stolz darauf, mit Euch im Projektmanagement zusammen gearbeitet haben zu dürfen.

In jungen Jahren bereits in die Konzeption und Leitung eines Projekts wie Verisoft eingebunden zu sein, ist ungewöhnlich und es bedarf ungewöhnlicher Menschen, um dies möglich zu machen. Der leider viel zu früh verstorbene Bernd Reuse war so ein Mensch. Ich hätte Ihnen Ihre Standardfrage *'Herr In der Rieden, was macht Ihre Promotion?'* gerne endlich positiv beantwortet. Vielen Dank für Ihr Vertrauen und Ihre Visionen.

Es gibt Dinge, auf deren Verlauf man nur bescheidenen Einfluss hat. Das

Gelingen eines Verbundvorhabens wie Verisoft gehört dazu. Dass es ein Erfolg geworden ist, verdanke ich vor allem den vielen Projektmitarbeitern, die hart gearbeitet haben. Euch allen, vor allem denen des Lehrstuhls Paul, ein herzliches Dankeschön.

Zum Schluss möchte ich noch ein paar besonderen Menschen danken: Sabine Nermerich, meiner ersten und dienstältesten Assistentin, dafür, dass Sie nie ein Blatt vor den Mund nimmt. Meinen ganzen Damen im Cluster Office, Ihr seid eine Wucht.

Eyad Alkassar, dessen körperlichen Widerstandsfähigkeit nicht immer ganz mit seiner geistigen und moralischen Integrität korreliert. Ich danke Dir für Deine unzähligen (liebenswürdigen) Angriffspunkte. Bitte beschäftige Dich weiterhin mit bärensicheren Mülleimern, das hat Zukunft ;).

Norbert Schirmer, auch Dir ein herzliches Dankeschön. Abgesehen davon, dass ich ohne Dich heute noch ein Beweisbarbar in Isabelle wäre, haben mich Deine Essgewohnheiten immer wieder zu interessanten Anregungen geführt und Deine Verzückung beim Bassspiel inspiriert.

Diese Arbeit wurde teilweise im Rahmen der Verisoft und Verisoft XT Vorhaben vom Bundesministerium für Bildung und Forschung (BMBF) unter dem Förderkennzeichen 01 IS C38 gefördert.

This thesis has been partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft and Verisoft XT projects under grant 01 IS C38.

## Abstract

This thesis is split into two parts. In the first one, we will present a formal specification of *Communicating Virtual Machines (CVM)*, a hardware-abstracting programming framework for microkernel implementers.

In the second part, we present the sketch of a simulation theorem between the CVM model and the hardware, represented by an integrated processor/device model. For parts of the CVM model, namely the microkernel, certain correctness properties are formalized and verified.

This work is part of the Verisoft project. Verisoft aims at the pervasive formal verification of integrated computer systems. This means that correctness properties from the top layers of such systems are transferred down to the lowest level, the hardware, where they have to be discharged. Following this procedure guarantees that we have not made too strong assumptions on the top layers.

To specify CVM, we integrate several computational models, for example for devices, assembly language, and a C-like programming language called $C0$. In the verification part, we also use fundamental results from the Verisoft project, for example a correctness theorem for a non-optimizing $C0$ compiler.

The major part of the results presented in this thesis has been formalized and verified in the interactive theorem prover Isabelle/HOL.



## Kurzzusammenfassung

Die vorliegende Arbeit befasst sich im ersten Teil mit der formalen Spezifikation von *Communicating Virtual Machines (CVM)*, einer von der Hardware abstrahierenden Programmierumgebung für Mikrokernel Implementierer.

Im zweiten Teil wird der Entwurf eines Simulationstheorems zwischen dem CVM Model und der Hardware, realisiert durch ein integriertes Prozessor-/Gerätemodell, präsentiert. Für einen Teil des CVM Modells, den Mikrokernel, werden diverse Korrektheitseigenschaften formalisiert und verifiziert.

Die vorliegende Arbeit ist im Rahmen des Verisoft Projekts entstanden. Verisoft zielt auf die durchgängige formale Verifikation von integrierten Computersystemen ab. Das bedeutet, dass Korrektheitsaussagen und -eigenschaften von den obersten Schichten solcher Systeme bis auf die unterste Schicht der Hardware heruntergebrochen und entlastet werden müssen. So ist gewährleistet, dass keine zu starken Annahmen auf den oberen Ebenen gemacht worden sind.

Zur Spezifikation von CVM werden diverse Berechnungsmodelle integriert, unter anderem für Geräte, Assembler und eine C-ähnliche Programmiersprache namens $C0$. Bei der Verifikation werden ebenfalls grundlegende Ergebnisse aus Verisoft verwendet, beispielsweise ein Korrektheitssatz für einen nicht-optimierenden $C0$ Compiler.

Der größte Teil der Resultate, die in dieser Arbeit vorgestellt werden, sind im interaktiven Theorembeweiser Isabelle/HOL formalisiert und verifiziert worden.

## Extended Abstract

Operating systems are crucial components in nearly every computer system. They provide plenty of services and functionalities, e.g., managing inter-process communication, device access, and memory management. Obviously, they play a key role in the reliability of such systems, and in fact, a considerable share of hacker attacks target operating system vulnerabilities. Thus, proving a computer system to be safe and secure requires to prove its operating system to be safe and secure.

We introduce a computational model called CVM (communicating virtual machines) that formalizes concurrent user processes interacting with a generic (abstract) microkernel and devices. To establish interaction, the abstract kernel features special functions called CVM primitives, which allow to alter process or device configurations, e.g., by copying data from one process to another. By linking a CVM implementation to an abstract kernel, we obtain a concrete kernel ('personality'). We describe how a whole framework, featuring virtual memory support, memory management, system calls, user defined interrupts, etc.—thus providing a trustworthy platform for microkernel programmers—can be proven correct.

One main result of this thesis are a complete formal definition of the computational model for CVM, including all of its primitives. Furthermore, we present a complete definition of abstract linking (to obtain the concrete kernel) and its pre-requisites. Last but not least, we elaborate on parts of the overall CVM implementation correctness theorem: we formalize the correctness relation between the abstract and the concrete kernel and present the formal proof that this relation is preserved by certain kernel steps. To a very large extent, this proof has also been formalized in the interactive theorem prover Isabelle/HOL [NPW02]. The specifications have been fully formalized.

This thesis has been written in the context of the Verisoft project [IP08, Ver03], which aims at formal and pervasive verification of entire computer systems. *Pervasive* stands for the policy that all correctness properties from the abstract upper layers are transferred down through all the intermediate layers to the actual hardware—given through an instruction set architecture with devices—, where they are to be discharged using only a very small set of well-known axioms.

Due to the nature of such an endeavor, we use results from various sources in the Verisoft project: the low-level hardware and its correctness [Tve09], assembler language semantics (partially [Tsy09], partially own work) *C*0 C-like programming language semantics and compiler correctness [Lei08], stack verification and top-level correctness [AHL+09, Alk09, Tsy09].

The results of this thesis contribute to two sub-projects within Verisoft: in the academic system, representing a vertical cross section of a real general-purpose computer system, covering all layers from the gate-level hardware description to communicating concurrent programs, a microkernel named *VAMOS* uses CVM. The second sub-project is dedicated to the development of a representative automotive system stack. Here, a real-time operating system named *OLOS* [Kna08] bases on CVM.

## Zusammenfassung

Betriebssysteme gehören zu den essentiellen Komponenten eines jeden Computersystems. Sie bieten Mechanismen und Funktionalitäten, die die Kommunikation zwischen Prozessen, den Zugriff auf Geräte, sowie die Speicherverwaltung regeln. Es ist daher offensichtlich, dass die Verlässlichkeit eines solchen Computersystems ganz erheblich von der des Betriebssystems abhängt—tatsächlich ist es so, dass ein beträchtlicher Anteil von Hackerangriffen auf Lücken im Betriebssystem abzielt. Möchte man daher beweisen, dass ein Computersystem sicher und zuverlässig ist, so muss man diese Eigenschaften auch für das Betriebssystem zeigen.

Wir führen in dieser Arbeit ein Berechnungsmodell namens CVM (communicating virtual machines) ein, wo nebenläufige Benutzerprozesse mit einem generischen (abstrakten) microkernel und Geräten interagieren. Dazu bietet der abstrakte Kernel spezielle Funktionen namens CVM Primitive an, die von den Benutzerprozessen aufgerufen werden können um Prozess- oder Gerätekonfigurationen zu verändern—beispielsweise in dem Daten von einem Prozess zu einem anderen kopiert werden. Indem wir die CVM Implementierung zu einem abstrakten Kernel linken, erhalten wir einen konkreten Kern ('personality'). Wir beschreiben in der vorliegenden Arbeit, wie ein solches Framework mit Unterstützung für virtuellen Speicher, Speicherverwaltung, etc. als korrekt bewiesen werden kann und somit als verlässliche Plattform für die Programmierer von Mikrokerneln dienen kann.

Ein Hauptergebnis dieser Dissertation ist die komplette formale Definition des CVM Berechnungsmodells, inklusive aller seiner Primitive. Desweiteren stellen wir eine Definition des abstrakten Linkens (zur Konstruktion des konkreten Kerns) vor, sowie der dazugehörigen Vorbedingungen. Dann verwenden wir diese Ergebnisse um einen Teil des Theorems zur CVM Implementierungskorrektheit vorzustellen: wir formalisieren die Korrektheitsrelation zwischen abstraktem und konkreten Kern und präsentieren den Beweis, dass diese Relation von gewöhnlichen Kernschritten erhalten wird. Der Beweis ist zum allergrößten Teil im interaktiven Theorembeweiser Isabelle/HOL [NPW02] formalisiert worden, die Spezifikationen liegen vollständig formal vor.

Diese Dissertation ist im Rahmen des Verisoft Projekts [IP08, Ver03] entstanden. Verisoft zielt auf die formale und durchgängige Verifikation ganzer Computersysteme ab. Durchgängig heißt dabei, dass all Korrektheitseigenschaften der abstrakten oberen Schichten eines Systems nach unten bis auf die eigentliche Hardware herunter gebrochen werden, wo sie dann entlastet werden unter zu Hilfenahme weniger wohlbekannter Axiome.

Wir verwenden Resultate verschiedener Quellen in Verisoft: die Hardware und ihre Korrektheit [Tve09], die Semantik der Assemblersprache (teilweise [Tsy09], teilweise eigene Arbeit), die Semantik der C-ähnlichen Programmiersprache *C*0 und Compilerkorrektheit [Lei08], sowie Ergebnisse zur Stackverifikation und ihrer Top-level Korrektheit [AHL+09, Alk09, Tsy09].

Die Resultate dieser Arbeit werden in zwei Teilprojekten in Verisoft verwendet: im akademischen System, das einen vertikalen Querschnitt eines realen Standard-Computersystems, das alle Ebene umfasst, verwendet eine Mikrokernel Implementierung namens *VAMOS* CVM. Das zweite Teilprojekt ist der Entwicklung eines representativen Automotive Systems gewidmet. Hier basiert ein Echtzeit-Betriebssystem namens *OLOS* [Kna08] auf CVM.

# Contents

# List of Figures

# List of Tables

> Programming today is a race between software engineers stirring to build bigger and better idiot-proof programs, and the universe trying to produce bigger and better idiots. So far, the universe is winning.
>
> Unknown

Chapter 1

# Introduction

**Contents**



Computer systems are faulty. We all know that: who has never experienced the annoyance of a so-called blue screen?

Most of the time, these faults are frustrating, sometimes they are infuriating. We have to redo some of our recent work (of course because we had not saved our files regularly), maybe we have to wait a couple of minutes until we can continue.

Obviously, we know how to repair a faulty system: *'We turn it off, we pull the plug, we count to ten, we plug it in, and finally we turn it on again.'*, says Sir Tony Hoare—and he is right, most of the times this works and looking for the bug, which caused the whole dilemma, would be just a waste of time.

Unfortunately, we can't use these repair instructions with all computer systems in the world. Some not, because they don't have a plug (have you ever tried to reset a locked-up mobile phone with an integrated battery?), others because even a second offline would have disastrous consequences—think of air-crafts, space ships, incubators, pacemakers and so on.

We call such a thing a critical computer system: if human life or health is at stake, we talk of *safety-critical systems*, if it is about money or data privacy, we call it *security-critical*.

There is a long, sad track of accidents caused by faulty hard- and software:

- In 1982, the CIA slipped a bug into a piece of a Canadian computer system. This computer system was known to be obtained by the Soviets to control the trans-Siberian gas pipeline. The Reagan administration was trying to stop Western Europe from importing Soviet natural gas. The resulted event has been reported to be the largest non-nuclear explosion in history [Dav04].

- The Terac-25 [Lev95] was a radiation therapy device that could deliver two kinds of radiation: either low-intensity electron beam or high-intensity

X-rays. The latter ones were generated by shooting high-power electrons into a metal target situated between the electron gun and the patient. Due to a race condition in the operating system of the Therac-25 and the replacement of the old electromechanical safeties with software control, a maloperation could lead to the X-ray been fired without the metal target in place. At least five casualties have been reported in the years 1985 to 1987, not counting the many seriously injured patients.

- Probably the first computer bug of which the world became widely aware of is the Intel Pentium floating point divide bug from 1993. The Pentium chip produced faulty results for floating point divisions in a certain range. At the time Intel became aware of the bug, already 3 to 5 million chips had been sold. Though only a very few people were actually affected by the impreciseness, the bug ultimately created costs of $475 million for Intel [Jan95].

- On June 4, 1996, Ariane 5 flight 501 crashes 40 seconds after launch. Working code of predecessor Ariane 4 has been reused in Ariane 5. The bug is in a routine, which converts a 64-bit floating-point number to a 16-bit signed integer. The problem was that the engines powering Ariane 5 are much faster, so the floating-point numbers are larger than before causing an overflow interrupt. Since the soft- and hardware of the backup system is identical to the primary one, both systems fail. In the consequence, the rocket is overpowered and starts to disintegrate as the self-destruct system of the launcher is triggered [Boa96]. The damage has been estimated with $500 million.

- In 2000, once again a software bug in radiation therapy claimed at least eight casualties while leaving another 20 with serious overdoses. The software allowed to place four shielding blocks to protect healthy tissue from radiation. But doctors in Panama wanted to use five blocks and discovered a trick to cheat the software by drawing one single large block with a hole in the middle. Unfortunately the software, which comes from a U.S. company, gives different answers depending on how the hole is drawn: in one direction, the correct dose is calculated. but in another direction the dose is twice as much as it should be. All physicians involved have been indicted for murder, since they were legally obliged to check the computer results by hand.

If we have a look around in the world, there are fortunately not too many really evil things happening due to computer errors. Industry is aware of the problems that arise from faulty systems and has come up with a bunch of approaches to minimize the risks:

- One approach is to define the development process and its guidelines as precisely as possible. This is usually done using natural language and the results are put together in a so-called standard.

  Depending on the application area, a vast number of standards have been produced, e.g., the *Common Criteria for Information Security Evaluation* [The99].

- Intensive testing is another way to improve the quality of critical systems. Along with an increasing severance in the case of failure, the testing becomes more exhaustive.

- Redundancy is still the primary choice for highly critical systems. In avionic and space industry, most core systems come in triplicate, so that if one system fails, a spare one takes over.

- Formal methods in specifying and verifying individual parts or modules of critical systems are becoming more and more familiar tools in industry. Here, formal methods means paper-and-pencil proofs or computer-aided verification (CAV) .

All of the above approaches have flaws. When using natural language, ambiguity comes into play. Besides all of the 'shoulds' and 'coulds', there is often plenty of space left for individual interpretation by the reader.

Exhaustive testing is neither an answer. It is helpful for debugging, i.e., in finding bugs, but it won't guarantee their absence. You can compare it to looking for the needle in the haystack: you have found one, two, three, and so on, but who tells you that there are none left? And often enough, test developer and programmer are the same people. So when they develop tests, they have the 'right' way to handle their software always in mind. Who would have thought about an ingenious trick as the one the Panama doctors had come up with?

Formal methods are useful, no doubt. Mathematical precision allows to avoid the impreciseness of natural language. Computers check or even lead the proofs that a piece of software or a hardware component is fault-free. Unfortunately, a system consisting of correct modules is not automatically correct itself: you have to consider the interfaces and the interplay of the different components: they might be correct, but do they also fit together?

The Verisoft project [IP08, Ver03] pursues a different strategy, namely to formal and pervasive verification of entire computer systems. Here, formal means that all proofs are machine-checked or machine-made using computer-aided verification systems. Pervasive stands for the policy that all correctness properties from the abstract upper layers are transferred down through all the intermediate layers to the actual hardware, where they are to be discharged using only a very small set of well-known axioms.

This procedure guarantees that the assumptions made are not too strong, since they have to be justified against the 'real world', the hardware[1]. On the other side we exclude human shortcoming by having the proofs rigorously machine-checked.

Hence, the so-verified systems are of extreme quality as required in many industrial sectors, such as automotive engineering, security, and medical technology.

## 1.1 The Context: The Verisoft Academic System

Another goal of Verisoft was the prototypical application of the methodology to four concrete scenarios, of which three came from the industrial sector. In

[1] Of course there can still be errors on the lowest level not captured by the proofs, e.g., physical damage, external influences, etc.

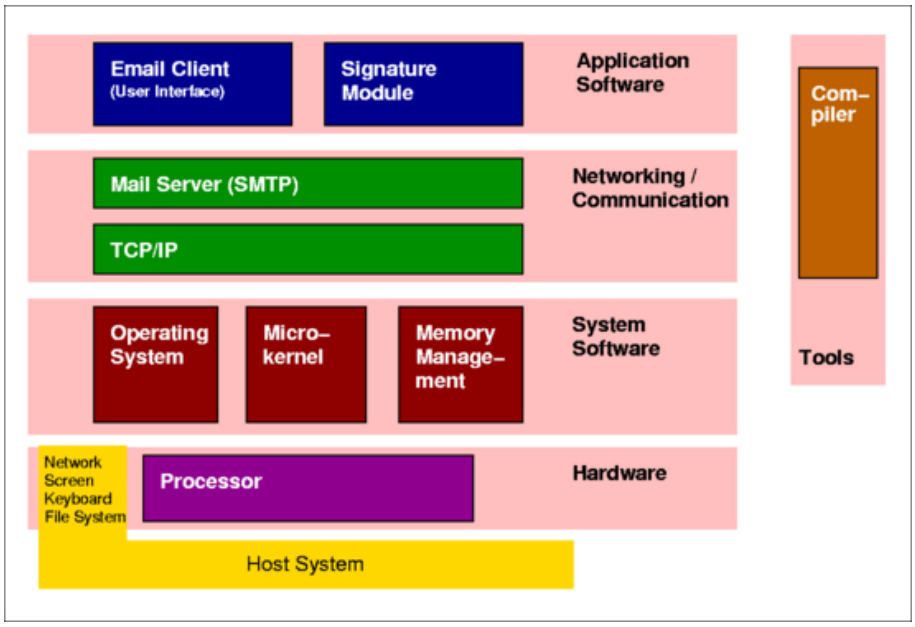


Figure 1.1: The Verisoft Academic System

one of these scenarios, a pervasively verified system (the academic system) for writing, signing, and sending emails had to be constructed. As such, the academic system represents a vertical cross section of a real general-purpose computer system, covering all layers from the gate-level hardware description to communicating concurrent programs. The properties of the academic system correspond to the required standards for real systems.

The academic system design comprises four layers. At the bottom, we have the *hardware layer* which mainly consists of a microprocessor. In order to interact with the academic system, this processor is realized on a FPGA, which itself is hooked up to a host PC.

The second lowest level is the one of the *system software*. Here, we find the microkernel, on top of which an operating system runs, and the memory management mechanisms.

Above the system software, we find the *networking and communication* layer. Three protocols are part of the academic system in order to realize network communication and email transfer.

On the topmost layer, the *application software* is located: a simple email client, serving as the user interface, and a cryptographic signature module.

Orthogonally to the system stack, we also have a tool layer, which comprises the compiler to translate the high-level language implementations.

The different implementation layers are shown in Fig. 1.2. We will now describe the components of the academic system in bottom-up fashion.

**Processor Hardware** The hardware platform is given by the VAMP (Verified Architecture Microprocessor) [BJK+06]. The VAMP is a 32-bit RISC CPU

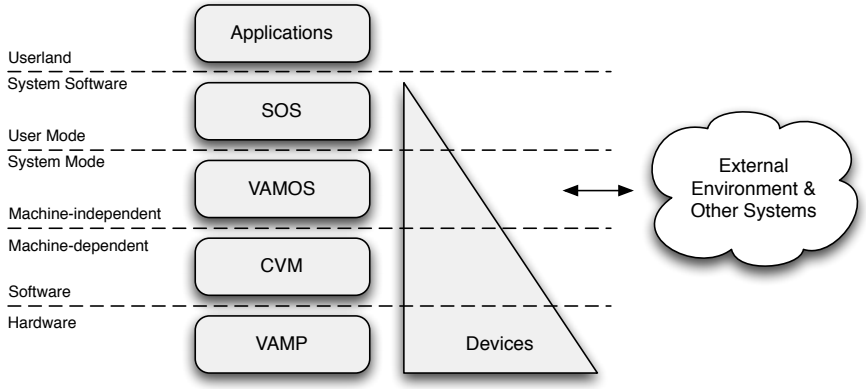


Figure 1.2: Implementation Layers of the Academic System

with full DLX-instruction set including a set of IEEE-compliant floating point instructions, delayed PC, address translation, and support for maskable nested precise interrupts.

The VAMP hardware consists of a 5-stage pipeline with an out-of order Tomasulo scheduler with reorder buffer [Kro01]. There are five execution units, the XPU (fix point unit), the MU (Memory Unit), and three FPUs (Floating Point Units). Instructions have up to six 32-bit source operands and deliver up to four 32-bit results.

The floating-point units [Jac02] are fully IEEE compliant and support single and double precision operations as well as denormal numbers and the full set of IEEE exceptions in hardware. The FPUs are pipelined and the multiplicative FPU has a cycle in the pipeline structure in order to compute quotients with Newton-Raphson iteration.

The memory interface of the VAMP consists of two MMUs (Memory Management Units) that access instruction and data cache, respectively, which in turn access a physical memory via a bus protocol. The caches [Bey05] support write-back and are kept consistent with snooping.

A verified version of the VAMP with MMUs in the interactive theorem prover PVS [OSR92] can be found in [Dal06], basing on results from [Hil05, DHP05]. The results have been repeated in Isabelle/HOL [NPW02] with a much higher degree of automation by integrating automatic tools [Tve09, TS08, Tve05].

**Compiler** A compiler has been implemented and formally verified, based on a small-step semantics for $C0$, our restricted version of C. The implementation of the compiler has been done in $C0$ itself.

The specification of the $C0$ compiler in Isabelle/HOL and its implementation in $C0$ have been completed. There exists a formal proof based on a Hoare logic for $C0$ [Sch06, Sch05] that the implementation generates the same assembly code as the compiler specification [Pet07]. A formal proof that the generated assembly program simulates the original $C0$ program has been conducted in Isabelle/HOL and presented in [Lei08].

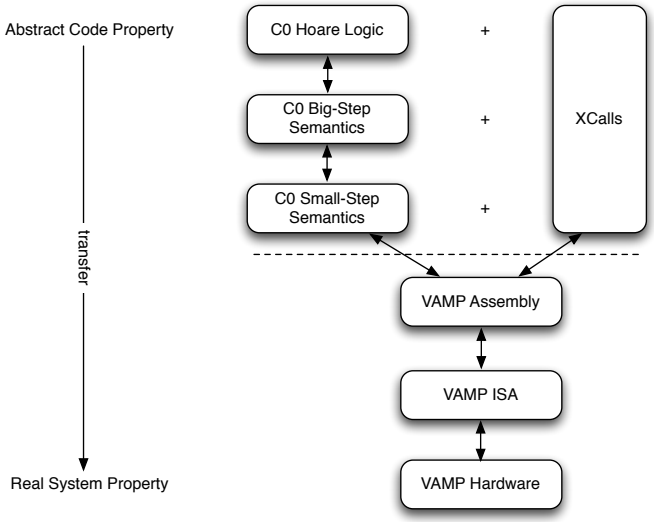


Figure 1.3: Semantic Layers in Verisoft

**Microkernel and CVM Layer** The layer of communicating virtual machines (CVM) has been introduced for the first time in [GHLP05]. A refined and improved version can be found in [IT08].

The CVM layer establishes a hardware-independent programming interface for a microkernel and a virtual computation environment for concurrently running processes. Some parts of CVM must be implemented in assembler since *C*0 lacks—for good reason—low-level programming constructs. The microkernel, named VAMOS and implemented in *C*0, is based on the CVM interface and contains no assembler parts.

This work as well as [Tsy09, ST08] deals with aspects of CVM correctness. More details about VAMOS can be found in [DDB08, DDWS08].

**Operating System** The kernel calls of the above microkernel are targeted for the implementation of a simple operating system (SOS) on top of this. It offers file I/O, inter-process communication, remote procedure calls and access to the peripherals as system calls to user processes. A formal specification of SOS has been presented in [Bog08].

**Networking and Communication Protocols** To realize network connectivity, a *C*0 implementation of the standard protocols Internet Protocol (IP, RFC 791 [Inf81a]) , which provides packet-oriented communication, and Transport Control Protocol (TCP, RFC 793 [Inf81b]), providing reliable connection-based communication, has been created.

To establish email communication, a mail server implementing the Simple Mail Transfer Protocol (SMTP, RFC 2821 [Gro01]) has been implemented. It provides the functionality for sending emails via SMTP (sendmail functionality), and it receives emails via SMTP and delivers them to the correct recipient/user

(mail server functionality).

**Application Software** There are two applications running atop of the simple operating system. One is a signature module providing the functionality for cryptographically signing texts (in particular emails) and for checking such signatures. It uses an asymmetric signature algorithm (RSA-PSS) with an SHA-1 hash function.

There is a formal specification of the module's interfaces and of RSA-PSS (including error handling). RSA-PSS and SHA-1 have been implemented in $C0$, and RSA-PSS has also been formally verified [LWB06].

A simple email client is the top-level application, providing the user interface for the whole academic system. It provides functionality for editing, signing, and sending emails, as well as receiving, checking the signature of, and reading emails.

The client, implemented in $C0$, has been fully formally verified [BHW06]. Work on how to formally define security requirements of user interfaces (such as the email client) has been published in [BB06b, BB06a, BB04].

User interaction is handled via a standard terminal hooked up to the system by a serial interface . A formal device and programming model for such a device has been presented in [AHK+07].

A comprehensive overview of the overall stack verification of the academic system from the low-level hardware to the level of CVM can be found in [AHL+09].

## 1.2 Motivation

In the last forty years, several major projects have been dedicated to operating system verification (see also Chapter 2). Most of the work has been invested in the formal specification of operating systems and kernels. Only a very few projects involved mentionable verification efforts on code-level.

L4.verified/seL4 [KEH+09] and KIT [Bev87] are exemptions. The first one features a high-performance microkernel, whose implementation seems to be fully formally verified. Since pervasiveness was not one of the project's goals, the trusted base is pretty large (compiler, hardware, assembler code portions).

On the other hand, KIT has been part of the famous CLI stack, which has been formally and pervasively verified. Here, the kernel functionality measured by the services it offers is very limited and cannot be compared to a modern microkernel.

The CVM model presented in this work aims to cover both aspects: its features are comparable to those of a second generation microkernel and is at the same time part of the Verisoft pervasive model stack (cf. Fig. 1.3).

Another important aspect we consider in this thesis is modularization and compositionality. While KIT was still written in assembler for a particular processor, nowadays microkernels are supposed to target several hardware platforms. Necessarily, any kernel implementation has to include low-level code parts. Without proper separation of the hardware-specific code from the other parts, verification work has to be redone for all target architectures.

For this reason we have encapsulated the hardware-dependent low-level implementation. A user of our framework can use his own hardware-independent

kernel implementation (*abstract kernel*). Furthermore we provide a formal linking mechanism, which allows a kernel implementer to merge the code parts and thus obtain a compilable, runnable kernel (*concrete kernel*).

Implementation correctness for the abstract kernel can then be proven independently from the rest of the implementation. Since this abstract implementation can be written completely in our high-level programming language $C0$, more sophisticated and powerful code verification tools like [Sch06] can be used. The verification results of the CVM layer can then be used in an *assume/guarantee* style of reasoning.

A considerable part of the overall correctness theorem for CVM deals with the correctness relation between the abstract and the concrete kernel. For standard abstract kernel steps, i.e., no kernel calls, the preservation of this correctness relation has been proven and will be presented in this work (see Chapter 8). To a very large extent, this proof has been conducted in the interactive theorem prover Isabelle/HOL [NPW02].

Major parts of the formal theories of Verisoft have been published or are in the process of publication [HP07]. This work relies on other Verisoft results: the low-level hardware and its correctness [Tve09], assembler language semantics [Tsy09], $C0$ C-like programming language semantics and compiler correctness [Lei08], stack verification and top-level correctness [AHL+09].

## 1.3 Outline

In the rest of this chapter, we will introduce some basic notation. In Chapter 2, we discuss related work.

The remainder of this thesis is split into two parts. The first one is dedicated to the computational models required to finally define CVM:

- In Chapter 3, we will introduce the C-like programming language $C0$. We start with the concrete $C0$ syntax and proceed then to the formal representation of $C0$ programs. In the end of the chapter, we work out on expression evaluation and the $C0$ transition function.
- In Chapter 4, a generic framework for devices interacting with a processor and an (not modeled) environment will be introduced.
- In Chapter 5, we will discuss the underlying hardware. The bottom layer is given through the instruction set architecture of the so-called VAMP (Verified Architecture Microprocessor), whose semantics will shortly be introduced. Subsequently, we will combine this model with the generalized device model from Chapter 4.

  The next abstraction layer is specified by the formal assembler semantics, the natural machine model for a system level programmer. The definitions of its data types and transition functions make up for the rest of this chapter.
- In Chapter 6, we will introduce *Communicating Virtual Machines*—short: CVM—, a hardware-abstracting computational model for user processes interacting with each other and devices via a so-called abstract kernel. The mechanisms for communication are established by special functions called *primitives*, whose semantics will be defined subsequently. Finally,

we present a concept allowing to build a runnable, compilable and hence concrete kernel by linking the hardware-dependent parts to it.

In the second part, we will deal with CVM correctness:

- In Chapter 7, we will work out the idea of a CVM correctness theorem, a simulation theorem between the hardware and CVM. Here, we will define the necessary abstraction relations between the components and the hardware. Finally, we will give a sketch of how an overall correctness theorem could look like.
- In Chapter 8, we will present part of the correctness proof of CVM. In particular, we show how the abstraction relations between the two kernels is preserved for non-primitive kernel steps.

We conclude in Chapter 9 with a summary and an outlook on future work.

## 1.4 Notation

In this section, we introduce the notation necessary to understand the rest of this thesis.

### 1.4.1 Basics

We use $\Sigma^+$ to denote the set of identifiers, for instance variable or type names. $\mathbb{N}$ is the set of the natural numbers, while $\mathbb{Z}$ stands for the integers. $\{i, \ldots, j\}$ denotes the integer interval from $i$ to $j$ for $i < j$. The Boolean set $\{True, False\}$ is abbreviated by $\mathbb{B}$.

Sometimes, we introduce anonymous functions using . For instance, $\lambda x, y.x \cdot y$ is a function that yields the product of two operands.

$f^n$ is the $n$-time application of a function $f$ with $f^n(x) = f(f^{n-1}(x))$, with $f^1(x) = f(x)$.

We define a finite sequence with $n$ elements of type $t$ formally by a mapping $s : [0 : n-1] \to t$. For $0 \leq i \leq j < n$, $s[i : j]$ denotes the sub sequence $(s[i], \ldots, s[j])$. Sometimes, we specify a sub sequence by $s[i, l]$, where $0 \leq i < n$ defines the start index and $l$ with $i + l < n$ the length of the sub sequence.

In particular, we treat bit vectors as sequences of bits. The type of all bit vectors of length $n$ is denoted by $\mathbb{B}^n$.

Frequently, we will use *records* in this thesis. We access their components with the . operator, that is $r.c$ returns the component $c$ of record $r$. We denote record updates with $r[c_0 := this, c_1 := that, \ldots, c_i := 42]$ where $r$ is a record and $c_0, \ldots, c_i$ are components of it. We assume that components that are not explicitly mentioned in a record update stay unchanged. On the fly creation of a record is written $[c_0 = this, c_1 = that, \ldots, c_i = 42]$.

Not surprisingly, *predicates* are functions from an arbitrary domain into the Boolean set. A predicate $P$ *holds* or *is valid* for an input $x$, iff $P(x) = True$. Shorthand notations for $P(x) = True$ and $P(x) = False$ are $P(x)$ and $\neg P(x)$.

Implications are often given by inference rules . For instance, $(A \wedge B) \Rightarrow C$ is denoted by

$$\frac{A \qquad B}{C}$$

We will sometimes use this notation to define predicates. In these cases, we assume that everything not covered explicitly by inference rules, evaluates to *False*.

In places where their necessity is obvious, we will omit universal and/or existential quantifiers . We say $d$ *divides* $n$*, iff there exists an* $i \in \mathbb{Z}$ *such that* $i \cdot d = n$ and write:

$$\frac{i \cdot d = n \qquad i \in \mathbb{Z}}{d \mid n}$$

Often, we will have to deal with pairs. We define $fst : t_1 \times t_2 \to t_1$ and $snd : t_1 \times t_2 \to t_2$ that return the first element or the second element of a pair respectively.

### 1.4.2 Abstract Data Types

We use abstract terms to define terms in a structured way. The construction rules are given by so-called *constructors*. The set of all terms, which can be build using the constructors of an abstract data type $t$ is called *type* $t$. The *type* $t$ is the set of all terms that can be built by using the constructors of an abstract data type $t$. Each constructor is a function with a result in the range of $t$.

For instance, an abstract data type for lists with elements of type $t$ could look as follows:

$$\begin{aligned} t\ list &= Nil \\ &\mid Cons(t, t\ list) \end{aligned}$$

Here, we have two constructors: $Nil : t\ list$ for the empty list, and $Cons : t \times t\ list \to t\ list$ for the concatenation of an element and a list.

### 1.4.3 Partial Functions and Option Type

Isabelle/HOL has no support for partial functions, but there are two ways to simulate them. First, we can introduce a constant representing the undefined value. In this thesis, we name this constant *undef*.

Furthermore, there is the so-called *option type*. Formally, an option type is an abstract data type with two constructors.

For a type $t$ we define the corresponding option type as:

$$\begin{aligned} t\ option &= None \\ &\mid Some(t) \end{aligned}$$

Here, *None* specifies an undefined value and $Some(x)$ a defined value $x$. For each option type $t$ there is a function $the : t\ option \to t$ that converts the argument back to the base type. We define

$$the(x) = \begin{cases} y & \text{if } x = Some(y) \\ undef & \text{otherwise} \end{cases}$$

For the rest of this thesis, we use $t_\perp$ for $t\ option$ and $\lfloor x \rfloor$ for $Some(x)$ as shorthand notation.

### 1.4.4 Lists

We have seen the abstract data type for lists. It is very cumbersome to use the constructors explicitly—and lists are vastly used in this thesis. For conciseness we denote the empty list *Nil* with $[\,]$ and write $h\#t$ for $Cons(h,t)$. We can describe lists explicitly: for instance, $[a,b,c]$ is short for $a\#b\#c\#[\,]$, which is again pretty printing for $Cons(a, Cons(b, Cons(c, Nil)))$ .

We can apply a function $f$ to all elements of a list by the *map* function: $map : (t_1 \to t_2) \times t_1\ list \to t_2\ list$, which is defined inductively:

$$\begin{aligned} map(f,[\,]) &= [\,] \\ map(f,h\#t) &= f(h)\#(map(f,t)) \end{aligned}$$

The length of a list is equal to the number of its elements:

$$\begin{aligned} |\,[\,]\,| &= 0 \\ |\,h\#t\,| &= Suc\,|\,t\,| \end{aligned}$$

We need some more auxiliary functions to facilitate list handling: $hd : t\ list \to t$ returns the first element—the head—of a list, $tl : t\ list \to t\ list$ the tail of the list, i.e., the list without its head. :

$$\begin{aligned} hd([\,]) &= undef \\ hd(h\#t) &= h \\ tl([\,]) &= undef \\ tl(h\#t) &= t \end{aligned}$$

The append operator $@ : t\ list \times t\ list \to t\ list$ allows to concatenate two lists of equal types:

$$\begin{aligned} [\,]@l &= l \\ (h\#t)@l &= h\#(t@l) \end{aligned}$$

$ith(l,i)$—or short: $l!i$—returns the $i$-th element of list $l$ . Note that the first element of a list is its 0-th element. For $i \geq |\,l\,|$, the result is *undef*. Updating the $n$-th element of a list with some value $x$ is denoted by $l[n := x]$.

The function $butlast : t\ list \to t\ list$ takes a list and returns this list without its last element We define:

$$\begin{aligned} butlast([]) &= [] \\ butlast(xs\#x) &= xs \end{aligned}$$

In some places, we need to flip a pair, that is the first element becomes the second and vice versa. For this purpose, we define $flip : (t_1 \times t_2) \to (t_2 \times t_1)$ and set $flip(a,b) = (b,a)$.

The function $replicate : \mathbb{N} \times t \to t\ list$ creates a list with as many copies as specified by the first argument of the functions second argument, e.g.,

$$replicate(5,a) = [a,a,a,a,a].$$

.

Frequently, we have to convert lists into sets . In this case, $\{l\}$ denotes the set derived from a list $l$.

We define an auxiliary function $mapof : (t_1 \times t_2)\, list \times t_1 \to t_{2\perp}$, which works as a pattern matching on lists of pairs . $mapof$ takes two arguments: a list of pairs to be searched and a pattern to be compared. The second element of the first pair in $l$ matching the pattern will be returned. If there is no such element, the result is $None$.

We define formally:

$$\begin{aligned} mapof([], x) &= None \\ mapof((a,b)\#t), x) &= \begin{cases} \lfloor b \rfloor & \text{if } a = x \\ mapof(t, x) & \text{otherwise} \end{cases} \end{aligned}$$

We define a function $filter : (t \to \mathbb{B}) \times (t\, list) \to t\, list$, which filters lists using a predicate as the filter criterion. For a given predicate $P$, we define recursively:

$$\begin{aligned} filter(P, [\,]) &= [\,] \\ filter(P, x\#xs) &= \begin{cases} x\#(filter(P, xs)) & \text{if } P(x) \\ filter(P, xs) & \text{otherwise} \end{cases} . \end{aligned}$$

I'm not the smartest fellow in the world, but I can sure pick smart colleagues.

Franklin D. Roosevelt

# Chapter 2

# Related Work

We start with the related work on operating system verification in Sect. 2.1, where older work is treated in Sect. 2.1.1 and more recent work in Sect. 2.1.2. For a brilliant and exhaustive overview on this topic, we strongly recommend [Kle09].

Work, which is not directly dedicated to operating system verification but contributing to it, is discussed in Sect. 2.2.

We conclude with the past and on-going work related to microkernel and operating system verification in Verisoft and its successor Verisoft XT (cf. Sect. 2.3).

## 2.1 Operating System Verification

### 2.1.1 The Early Days

The first major attempts to specify and verify complete operating systems—PSOS and UCLA Secure Unix—date back to the seventies of the last century.

**PSOS—A Provably Secure Operating System** The PSOS—provably secure operating system—project at SRI started in 1973. According to [NF03], PSOS *'was designed as a useful general-purpose operating system with demonstrable security properties'*.

The PSOS design and specification principles relies heavily on hierarchical modeling, where the abstraction objects of each layer were meant to be implemented by the abstract and primitive objects of lower layers.

This modeling approach was named *Hierarchical Development Methodology (HDM)* and had been developed very early in the project, coming along with a specification and assertion language SPECIAL [RLNS75, RL77]. Using SPECIAL, each module in the system had been specified where a layer could comprise a number of encapsulated modules. Furthermore, SPECIAL was supporting mapping and abstraction functions between modules on different layers, a feature that could have been used as a base for implementation proofs later on.

The design of PSOS seems to be basically complete, though it remains unclear how much of it has actually been implemented—considering the fact that new hardware had been involved. According to the project's final report [NBFR80],

no code proofs had been accomplished but only *'some simple illustrative proofs were carried out'*. Research on information flow analysis on PSOS went on until 1983 [GM82, GM84].

Nevertheless, the PSOS project has a long-lasting and far-reaching impact. The experiences made with HDM have led to the development of SRI's Prototype Verification System (PVS) [OSR92]. Furthermore, the hierarchical approach to system modeling and proving was inspiring to the famous CLI stack project [Moo89, BHMY89] (cf. Sect. 2.1.1).

The PSOS methodology was used in the Ford Aerospace Kernelized Secure Operating System (KSOS) [BBJ79, MD79, PCH84].

**UCLA Secure Unix** The UCLA[1] Secure Unix had been developed as an operating system for the DEC PDP-11/45 computer [WKP80]. Though the term 'microkernel' had not been invented at this point in time, this was already very close to the concepts of modern microkernels, and thus an early attempt in separating operating system from kernel functionality. The verification efforts were concentrating on the kernel component.

The security proof for the kernel was split into two part. In a first step, a four-level specification, ranging from Pascal code at the bottom to the top-level security criterion, had to be developed. Then, the verification of the system was completed by proving that these different levels of abstractions were consistent with each other, more precise: that the state machines simulate each other.

The upper three levels were formulated by means of finite state machines. The top-level security criterion was supposed to capture the common notion of *data security* based on capabilities (cf. Sect. 2.1.1): state components are only allowed to be modified or referenced if the current process allows for it.

The authors of [WKP80] state, over ninety percent of the UCLA kernel has been specified—though they think that *'the task is still too difficult and expensive for general use'*. Yet, the specification work has uncovered already significant security bugs, which had slipped conventional testing.

An implementation of the kernel in a subset of Pascal has been completed, of which less than twenty percent have been verified. The authors write that *'even accomplishing even this much was quite painful'* and recommend to wait for more effective machine aids in order to complete the task. The choice went for Pascal, since it was both—relatively—suitable for low-level system software implementation and it had a clear formal semantics [HW73].

Furthermore, the authors state that *'the recommended approach to program verification—developing the proof before or during software design and development—is often not practical'* [WKP80, Sect. 4].

Standard Pascal had been extended by the ability to locate variables at specific memory regions, so that hardware register manipulation could be realized without in-line assembly parts. Code fragments which were referencing or manipulating hardware registers where isolated and put in separate functions. Obviously, the Pascal semantics as described in [HW73] do not capture their effects. It seems that the pre- and postconditions of these functions have been added manually in an axiomatic way.

An interesting observation by the authors is the negligence of hardware finiteness, in particular when dealing with arithmetic and the corresponding

[1] University of California, Los Angeles

translation from high-level languages to lower levels. In Verisoft, the same insight led to the introduction of so-called *guards* in high-level code verification, triggering in the case of an arithmetic overflow.

The overall system performance was pretty poor—an order of magnitude slower than the standard Bell Unix of those days in some cases. This is due to the use of a nearly non-optimizing compiler (the people in the project wanted to be able to at least manually check the generated code) and a bad overhead in process switching.

**KIT** Probably the first kernel deserving to be called formally verified, is KIT—Kernel for Isolated Tasks—from the end of the 1980s [Bev87, Bev88, Bev89a, Bev89b].

Besides process isolation—as implied by the name—KIT offered services for asynchronous I/O device access, single-word message passing and exception handling. There was no support for shared memory or virtual memory in the modern sense, nor did KIT offer dynamic process creation.

KIT has been implemented in an artificial, yet realistic assembly language. The code size is with 620 lines, out of which are 300 lines of actual instructions, is very small. No surprise that the kernel is rather primitive compared to modern microkernels.

The verification has been conducted in the Boyer-Moore theorem prover Nqthm [BKM95], which was the precursor to the well-known ACL2 prover [Moo00]. Similar to UCLA Secure Linux (cf. Sect. 2.1.1), the correctness proof for KIT relies on the correspondence between finite state machines.

The correspondence proof shows that the kernel implements this abstraction correctly, that is no implementation bugs have been introduced. The underlying hardware is assumed to be fault-free.

The top-level specification seems to be strong enough to guarantee process isolation, a property which is described as *'that a task's private state can change only when it is active'* [Bev87, p.4].

### 2.1.2 Recent Work

The decade after KIT was very quiet regarding projects in operating system verification. Since the dawn of the new millennium, it seems that the topic has been revived.

**VFiasco** The VFiasco project, short for Verified Fiasco, started in 2001. Fiasco [HH01] is a C++ re-implementation of the high-performance microkernel L4 [Lie95]. The Fiasco kernel is built with the intention to be used in real-time systems, so almost all of its code is pre-emptible, that is interruptible guaranteeing for short reaction times to external interrupts.

The idea of the project is the direct translation of the programming language C++ into its semantics in the theorem prover PVS [OSR92], thus avoiding the formalization of C++ syntax there.

VFiasco has not seen any publications since 2005, the last one [HT05] summarizing on the C++ verification approach, which has though been continued in the Robin project on the Nova Hypervisor [Tew07].

Yet, both projects lack any successes in verifying representative portions of the implementation.

**EROS/Coyotos** Coyotos [SDNM04] is a secure, microkernel-based operating system that builds on the ideas and experiences of the EROS project [SSF99]. EROS (Extremely Reliable Operating System) is a capability-based second generation microkernel succeeding KeyKOS [Har85].

In [SW00], a paper-and-pencil formalization with a proof for the security model have been presented. Yet, this security model was not related to the actual EROS implementation.

This was part of the successor kernel Coyotos. For this purpose, a new implementation language called BitC [SS06] has been developed, which is both powerful enough to implement low-level system software and also eases verification efforts. This idea is very appealing, since all common implementation languages for kernel (C, C++, assembly) have significant flaws and insecure features.

It seems that the Coyotos project makes rapid progress on the design and implementation of the kernel. Until now, there has been no publications on verification efforts or on a framework for reasoning on BitC programs.

**L4.verified/seL4** The goal of the seL4 (secure embedded L4) project was the enhancement of Liedtke's L4 microkernel conception [Lie95] with attention to security-relevant embedded systems [Elp04].

According to [EKD+07, KEH+09], the project has been successfully concluded by the end of 2007. The result is a C and assembly implementation (8,700 LOC C, 600 LOC assembly) running on the ARM11 hardware platform, offering threads, IPC, memory virtualization and interrupt handling and is of commercially competitive performance.

The design approach of seL4 is very appealing, since it combines aspects of both OS design and formal verification. The OS design group was using the programming language Haskell [Jon03] for rapid prototyping and—enhanced with a hardware simulated—testing with user applications.

Since Haskell is very close to Isabelle/HOL, it is automatically translatable into the theorem prover and hence available as a low-level formal design for further verification efforts.

Strongly related to seL4 is the L4.verified project, in which a model stack above the concrete implementation has been developed. The topmost layer is an access control model for seL4 [EKE08], which is refined to an operational model of user-visible kernel operations. The two lower layers are made up by the Haskell low-level formalization and the actual implementation.

Verification in L4.verified aims at functional correctness, in the sense that *'the implementation always strictly follows our high-level abstract specification of kernel behavior'*. The refinement proof between the bottom two layers is conducted within Verisoft's Hoare-style verification framework [Sch06].

As mentioned above, hardware, compiler, and assembly code belong to the trusted base. [KEH+09] states that the verification efforts have been completed.

## 2.2 Other Related Projects

There are a number of projects, which do not explicitly deal with operating system verification, but are of interest due to their scientific contributions to the topic.

**FLINT** The FLINT project has a special interest in developing the infrastructure needed to construct large-scale certified system software. In particular, the enhancement of theorem provers to be more suitable for building large proofs for system level software is in the focus.

In [FSDG08, FSGD09], the authors present a novel Hoare-logic-like framework for certifying low-level system programs involving both hardware interrupts and preemptive threads.

In [NYS07], a verification framework named XCAP is introduced and exemplarily applied to prove the correctness of context switching code on the Intel x86 architecture.

**AAMP7** The AAMP7 microprocessor of Rockwell Collins, Inc., basically implements the properties of a separation kernel in hardware. In order to obtain Common Criteria EAL7 certification [The99] for it, various efforts have been undertaken [HSY06]. The separation kernel model used is based on [MWE03, MWE05], the proof work itself is done in ACL2 [Moo00]. A low-level model is closely related to the actual implementation (processor microcode), but there is no formal correspondence proof, though the work done surely goes beyond the requirements of an EAL7 evaluation.

**Specification of an Operating System in Focus** In [Spi98], a high-level abstract model for a generic operating system has been formalized in Focus [BDD+92]. The model covers several main aspects of an operating system, such as processor management, memory management, process cooperation, and management.

**Specification of the Mach Kernel** The term microkernel is reported first to be used in the context of the Mach kernel [RBF+89] developed at Carnegie-Mellon in the years from 1985 to 1994.

In [BS93b, BS93a], a formal specification of the kernel is given, though to the best of our knowledge, no implementation proofs had been conducted.

**Linking** Unfortunately, there is not much related work on the formalization of linking (cf. Sect. 6.5.2), in particular not on the level of programming languages semantics.

As Cardelli says in [Car97]: *'We suggest that linking and separate compilation should be seriously taken into account when designing a language and module system. This sentence may seem a truism, but these issues have been surprisingly under-emphasized in the technical literature.'*

This statement is more than ten years old, but the community still neglects this topic carelessly.

Some further related work on compilation management for Standard ML code fragments can be found in [HPLR94]. Recompilation, which is strongly connected to linking and dependency analysis, and its optimization is addressed in [Tic86] and [SA93].

## 2.3 Verisoft/Verisoft XT Context

The development of CVM has been application-driven. In the Verisoft project, two microkernels have been developed, which make use of the low-level CVM implementation.

In the successor project, Verisoft XT, work on operating system verification has been continued. Here, two commercially available products are subject to verification: Microsoft's virtualization platform Hyper-V [Hyp09] and SYSGO's microkernel PikeOS [Pik09].

**VAMOS** VAMOS (Verified Architecture Microkernel Operating System) is a microkernel implementation, which is used in Verisoft's academic system (see Sect. 1.1).

VAMOS provides a process scheduler, an infrastructure for communication with hardware devices, and message passing between processes. All software layers are formally specified; refinement relations correlate the adjacent layers such that eventually all specification layers can be mapped down to our hardware-processor model. The whole model stack is formalized in the theorem prover Isabelle/HOL.

For considerable parts of the kernel, correctness proofs have been conducted, for instance, fairness and implementation correctness of the scheduler have been shown [DDW09].

Currently, implementation correctness for the inter-process communication has come close to an end. We expect the results to be published in 2010.

**OLOS** OLOS (OSEKtime-like Operating System) has been implemented as a time-triggered microkernel for the Verisoft automotive project. Considerable parts of OLOS—including the communication layer and bus controller device—have been formally verified in Isabelle/HOL [ABK08, Kna08, BBG+05, IK05], with a special attention to the real-time properties of the system [KP07b, KP07a].

Basing on OLOS, a distributed automotive system has been exemplarily set up, where the single nodes where interconnected via a real-time bus [KS06]. An application, developed together with BMW, has been implemented and formally verified [BGH+06, BKKS05].

### 2.3.1 Hyper-V

Hyper-V [Hyp09] is a virtualization platform, which allows to run multiple operating systems at the same time on Intel's and AMD's latest x64 hardware, by providing a transparent interface to the underlying hardware. It is part of the Windows Server 2008 distribution. Hyper-V comprises about 100KLOC of C code and another 5KLOC of assembly code.

Besides the sheer complexity of highly optimized, industrial performance code, Hyper-V itself runs multithreaded in a concurrent setting and has not been implemented with formal verification in mind.

This makes conventional (sequential) code verification not applicable. A new tool, VCC, had to be developed [CDH+09] along with the corresponding methodology [CMST09, MMS08, BLW08]. VCC allows to write verification conditions directly into the program code (of course in a structured way, so that

they can easily be removed before compiling), hence assuring the consistency of specification and implementation by design.

Due to the secrecy necessary when dealing with commercial products, only a few publications are available to date dealing with details of the actual verification [LS09].

### 2.3.2 PikeOS

PikeOS is a microkernel for x86, PowerPC, and ARM hardware platforms to develop embedded systems in which multiple guest operating systems and applications can run simultaneously in a secure environment. In particular, PikeOS realizes partitioning, such that several operating systems can run isolated on the same processor.

As in the Hyper-V project, verification is done directly on code level using the VCC tool. The PikeOS team reports on the on-going work in [BBBB09b, BB09, BBBB09a]

# Part I

# Computational Models

> Software is like entropy. It is difficult to grasp, weighs nothing, and obeys the second law of thermodynamics; i.e., it always increases.
>
> Norman R. Augustine

# Chapter 3

# The C-like Programming Language $C0$

**Contents**



In this chapter, we are going to introduce the C-like programming language $C0$. $C0$ has been developed in the first Verisoft project and serves as the programming language for most of the system level software within the project. A simple, non-optimizing compiler from $C0$ to VAMP assembly code (cf. Sect. 5.3) has been developed and completely formally verified. These proofs are covered in detail in [Lei08] and [Pet07].

For the rest of this chapter, we will proceed as follows. In Sect. 3.1, we will introduce the concrete $C0$ syntax.

In Sect. 3.2, we describe formally the representation of programs for the $C0$ semantics. $C0$ configurations keep track of the run-time information during the execution of such a program . We define these configurations formally in Sect. 3.3.

In Sect. 3.4, we treat the evaluation of $C0$ expressions . Last but not least, we complete $C0$ program execution with the formal definition of the $C0$ transition function in Sect. refc0semantics::sect::delta.

The semantics of $C0$ will be used later on in Chapter 6 to model the behavior of CVM's abstract kernel .

## 3.1 An Informal View on $C0$

C has been developed in the early seventies of the last century. It has been developed for the implementation of system level software, in particular for kernel implementation.

Throughout the years, several standards for C have been developed and published, the latest dating back to 1999 [ISO99]. Based on considerations regarding formal verification, certain design limitations compared to these

standards have been applied to $C0$. For instance, we do not allow side effects during expression evaluation; this includes function calls as part of expressions, making the introduction of a particular $C0$ function call statement necessary (see Sect. 3.2.3).

$C0$ features a dedicated type for arrays . Their size has to be fixed at compile time.

We restrict the use of pointers, which are typed in $C0$. Pointers must not point to functions or local variables . There are no void pointers and pointer arithmetic is forbidden.

We do not allow explicit memory deallocation, like the C *free* statement. Instead, we are planning to use a garbage collector that frees unreachable heap portions periodically. This garbage collector has already been code verified, but yet not been integrated into the $C0$ semantics.

Despite all these restrictions, the various efforts in Verisoft have produced evidence for the usability of $C0$ in system programming. For instance, two operating systems, *VAMOS* [AHL+09] and *OLOS* [IK05] have been implemented in $C0$. In Verisoft's academic sub-project, a whole system stack including protocol, driver and application implementations has been developed and formally verified in large parts.

### 3.1.1 Types

In $C0$, we support four basic types:

- Boolean: *bool* with values in $\{True, False\}$
- 32-bit signed integers: *int*, with values in $\{-2^{31}, \ldots, 2^{31}-1\}$
- 32-bit unsigned integers: *unsigned int*, with values in $\{0, \ldots, 2^{32}-1\}$
- 8-bit signed integers: *char*, with values in $\{-2^7, \ldots, 2^7-1\}$

We allow arithmetic operations only on 32-bit integers and unsigned integers. In addition, we define so-called aggregate types:

- For a type $t$, $*t$ denotes a (typed) pointer.
- For a type $t$ and $n \in \mathbb{N}$, $t[n]$ denotes an array of (fixed) size $n$ of type $t$.
- For types $t_0, \ldots, t_{i-1}$ and names $\sigma_0, \ldots, \sigma_{i-1}$, $struct\{\sigma_0 : t_0, \ldots, \sigma_{i-1} : t_{i-1}\}$ denotes a structure with $i$ components.

Note that we do not support unions or bitfields.

### 3.1.2 Expressions

Variable names and literals are expressions. For an expression $e$, $(e)$ is an expression.

If both $e$ and $i$ are expressions, than the array access $e[i]$ is also an expression. Let $\sigma \in \Sigma^+$ be a name, and $e$ an expression; then $e.\sigma$, that is access of a structure component, is an expression.

Pointer dereferencing $*e$ and application of the 'address-of' operator $\&e$ are expressions, too.

Table 3.1: Unary C0 Operators

| Operator | Meaning | Supported Operand Types | Result Type |
|---|---|---|---|
| ~ | bit-wise negation | $t \in \{int, unsigned\ int\}$ | $t' = t$ |
| ! | logical negation | $t \in \{bool\}$ | $t' = t$ |
| - | unary minus | $t \in \{int\}$ | $t' = t$ |
| *int* | cast to *int* | $t \in \{char, unsigned\ int, int\}$ | $t' = int$ |
| *unsigned* | cast to *unsigned int* | $t \in \{char, unsigned\ int, int\}$ | $t' = unsigned\ int$ |
| *char* | cast to *char* | $t \in \{char, unsigned\ int, int\}$ | $t' = char$ |

$t$ is the type of the operand, $t'$ is the type of the result

For an expression $e$ and a unary operator $\oplus_1$ from table 3.1, $\oplus_1 e$ is an expression. For expressions $e_1$, $e_2$ and a binary operator $\oplus_2$ from Table 3.2, $e_1 \oplus_2 e_2$ is an expression.

Though pointer arithmetic is prohibited, we allow pointers to sub variables of aggregate variables.

### 3.1.3 Statements

Let $f$ denote a function name, $e$ and $e_i$ functions, $t$ a type, $s$ and $s_i$ statements. Then, $C0$ offers support for the following statements:

- Assignments: $e_1 = e_2$. Unlike standard C, $C0$ allows to assign array values.
- Dynamic heap memory allocation: $e_1 = new(t)$. As mentioned before, there is no support for manual deallocation.
- Function calls: $e = f(e_0, \ldots, e_{n-1})$. The dedicated function call statement meets the constraints regarding side effects in expression evaluation.
- Return from function call: *return* $e$
- While loops: *while* $e\ s$
- For loops: *for* $(e_0 = e_1; e_2; e_3 = e_4)\ s$
- If statements: *if* $e\ s_0$ or *if* $e\ s_0$ *else* $s_1$

Sequences of statements—separated by semicolons—are allowed. In contrast to C, $C0$ does not provide support for goto, switch or long jump statements.

#### Complex Literals

Complex literals comprise literals of aggregate types. They can be used in atomic assignments to non-elementary variables:

- For complex literals $c_i$ of type $t$, $\{c_0, c_1, \ldots, c_{n-1}\}$ denotes a complex literal of the array type $t[n]$.

Table 3.2: Binary C0 Operators

| Operator | Meaning | Supported Operand Types | Result Type |
|---|---|---|---|
| `+` | addition | $t_1 = t_2 \in \{int, unsigned\ int\}$ | $t' = t_1$ |
| `-` | subtraction | $t_1 = t_2 \in \{int, unsigned\ int\}$ | $t' = t_1$ |
| `*` | multiplication | $t_1 = t_2 \in \{int, unsigned\ int\}$ | $t' = t_1$ |
| `/` | division | $t_1 = t_2 \in \{unsigned\ int\}$ | $t' = t_1$ |
| `%` | modulo | $t_1 = t_2 \in \{unsigned\ int\}$ | $t' = t_1$ |
| `\|` | bit-wise or | $t_1 = t_2 \in \{int, unsigned\ int\}$ | $t' = t_1$ |
| `&` | bit-wise and | $t_1 = t_2 \in \{int, unsigned\ int\}$ | $t' = t_1$ |
| `^` | bit-wise exclusive or | $t_1 = t_2 \in \{int, unsigned\ int\}$ | $t' = t_1$ |
| `<<` | logical left shift | $t_1 \in \{int, unsigned\ int\}$, $t_2 \in \{char, int, unsigned\ int\}$ | $t' = t_1$ |
| `>>` | logical right shift | $t_1 \in \{int, unsigned\ int\}$, $t_2 \in \{char, int, unsigned\ int\}$ | $t' = t_1$ |
| `<` | less | $t_1 = t_2 \in \{char, int, unsigned\ int\}$ | $t' = bool$ |
| `<=` | less or equal | $t_1 = t_2 \in \{char, int, unsigned\ int\}$ | $t' = bool$ |
| `>` | greater | $t_1 = t_2 \in \{char, int, unsigned\ int\}$ | $t' = bool$ |
| `>=` | greater or equal | $t_1 = t_2 \in \{char, int, unsigned\ int\}$ | $t' = bool$ |
| `==` | equal | $c_{\text{eq}}$ | $t' = bool$ |
| `!=` | not equal | $c_{\text{eq}}$ | $t' = bool$ |

$t_1$ and $t_2$ are the types of the operands, $t'$ is the type of the result, the type condition $c_{\text{eq}}$ for equality tests is fulfilled if both types are equal or if one of them is a pointer and the other the special null pointer type

Table 3.3: Lazy C0 Operators

| Operator | Meaning | Supported Operand Types | Result Type |
|---|---|---|---|
| `&&` | logical and | $t_1, t_2 \in \{bool\}$ | *bool* |
| `\|\|` | logical or | $t_1, t_2 \in \{bool\}$ | *bool* |

$t_1$ and $t_2$ are the types of the operands, $t'$ is the type of the result

- For complex literals $c_i$ of types $t_i$ and component names $\sigma_i$, $\{.\sigma_0 = c_0, .\sigma_1 = c_1, \ldots, .\sigma_{n-1} = c_{n-1}\}$ is a complex literal of the structure type $struct\{\sigma_0 : t_0, \sigma_1 : t_1, \ldots, \sigma_{n-1} : t_{n-1}\}$.

For an expression $e$ and a complex literal $c$, the assignment $e = c$ is also a $C0$ statement. Note, that complex literals are not allowed within expressions.

#### In-line Assembly

Low-level system software often addresses parts of the underlying hardware— for instance processor registers— that are not visible in higher programming languages. For this purpose we have to extend our $C0$ language with a statement, which allows to embed assembly code into $C0$ code.

Let $u$ be a list of assembler instructions; then $asm\{u\}$ is a $C0$ statement.

The concrete kernel introduced in Sect. 6.5.1 uses in-line assembly code in its hardware-dependent low-level parts (see [Tsy09]). In [Alk09], a low-level device driver makes use of in-line assembly to address a hard disk device.

## 3.2 $C0$ Programs

In this section, we will introduce formally the notion of $C0$ programs . Such a program is given through a *type name environment*—a mapping of type names to types—, its functions, and information about its global variables.

### 3.2.1 Types and Type Name Environment

$C0$ offers four basic types: $Bool_T$ for Booleans, $Int_T$ for signed 32-bit integers, $Unsgnd_T$ for 32-bit unsigned integers, and $Char_T$ for 8-bit signed integers. A structure type with $n$ components of types $t_0, t_1, \ldots, t_{n-1}$ and names $\sigma_0, \sigma_1, \ldots, \sigma_{n-1}$ is given by $Str_T([(\sigma_0, t_0), (\sigma_1, t_1), \ldots, (\sigma_{n-1}, t_{n-1})])$. An array type with $n$ elements of type $t$ is defined by $Arr_T(n, t)$.

$C0$ supports fully recursive data types. This makes modeling by abstract data structures a bit harder, since we have to introduce an additional level of indirection: pointers do not directly give the type, to which they map, but a *type name*. Type names are mapped to names via the type name environment. Thus, a pointer to a type name $tn$ is given by $Ptr_T(tn)$. The type $Null_T$ has only one element, namely the null pointer literal.

We can now define types formally by the abstract data type below:

$$\begin{array}{rcl} ty & = & Bool_T \mid Int_T \mid Unsgnd_T \mid Char_T \\ & \mid & Str_T((\Sigma^+ \times ty)\, list) \\ & \mid & Arr_T(\mathbb{N}, ty) \\ & \mid & Ptr_T(\Sigma^+) \mid Null_T \end{array}$$

Type name environments are defined by $tenv : (\Sigma^+ \times ty)\, list$.

**Definition 3.1 (Elementary Types)** A type $t$ is called *elementary* iff it is not a structure type or an array type:

$$?elem_T(t) = t \in \{Bool_T, Int_T, Unsgnd_T, Char_T, Ptr_T, Null_T\}$$

**Definition 3.2 (Abstract Size of Types)** The *abstract size* of a type $t$ is defined inductively:

$$size_T(t) = \begin{cases} 1 & \text{if } ?elem_T(t) \\ n \cdot size_T(t') & \text{if } t = Arr_T(n, t') \\ 0 & \text{if } t = Str_T([]) \\ size_T(t) + size_T(Str_T(clist)) & \text{if } t = Str_T((\sigma, t)\#clist) \end{cases}$$

### 3.2.2 Expressions and Complex Literals

The building blocks of $C0$ expressions are *literals* . We define a corresponding abstract data type *lit* providing a constructor for each basic $C0$ type plus a literal for null pointers:

$$lit = Boolean(\mathbb{B}) \mid Int(\mathbb{Z}) \mid Unsgnd(\mathbb{N}) \mid Char(\mathbb{Z}) \mid NullPointer$$

Expressions are defined inductively with two base cases:

- *literal expressions* $Lit(l)$ with a literal $l \in lit$ and
- *variable accesses* $Var(\sigma)$ to global or local variables with name $\sigma \in \Sigma^+$.

For expressions $e$, $e_1$, $e_2$, $i$, a name $\sigma$, and an operator $op$, the inductive cases of expressions are:

- *array access* $Arr(e, i)$,
- *structure access* $Str(e, \sigma)$,
- *addr-of computation* $AddrOf(e)$,
- *pointer dereferencing* $Deref(e)$,
- *arithmetic and logical computations* $UnOp(op, e)$, $BinOp(op, e_1, e_2)$, and $LazyBinOp(op, e_1, e_2)$.

The abstract data type, which we use for modeling $C0$ expressions in Isabelle, is defined as follows:

$$\begin{aligned} expr \quad = \quad & Lit(lit) \\ & \mid Var(\Sigma^+) \\ & \mid Arr(expr, expr) \\ & \mid Str(expr, \Sigma^+) \\ & \mid UnOp(unop, expr) \\ & \mid BinOp(binop, expr, expr) \\ & \mid LazyBinOp(lazybinop, expr, expr) \\ & \mid AddrOf(expr) \\ & \mid Deref(expr) \end{aligned}$$

The list of supported operators is given in Tables 3.4, 3.5, 3.6, and 3.7.

Table 3.4: Unary Operators

| Operator | Meaning |
|---|---|
| *minus* | unary minus |
| *neg* | bitwise negation |
| *not* | logical not |
| $to_{int}$ | conversion to *Int* |
| $to_{unsgnd}$ | conversion to *Unsgnd* |
| $to_{char}$ | conversion to *Char* |

Table 3.5: Binary Operators

| Operator | Meaning |
|---|---|
| *add* | addition |
| *sub* | subtraction |
| *mult* | multiplication |
| *div* | division |
| *mod* | modulo |
| $or_{\mathrm{b}}$ | bitwise or |
| $and_{\mathrm{b}}$ | bitwise and |
| $xor_{\mathrm{b}}$ | bitwise exclusive or |
| $shift_{\mathrm{l}}$ | arithmetic left shift |
| $shift_{\mathrm{r}}$ | arithmetic right shift |

Table 3.6: Comparison Operators

| Operator | Meaning |
|---|---|
| $comp_{\mathrm{less}}$ | less |
| $comp_{\mathrm{le}}$ | less or equal |
| $comp_{\mathrm{greater}}$ | greater |
| $comp_{\mathrm{ge}}$ | greater or equal |
| $comp_{\mathrm{eq}}$ | equal |
| $comp_{\mathrm{neq}}$ | not equal |

Table 3.7: Lazy Binary Operators

| Operator | Meaning |
|---|---|
| $and_{\mathrm{l}}$ | logical and |
| $or_{\mathrm{l}}$ | logical or |

#### Complex Literals

Complex literals are literals for aggregate types, i.e., structures and arrays:

$$\begin{aligned} lit_c \quad = \quad & clPrim(lit) \\ & \mid clArr(lit_c\ list) \\ & \mid clStr((\Sigma^+ \times lit_c)\ list) \end{aligned}$$

### 3.2.3 Statements

$$\begin{aligned} stmt \quad = \quad & Skip \\ & \mid Comp(stmt, stmt) \\ & \mid Ass(expr, expr, sid) \\ & \mid Ass_c(expr, lit_c, sid) \\ & \mid PAlloc(expr, \Sigma^+, sid) \\ & \mid SCall(\Sigma^+, \Sigma^+, expr\ list, sid) \\ & \mid Return(expr, sid) \\ & \mid Ifte(expr, stmt, stmt, sid) \\ & \mid Loop(expr, stmt, sid) \\ & \mid Asm(asm\ list, sid) \\ & \mid XCall(\Sigma^+, expr\ list, expr\ list, sid) \\ & \mid ESCall(\Sigma^+, \Sigma^+, expr\ list, sid) \end{aligned}$$

All statements but *Skip* and *Comp* are uniquely tagged with an identifier $sid \in \mathbb{N}$. These identifiers are mainly needed for the compiler correctness proof in [Lei08], thus we omit them most of time due to readability.

*Skip* is the empty statement. We can combine two statements $s_1$ and $s_2$ with the compound statement $Comp(s_1, s_2)$.

In Isabelle, we actually have only one assignment statement; to achieve a better readability, we use two versions of assignments: one for complex literals $Ass_c(e_{left}, l_c)$ and one for expressions $Ass(e_{left}, e_{right})$. We allocate new heap objects of a type with name $tn$ with the allocation statement $PAlloc(e_{left}, tn)$; the newly created pointer will be assigned to the left-hand expression $e_{left}$.

Calling a function with name $fn$ is done via the $SCall(fn, vn, [e_0, \ldots, e_{n-1}])$ statement, with $[e_0, \ldots, e_{n-1}]$ defining a—possibly empty—list of parameters and $vn$ being the variable name, where the return value will be stored. We return from a function call with the $Return(e)$ statement, $e$ being the return expression.

With the statement $Ifte(e, s_{then}, s_{else})$, we realize conditional program execution. If $e$ evaluates to *True*, execution continues with $s_{then}$, otherwise with $s_{else}$. Loop statements are modeled by $Loop(e, s_{lbody})$, where $s_{lbody}$ is executed as long as $e$ evaluates to *True*. Note that there is no special statement for *for* loops, since they can be syntactically transformed into semantically equivalent *while* loops during pre-processing.

The $Asm(u)$ statement allows to embed assembler code in $C0$ code, where $u$ is a list of VAMP assembler instructions (cf. Chapter 5). Note that the execution

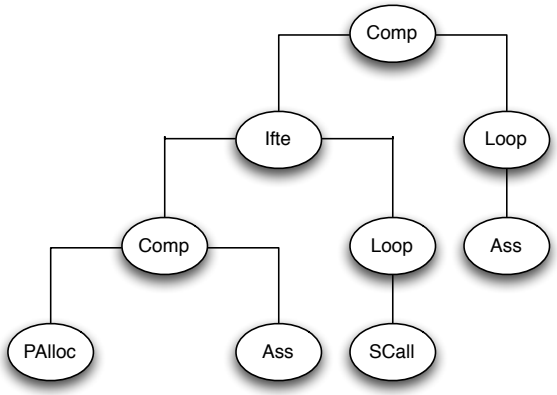


Figure 3.1: Statement Tree

of $u$ is defined by the VAMP assembler semantics, i.e., it is not covered by the $C0$ semantics.

Some $C0$ programs are not 'complete', e.g., the abstract kernel introduced in Sect. 6.5.1. These programs declare functions, but don't define them, i.e., don't specify a function body . We call these functions *external* and model calls to them with the $ESCall(fn, vn, [e_0, \ldots, e_{n-1}])$. In the process of *abstract linking*, these external function calls will be replaced as the corresponding function implementations are added (cf. Sect. 6.5.2).

To make the effects of assembler execution visible at higher levels, we define the $XCall(fn, [e_0, \ldots, e_{n-1}], [p_0, \ldots, p_{m-1}])$ statement : $p_i$ are parameters to the XCall, $e_i$ are the return values of it. In [AHL+09], XCalls are used in an integrated correctness proof for a Verisoft system stack.

**Definition 3.3 (Top-level Statement Flattening)** A statement tree (see Fig. 3.1 spanned by *Comp* statements can be flattened into a list of top-level statements by the function $s2l : stmt \to stmt$. We define:

$$s2l(s) = \begin{cases} s2l(s_1)@s2l(s_2) & \text{if } s = Comp(s_1, s_2) \\ s & \text{otherwise} \end{cases}$$

Correspondingly, we define $s2l_{noskip} : stmt \to stmt$, which additionally removes all *Skip* statements:

$$s2l_{noskip} = \begin{cases} s2l_{noskip}(s_1)@s2l_{noskip}(s_2) & \text{if } s = Comp(s_1, s_2) \\ [\,] & \text{if } s = Skip \\ s & \text{otherwise} \end{cases}$$

Note, that the flattening only affects the top-level statements, that is inner statements—like loop bodies—stay untouched (cf. Fig. 3.2).

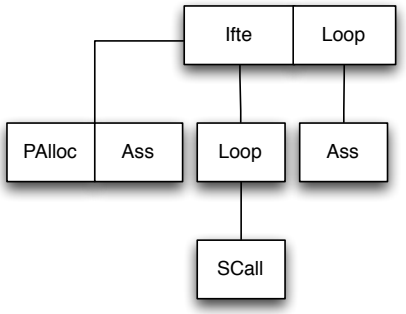


Figure 3.2: Flattened Statement Tree

### 3.2.4 Procedure Table

The procedure table of a $C0$ program contains the information of all of the program's functions. Each function is defined by one procedure descriptor, which is defined as a record $procT$ of four components:

- $p.body \in stmt$, the function body,
- $p.params \in (\Sigma^+ \times ty)\ list$, the parameters of the function given by pairs of names and types ,
- $p.rtype \in ty$, the return type, and
- $p.lvars \in (\Sigma^+ \times ty)\ list$, the local variables given by pairs of names and types.

Note, that for external functions the component *p.body* just contains a single *Skip* statement.

Formally, the procedure table is a list of pairs of function names and their corresponding descriptors and defined by the type $proctableT : (\Sigma^+ \times procT)\ list$.

## 3.3 $C0$ Small-Step Configurations

$C0$ small-step configurations keep track of the run-time information during the execution of a $C0$ program. They consist of two components:x

1. the *memory configuration* stores information about the global, local, and heap variables and their values;
2. the *program rest*, which holds the statements that haven't been executed yet.

### 3.3.1 Memory Configurations

Memory configurations store information about variables and the corresponding values. They are organized in *memory frames*: one frame for each the global variables and heap variables, and a list of frames and return destinations for the local variables .

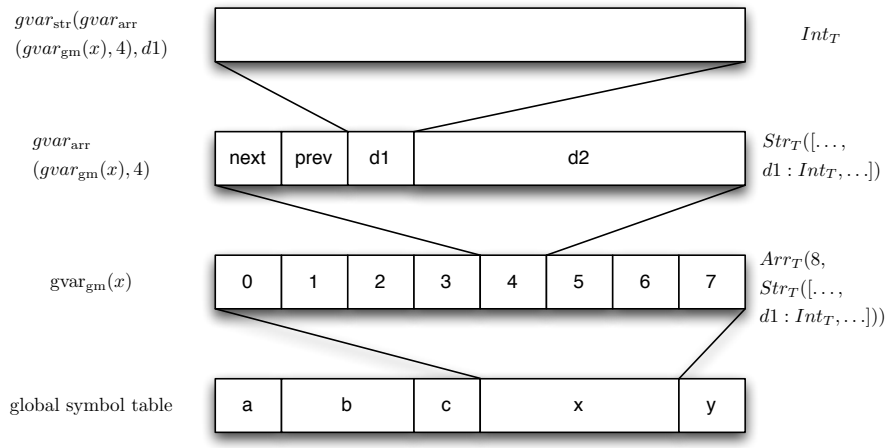


Figure 3.3: g-Variable Hierarchy

### Generalized Variables

Small-step semantics defines variables structurally as so-called *generalized variables* or *g-variables*. g-variables are defined inductively, with three base cases (global, local, and heap variables) and two inductive cases (structure and array access).

For $vn \in \Sigma^+$, $i, j \in \mathbb{N}$, $gvar_{gm}(vn)$ denotes the global variable with name $vn$, $gvar_{lm}(i, vn)$ the local variable with name $vn$ in the local memory frame $i$, and $gvar_{hm}(j)$ the heap variable with index $j$. Note that global and local variables are referenced by their name—and thus called *named variables*—while heap variables are *nameless* and referenced by their index only.

For a name $cn \in \Sigma^+$ and a g-variable $g$ of structure type, the component $gvar_{str}(g, cn)$ is a g-variable, too. This is also true for the array element $gvar_{arr}(g, i)$ of a g-variable $g$ of array type at index $i \in \mathbb{N}$.

The following abstract data type defines g-variables formally:

$$\begin{aligned} gvar \quad = \quad & gvar_{gm}(\Sigma^+) \\ & \mid gvar_{lm}(\mathbb{N}, \Sigma^+) \\ & \mid gvar_{hm}(\mathbb{N}) \\ & \mid gvar_{arr}(gvar, \mathbb{N}) \\ & \mid gvar_{str}(gvar, \Sigma^+) \end{aligned}$$

Obviously, this definition leads to a hierarchical, tree-like structure of g-variables (see Fig. 3.3). We will now define several functions that allow to handle this structure.

**Definition 3.4 (Sub g-Variables)** All g-variables in a sub-tree with a root $g$—including the root itself—are called *sub g-variables* of $g$. We define the set

of sub g-variables for a given $g$ inductively:

$$\frac{}{g \in sub_{\mathrm{g}}(g)}$$
Base Case

$$\frac{h \in sub_{\mathrm{g}}(g)}{gvar_{arr}(h,i) \in sub_{\mathrm{g}}(g)} \qquad \frac{h \in sub_{\mathrm{g}}(g)}{gvar_{str}(h,cn) \in sub_{\mathrm{g}}(g)}$$
Inductive Cases

**Definition 3.5 (Parent g-Variable)** Correspondingly, we define the function $idparent_g : gvar \to gvar_\perp$ to walk up in the hierarchy of g-variables:

$$parent_g(g) = \begin{cases} h & \text{if } g = gvar_{arr}(h,i) \\ h & \text{if } g = gvar_{str}(h,cn) \\ None & \text{otherwise} \end{cases}$$

**Definition 3.6 (Root g-Variable)** Obviously, application of $parent_g$ to a g-variable $g$ several times would lead us to the first ancestor—the root of the tree— of $g$. We call this variable the *root g-variable* of $g$ and define the function $root_g : gvar \to gvar$ that calculates it formally as follows:

$$root_g(g) = \begin{cases} root_g(h) & \text{if } g = gvar_{arr}(h,i) \\ root_g(h) & \text{if } g = gvar_{str}(h,cn) \\ g & \text{otherwise} \end{cases}$$

### Memory Frames

The $C0$ small-step semantics uses a flat and rather explicit memory model, similar to the one in [Nor98], defined by a mapping of of addresses—natural numbers—to memory cells. In order to store the value of a variable of type $t$, we need $size_T(t)$ many memory cells. This means that the value of an elementary type variable uses exactly one memory cell. Aggregate type values are stored in consecutive sequences of memory cells.

Memory cells are formalized by an abstract data type, where we define a constructor for each of the five elementary types:

$$mcellT = Int(\mathbb{Z}) \mid Unsgnd(\mathbb{N}) \mid Char(\mathbb{Z}) \mid Bool(\mathbb{B}) \mid Ptr(gvar \cup \{NullPointer\})$$

Note that pointers are represented by a g-variable or the special null pointer value $NullPointer$.

In addition to the content of a memory frame, we need to know which variables have already been initialized. We store this information in a list of variable names.

Last but not least, each memory frame has a list of its variables—given by their names—along with their types. Such lists are called *symbol tables*.

So, each memory frame is specified by a record $mf$ of type $mframeT$ with the above three components:

- $mf.cont : \mathbb{N} \to mcellT$, the *content* of the memory frame ,
- $mf.init : \Sigma^+\ list$, the list of *initialized variables*, and
- $mf.st : (\Sigma \times ty)list$, the *symbol table* of the frame.

**Definition 3.7 (Base Addresses of Variables)** The base address of a variable is defined recursively over its position in the symbol table. If the variable is not present in the symbol table or if the symbol table is empty, the base address is *None*. In all other cases, we compute recursively: if the variable is the first entry in the symbol table, then the base address is zero. Otherwise, we set its base address to the base address in the tail of the list plus the abstract size of the type at the first entry.

We define $ba_v : (\Sigma^+ \times ty)\ list \times \Sigma^+ \to \mathbb{N}_\perp$ and set:

$$\begin{aligned} ba_v([], \sigma) &= None \\ ba_v((vn, t)\#xs, \sigma) &= \begin{cases} \lfloor 0 \rfloor & \text{if } \sigma = vn \\ size_T(t) \oplus ba_v(xs, \sigma) & \text{otherwise} \end{cases} \end{aligned}$$

with

$$i \oplus j = \begin{cases} the(i) + the(j) & \text{if } i \neq None \text{ and } j \neq None \\ None & \text{otherwise} \end{cases}$$

**Definition 3.8 (Type of Variables)** The type of a variable in a symbol table is defined by the second component of the corresponding pair in that symbol table. Formally, we define $type_v : (\Sigma^+ \times ty)\ list \times \Sigma^+ \to ty_\perp$ with

$$type_v(st, x) = mapof(st, x)$$

### Memory Configuration

As described in the introduction above, a memory configuration consists of memory frames. We define a configuration $m$ formally by a record $memconfT$ with components:

- $m.gm \in mframeT$, the global variables' memory frame.
- $m.lm : (mframeT \times gvar)$ list, the stack of local memories. Each memory frame stands for one stack frame. The second component, a g-variable, defines, where the return value of the function associated with that frame will be stored (*return destination*).
- $m.hm \in mframeT$, the heap variables' memory frame. Since heap variables are nameless, the names in the symbol table $m.hm.st$ are not used. Heap variables are initialized by definition, thus the field $m.hm.init$ is ignored.

Whenever we extend the stack, we put a new memory frame at the *end* of the existing list $m.lm$. This means, the current stack frame is the last stack frame in the list, which we abbreviate by $toplm(m) = m.lm!(|\ lm\ | -1)$. Thus, we can keep the index of stack frames and the index of local variables (see Sect. 3.1) invariant during program execution.

**Symbol Configuration**

We introduce some auxiliary functions to refer to the symbol tables of the different memory frames in a readable way:

$$\begin{aligned} gst &: memconfT \to (\Sigma^+ \times ty)\ list \\ toplst &: memconfT \to (\Sigma^+ \times ty)\ list \\ hst &: memconfT \to (\Sigma \times ty)\ list \end{aligned}$$

$$\begin{aligned} gst(m) &= m.gm.st \\ toplst(m) &= toplm(m).st \\ hst(m) &= m.hm.st \end{aligned}$$

Often, we are just interested in the symbol tables of a certain memory configuration, but not its content. To improve the formalism in these sections, we define a new record type named $symbolconfT$, which will store a *symbol configuration*, consisting of the three symbol tables . For $sc \in symbolconfT$ we define:

- $sc.gst : (\Sigma^+ \times ty)\ list$, the symbol table of the global memory frame,
- $sc.lst : (\Sigma^+ \times ty)\ list\ list$, a list of symbol tables of all local memory frames, and
- $sc.hst : (\Sigma \times ty)\ list$, the heap memory frame symbol table.

For a given memory configuration $m$, we write short $sc(m)$ to refer to its symbol configuration.

### 3.3.2 Program Rest

The program rest stores those statements that haven't been executed yet. On initialization, we start with the body of the main function of the $C0$ program. Afterward, the program rest grows or shrinks as defined through the execution of the program. Formally, a program rest $pr$ is just a common $C0$ statement: $pr \in stmt$.

### 3.3.3 C0 Configuration

We can now give a complete formal definition of a configuration $c$ as used in $C0$ small-step semantics. Therefore, we introduce a new record type $confT_{C0}$, which consists of two components

- $c.m \in memconfT$, the memory configuration, and
- $c.pr \in stmt$, the program rest.

## 3.4 Expression Evaluation

### 3.4.1 Address of g-Variables

**Definition 3.9 (Type of g-Variables)** Similar to Def. 3.8, we define the type of g-variables $type_g : symbolconfT \times gvar \to ty$. For $\sigma \in \Sigma^+$ and $i, j \in \mathbb{N}$ we set the base cases:

$$\begin{aligned} type_g(sc, gvar_{gm}(\sigma)) &= type_v(sc.gst, \sigma) \\ type_g(sc, gvar_{lm}(i, \sigma)) &= type_v(sc.lst!i, \sigma) \\ type_g(sc, gvar_{hm}(j)) &= type_v(sc.hst, j) \end{aligned}$$

The inductive cases, i.e., array and structure accesses, are defined partially as follows:

$$type_g(sc, gvar_{arr}(g, i)) = \begin{cases} t & \text{if } type_g(g) = Arr_T(t, j) \\ undef & \text{otherwise} \end{cases}$$

$$type_g(sc, gvar_{str}(g, x)) = \begin{cases} the(mapof(c, x)) & \text{if } type_g(g) = Str_T(c) \\ undef & \text{otherwise} \end{cases}$$

**Definition 3.10 (Named and Nameless g-Variables)** To determine, if a variable is in global, local, or heap memory, we introduce the function $mem_g : gvar \to memname$:

$$\begin{aligned} mem_g(gvar_{gm}(\sigma)) &= gm \\ mem_g(gvar_{lm}(i, \sigma)) &= lm(i) \\ mem_g(gvar_{hm}(i)) &= hm \\ mem_g(gvar_{arr}(g, i)) &= mem_g(g) \\ mem_g(gvar_{str}(g, \sigma)) &= mem_g(g) \end{aligned}$$

We distinguish between *nameless* variables, i.e., heap variables, and *named* variables . Thus, we define a predicate $?named_g : gvar \to \mathbb{B}$ formally:

$$\frac{mem_g(g) = gm \vee (\exists i \in \mathbb{N} : mem_g(g) = lm(i))}{?named_g(g)}$$

**Definition 3.11 (Initialized g-Variables)** We say, a g-variable $x$ is *initialized* in a memory configuration $m$, if its root g-variable is in the set of initialized variables of the corresponding memory frame. Heap variables are by definition always initialized.

We set for $\sigma \in \Sigma^+$, $i \in \mathbb{N}$ and $g \in gvar$:

$$\begin{aligned} ?init_g(m, gvar_{gm}(\sigma)) &= \sigma \in m.gm.init \\ ?init_g(m, gvar_{lm}(i, \sigma)) &= \sigma \in m.lm!i.init \\ ?init_g(m, gvar_{hm}(i)) &= True \\ ?init_g(m, gvar_{arr}(g, i)) &= ?init_g(m, g) \\ ?init_g(m, gvar_{str}(g, \sigma)) &= ?init_g(m, g) \end{aligned}$$

**Definition 3.12 (Base Address of g-Variables)** The *base address* of a g-variable is defined by the function $ba_g : symbolconfT \times gvar \to \mathbb{N}$. For named g-variables, we set:

$$\begin{aligned} ba_g(sc, gvar_{gm}(\sigma)) &= ba_v(sc.gst, \sigma) \\ ba_g(sc, gvar_{lm}(i, \sigma)) &= ba_v(sc.lst!i, \sigma) \end{aligned}$$

Unfortunately, $ba_v$ cannot be used with nameless variables. In order to determine the base address of a heap variable with index $i$ in a symbol table $sc.hst$, we sum up the abstract sizes of the types with index less than $i$:

$$ba_g(sc, gvar_{hm}(i)) = \begin{cases} 0 & \text{if } i = 0 \\ \sum_{j=0}^{i-1} size_T(snd(sc.hst!j)) & \text{otherwise} \end{cases}$$

For array accesses, we define partially:

$$ba_g(sc, gvar_{arr}(g, i)) = \begin{cases} ba_g(g) + i \cdot size_T(t) & \text{if } type_g(g) = Arr_T(t, j) \\ undef & \text{otherwise} \end{cases}$$

Dealing with structure accesses is more convenient, since the list of struct components is of the same type as a symbol table: $(\Sigma^+ \times ty)\, list$. Thus we can use $ba_v$ again in order to compute the offset of a given component $\sigma$ *within* a component list $cl$:

$$ba_g(sc, gvar_{str}(g, \sigma)) = \begin{cases} ba_g(sc, g) + the(ba_v(cl, \sigma)) & \text{if } type_g(g) = Str_T(cl) \\ undef & \text{otherwise} \end{cases}$$

### 3.4.2 Value of a g-Variable

Let's first define a function that allows to read memory cells, the building blocks of the content of a memory frame.

**Definition 3.13 (Memory Content)** For a given memory configuration $m$, a memory name $\sigma$, and $i, j \in \mathbb{N}$, we define the content of the memory in the range from $i$ to $j$ as follows:

$$m_\sigma[i : j] = \begin{cases} m.gm.ct[i : j] & \text{if } \sigma = gm \\ m.lm!i.ct[i : j] & \text{if } \sigma = lm(i) \\ m.hm.ct[i : j] & \text{if } \sigma = hm \end{cases}$$

**Definition 3.14 (Value of a g-Variable)** We can now use the above definition in order to describe formally the *value* of a g-variable $g \in gvar$ in a memory configuration $m \in memconfT$:

$$value_g(m, g) = m_{mem_g(g)}[ba_g(sc(m), g), size_T(type_g(sc(m), g))]$$

### 3.4.3 Evaluating Expressions

In this subsection, we will deal with the actual functions for expression evaluation in C0 small-step semantics. We will only consider those expressions being of

relevance for the correctness proof in Chapter 8. This means, we won't look into the details of literal or operator evaluation, but which are covered in [Lei08].

In Isabelle, the evaluation of an expression returns either *None* or a so-called *data slice* data type. Here, we split up the monolithic Isabelle *eval* function into five functions, each returning one of the components of the data slice data type:

- $lval : tenv \times memconfT \times expr \to gvar_\bot$ computes the left hand value of a given expression, that is its address. For expressions with no address, like literals, *lval* returns None. This corresponds to the *lval* component of a data slice.
- $rval : tenv \times memconfT \times expr \to (\mathbb{N} \to mcellT)_\bot$ computes the right hand value of a given expression. For uninitialized expressions, *rval* returns None. This corresponds to the *data* component of a data slice.
- $type : tenv \times (\Sigma \times ty)\ list \times (\Sigma \times ty)\ list \times expr \to ty_\bot$ returns the type of an expression. Assuming a type correct memory, this corresponds to the *type* component.
- $?init : tenv \times memconfT \times expr \to \mathbb{B}$ is *True* for initialized expressions, otherwise *False*. *initialized* is the corresponding component in data slices.
- $?inter : tenv \times (\Sigma \times ty)\ list \times (\Sigma \times ty)\ list \times expr \to \mathbb{B}$ is *True*, if the expression is *intermediate*—i.e., not a memory object. Data slices have a corresponding component named *intermediate*.

Note that not all of the above functions require a memory configuration as input. Instead, *type* and *init* take two symbol tables, one for the global memory, one for the local memory (usually the latter one is the symbol table of the topmost stack frame).

### Literals

We start with evaluating literal expressions. Regarding initialization, left value and memory status, we define as follows:

$$\begin{aligned} ?init(te, m, Lit(l)) &= True \\ ?inter(te, gst, lst, Lit(l)) &= True \\ lval(te, m, Lit(l))) &= undef\,. \end{aligned}$$

We do not define these functions for complex literals.

Regarding type of literals, we define two functions.

**Definition 3.15 (Type of Literals)** The type of a literal is given through the function $type_{\text{lit}} : lit \to ty$. We set:

$$\begin{aligned} type_{\text{lit}}(Bool(b)) &= Bool_T \\ type_{\text{lit}}(Unsgnd(u)) &= Unsgnd_T \\ type_{\text{lit}}(Int(i)) &= Int_T \\ type_{\text{lit}}(Char(c)) &= Char_T \\ type_{\text{lit}}(NullPointer) &= Null_T \end{aligned}$$

**Definition 3.16 (Type of Complex Literals)** The type of a complex literal is given through the function $type_{\text{clit}} : lit_c \to ty$. We set:

$$\begin{aligned} type_{\text{clit}}(clPrim(l)) &= type_{\text{lit}}(l) \\ type_{\text{clit}}(clArr(x\#xs)) &= Arr_T(|x\#xs|, type_{\text{clit}}(x)) \\ type_{\text{clit}}(clStr(xs)) &= Str_T(map(type_{\text{clit}}, map(snd, xs))) \end{aligned}$$

**Definition 3.17 (Right Value of Literals)** We define a function $rval_{\text{lit}} : lit \to mcellT$, which takes a literal and returns its right value. We set:

$$\begin{aligned} rval_{\text{lit}}(Bool(b)) &= Bool(b) \\ rval_{\text{lit}}(Unsgnd(u)) &= Nat(u) \\ rval_{\text{lit}}(Int(i)) &= Int(i) \\ rval_{\text{lit}}(Char(c)) &= Char(c) \\ rval_{\text{lit}}(NullPointer) &= Ptr(NullPointer) \end{aligned}$$

**Definition 3.18 (Right Value of Complex Literals)** We define the function $rval_{\text{clit}} : lit_c \to mcellT\ list$ to obtain the right value of complex literals. We set:

$$\begin{aligned} rval_{\text{clit}}(clPrim(l)) &= rval_{\text{lit}}(l) \\ rval_{\text{clit}}(clArr([\,])) &= [\,] \\ rval_{\text{clit}}(clArr(x\#xs)) &= rval_{\text{clit}}(x)@rval_{\text{clit}}(clArr(xs)) \\ rval_{\text{clit}}(clStr([\,])) &= [\,] \\ rval_{\text{clit}}(clStr([x\#xs])) &= rval_{\text{clit}}(snd(x))@rval_{\text{clit}}(clStr(xs)) \end{aligned}$$

Using the above definitions, we can now define the type and value of literal expressions as follows:

$$\begin{aligned} type(te, gst, lst, lit(l)) &= \lfloor type_{\text{lit}}(l) \rfloor \\ rval(te, m, lit(l)) &= \lfloor rval_{\text{lit}}(l) \rfloor. \end{aligned}$$

### Variable Access

Accessing a variable $\sigma$ means accessing a memory object, thus

$$?inter(te, gst, lst, Var(\sigma)) = False.$$

$C0$ supports *shadowing*. This means that if both *gst* and *lst* define a variable with name $\sigma$, we will access the local one. If there is no local variable of that name, we access the global one. If the variable $\sigma$ is neither in *gst* nor in *lst*, the evaluation fails. In this case, we set:

$$\begin{aligned} ?init(te, m, Var(\sigma)) &= False \\ type(te, gst, lst, Var(\sigma)) &= None \\ rval(te, m, Var(\sigma)) &= None \\ lval(te, m, Var(\sigma)) &= None \end{aligned}$$

In the other cases, that is $\sigma$ is either in *gst* or *lst*, we set:

$$type(te, gst, lst, Var(\sigma)) = \begin{cases} type_v(lst, \sigma) & \text{if } \sigma \in \{map(fst, lst)\} \\ type_v(gst, \sigma) & \text{otherwise} \end{cases}$$

for the type and

$$lval(te, mc, Var(\sigma)) = \begin{cases} \lfloor gvar_{lm}(\mid m.lm \mid -1, \sigma) \rfloor & \text{if } \sigma \in map(fst, toplst(m)) \\ \lfloor gvar_{gm}(\sigma) \rfloor & \text{otherwise} \end{cases}$$

for the left hand value.

Furthermore, we define the to $\sigma$ corresponding memory frame *mf* as

$$mf = \begin{cases} toplm(m) & \text{if } \sigma \in \{map(fst, toplst(m))\} \\ m.gm & \text{otherwise} \end{cases}$$

This allows finally to define the two remaining evaluation functions in a comprehensive way:

$$\begin{aligned} ?init(te, m, Var(\sigma)) &= \sigma \in mf.init \\ rval(te, m, Var(\sigma)) &= \lfloor mf.ct[ba_v(mf.st, \sigma), size_T(type_v(mf.st, \sigma))] \rfloor \end{aligned}$$

**Pointer Dereferencing**

Pointer dereferencing again means dealing with a memory object, i.e.,

$$?inter(te, gst, lst, Deref(e)) = False.$$

The evaluation of a $Deref(e)$ expression will fail, if certain constraints are violated. These are:

- the sub expression $e$ has to be of pointer type: $type(te, gst, lst, e) = \lfloor Ptr_T(tn) \rfloor$,
- the type name $tn$ has to be defined in the type name environment: $mapof(te, tn) = \lfloor t \rfloor$;
- the right hand value of $e$ has to be a pointer, but not the null pointer: $rval(te, m, e) = \lfloor Ptr(p) \rfloor \wedge p \neq NullPointer$,
- $p$ must not be a local variable, that is $mem_g(p) \in \{gm, hm\}$, and last but not least,
- $e$ has to be initialized: $?init(te, m, e) = True$.

In the case of failure, we set $?init$ to *False* and *lval*, *rval*, and *type* to *None*. Otherwise, we set the remaining four evaluation functions as follows:

$$\begin{aligned} type(te, gst, lst, Deref(e)) &= mapof(te, tn) \\ ?init(te, m, Deref(e)) &= ?init_g(m, p) \\ lval(te, m, Deref(e)) &= \lfloor p \rfloor \\ rval(te, m, Deref(e)) &= \lfloor value_g(m, p) \rfloor \end{aligned}$$

### Address-Of Operator

The Address-Of operator $\mathit{AddrOf}(e)$ returns the address of a memory object, which itself is not a memory object, but always initialized:

$$?\mathit{inter}(te, \mathit{gst}, \mathit{lst}, \mathit{AddrOf}(e)) = \mathit{True}$$
$$?\mathit{init}(te, m, \mathit{AddrOf}(e)) = \mathit{True}$$

As with pointer dereferencing, applying the Address-Of operator can fail, if one of the following requirements is not met:

- $e$ has to be a memory object: $?\mathit{inter}(te, \mathit{gst}, \mathit{lst}, e) = \mathit{False}$, and
- its left evaluation has to return a g-variable: $\mathit{lval}(te, m, e) = \lfloor g \rfloor$, and finally
- there has to be a type name for the type of $e$ in the type name environment: $\mathit{mapof}(\mathit{map}(\mathit{flip}, te), \mathit{type}(te, \mathit{gst}, \mathit{lst}, e)) = \lfloor tn \rfloor$.

If any of these requirements are not met, we set *type*, *lval*, and *rval* to $None$.

Otherwise, we define:

$$\begin{aligned} \mathit{type}(te, \mathit{gst}, \mathit{lst}, \mathit{AddrOf}(e)) &= \lfloor \mathit{Ptr}_T(tn) \rfloor \\ \mathit{lval}(te, m, \mathit{AddrOf}(e)) &= \mathit{undef} \\ \mathit{rval}(te, m, \mathit{AddrOf}(e)) &= \lfloor [\mathit{Ptr}(g)] \rfloor \end{aligned}$$

### Array Element Access

The expression for array element access $\mathit{Arr}(e_a, e_i)$ has two sub expressions: $e_a$ for the array expression and $e_i$ for the index expression. An array element access is a memory object, iff the array expression is a memory object, that is

$$?\mathit{inter}(te, \mathit{gst}, \mathit{lst}, \mathit{Arr}(e_a, e_i)) = ?\mathit{inter}(te, \mathit{gst}, \mathit{lst}, e_a),$$

and it is initialized, iff both sub expressions are initialized:

$$?\mathit{init}(te, m, \mathit{Arr}(e_a, e_i)) = ?\mathit{init}(te, m, e_a) \wedge ?\mathit{init}(te, m, e_i).$$

Furthermore, we require

- the array expression can be evaluated, which is $\mathit{lval}(te, m, e_a) = \lfloor g_a \rfloor$ and $\mathit{rval}(te, m, e_a) = \lfloor v_a \rfloor$;
- the right evaluation of the index expression returns a numerical value, i.e.,

$$\begin{aligned} \mathit{rval}(te, m, e_i) &= \lfloor \mathit{Int}(i) \rfloor \\ \vee\, \mathit{rval}(te, m, e_i) &= \lfloor \mathit{Unsgnd}(i) \rfloor \\ \vee\, \mathit{rval}(te, m, e_i) &= \lfloor \mathit{Char}(i) \rfloor \end{aligned}$$

- that $e_a$ is an array with $n$ elements of type $t$, i.e., $\mathit{type}(te, \mathit{gst}, \mathit{lst}, e_a) = \lfloor \mathit{Arr}_T(n, t) \rfloor$, where

- the index $i$ is in range: $i < n$.

If the evaluation of the array element access fails, since one of the above criteria is not fulfilled, we set *type*, *lval* and *rval* to $None$.

In all other cases we define

$$\begin{aligned} type(te, gst, lst, Arr(e_a, e_i)) &= \lfloor t \rfloor \\ lval(te, m, Arr(e_a, e_i)) &= \lfloor gvar_{arr}(g_a, i) \rfloor \\ rval(te, m, Arr(e_a, e_i)) &= \lfloor v_a[i \cdot size_T(t), size_T(t)] \rfloor \end{aligned}$$

### Structure Component Access

Similar to array element accesses, a structure component access $Str(e, cn)$ consists of two parts: a sub expression for the structure access $e$ and a component name $cn$. A structure component access' initialization and memory object status is the same as those of the structure access, thus

$$\begin{aligned} ?inter(te, gst, lst, Str(e, cn)) &= ?inter(te, gst, lst, e) \\ ?init(te, m, Str(e, cn)) &= ?init(te, m, e) \end{aligned}$$

The evaluation of $Str(e, cn)$ fails, if one or more of the following constraints are violated:

- $e$ has to be of structure type, i.e., $type(te, gst, lst, e) = \lfloor Str_T(cl) \rfloor$, and
- $e$ can be evaluated in the sense that $lval(te, m, e) = \lfloor g \rfloor$ and $rval(te, m, e) = \lfloor v \rfloor$;
- there is a $t$ such that $(cn, t)$ is in the component list $cl$ of the struct, that is $mapof(cl, cn) = \lfloor t \rfloor$.

In the case of failure, we set *lval*, *rval* and *type* to $None$.

Otherwise we define

$$\begin{aligned} type(te, gst, lst, Str(e, cn)) &= \lfloor t \rfloor \\ lval(te, m, Str(e, cn)) &= \lfloor gvar_{str}(g, cn) \rfloor \\ rval(te, m, Str(e, cn)) &= \lfloor v[ba_v(cl, cn), size_T(t)] \rfloor \end{aligned}$$

### Operators

As mentioned in the introduction to this subsection, we will not fully work out the details of operator evaluation. More precisely, we do not explicitly take care of how the right hand value and the result type are obtained—we will use two uninterpreted functions for this purpose. The details of operator evaluation are covered in [Lei08].

For unary operators, we define

$$\begin{aligned} type_{\oplus_1} &: unop \times ty \to ty \\ value_{\oplus_1} &: unop \times mcellT_\perp \to mcellT_\perp \end{aligned}$$

and correspondingly for binary operators

$$\begin{aligned} type_{\oplus_2} &: binop \times ty \times ty \to ty \\ value_{\oplus_2} &: binop \times mcellT_\perp \times mcellT_\perp \to mcellT_\perp \end{aligned}$$

**Unary Operators** A unary operator expression is never a memory object, thus we set

$$?inter(te, gst, lst, UnOp(\oplus_1, e)) = True,$$

and it is initialized, iff the sub expression $e$ is initialized:

$$?init(te, m, UnOp(\oplus_1, e)) = ?init(te, m, e).$$

Successful operator application requires the following conditions to be fulfilled:

- the sub expression $e$ can be evaluated, that is $rval(te, m, e) = \lfloor v_e \rfloor$, and
- applying the unary operator to $v_e$ returns some result: $value_{\oplus_1}(\oplus_1, v_e) = \lfloor v \rfloor$.

If one or more of these requirements are not met, we set *type*, *lval*, and *rval* to *None*. Else, we set

$$\begin{aligned} type(te, gst, lst, UnOp(\oplus_1, e)) &= \lfloor type_{\oplus_1}(\oplus_1, the(type(te, gst, lst, e))) \rfloor \\ lval(te, m, UnOp(\oplus_1, e)) &= undef \\ rval(te, m, UnOp(\oplus_1, e)) &= \lfloor [v] \rfloor \end{aligned}$$

**Binary and Lazy Operators** Similarly, we proceed with binary and lazy operators. For the rest of this paragraph, we set $e = BinOp(\oplus_2, e_1, e_2)$ or $e = LazyBinOp(\oplus_2, e_1, e_2)$. As with unary operators, binary and lazy operator expressions are never a memory object, that is

$$?inter(te, gst, lst, e) = True,$$

and they are initialized, iff both sub expressions are initialized:

$$?init(te, m, e) = ?init(te, m, e_1) \wedge ?init(te, m, e_2).$$

Here, the requirements for successful expression evaluation are:

- both sub expressions can be right evaluated, i.e., $rval(te, m, e_1) = \lfloor v_1 \rfloor$ and $rval(te, m, e_2) = \lfloor v_2 \rfloor$, and
- application of the operator yields some result: $value_{\oplus_2}(\oplus_2, v_1, v_2) = \lfloor v \rfloor$, and finally
- we can determine some type both for $e_1$ and $e_2$:

$$\begin{aligned} type(te, gst, lst, e_1) &= \lfloor t_1 \rfloor \\ type(te, gst, lst, e_2) &= \lfloor t_2 \rfloor \end{aligned}$$

If expression evaluation fails, we set *lval*, *rval*, and *type* to *None*. In all other cases, we set:

$$\begin{aligned} type(te, gst, lst, e) &= \lfloor type_{\oplus_2}(\oplus_2, e_1, e_2) \rfloor \\ lval(te, m, e) &= undef \\ rval(te, m, e) &= \lfloor [v] \rfloor \end{aligned}$$

## 3.5 Execution of $C0$ Programs

In this section we will treat the execution of $C0$ programs using the small-steps semantics. We first start with the definition of initial configurations (cf. Sect. 3.5.1).

We proceed with a formal definition of memory updates in Sect. 3.5.2, before we conclude with the definition of the $C0$ transition function (cf. Sect. 3.5.3)

### 3.5.1 Initial Configuration

As we have learned in Sect. 3.3.3, a $C0$ configuration consists of two components: the memory configuration and the program rest. In this section, we will define, how these two components are initialized.

#### Initial Memory

Global and heap variables in $C0$ are initialized with a default value depending on their type. Then, we use these initial values to construct the initial memory.

**Definition 3.19 (Initial Values)** Let $init_{val} : ty \rightarrow mcellT\ list$ define the initial value for a given type. Let $tn$ denote a type name. For the base case we set:

$$\begin{aligned} init_{\text{val}}(Int_T) &= [Int(0)] \\ init_{\text{val}}(Unsgnd_T) &= [Unsgnd(0)] \\ init_{\text{val}}(Bool_T) &= [Bool(False)] \\ init_{\text{val}}(Char_T) &= [Char(0)] \\ init_{\text{val}}(Ptr_T(tn)) &= [Ptr(NullPointer)] \\ init_{\text{val}}(Null_T) &= [Ptr(NullPointer)] \end{aligned}$$

Given a type $t$, a natural number $n$, and a component name $cn$, we define for the inductive case:

$$\begin{aligned} init_{\text{val}}(Arr_T(n,t)) &= replicate(n, init_{\text{val}}(t)) \\ init_{\text{val}}(Str_T[\,]) &= [\,] \\ init_{\text{val}}(Str_T((cn,t)\#xs)) &= init_{\text{val}}(t)@init_{\text{val}}(Str_T(xs)) \end{aligned}$$

We can now use the function $init_{\text{val}}$ to define recursively $init_{st}$, which takes a whole symbol table as input and returns a corresponding initialized content for it:

$$\begin{aligned} init_{st}([\,]) &= [\,] \\ init_{st}((vn,t)\#xs) &= init_{\text{val}}(t)@init_{st}(xs) \end{aligned}$$

**Definition 3.20 (Initial Memory Frame)** Given a symbol table $st$, we define an initial memory frame as follows:

$$init_{mem}(st) = \begin{bmatrix} ct = undef \\ st = st \\ init = \emptyset \end{bmatrix}$$

**Definition 3.21** Now, we take this wrapper together with the memory content generator from Def. 3.19 to initialize all variables of a memory frame. Therefore, we define $init_{\text{vars}} : mframeT \to mframeT$ and set for a given memory frame $frame$:

$$init_{\text{vars}}(frame) \quad = \quad \begin{bmatrix} ct = \lambda i.init_{st}(frame.st)!i \\ init = \{map(fst, frame.st)\} \end{bmatrix}$$

We have now all the ingredients to define the initial memory of a $C0$ machine. For the global memory frame, we take the global variable's symbol table to construct the initial content. Since there haven't been any objects allocated yet, the heap is empty.

The situation for the stack is a little trickier: we search the procedure table for the main function of the program. Then we take its parameters and local variables and use them to construct the initial stack frame of our initial $C0$ configuration.

**Definition 3.22 (Initial Memory Configuration)** Let $pt$ be a procedure table and $gst$ a symbol table for the global variables. An *initial memory configuration* of a $C0$ program is given through the function $init_{mc} : proctableT \times (\Sigma^+ \times ty)\ list \to memconfT_\perp$.

If $mapof(pt, \texttt{`main'}) = None$, i.e., there is no main function in the program, we set $init_m(pt, gst) = None$. Otherwise, denote with $mp$ the main function of the program, which is $mapof(pt, \texttt{`main'}) = \lfloor mp \rfloor$. Then we set

$$init_m(pt, gst) \quad = \quad \left\lfloor \begin{bmatrix} gm = init_{\text{vars}}(init_{mem}(gst)) \\ lm = [(init_{mem}(mp.params@mp.lvars), undef)] \\ hm = [init_{mem}([])] \end{bmatrix} \right\rfloor$$

There is only the second component of an initial $C0$ machine configuration missing, the initial program rest. This rest is defined by the body of the program's main function after removing the last statement of it. This is in fact a *Return* statement and would be the last statement of the program to be executed. Unfortunately, this would leave the stack in an undefined state. For this reason, termination of a $C0$ program is defined in a different way. We will have a more precise look at this in Sect. 3.5.3.

**Definition 3.23 (Removing the Last Statement)** We define the function $remlast : stmt \to stmt$, which replaces the last statement in a tree with *Skip*. For an input statement $s$, we define recursively:

$$remlast(s) = \begin{cases} Comp(s_1, remlast(s_2)) & \text{if } s = Comp(s_1, s_2) \\ Skip & \text{otherwise} \end{cases}$$

**Definition 3.24 (Initial $C0$ Configuration)** We will now define the function $init_{\text{conf}} : proctableT \times (\Sigma^+ \times ty)\ list \to c_{C0\perp}$. If there is no main function in the corresponding $C0$ program, that is $mapof(pt, \texttt{`main'}) = None$, we return $init_{\text{conf}}(pt, gst) = None$. Else, let $mp$ denote the main procedure of the program: $mp = the(mapof(pt, \texttt{`main'}))$.

$$init_{\text{conf}}(pt, gst) = \left\lfloor \begin{bmatrix} mem = the(init_{mem}(pt, gst)) \\ prog = remlast(mp.body) \end{bmatrix} \right\rfloor$$

### 3.5.2 Memory Updates

Statements like *SCall*, *PAlloc*, *Return*, and *Ass* change the memory configuration of a $C0$ machine. In this section, we are going to define a function $upd_{mm} : memconfT \times gvar \times (\mathbb{N} \to mcellT) \to memconfT_\perp$ for memory updates. In Sect. 3.5.3, we will use this function to describe the semantics of the statements mentioned above.

For the rest of this section, let $m$ denote a memory configuration, $g$ a g-variable, and $v$ a value. In $C0$, partial updates of uninitialized variables are forbidden. This is why we require that the variable to be updated is either initialized or a root g-variable.

If this is not the case, the memory update returns *None*:

$$\frac{\neg(?init(m,g) \vee root_g(g) = g)}{upd_{mm}(m,g,v) = None}$$

Otherwise, we define the memory update semantics by a case distinction on the root of $g$. Let $b = ba_g(sc(m), g)$ denote the base address of $g$, and $s = size_T(type_g(sc(m), g))$ its size.

For a global root g-variable—$root_g(g) = gvar_{gm}(\sigma)$—we set

$$upd_{mm}(m,g,v) := \left\lfloor m \left[ \begin{array}{l} gm.init := gm.init \cup \sigma \\ gm.ct := gm.ct([b,s] := v[0,s]) \end{array} \right] \right\rfloor$$

For a local root g-variable—$root_g(g) = gvar_{lm}(i,\sigma)$—we set

$$upd_{mm}(m,g,v) := \left\lfloor m \left[ \begin{array}{l} lm!i.init := lm!i.init \cup \sigma \\ lm!i.ct := lm!i.ct([b,s] := v[0,s]) \end{array} \right] \right\rfloor$$

Finally, for a heap root g-variable—$root_g(g) = gvar_{hm}(j)$—we set

$$upd_{mm}(m,g,v) := \lfloor m \left[ \; hm.ct := hm.ct([b,s] := v[0,s]) \; \right] \rfloor$$

Note, that in the latter case we do not care about the init component of the heap memory frame, since heap variables are by definition always initialized.

### 3.5.3 $C0$ Transition Function

The execution of a $C0$ program is modeled by the small-step semantics transition function $\delta_{C0}$. Parametrized with a type name environment and a procedure table, $\delta_{C0}$ computes for a given configuration $c$ either its successor configuration $c'$ or—in the case of a run-time error—*None*.

$$\delta_{C0} : tenv \times proctableT \times confT_{C0} \to confT_{C0\perp}$$

The function is defined by induction over the program rest. If the program rest is a single statement, we apply the transition function as described in the paragraphs below. Otherwise—i.e., the program rest is a tree spanned by compound statements—we apply $\delta_{C0}$ recursively.

#### Skip

A program rest consisting of a single *Skip*, that is $c.prog = Skip$, means that the program has terminated. In $C0$, termination is modeled by a infinite fix point loop: $\delta_{C0}(te, pt, c) = \lfloor c \rfloor$.

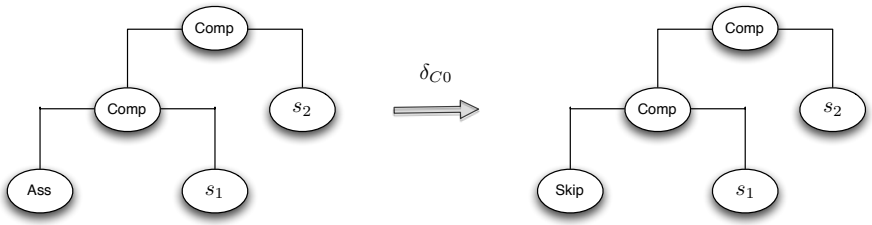


Figure 3.4: Execution of Compound Statements with 'Real' Statements

### Compound Statements

A compound statement $Comp(s_1, s_2)$ combines two sub trees $s_1$ and $s_2$. Execution is defined by recursion on the left sub tree.

The execution of that sub tree has finished, when there is only a *Skip* statement left, i.e., $s_1 = Skip$. In this case, we go on with the recursive execution of the right sub tree $s_2$. We define

$$\delta_{C0}(te, pt, c) = \begin{cases} \lfloor c[prog := s_2] \rfloor & \text{if } s_1 = Skip \\ \lfloor c'[prog := Comp(c'.prog, s_2)] \rfloor & \text{if } \delta_{C0}(te, pt, c[prog := s_1]) = \lfloor c' \rfloor \\ None & \text{otherwise} \end{cases}$$

This means that we always execute the left-most statement in statement tree. Consumed left sub trees—those that just consist of a *Skip* statement—are pruned together with the corresponding compound statement (see Figures 3.4 3.5).

### Conditional Statements

Executing a conditional statement $Ifte(e, s_1, s_2)$ means that there are two possible new program rests: either $s_1$ or $s_2$, depending on $e$. There are some requirements on the expression $e$ for a successful execution:

- $e$ has to be of Boolean type: $type(te, gst(c.m), lst(c.m), e) = \lfloor Bool_T \rfloor$,
- it is initialized, that is $?init(te, c.m, e) = True$, and
- it can be right evaluated, that is $rval(te, c.m, e) = \lfloor v \rfloor$.

If one or more of these conditions are violated, execution fails and we set $\delta_{C0}(te, pt, c) = None$. Otherwise, we define for $c.prog = Ifte(e, s_1, s_2)$:

$$\delta_{C0}(te, pt, c) = \begin{cases} \lfloor c[prog := s_1] \rfloor & \text{if } v = True \\ \lfloor c[prog := s_2] \rfloor & \text{otherwise} \end{cases}$$

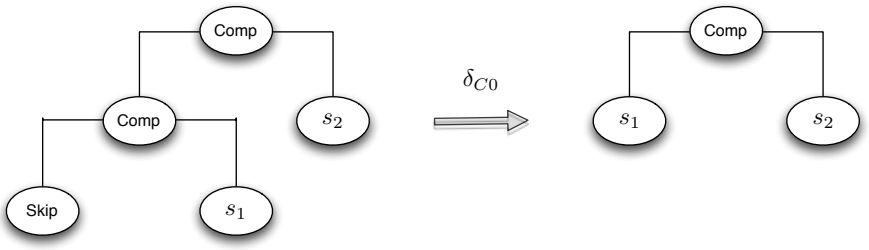


Figure 3.5: Execution of Compound Statements with Skip Statements

### While Loops

If the program rest consists of a loop, i.e., $c.prog = Loop(e, s)$, there are two possible program rests: *Skip*, if $e$ evaluates to *False*, or $Comp(s, Loop(e, s))$ otherwise—which means, we execute the loop body $s$ as long as $e$ is *True*.

Again, execution can fail, if not all of the following requirements on $e$ are met:

- $e$ has to be of Boolean type: $type(te, gst(c.m), lst(c.m), e) = \lfloor Bool_T \rfloor$,
- it is initialized, that is $?init(te, c.m, e) = True$, and
- it can be right evaluated: $rval(te, c.m, e) = \lfloor v \rfloor$.

In the case of failure, we set $\delta_{C0}(te, pt, c) = None$, otherwise we define:

$$\delta_{C0}(te, pt, c) = \begin{cases} \lfloor c[prog := Comp(s, Loop(e, s))] \rfloor & \text{if } v = True \\ \lfloor c[prog := Skip] \rfloor & \text{otherwise} \end{cases}$$

### Assignments

Given a program rest $c.prog = Ass(e_{\text{left}}, e_{\text{right}})$ and the following requirements:

- $e_{\text{left}}$ can be left evaluated to some g-variable $g$:

$$lval(te, c.m, e_{\text{left}}) = \lfloor g \rfloor,$$

- $e_{\text{right}}$ is initialized, $?init(te, c.m, e_{\text{right}}) = True$, and
- can be right evaluated to some value $v$:

$$rval(te, c.m, e_{\text{right}}) = \lfloor v \rfloor,$$

and finally

- the memory update succeeds: $upd_{mm}(c.m, g, v) = \lfloor m' \rfloor$.

If one or more of the above conditions are violated, we set $\delta_{C0}(te, pt, c) = None$. Otherwise, we define

$$\delta_{C0}(te, pt, c) = \left\lfloor c \left[ \begin{array}{l} prog := Skip \\ m := m' \end{array} \right] \right\rfloor$$

**Aggregate Literals** We have already mentioned in Sect. 3.2.3 and 3.5.1, there is an extra statement for the assignment of aggregate literals: $Ass_c$. It allows to assign complex values in *one* step of the $C0$ machine, which is essential in the equivalence proof of the small-step semantics and the Hoare logic as defined in [Sch06]. Since this proof is not relevant for our work, we do not get further into the details of it, but merely describe the semantics of $Ass_c$, which are very similar to those of a 'normal' assignment.

For a program rest $c.prog = Ass_c(e_{\text{left}}, al)$, we require that

- $e_{\text{left}}$ can be left evaluated to some g-variable $g$, that is
$$lval(te, c.m, c.m) = \lfloor g \rfloor;$$
- the aggregate literal $al$ can be right evaluated to some value $v$:
$$rval_{al}(te, c.m, al) = \lfloor v \rfloor,$$
- and finally the memory update is successful: $upd_{mm}(c.m, g, v) = \lfloor m' \rfloor$.

In the case of failure, we set $\delta_{C0}(te, pt, c) = None$, otherwise we define

$$\delta_{C0}(te, pt, c) = \left\lfloor c \left[ \begin{array}{c} prog := Skip \\ m := m' \end{array} \right] \right\rfloor$$

**Memory Allocation**

Whenever we are to allocate a new object on the heap, we have to make sure first, that there is enough memory available to do so. Thus, we define a predicate $?heap : memconfT \times ty \to \mathbb{B}$, which determines this for a given memory configuration and a type. On the semantic level, we leave this predicate uninterpreted. In a concrete setting as described in [Lei08], parameters like overall memory size, current heap size, and compiler construction would be part of such a definition.

For a successful execution of the $PAlloc(e_{\text{left}}, tn)$ statement we require,

- that $e_{\text{left}}$ can be left evaluated to some g-variable $g$: $lval(te, c.m, e_{\text{left}}) = \lfloor g \rfloor$, and
- that the type name $tn$ is defined in $te$: $mapof(te, tn) = \lfloor t \rfloor$.

As before, we set $\delta_{C0}(te, pt, c) = None$ for the failure case.

If both requirements are met, there are two ways to proceed depending on the value of $?heap$. Assuming that there is enough memory available, we first extend the heap with a new variable of the specified type and assign the appropriate initial value to it. In a second step, we create a pointer to this variable and assign it to the left expression. If there is not enough memory, we only assign a null pointer to the left expression.

**Definition 3.25 (Extending the Heap)** We define now a function

$$ext_{\text{heap}} : (memconfT \times ty \to \mathbb{B}) \times memconfT \times ty \to memconfT,$$

which handles the extension of the heap as described above. Let $m$ denote a memory configuration and $t$ the type of the variable to be added to the heap.

The base address of this variable is equal to the overall abstract size of the whole heap—since it will be appended at its end. Thus we set

$$b = \sum_{i=0}^{|hst(m)|-1} size_T(snd(hst(m)!j)).$$

If $?heap(m,t) = True$, that is there is enough memory available for a new variable of type $t$, we define the new heap memory $hm'$ by

$$hm' = m.hm \left[ \begin{array}{l} st := hst(m)@[(undef, t)] \\ ct := m.hm.ct([b, size_T(t)] := init_{\text{val}}(t)[0, size_T(t)]) \end{array} \right]$$

In the case of insufficient memory—$?heap(m,t) = False$—, we just return the old heap: $hm' = m.hm$.

We still have to deal with the second step: assigning either the null pointer or a pointer to the newly created variable to the left hand expression. We set:

$$p = \begin{cases} Ptr(gvar_{hm}(|hst(c.m)|)) & \text{if } ?heap(c.m,t) = True \\ NullPointer & \text{otherwise} \end{cases}$$

Since $p$ is initialized by definition (cf. Sect. 3.4) and the requirements described above are met, the following memory update will not fail and yield a new memory configuration $m'$:

$$upd_{mm}(ext_{\text{heap}}(?heap, c.m, t), g, p) = \lfloor m' \rfloor.$$

Finally, we define the resulting configuration by

$$\delta_{C0}(te, pt, c) = \left\lfloor c \left[ \begin{array}{l} prog := Skip \\ m := m' \end{array} \right] \right\rfloor$$

### Function Calls

Given a program rest $SCall(vn, fn, plist)$, i.e., a call of the function with name $fn$ and a list of parameter expressions $plist$. $vn$ denotes the name of the variable, where the return value will be stored later on. A function call is realized in three steps:

1. We start with the extension of the stack by a new frame. The frame's symbol table is made up by the function parameters and its local variables.
2. Subsequently, we evaluate the parameter expression list and copy the results into the new stack frame.
3. In the end, we set the function body as the new program rest.

**Definition 3.26 (Parameter Passing)** We start by defining the function $copy_{\text{pars}} : memconfT \times (\mathbb{N} \to mcellT)\ list \times \Sigma^+\ list \to memconfT_\perp$. $copy_{\text{pars}}$ takes a list of values and tries to copy them into the topmost stack frame, where a list of variable names specifies the variables to be updated.

Let $vl$ denote a list of values, $pnl$ a list of parameter names, and $m$ a memory configuration. We then define $copy_{\text{pars}}(m, vl, pnl)$ recursively on the value list $vl$ with the base case

$$copy_{\text{pars}}(m, [\,], pnl) = \lfloor m \rfloor.$$

For the recursive case $copy_{\text{pars}}(m, v\#vl, pnl)$, we examine the memory update with the head of the value list $v$. If it fails, we set

$$copy_{\text{pars}}(m, v\#vl, pnl) = None.$$

Otherwise, there exists a memory configuration $m'$ with

$$upd_{mm}(m, gvar_{lm}(|m.lm| - 1, hd(pnl)), v) = \lfloor m' \rfloor$$

and we set

$$copy_{\text{pars}}(m, v\#vl, pnl) = copy_{\text{pars}}(m', vl, tl(pnl))$$

For reasons of better readability, we introduce the shorthand notation $st_{\text{func}}(f)$, which refers to the symbol table of a function $f \in procT$. This symbol table consists of the parameters of the function and its local variables: $st_{\text{func}}(f) = f.params@f.lvars$.

**Definition 3.27** We will now take the stack extension and the parameter passing and put them into one function $ext_{\text{stack}} : tenv \times memconfT \times \Sigma^+ \times procT \times expr\ list \to memconfT_\perp$. Let $ext_{\text{stack}}(te, m, vn, f, pl)$ denote an application of this function.

First, we define the new stack frame $mf'$ by

$$mf' = \begin{bmatrix} st = st_{\text{func}}(f) \\ ct = undef \\ init = \emptyset \end{bmatrix}$$

Then we add this new stack frame at the end of the existing stack and obtain a new intermediate memory configuration $m_{imed}$:

$$m_{imed} = m[lm := m.lm@[(mf', lval(te, m, Var(vn)))]]$$

Here, $lval(te, m, Var(vn))$ determines the g-variable where the return value will be stored at the end of the function execution.

Now, we take this intermediate memory configuration and copy the parameter values into it and obtain thereby the extended memory configuration:

$$ext_{\text{stack}}(te, m, vn, f, pl) = \\ copy_{\text{pars}}(m_{imed}, map(the(rval(te, m)), pl), map(fst, f.params)).$$

Stack extension can fail for two reasons:

- the left evaluation for the return destinations fails: $lval(te, m, Var(vn)) = None$.
- the evaluation of one or more parameter expressions fails: $\exists p \in \{pl\} : rval(te, m, p) = None$.

In these cases we set $ext_{\text{stack}}(te, m, vn, f, pl) = None$.

We can now define the successor configuration of a function call $c.prog = SCall(vn, fn, pl)$ by using the definition above and setting the new program rest

appropriately. Let $f$ denote the called function, i.e., $the(mapof(pt, fn)) = f$, and assume $ext_{\text{stack}}(te, m, vn, pl) = \lfloor m' \rfloor$. Then we define

$$\delta_{C0}(te, pt, c) = \left\lfloor c \left[ \begin{array}{l} prog := f.body \\ m := m' \end{array} \right] \right\rfloor$$

If our assumption fails, that is $ext_{\text{stack}}(te, m, vn, pl) = None$, the execution of the function call fails and we set $\delta_{C0}(te, pt, c) = None$.

### Returns

The execution of a program rest $c.prog = Return(e)$ can be split into two steps. First, we evaluate the return expression $e$ and use its value to update the return destination of the topmost stack frame. Then, we remove this frame from the stack.

Successful execution has several requirements to be met:

- the return expression can be right evaluated, that is

$$rval(te, c.m, e) = \lfloor v \rfloor,$$

- it is initialized, i.e., $?init(te, c.m, e) = True$, and finally
- the memory update of the return destination succeeds:

$$upd_{mm}(c.m, snd(toplm(c.m)), v) = \lfloor m' \rfloor.$$

In the failure case, we set again $\delta_{C0}(te, pt, c) = None$. Otherwise, we define the transition function for return statements by

$$\delta_{C0}(te, pt, c) = \left\lfloor c \left[ \begin{array}{l} prog := Skip \\ m := m'[lm := butlast(m'.lm)] \end{array} \right] \right\rfloor$$

### External Function Calls

As mentioned in Sect. 3.2.3, external function calls refer to functions that are just declared, but not defined. In Sect. 6.5.2, the occurrences of external function calls will be replaced by actual function calls as described further above. There is not much sense in defining a transition function for external function calls. Nevertheless for reasons of completeness, we set for a program rest $c.prog = ESCall(vn, fn, plist)$

$$\delta_{C0}(te, pt, c) = \lfloor c\,[prog := Skip] \rfloor$$

### XCalls and In-line Assembler

**In-line Assembler** We will deal here with in-line assembly code only superficially, since the part of the CVM correctness proof, with which we deal in this work, relies only on pure $C0$ small step semantics.

In the early stages of the Verisoft project, we originally considered to introduce an extra computational model of C with in-line assembler code. Practical formal verification work within Verisoft's academic sub-project has taught us yet that a dual model approach is more feasible. This means, we

separate the computational model for assembler programs from the one for $C0$ programs and thus also separate the proof work. Later on, we combine the two results with each other.

This is possible, because the compiler correctness theorem, as introduced in [Lei08], relates each $C0$ configuration to a corresponding configuration of the assembler machine, thus providing two traces of relating configurations. So, whenever the program rest consists of an in-line assembler statement, we can move from this $C0$ configuration down to the relating assembler configuration. As we assume termination of in-line assembler portions (for a full list of assumptions on in-line assembler and its formalization see [Tsy09]), we can now execute the whole instruction list provided by the *Asm* statement. The final assembler configuration is then used to derive again the related $C0$ configuration from it. Note that this might include memory updates of variables of the $C0$ program.

There are a couple of examples in Verisoft, where this methodology has been successfully applied. In [Tsy09], correctness for some of the CVM kernel primitives has been shown. [Sta09] has formally verified a demand paging algorithm. Finally, [Alk09] has produced a correctness proof for a device driver.

**XCalls** XCalls are a way to make the results of in-line assembly code visible on higher levels of the semantic stack, i.e., in the Hoare Logic of [Sch06]. The meaning of each XCall is defined in an axiomatic way. The authors of [ASS08, AHL+09] have used this methodology in the pervasive verification of the paging mechanism used in Verisoft's academic sub-project.

> Most of the fundamental ideas of science are essentially simple, and may, as a rule, be expressed in a language comprehensible to everyone.
>
> Albert Einstein

# Chapter 4

# Devices

**Contents**



In our device model [HIP05], we define single devices as finite state transition systems, that interact on one hand with the processor and on the other hand with an—not modeled—external environment (cf. Fig. 4.1). The external environment can, for instance, represent a user typing on a keyboard or a network, which is connected to a device.

## 4.1 Device Configurations

In Verisoft, we support the following device types:

- a *timer*, which can be used, for instance, for scheduling,
- a *hard disk*,
- an *UART serial interface*, as it is used to connect terminals for example,
- an *automotive bus controller (ABC)*, which is used in the Automotive sub-project of Verisoft, and
- a *network interface controller (NIC)*.

Three of the above devices have been formally modeled in Isabelle/HOL: the hard disk [AH08], the serial interface [AHK+07], and the automotive bus controller [Kna08, ABK08, IK05] used in the Verisoft automotive sub-project [BBG+05, ILP05].

The detailed discussion of these devices goes far beyond the scope of this work and it is not necessary for the further understanding. Each type of device has its own disjoint state space, which we abbreviate by $confT_x$ with $x$ denoting the device type: $x \in \{timer, hd, uart, abc, nic\}$. The union of all possible device states is denoted by $confT_{\text{dev}}$.

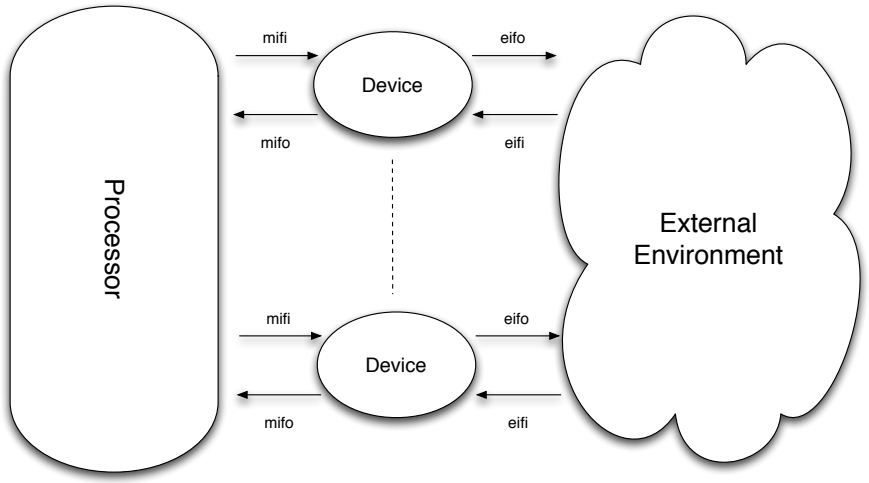


Figure 4.1: Devices Interacting with External Environment and Processor

Remark (Isabelle/HOL) In Isabelle/HOL, we model $confT_{\text{dev}}$ by an abstract data type with one constructor for each device type.

Let $d_{max}$ denote the maximal number of devices in a system. Then each device is associated with an unique identifier in the set $\{0, \ldots, d_{max} - 1\} = \mathbb{D}$. The overall generalized device configuration of such a system is then given through the mapping

$$confT_{\text{devs}} = \mathbb{D} \rightarrow confT_{\text{dev}}.$$

The type for a given device state $c$ can be obtained by the mapping $type_{\text{dev}}$ with

$$type_{\text{dev}}(c) = t \Leftrightarrow c \in confT_t.$$

We additionally define a function $irq_{\text{dev}} : confT_{\text{devs}} \times \mathbb{D} \rightarrow \mathbb{B}$. For a given generalized device configuration $c_{\text{devs}}$, we say the device with id *did* is currently signaling an interrupt, iff $irq_{\text{dev}}(c_{\text{devs}}, did) = True$.

## 4.2 Device Communication

Device communication is bidirectional with both the processor and the external environment. For the processor part of communication, the devices are *memory mapped*. This means that certain parts of the device, the so-called *ports*, are mapped into the memory of the processor. Conventional memory operations can then be used to communicate with these devices, that is with load and store operations on the corresponding addresses. In our device model, we restrict the amount of ports per device to $port_{\max} = 2^{10}$.

### Device Communication on the Upper Layers

On upper-layer computational models, we abstract from assembly instructions and memory. Plus, we want to work with block operations on devices, that is we want to read and write chunks of data of size larger than one word. For this purpose we introduce two new types to allow for modeling the communication between processor and devices in that way: the *memory interface input mifi*—input from the processor to the device—and the *memory interface output Mifo*—output from the device to the processor. In CVM, we assume that device output is a list of integers and thus set $Mifo = \mathbb{Z}\ list$.

A memory interface input $din \in mifi$ is a record type with the following components:

- $rep \in \mathbb{B}$: if $rep = True$, we will read or write repeatedly from the same port of the device. Otherwise, we read from or write the words to consecutive ports starting with the port specified in the *port* field.
- $wr \in \mathbb{B}$: the write flag signals a write operation, iff $wr = True$ and a read operation else.
- $port \in \{0, \ldots, port_{max} - 1\}$ denotes the port on which we operate in the case of $rep = True$; otherwise, it specifies the port with which we start.
- $data \in \mathbb{Z}\ list$: in the case of $wr = True$, this component holds the data to be written to the device. In the read case, only the length $|data|$ of the list matters, since it determines how many words we are going to read from the device.

Furthermore, the element

$$mifi_\epsilon = (rep = False, wr = False, port = 0, data = [0])$$

denotes the so-called *idle mifi*.

In order to abstract from a single device, we extend this type by an extra field *did* to determine, for which device the input is meant. We name this new, extended type *Mifi*.

Communication with the external environment is again device-specific. So, there is for each device type $t$ a particular type for the input from the environment to the device named $Eifi_t$, the *external interface input*, and a type for the output from the device to the environment named $Eifo_t$, the *external interface output*. The union of all possible external inputs and outputs are named *Eifi* and *Eifo*, respectively.

Remark (Isabelle/HOL) In Isabelle/HOL, we model both *Eifi* and *Eifo* by abstract data types with one constructor for each device type.

### Device Communication on the ISA Level

On the ISA level, devices are mapped into the memory. A certain address range is reserved for the devices, so that they can be accessed with conventional load and store operations.

For memory interface input, the effective address of such an operation can be used to determine the corresponding device id and port, while the type of

the operation determines if the write flag is set or not. In the case of a write operation, we use the natural number representation of the value associated with this operation for the data field of the device input. Note that on the ISA level, we do not support block operations.

A detailed definition on how to compute memory interface input for devices on the ISA level can be found in [Tve09, Alk09].

In addition, memory interface output on this level consists only of a single integer value—compared to a list of integers for the upper layer device communication.

A upper layer memory interface input, which constitutes a block operation, can easily be translated into a finite sequence of low-level memory inputs.

## 4.3 Transition Functions

The device model which we introduce here is pseudo-parallel in the sense that we either do an internal step—the device consumes a processor input—or an external step—the device consumes an external input, but never both at the same time. Thus, for each device there is an external and an internal transition function. Again, we do not look into the details of these transition functions but merely treat them as uninterpreted functions.

While external steps can only produce external output, internal steps can result both in external and internal output.

Let $t$ denote a device type with $t \in \{hd, uart, nic, timer, abc\}$. Then, the internal transition function for that device is given through

$$\delta_t^{\text{int}} : mifi \times confT_t \to confT_t \times Mifo \times Eifo_t$$

and the external transition function through

$$\delta_t^{\text{ext}} : Eifi_t \times confT_t \to confT_t \times Eifo_t.$$

For CVM again, we want to abstract from single devices and obtain generalized transition functions for overall device configurations

$$\delta_{devs}^{\text{int}} : Mifi \times confT_{\text{devs}} \to confT_{\text{devs}} \times Mifo \times Eifo$$

and

$$\delta_{devs}^{\text{ext}} : (\mathbb{D} \times Eifi) \times confT_{\text{devs}} \to confT_{\text{devs}} \times Eifo.$$

With each application of the transition functions, one device makes a step.

Let $din_{\text{int}} \in Mifi$ denote an internal device input, $c_{\text{devs}} \in confT_{\text{devs}}$ a generalized overall device configuration. The device, which is supposed to progress, is given through $din_{\text{int}}.did$. We construct the internal input for a single device $din'_{\text{int}} \in mifi$ by throwing away the device identifier $did$. Additionally, we identify the device type $t = type_{\text{dev}}(c_{\text{devs}}(din_{\text{int}}.did))$.

Now, we step the device using the appropriate device specific internal transition function and obtain a new device state, internal output $dout_{\text{int}}$ and external output $dout_{\text{ext}}$:

$$(c'_t, dout_{\text{int}}, dout_{\text{ext}}) = \delta_t^{\text{int}}(din'_{\text{int}}, c_{\text{devs}}(din_{\text{int}}.did)).$$

Finally, we can specify the result of the generalized internal transition function as

$$\delta^{\text{int}}_{devs}(din_{\text{int}}, c_{\text{devs}}) = (c_{\text{devs}}(din_{\text{int}}.did := c'_t), dout_{\text{int}}, dout_{\text{ext}}).$$

This works similar for the generalized external transition function. Let $did \in \mathbb{D}$ denote a device id, $din_{\text{ext}} \in Eifi_t$ an external device input and $c_{\text{devs}} \in confT_{\text{devs}}$ a generalized device configuration. We identify the type of the addressed device by $t = type_{\text{dev}}(c_{\text{devs}}(did))$. Then, we step the device with the specific external transition function and obtain a new device state and external output:

$$(c'_t, dout_{\text{ext}}) = \delta^{ext}_t(din_{\text{ext}}, c_{\text{devs}}(did)).$$

The result of the generalized external transition function can then be defined as

$$\delta^{\text{ext}}_{devs}((did, din_{\text{ext}}), c_{\text{devs}}) = (c_{\text{devs}}(did := c'_t), dout_{\text{ext}}).$$

> Engineers now have the ability to formally specify properties of their hardware design model using an industry standard, and then verify these properties in dynamic verification (that is, simulation) or static verification (that is, formal verification), ... Prior to IEEE 1850, there were multiple proprietary ways of specifying properties and assertions, but not a standard. This meant that the same specification could not be used across multiple tools. With a new standard, a single form of specification can be reused across multiple processes.
>
> Harry Foster

Chapter 5

# VAMP ISA and Assembler

**Contents**



User processes are applications running on top of the microkernel. We could use the *C*0 small-step semantics as introduced in Chapter 3. Yet, we do not only want to model well-behaving friendly user processes, but also malevolent processes. Modeling in *C*0 would impose programming restrictions that do not cope with real world scenarios. For instance, a hacker would probably implement his malicious code directly in assembler to make use of certain exploits. Thus we restrain from *C*0 program user process and will instead use *assembler machines* to model user processes.

The hardware platform for CVM is given by the *Verified Architecture Microprocessor (VAMP)*, based on the DLX architecture [HP96] and introduced in [MP00]. A first VAMP implementation correctness proof has been presented in [BJK+03, BJK+06]. During the Verisoft project, the VAMP has been extended by I/O devices and address translation [DHP05].

There are several formal models related to the VAMP architecture. For each model, there exist two variants, one with device support and one without. Adjacent layers are related by simulations proofs [Tve09, Tsy09].

The most concrete layer is the *gate-level implementation*, above which we have the *instruction set architecture (ISA)*. Here, we already abstract from the actual hardware layout and define state transitions using the semantics of the processor's instruction set. For the correctness proof sketched in Chapter 7, this model of the hardware with devices represents the lowest layer in the simulation proof.

Yet, data and instructions are still given through bit vectors, resulting in an inconvenient model for a programmer. For this purpose, we introduce the VAMP assembly language, which is a slight abstraction of the VAMP ISA. Here, addresses are represented by natural numbers, and register and memory contents by integers. We consider this layer to be the programming model for low-level applications, and user processes will be represented by assembler machines.

## 5.1 VAMP ISA

In this section, we will briefly introduce the instruction set architecture of the VAMP. Then, we will combine this model with the generalized device model from Chapter 4. We will omit many details that can be found in [Tve09].

Remark (Isabelle/HOL) The following definitions are based on the formal definitions in Isabelle. While originally ISA memory was byte addressed (cf. [MP00]), the Isabelle version of it is word addressed.

### 5.1.1 Instruction Set Architecture

The VAMP instruction set architecture serves as the specification for the gate-level implementation. On the other hand, it offers a system software programmer's view of the hardware with all details as interrupt handling, address translation, and execution modes.

#### ISA Configuration

An ISA configuration $c_{\text{isa}}$ is modeled by the record type $confT_{\text{isa}}$ with fields:

- $pcp \in \mathbb{B}^{30}$, the program counter, and
- $dpc \in \mathbb{B}^{30}$, the delayed program counter which specifies the address for the next instruction fetch (for details on delayed branch mechanism cf. [MP00]);
- two register files, namely $gprs : \mathbb{B}^5 \to \mathbb{B}^{32}$, the general purpose register file and
- the special purpose register file $sprs : \mathbb{B}^5 \to \mathbb{B}^{32}$;
- a memory $mm : \mathbb{B}^{30} \to \mathbb{B}^{32}$.

Frequently, we refer to the special purpose registers by an acronym denoting their meaning (cf. Tab. 5.1). For instance, $sprs(ptl)$ denotes the page table length register $sprs(10)$.

Remark (Isabelle/HOL) Unlike in the hardware, in Isabelle/HOL bit vectors can have variable length. The preservation of the constant length is a transition invariant that had to be proven.

The VAMP processor offers support for two execution modes. In *system mode*, programs have direct access to the memory and have full control over the hardware via a set of privileged instructions. In *user mode*, memory access is subject to address translation and the use of privileged instructions will raise an exception. The current execution mode is defined by the content of the register $sprs.mode$: if it is 0, we are in system mode, otherwise we are in user mode.

| Index | Function | Name |
|---|---|---|
| 0 | Status Register | *sr* |
| 1 | Exception status register | *esr* |
| 2 | Exception cause register | *eca* |
| 3 | Exception PC | *epc* |
| 4 | Exception DPC | *edpc* |
| 5 | Exception data | *edata* |
| 6 | Rounding mode | *rm* |
| 7 | IEEE flags register | *ieeef* |
| 8 | Floating point condition code | *fcc* |
| 9 | Page table origin | *pto* |
| 10 | Page table length | *ptl* |
| 11 | Exception mode | *emode* |
| 12 | Not used | |
| 13 | Not used | |
| 14 | Not used | |
| 15 | Not used | |
| 16 | Used to distinguish system and user mode | *mode* |

Table 5.1: Special Purpose Registers

### ISA Transitions

The transition function for the ISA

$$\delta_{\text{isa}} : confT_{\text{isa}} \times \mathbb{B}^{19} \times \mathbb{B}^{32} \rightarrow confT_{\text{isa}}$$

takes an ISA configuration $c_{\text{isa}}$, an external interrupt vector *eev*, and data $dout_{\text{int}}$ provided by the devices, returning the successor configuration $c'_{isa}$. Each bit in *eev* indicates, if the corresponding external interrupt is raised (cf. Tab. 6.1). Then, we continue with a check, whether $c_{\text{isa}}$ together with *eev* is causing an interrupt. If so, we proceed with interrupt handling as described in the next paragraph. Otherwise, we execute the next instruction from the address the *dpc* points to and proceed with a case-split on this instruction.

**Interrupt Handling** We distinguish between internal and external interrupts (cf. Tab. 6.1). Some interrupts are maskable, i.e., their occurrences are ignored when they are masked. This mechanism is controlled by the status register $sprs(sr)$, where a 0 bit denotes that this interrupt is disabled. We use this mechanism for instance to prevent the CVM kernel from interruption as described in Sect. 6.2.2.

When an unmasked interrupt occurs, the VAMP switches to system mode and enters the *Interrupt Service Routine (ISR)*, which starts at address 0. The actual interrupt handling is then controlled by the code starting there. Interrupt handling ends usually with a `rfe` (return from exception) privileged instruction, returning to user mode. Then, execution is continued: in the case of a repeat interrupt with the instruction that caused the interrupt, in the continue case with the instruction that follows.

### Address Translation

The VAMP offers a single-level address translation support. Its memory is divided into chunks of consecutive $4096 = 2^{10} = 1K$ words called *pages*. In user mode, we interpret addresses of load/store operations as *virtual addresses*, that is they are subject to address translation.

Address translation is realized using a *page table*. A page table is a dedicated memory region in the main memory, which starts at the address specified in the *page table origin* register $sprs(pto)$ and has $sprs(ptl) + 1$—the so-called (*page table length*)—many *page table entries (PTEs)* of word-size.

Let $va$ denote a virtual address. The upper 20 bits $px(va) = va[29:10]$ are called *(virtual) page index* and are used as an index into the page table. The lower 10 bit $wx(va) = va[9:0]$ are named *word index*.

For $px(va) > sprs(ptl)$, a page fault interrupt is thrown . Otherwise, we compute the corresponding page table entry as

$$pte = mm(sprs(pto) \cdot 2^{10} + px(va)).$$

This entry is then interpreted as follows:

- the upper 20 bits $pte[31:12]$ are the *physical page index* $ppx$,
- $pte[11]$ is the *valid bit*, and
- $pte[10]$ is the *protection bit*.

If the valid bit is set, read operations on the virtual address are allowed. Furthermore, if the protection bit is cleared, writes are allowed, too. If these requirements are met, the access of the virtual address $va$ will be performed on the physical address $pa(va) = [ppx(va); wx(va)]$. Otherwise, a page fault interrupt will be generated.

## 5.2 Combining ISA and Devices

When moving from gate-level implementation to instruction set architecture, we basically abstract from timing since we consider instructions and not cycles any longer. The same instruction can have different run-times, depending on the state of the gate-level machine. For instance, execution duration of load/store operations heavily depends on cache content, which is no longer visible at the instruction set architecture level.

At the gate-level, processor and devices run with the same clock in lock-step. We make up for this loss of granularity on the ISA level by the introduction of execution interleaving of devices and processor.

**Definition 5.1 (Configuration of ISA and Devices)** A combined configuration of VAMP ISA with devices is modeled by a record type $confT_{\mathrm{i\&d}}$ with two components: $proc \in confT_{\mathrm{isa}}$, the VAMP ISA configuration of the processor, and $devs \in confT_{\mathrm{devs}}$, a generalized device configuration.

For an ISA configuration $c_{\mathrm{isa}}$, we introduce the notion of the *effective address* $ea(c_{\mathrm{isa}})$. In the case that the current instruction of this configuration is a load or store operation, this is the address from which we read or to which we write. In all other cases it is undefined.

The predicate $?store(c_{\text{isa}})$ will return *True*, if the current instruction of $c_{\text{isa}}$ is a store operation, and *False* otherwise. Moreover, $val(c_{\text{isa}})$ determines the integer representation of the value to be stored in case that the current instruction is a store instruction.

Finally, let $DA$ denote the address range for devices. For an address $ad$ in this range, $did(ad)$ yields the device id and $port(ad)$ the port encoded by this address. The detailed definition of the above functions can be found in [Tve09] and [Alk09].

We can now define the function $mifi_{\text{ISA}}$, which returns the processor's device input in a configuration $c_{\text{isa}}$.

$$mifi_{\text{ISA}}(c_{\text{isa}}) = \begin{cases} (False, False, port(ea(c_{\text{isa}})), [0]) & \text{if } ea(c_{\text{isa}}) \in DA \wedge \neg ?store(c_{\text{isa}}) \\ (False, True, port(ea(c_{\text{isa}})), [val(c_{\text{isa}})]) & \text{if } ea(c_{\text{isa}}) \in DA \wedge ?store(c_{\text{isa}}) \\ mifi_{\epsilon} & \text{otherwise} \end{cases}$$

Furthermore, the function $\omega(c_{\text{devs}})$ computes the external event vector $eev$ for the processor based on the current device configurations.

The transition function $\delta_{\text{i\&d}}$ for the combined ISA and device model has to distinguish, if the processor or a device are progressing next. Thus, it additionally takes an oracle, called event, for input, modeled by the option type $(\mathbb{D} \times Eifi)_{\perp}$. It returns a successor configuration and potentially an output to the external environment.

If an event $din_{\text{ext}}$ is equal to *None*, the processor component makes a step, otherwise a device makes a step.

If there is neither external device input nor does the processor access a device, that is $din_{\text{ext}} = None$ and $ea(c_{\text{isa}}) \notin DA$, the processor performs a local step. In this case, we use the ISA transition function to obtain an update of the current processor configuration, while there is no output to the environment:

$$\delta_{\text{i\&d}}(c_{\text{i\&d}}, None) = (\delta_{\text{isa}}(c_{\text{i\&d}}.proc, \omega(c_{\text{i\&d}}.devs), 0^{32}), c_{\text{i\&d}}.devs, eifo_{\epsilon}).$$

In the case of $din_{\text{ext}} = None$ and $ea(c_{\text{i\&d}}.proc) \in DA$, a processor-device step is taken. Then, the device input $mifi_{\text{ISA}}(c_{\text{i\&d}}.proc)$, generated by the processor for the device with id $did(ea(c_{\text{i\&d}}.proc))$, is used with the internal step function for devices:

$$(devs', dout_{\text{int}}, dout_{\text{ext}}) = \\ \delta^{\text{int}}_{devs}((did := did(ea(c_{\text{i\&d}}.proc)), mifi_{\text{ISA}}(c_{\text{i\&d}}.proc)), c_{\text{i\&d}}.devs).$$

Then, the processor makes a step using the external event vector and output of the updated device configuration:

$$proc' = \delta_{\text{isa}}(c_{\text{i\&d}}.proc, \omega(devs), dout_{\text{int}}).$$

Finally, the result of a processor-device transition is given through

$$(proc', devs', dout_{\text{ext}}).$$

At last, we consider external device steps, which happen when the event is some $din_{\text{ext}}$, that is there exists environmental input for a device. In this case,

we use the external device transition function to update the devices,

$$(devs', dout_{\text{ext}}) = \delta^{\text{ext}}_{devs}(din_{\text{ext}}, c_{\text{i\&d}}.devs),$$

while the processor component stays unchanged. So, we obtain

$$\delta_{\text{i\&d}}(c_{\text{i\&d}}, \lfloor din_{\text{ext}} \rfloor) = (c_{\text{i\&d}}.proc, devs', dout_{\text{ext}}).$$

For a whole run, that is several steps of the combined model, we introduce the $n$-step transition function

$$\delta^n_{i\&d} : confT_{i\&d} \times (\mathbb{N} \to (\mathbb{D} \times Eifi)_\perp) \to confT_{i\&d} \times Eifo\ list.$$

It takes a combined start configuration $c_{\text{i\&d}}$ and a sequence of events $seq$ as inputs and returns both the combined configuration after applying the transition function $\delta_{\text{i\&d}}$ $n$ times and the external output generated during the computation. We set

$$\delta^0_{i\&d}(c_{\text{i\&d}}, seq) = (c_{\text{i\&d}}, []).$$

We assume that

$$(c_{\text{i\&d}}', eifos) = \delta^n_{i\&d}(c_{\text{i\&d}}, seq)$$

and

$$(c_{\text{i\&d}}'', dout_{\text{ext}}) = \delta_{\text{i\&d}}(c_{\text{i\&d}}', seq(n+1))$$

to define

$$\delta^{n+1}_{i\&d}(c_{\text{i\&d}}, seq) = (c_{\text{i\&d}}'', dout_{\text{ext}} \# eifos).$$

## 5.3 Assembler

As mentioned in the introduction to this chapter, there exists a variant of VAMP assembler with devices, too. Yet in CVM, we use assembler as the programming model for user processes, which can only communicate with devices indirectly via the kernel. Thus, we will introduce here the computational model of assembler *without* devices.

Compared to the instruction set architecture, the assembly model abstracts from several machine features:

- We exchange bit vectors in favor of naturals for addresses and integers for values. Instructions are encoded by an abstract data type *Instr*.
- Address translation is not visible any longer. This means that any assembler computation either simulates a system mode execution or a user mode execution with already established memory virtualization.
- Interrupts are not visible any longer.

### 5.3.1 Abstracting from Bit Vectors

VAMP ISA uses 32-bit vectors for data and instructions, and 30-bit vectors for addresses. For the first two ones, we can either interpret these vectors as binary numbers or as 2's complement numbers. For the use at assembly level, we define two functions to convert between bit vectors and naturals/integers: $to_{int} : \mathbb{B}^{32} \to \mathbb{Z}$ and $to_{nat} : \mathbb{B}^{30} \to \mathbb{N}$ with

$$to_{int}(bv) = \begin{cases} \sum_{i=0}^{31} bv[i] \cdot 2^i & \text{if } bv[31] = 0 \\ -2^{31} + \sum_{i=0}^{30} bv[i] \cdot 2^i & \text{otherwise} \end{cases}$$

$$to_{nat}(bv) = \sum_{i=0}^{29} bv[i] \cdot 2^i$$

Any program is encoded as data in the main memory, thus given through integer values. We define a function $to_{\text{instr}} : \mathbb{Z} \to Instr \cup \{undef\}$. Since not all integers encode meaningful instructions, $to_{\text{instr}}(x)$ either returns an instruction in *Instr* or *undef*. A detailed introduction of all VAMP instructions can be found in [MP00, Bey05, Dal06, Tve09].

In the following, we refrain from the explicit use of the above defined conversion functions, if not needed for understanding.

**Definition 5.2 (Assembler Configuration)** An assembler configuration is modeled by the record type $confT_{\text{asm}}$ with components:

- $pcp \in \mathbb{N}$ and $dpc \in \mathbb{N}$ the two program counters,
- $gprs : \mathbb{Z}\ list$, the general purpose register file,
- $sprs : \mathbb{Z}\ list$, the special purpose register file,
- $mm : \mathbb{N} \to \mathbb{Z}$, the memory, a mapping from addresses to data.

### 5.3.2 Assembler Transitions

The VAMP assembler transition function

$$\delta_{\text{asm}} : confT_{\text{asm}} \to confT_{\text{asm}}$$

takes a configuration $c_{\text{asm}}$ and returns its successor configuration $c_{\text{asm}}'$. As for the VAMP ISA, assembler transitions are defined by a case distinction over the current instruction, whose location is given by the delayed program counter of the current configuration $c_{\text{asm}}$: $to_{\text{instr}}(c_{\text{asm}}.mm(c_{\text{asm}}.dpc))$. The transition function is defined explicitly in [Alk09].

A detail worth mentioning are the semantics for the two instructions dealing with interrupt handling. The `trap` instruction is used to raise the trap interrupt, a mechanism which is used in CVM to invoke kernel primitives (see Sect. 6.2.2). The `rfe` instruction returns from interrupt handling, from where we proceed as defined by the interrupt. Both instructions have effects that are not visible at the assembly level and are therefore modeled by dummy transitions.

There are no significant bugs in our released software that any significant number of users want fixed.

Bill Gates (1995)

# Chapter 6

# Communicating Virtual Machines

**Contents**



*Communicating Virtual Machines*—short: *CVM*—are a hardware-abstracting framework for microkernel programmers [IT08]. With CVM we introduce a computational model for a fixed number of user processes—modeled by assembler machines (cf. Chapter 5)—that can interact with each other and with a fixed number of devices (cf. Chapter 4) via an abstract kernel, modeled by a $C0$ machine (cf. Chapter 3).

CVM does not support shared memory—neither between processes nor between a process and a device—, so all communication is handled by the abstract kernel. This means that we assume an isolated memory for each user process (*memory separation*).

As implied by the term 'abstract', this kernel allows to abstract from low-level details like interrupt handling, memory virtualization, kernel entry and exit, and the actual implementation of the so-called *CVM primitives* (see Sect. 6.3).

From the viewpoint of the upper layers in a system stack, a microkernel programmer can take advantage of these primitives in order to implement more sophisticated kernel calls and task scheduling. In Verisoft, CVM is the platform for two microkernels: VAMOS, the kernel of the academic system (cf. Sect. 1.1), and OLOS, a real-time operating system for automotive systems [IK05, KP07b, Kna08].

The rest of this chapter is structured in the following way: in Sect. 6.1, we will make use of the computational models introduced in the chapters before, in order to define CVM configurations. In Sect. 6.2, we will discuss the CVM transition function in detail. The formal specification of the CVM primitives is presented in Sect. 6.3.

In Sect. 6.4, we define the initial CVM configuration and the $n$-step transition function for CVM configurations.

We obtain a concrete kernel by linking the low-level implementations with the abstract kernel. We introduce a formalism for linking on the level of $C0$ semantics and use this formalism to describe how to obtain such a concrete kernel in Sect. 6.5.

## 6.1 CVM Configuration

A CVM configuration $c_{\mathrm{CVM}}$ is formally modeled by a record type $confT_{\mathrm{CVM}}$ consisting of three components:

- the *user process component* $up \in upT$,
- the *abstract kernel component* $kern \in kernT$,
- and the *device component* $devs \in confT_{\mathrm{devs}}$.

The user component stores information about all the user processes, the process that is to be executed next, and the CVM interrupt mask. For a maximal number of user processes $p_{\max}$, let $\mathbb{P} = \{1, \ldots, p_{\max}\}$ denote the set of process ids. Then, we model the user component by the record type $upT$ with three fields:

- $procs : \mathbb{P} \to confT_{\mathrm{asm}}$, mapping of process ids to assembler machine configurations encoding these processes;
- $cp \in \mathbb{P}_{\perp}$, the current user process field. It is either *None* or $\lfloor pid \rfloor$, with $procs(pid)$ being the next process to be executed;
- $mask \in \mathbb{B}^{32}$, the CVM interrupt mask, identical for all user processes.

Kernel components are modeled by the type $kernT$, which has components:

- a *type name environment* $tn \in tenv$,
- a *procedure table* $pt \in proctableT$
- and the $C0$ *configuration* encoding the abstract kernel, $kconf \in confT_{C0}$.

## 6.2 Transitions

In CVM, all components take turns in execution, that is either the kernel progresses, or a user process, or a device. As already introduced in Sect. 5.2, we use an oracle to determine, whose turn is next.

Hence, a CVM transition $\delta_{\mathrm{CVM}}(c_{\mathrm{CVM}}, din_{ext})$ takes a CVM configuration $c_{\mathrm{CVM}}$ and an external device input $din_{ext}$ as parameters, yielding either a next state $\lfloor {c_{\mathrm{CVM}}}' \rfloor$ or *None*, if a run-time error has occurred, and potentially a device output to the environment.

Assuming the first case and depending on $din_{ext}$ and $c_{\mathrm{CVM}}.up.cp$, either one of the devices, a user process or the abstract kernel make a step:

- If $din_{ext} \neq None$, i.e., there is some external device input, we compute the new generalized device configuration and update $c_{\mathrm{CVM}}$ with it. In this case also some device output to the environment might be generated.

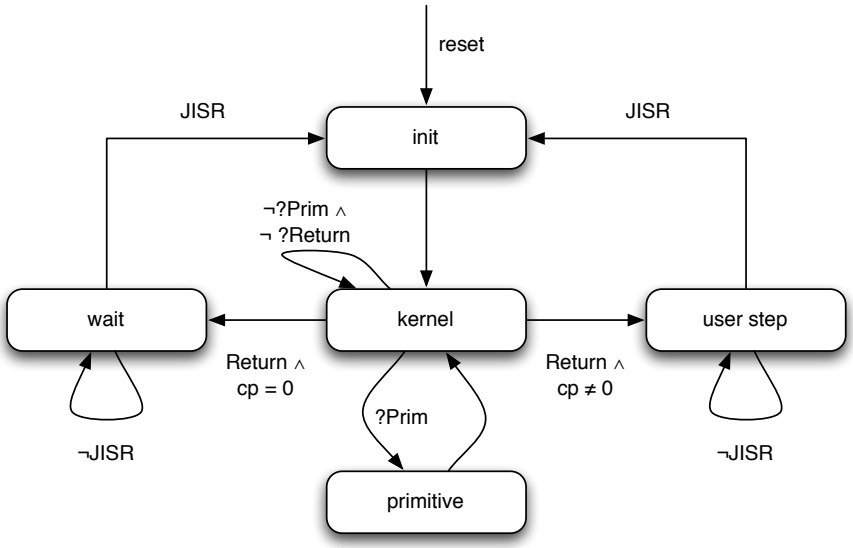


Figure 6.1: CVM Control Flow

- If there is no external device input and the current process identifier $c_{\mathrm{CVM}}.up.cp$ is *None*, the abstract kernel makes a step;
- Otherwise, the user process $c_{\mathrm{CVM}}.up.procs(the(c_{\mathrm{CVM}}.up.cp))$ progresses.

Operating systems provide idle mechanisms for times without user or kernel computation. This is either realized by an non-terminating *idle process* or via a dedicated wait state of the kernel, called *kernel wait*. As long as there is no interrupt, the kernel stays in this state. Otherwise, the kernel is re-entered as described in Sect. 6.2.2.

CVM primitives are functions, which allow to alter the state of the hardware, of a device, of the kernel, or of a user process. Though their implementation is not part of the abstract kernel, the effects of primitive execution are part of the CVM semantics.

In Fig. 6.1, we show the control flow for the kernel and user process case, respectively. Using this control flow, we we will now present in detail the formal definition for the CVM transition function

$$\delta_{\mathrm{CVM}} : confT_{\mathrm{CVM}} \times (\mathbb{D} \times Eifi)_{\perp} \to (confT_{\mathrm{CVM}} \times (\mathbb{D} \times Eifo)\, list)_{\perp}$$

in the above order: first device steps, then kernel steps, and finally user steps.

### 6.2.1 Device Steps

For an input $(c_{\mathrm{CVM}}, din_{ext})$ with $din_{ext} = \lfloor (did, eifi) \rfloor$, we use the generalized external device transition function and obtain

$$(devs', dout_{ext}) = \delta^{\mathrm{ext}}_{devs}(the(din_{ext}), c_{\mathrm{CVM}}.devs).$$

Thus, the next configuration of CVM is given by

$$\delta_{\mathrm{CVM}}(c_{\mathrm{CVM}}, din_{ext}) = \lfloor (c_{\mathrm{CVM}}[devs := devs'], dout_{ext}) \rfloor .$$

### 6.2.2 Kernel Steps

At first, we will define, how the abstract kernel is started. This happens each time, when we enter kernel mode, that is after a reset or when an unmasked interrupt has occurred during user process execution. In both cases, the abstract kernel main function—the *abstract kernel dispatcher*—is called, taking two unsigned integers as arguments: the *exception cause* and the *exception data*. Basically, these two values are taken from the hardware, though the concrete kernel, which is handling kernel entry, is providing them to the abstract kernel. This mechanism is described in detail in [Tsy09]. On entering the kernel, we set the current process identifier to *None* and zero the interrupt mask:

$$\begin{aligned} c_{\mathrm{CVM}}.up.cp &= None \\ c_{\mathrm{CVM}}.up.mask &= 0^{32}. \end{aligned}$$

Then, kernel execution can take different paths: when the program rest starts with a primitive call, we execute this primitive. If the execution of the kernel has terminated—the program rest is *Skip*—we leave the kernel and either switch to *kernel wait* or to a user process. Otherwise, we proceed by executing the next $C0$ statement of the abstract kernel's program rest.

For better readability, we will use the following abbreviations for the remainder of the section:

- *kconf* for $c_{\mathrm{CVM}}.kern.kconf$,
- *te* for $c_{\mathrm{CVM}}.kern.te$, and
- *pt* for $c_{\mathrm{CVM}}.kern.pt$.

#### Starting the Abstract Kernel

We define a function $start_{ak} : confT_{C0} \times \mathbb{N} \times \mathbb{N} \rightarrow confT_{C0}$ that is used in starting the abstract kernel. We assume that the abstract kernel's dispatcher function is named `dispatcher_kernel`, taking two parameters of type *Unsgnd*.

In order to capture the kernel start by the means of the $C0$ small-step semantics, we assume the existence of a dummy function[1] with a single local variable `abs_kernel_res` and two parameters `eca` and `edata`, exception cause and exception data, as provided by the hardware and delivered by the concrete kernel. Its body merely consists of a function call for the abstract kernel

[1] We will explain in Sect. 9.3 how to get rid of this artificial construct.

dispatcher followed by a *Return*. We denote this body by $s$ and set:

$$
\begin{array}{l}
s = \\
\mathit{Comp}( \\
\quad \mathit{SCall}( \\
\quad\quad \texttt{abs\_kernel\_res}, \\
\quad\quad \texttt{dispatcher\_kernel}, \\
\quad\quad \mathit{Var}(\texttt{eca}), \mathit{Var}(\texttt{edata}) \\
\quad ), \\
\quad \mathit{Return}(\mathit{Lit}(\mathit{Prim}(\mathit{Unsgnd}\,0))))-
\end{array}
$$

At the starting point of the abstract kernel, we initialize the local memory stack of the abstract kernel with the corresponding memory frame for this dummy function. This frame has an undefined memory content and its symbol table merely contains the variable name and type of `abs_kernel_res`. We define this initial memory frame $mf_{init}$ formally as follows:

$$
\begin{array}{lll}
mf_{init} = [ & ct = & \mathit{undef}, \\
& st = & [(\texttt{abs\_kernel\_res}, \mathit{Unsgnd}_T), \\
& & \quad (\texttt{eca}, \mathit{Unsgnd}_T), \\
& & \quad (\texttt{edata}, \mathit{Unsgnd}_T)], \\
& \mathit{init} = & \emptyset].
\end{array}
$$

The rest of the memory, that is the kernel heap and kernel global memory, stay unchanged compared to the point when we left the kernel last time. So, we obtain an updated memory $m'$ with

$$m' = \mathit{kconf}.m[\mathit{lm} := [mf_{init}, \mathit{undef}]].$$

The complete $C0$ configuration when starting the abstract kernel is then given through

$$\mathit{start}_{ak}(\mathit{kconf}, \mathit{eca}, \mathit{edata}) = \mathit{kconf}[m := m', \mathit{prog} := s].$$

### Leaving the Kernel

When execution of the abstract kernel has terminated—$\mathit{kconf}.\mathit{prog} = \mathit{Skip}$—we switch from the kernel either to kernel wait or to a user process. This is determined by the return value of the abstract kernel dispatcher. If the value of this variable is a valid user process id, i.e.,

$$\mathit{pid} = \mathit{value}_g(\mathit{kconf}.m, \mathit{gvar}_{lm}(0, \texttt{abs\_kernel\_res})) \in \mathbb{P},$$

we will switch to the corresponding user process $c_{\mathrm{CVM}}.\mathit{up.procs}(\mathit{pid})$, otherwise, we switch to kernel wait.

Switching to a user process is handled in the following way: Given a process id $\mathit{pid}$. Then, we analyze the corresponding user process $vm = c_{\mathrm{CVM}}.\mathit{up.procs}(\mathit{pid})$. If this user process has memory,[2] that is $vm.\mathit{sprs}!\mathit{ptl} \geq 0$, we make it the current

[2] In CVM, a ptl value of $-1$ by convention says that the process has no memory (cf. Sect. 5.1.1)—in contrast to regular hardware machines, where this denotes the maximum memory.

user process by setting the current process field $up' = c_{\mathrm{CVM}}.up[cp := pid]$. Otherwise, a run-time error has occurred and we return *None*:

$$\delta_{\mathrm{CVM}}(c_{\mathrm{CVM}}, [\,]) = \begin{cases} \lfloor (c_{\mathrm{CVM}}[up := up'], [\,]) \rfloor & \text{if } vm.sprs!ptl \geq 0 \\ None & \text{otherwise} \end{cases}.$$

Note, that the absence of run-time errors for a given abstract kernel is a correctness criterion the implementer would usually want to prove.

### Kernel Wait

Since our kernel is non-preemptive, that is non-interruptible, usually all interrupts are masked during kernel execution. When switching to kernel wait, we enable the external interrupts, so that the kernel can be 'woken up'. Then, we set the program rest to an assembler statement with an empty instruction list: $kconf' := kconf[prog := Asm[\,]]$.

The next state is given through

$$\delta_{\mathrm{CVM}}(c_{\mathrm{CVM}}, [\,]) = \lfloor (c_{\mathrm{CVM}}.kern[kconf := kconf'], [\,]) \rfloor.$$

In CVM kernel wait, the abstract kernel loops on a single assembler statement with an empty instruction list until an external interrupt occurs. We do not model this step using the $C0_A$ semantics, but treat it as a special case of the CVM transition function. Thus, the assembler statement merely serves as an indicator and since this is the only place in the abstract kernel with an assembler statement, we can easily use the program rest to determine, if we are in kernel wait: $hd(kconf.prog) = Asm[\,]$.

If a device interrupt has occurred, i.e.,

$$?irq = \bigvee_{i \in \mathbb{D}} irq_{\mathrm{devs}}(c_{\mathrm{CVM}}.devs, i) = 1,$$

we start the abstract kernel as described above and set

$$kern' := start_{ak}(kconf, eca, 0).$$

Note that devices do not generate exception data when raising an interrupt.

Otherwise, we leave the CVM configuration as it is:

$$\delta_{\mathrm{CVM}}(c_{\mathrm{CVM}}, [\,]) = \begin{cases} \lfloor (c_{\mathrm{CVM}}[kern := c_{\mathrm{CVM}}.[kern := kern'], [\,]) \rfloor & \text{if } ?irq = 1 \\ \lfloor (c_{\mathrm{CVM}}, [\,]) \rfloor & \text{otherwise} \end{cases}$$

### Kernel Primitive Steps

A user process invokes one of the the primitives using the `trap` instruction. On ISA level, this instruction raises the trap interrupt, and its argument—stored in the special purpose register $sprs(edata)$—specifies the number of the corresponding primitive. Since all primitives are numbered uniquely, the abstract kernel dispatcher uses the *edata* argument to determine the appropriate primitive to be executed.

The actual implementation of these functions is part of the *concrete kernel* (cf. Sect. 6.5.1). The abstract kernel merely declares these primitives, i.e. specifies their signature but without implementation.

Let $Prim \subset \Sigma^+$ denote the set of all primitive function names. For a $p \in Prim$, an abstract kernel program rest that starts with $ESCall(vn, p, plist)$ denotes a primitive call. The semantics of each of these primitives is specified by a function $p_s$, that takes a CVM configuration $c_{\text{CVM}}$ and a number of arguments corresponding to the parameter list *plist*. It returns either an updated CVM configuration and possibly some device output to the external environment, or *None* in the error case.

In Sect. 6.3, we give a complete definition of all primitive specification functions. We define:

$$\delta(c_{\text{CVM}}, [\,]) = p_s(c_{\text{CVM}}, plist).$$

#### $C0$ Kernel Steps

If the program rest of the kernel is neither *Skip*, nor an assembler statement, nor a CVM primitive call, the next CVM state is given through the standard C0 transition function. In the error case—$\delta_{C0}(te, pt, kconf) = None$—we set $\delta_{\text{CVM}}(c_{\text{CVM}}, [\,]) = None$. Else, we have $\delta_{C0}(te, pt, kconf) = \lfloor kconf' \rfloor$ and define

$$\delta_{\text{CVM}}(c_{\text{CVM}}, [\,]) = \lfloor (c_{\text{CVM}}[kern := c_{\text{CVM}}.kern[kconf := kconf']], [\,]) \rfloor$$

### 6.2.3 User Steps

We perform a user step when there is no external input for the devices and the current process field is not equal to *None*. We denote this value with $pid = the(c_{\text{CVM}}.up.cp)$ and the user process with $vm = c_{\text{CVM}}.up.procs(pid)$. Moreover we abbreviate the interrupt mask with $mask = c_{\text{CVM}}.up.mask$.

During user execution, either an interrupt occurs or not. In the latter case, we step the current user process using the assembler transition function as described in Sect. 5.3. In the interrupt case, we update the current process field and start the abstract kernel.

Dealing with interrupts at this level is a little tricky, since the assembly semantics has no notion of them. We define a function $mca : confT_{\text{asm}} \times confT_{\text{devs}} \times \mathbb{B}^{32} \to \mathbb{B}^{32}$ that takes a user process, a device configuration, and a CVM interrupt mask and returns a bit vector, the so-called *interrupt vector* according to Table 6.1. For a user process $vm$, a device configuration $c_{\text{devs}}$, we define $mca'$ as follows:

- $mca'[0] = 0$, because the reset case is treated differently.
- If the current instruction is undecodable, i.e. $to_{instr}(vm.mm(vm.dpc)) = undef$, or if it is a privileged instruction, we set $mca'[1] = 1$.
- For $vm.dpc \mod 4 \neq 0$, we have an instruction misalignment, and set $mca'[2] = 1$.
- In the case of data misalignment, we set $mca'[3] = 1$.
- We set $mca'[4] = 1$, if the delayed PC points to a region outside of the user process' memory: $vm.dpc \geq (vm.sprs.PTL + 1) \cdot 2^{12}$.
- The attempt to perform a memory operation on an address outside the user process' memory results in $mca'[5] = 1$.

- If the *dpc* points to an arithmetic operation, which would yield an integer overflow, we set $mca'[6] = 1$.
- If the current instruction is a `trap` instruction, we set $mca'[7] = 1$.
- Currently, we do not take care of floating point exceptions and thus set $mca'[8]$ to $mca'[12]$ to 0.
- For the device interrupts, we take the corresponding information from the CVM device configuration, i.e. for $13 \leq i \leq 20$, we set:

$$mca'[i] = irq_{dev}(c_{\text{devs}}, i - 13).$$

- The remaining bits are set to 0.

Finally, we apply the mask to $mca'$ by bitwise conjunction and obtain for $0 \leq i \leq 31$:

$$mca(vm, c_{\text{devs}}, mask)[i] = mca'[i] \wedge mask[i].$$

Note that if the instruction fetch already fails, i.e. in the cases of instruction misalignment, undecodable instructions, or a *dpc* outside the code region—we ignore the other interrupt causes.

Moreover, we define a predicate $JISR : \mathbb{B}^{32} \to \mathbb{B}$ and set

$$JISR(mca) = \bigvee_{i=0}^{31} mca[i].$$

### User Step without Interrupts

In the case of no interrupt, i.e. $JISR(mca(vm, c_{\text{CVM}}.devs, c_{\text{CVM}}.up.mask)) = 0$, we use the assembler next state function to obtain the new user process configuration $vm' := \delta_{\text{asm}}(vm)$ and denote the updated user component with

$$up' = c_{\text{CVM}}.up[procs := c_{\text{CVM}}.up.procs(pid := vm')].$$

Since we continue with the current user process, no other CVM component has to be updated and we define

$$\delta_{\text{CVM}}(c_{\text{CVM}}, [\,]) = \lfloor (c_{\text{CVM}}[up := up'], [\,]) \rfloor.$$

### User Step with Interrupts

If an interrupt has occurred, that is

$$JISR(mca(vm, c_{\text{CVM}}.devs, c_{\text{CVM}}.up.mask)) = 1,$$

we first determine the interrupt level according to Table 6.1 as following:

$$il = min\{j.\, mca(vm, c_{\text{CVM}}.devs, c_{\text{CVM}}.up.mask)[j] = 1\}.$$

If *il* is a continue interrupt, we step the user process and set $vm' = \delta_{\text{asm}}(vm)$. Otherwise, we leave the configuration as it is, i.e. $vm' = vm$.

| Bit | Name | Meaning | ext? | mask? | cont? |
|---|---|---|---|---|---|
| 0 | reset | Reset | yes | no | no |
| 1 | ill | Illegal Instruction | no | no | no |
| 2 | imal | Instruction Misalignment | no | no | no |
| 3 | dmal | Data Misalignment | no | no | no |
| 4 | pff | Page Fault on Fetch | no | no | no |
| 5 | pfls | Page Fault on Load/Store | no | no | no |
| 6 | ovf | Integer Overflow | no | yes | yes |
| 7 | trap | Trap Instruction | no | no | yes |
| 8–12 | | Floating Point Interrupts | no | yes | yes |
| 13–20 | | Device Interrupts | yes | yes | yes |
| 21–31 | | not used | | | |

Table 6.1: VAMP Interrupts

Additionally, we start the abstract kernel as described in Sect. 6.2.2, providing parameters $eca = mca(vm, c_{\mathrm{CVM}}.devs, c_{\mathrm{CVM}}.up.mask)$ and $edata$ as provided by the hardware. We denote the new kernel configuration with $kern'$ with

$$kern' = c_{\mathrm{CVM}}.kern[kconf := start_{ak}(c_{\mathrm{CVM}}.kern.kconf, eca, edata)]$$

and the updated user component with $up'$ with

$$up' = c_{\mathrm{CVM}}.up \left[ \begin{array}{l} procs := c_{\mathrm{CVM}}.up.procs(pid := vm') \\ cp := None \end{array} \right]$$

The transition is then given through

$$\delta(c_{\mathrm{CVM}}, [\,]) = \left\lfloor \left( c_{\mathrm{CVM}} \left[ \begin{array}{l} up := up' \\ kern := kern' \end{array} \right], [\,] \right) \right\rfloor$$

## 6.3 CVM Primitives

In this section, we will define the specification functions for the CVM primitives. We will first introduce some auxiliary functions, which will facilitate these definitions (cf. Sect. 6.3.1).

We have divided the CVM primitives in groups: those dealing with *process management*, e.g., the allocation and deallocation of user process memory, are presented in Sect. 6.3.2 (see Table 6.2). The second group contains these primitives used for inter-process communication (see Sect. 6.3.3). Device communication will be treated in Sect. 6.3.4. Primitives dealing with the kernel memory are introduced in Sect. 6.3.5.

Finally, all primitives that do not fit into one of the groups mentioned before are bundled in Sect. 6.3.6.

### 6.3.1 Auxiliary Functions

$read_{mm} : (\mathbb{N} \to \mathbb{Z}) \times \mathbb{N} \times \mathbb{N} \to \mathbb{Z}\ list$ takes an assembler memory, a start address and a length as input. $read_{mm}(mm, sa, am)$ returns the values stored at the

| Name | Function |
|---|---|
| *reset* | resets a user process to initial state |
| *clone* | clones a user process |
| *alloc* | allocates new virtual memory for a user process |
| *free* | frees virtual memory for a user process |
| *copy* | copies memory from one process to another |
| *GetGPR* | returns the content of a general purpose register |
| *SetGPR* | sets the content of a general purpose register |
| *SetMask* | sets the interrupt mask |
| *GetWord* | read a single word from user memory |
| *SetWord* | write a single word into user memory |
| *VirtIOIn* | allows user processes to read from a device |
| *VirtIOOut* | allows user processes to write to a device |
| *WordIn* | as *VirtIOIn*, but only for one word |
| *WordOut* | as *VirtIOOut*, but only for one word |
| *P2Vcopy* | copies kernel memory to user memory |
| *V2Pcopy* | copies user memory to kernel memory |
| *PhysIOIn* | read from a device to kernel memory |
| *PhysIOOut* | write from kernel memory to user memory |
| *PhysIOInRange* | as *PhysIOIn*, but on a range of ports |
| *PhysIOOutRange* | as *PhysIOOut*, but on a range of ports |
| *LoadOS* | load an operating system into memory |

Table 6.2: CVM Primitives

addresses from $sa$ to $sa + am - 1$ in the memory $mm$. We define recursively:

$$\begin{aligned} read_{mm}(mm, sa, 0) &= [\,] \\ read_{mm}(mm, sa, i+1) &= read_{mm}(mm, sa, i)@[mm(sa+i)] \end{aligned}$$

Symmetrically, $write_{mm} : (\mathbb{N} \to \mathbb{Z}) \times \mathbb{N} \times \mathbb{Z}\,list \to (\mathbb{N} \to \mathbb{Z})$ takes an assembler memory, a start address, and a sequence of integers as input. $write_{mm}(mm, sa, data)$ returns the updated memory, where the values stored at addresses $sa$ to $sa + |data| - 1$ are overwritten with the corresponding values from $data$. We define recursively:

$$\begin{aligned} write_{mm}(mm, sa, [\,]) &= mm \\ write_{mm}(mm, sa, x\#xs) &= write_{mm}(mm(sa := x), sa+1, xs) \end{aligned}$$

When we copy data from one memory to the other, we will use the function $copy_{mm} : (\mathbb{N} \to \mathbb{Z}) \times \mathbb{N} \times (\mathbb{N} \to \mathbb{Z}) \times \mathbb{N} \times \mathbb{N} \to (\mathbb{N} \to \mathbb{Z})$, which combines the above two functions. The first memory determines the destination to where we write and the second memory determines the source from which we read the data. The other parameters specify the start addresses and the amount of words to be copied. Let $data = read_{mm}(mm_s, sa_s, am)$ denote the data to be copied. Then the result of an execution of $copy_{mm}(mm_d, sa_d, mm_s, sa_s, am)$ is $write_{mm}(mm_d, sa_d, data)$.

Some primitives deal with devices and can thus produce output to the external environment. To convert this output to the type specified in the CVM transition function, we define $to_{\text{eifo}} : \mathbb{D} \times \mathit{Eifo}\ \mathit{list} \to (\mathbb{D} \times \mathit{Eifo})\ \mathit{list}$ and set:

$$\begin{aligned} to_{\text{eifo}}(did, [\,]) &= [\,] \\ to_{\text{eifo}}(did, x\#xs) &= (did, x)\#to_{\text{eifo}}(did, xs) \end{aligned}$$

For some primitives, we have to compute the amount of memory in words used by a single user process. We define the function $size_{mm} : \mathit{confT}_{\text{asm}} \to \mathbb{N}$ and set for a given machine $vm$:

$$size_{mm}(vm) = (vm.sprs!PTL + 1) \cdot 2^{10}.$$

The predicate $?inmem : \mathit{confT}_{\text{asm}} \times \mathbb{N} \to \mathbb{B}$ takes a virtual machine and an address as input and returns *True*, if the address lies within the memory of that user process and *False* otherwise. We define

$$?inmem(vm, ad) = \begin{cases} \mathit{True} & \text{if } ad < size_{mm}(vm) \\ \mathit{False} & \text{otherwise} \end{cases}.$$

Primitive calls in the concrete kernel are $C0$ function calls and returning from them is realized in the way defined in Sect. 3.5.3 for *SCall* and *Return* statements, respectively. Yet, in the abstract kernel, we deal with *external* function calls (*ESCall*), which are merely modeled by dummy transitions in the $C0$ semantics. To make the effects of a primitive call visible in the $C0$ configuration of the abstract kernel, we thus have to take care explicitly of both (i) the update of the return destination and (ii) the update of the program rest.

Given an abstract kernel component *kern* with

$$hd(kern.kconf.prog) = ESCall(vn, prim, e_1, \ldots, e_n)$$

denoting the call of a primitive with name *prim*. The corresponding g-variable for the return destination is then

$$rd = lval(kern.tn, kern.kconf.m, Var(vn)).$$

For a return value $v$, we obtain the updated memory by

$$m' = upd_{mm}(kern.kconf.m, rd, v).$$

The program rest of *kconf* is of the form

$$kern.kconf.prog = (ESCall(vn, prim, e_1, \ldots, e_n))\#pr.$$

The new program rest is then simply given by $prog' := \mathit{Skip}\#pr$. We combine these two steps in one function and define

$$upd_{ret}(kern, v) = kern\left[kconf := kern.kconf\left[\begin{array}{c} m := m' \\ prog := prog' \end{array}\right]\right]$$

### 6.3.2 Process Management

**Reset**

To reset a user process given by an assembler machine back into its initial state, we define the primitive

$$reset : confT_{\mathrm{CVM}} \times \mathbb{N} \to (confT_{\mathrm{CVM}} \times (\mathbb{D} \times Eifo)\, list)_{\perp},$$

which takes a single parameter *pid* denoting the process id of the process to be reset.

Resetting a user process means:

- The page table length register $PTL$ is set to $-1$, i.e. the machine has no memory.
- The mode register *mode* is set to 1, that is the machine runs in user mode.
- The *dpc* and *pcp* are set to their initial values 0 and 4.
- All other general and special purpose registers are zeroed.

For a primitive call $reset(c_{\mathrm{CVM}}, pid)$, we proceed as follows:

Let $vm$ denote the user process to be reset, i.e. $vm = c_{\mathrm{CVM}}.up.procs(pid)$. We denote with $gprs' = map(vm.gprs, \lambda x.0)$ and $sprs' = map(vm.sprs, \lambda x.0)$ the zeroed register files of $vm$. We can now define the reset virtual machine $vm'$ as follows:

$$vm' = vm \left[ \begin{array}{l} gprs := gprs' \\ sprs := sprs'[PTL := -1, mode := 1] \\ dpc := 0 \\ pcp := 4 \end{array} \right]$$

Using this, we update the user process component: $up' = c_{\mathrm{CVM}}.up[procs := c_{\mathrm{CVM}}.up.procs(pid := vm')]$.

Successful execution of the *reset* primitive depends on the input value *pid*. If *pid* is not a valid user process id, i.e. $pid \notin \mathbb{P}$, execution fails and we set $reset(c_{\mathrm{CVM}}, pid) = None$. Otherwise, we return the new CVM configuration with an updated kernel and user process component:

$$reset(c_{\mathrm{CVM}}, pid) = \left\lfloor \left( c_{\mathrm{CVM}} \left[ \begin{array}{l} up := up', \\ kern := upd_{ret}(c_{\mathrm{CVM}}.kern, 0) \end{array} \right], [\,] \right) \right\rfloor$$

**Clone**

The primitive

$$clone : confT_{\mathrm{CVM}} \times \mathbb{N} \times \mathbb{N} \to (confT_{\mathrm{CVM}} \times (\mathbb{D} \times Eifo)\, list)_{\perp}$$

takes besides a CVM configuration two process ids as input. The first id determines the process to be cloned—the cloner—and the second one determines the process which is replaced by an identical copy of the first one—the clonee.

For a primitive call $clone(c_{\mathrm{CVM}}, pid_1, pid_2)$ we proceed in the following way. Let $cloner = c_{\mathrm{CVM}}.up.procs(pid_1)$ denote the cloner process. Now we obtain

a new user process component $up'$ by merely overwriting the clonee with the cloner, i.e.

$$up' = c_{\mathrm{CVM}}.up[procs := c_{\mathrm{CVM}}.up.procs(pid_2 := cloner)].$$

Successful execution of the *clone* primitive depends on the following requirements:

- both process ids have to be valid, i.e $pid_1 \in \mathbb{P}$ and $pid_2 \in \mathbb{P}$,
- the clonee process has to have no memory, i.e.

$$(c_{\mathrm{CVM}}.up.procs(pid_2))!PTL = -1.$$

- Let $TVM$ denote a constant, which stores the total available virtual memory in words. Then, the overall amount of memory used after the cloning must not exceed this value:

$$\sum_{i=1}^{pmax} size_{mm}(c_{\mathrm{CVM}}.up.procs(i)) + size_{mm}(cloner) \leq TVM.$$

Violating one or more of these preconditions means execution of the primitive fails. In this case we set $clone(c_{\mathrm{CVM}}, pid_1, pid_2) = None$. Otherwise we set

$$clone(c_{\mathrm{CVM}}, pid_1, pid_2) = \left\lfloor \left( c_{\mathrm{CVM}} \left[ \begin{array}{l} up := up', \\ kern := upd_{ret}(c_{\mathrm{CVM}}.kern, 0) \end{array} \right], [\,] \right) \right\rfloor$$

**Alloc**

CVM allows to dynamically allocate more memory for user processes by means of the $alloc : confT_{\mathrm{CVM}} \times \mathbb{N} \times \mathbb{N} \to (confT_{\mathrm{CVM}} \times (\mathbb{D} \times Eifo)\ list)_\perp$ primitive. *alloc* takes two arguments: the first one is the id of the user process, of which the memory is to be enlarged, and the second one is the number of pages by which that memory is to be extended.

For a primitive call $alloc(c_{\mathrm{CVM}}, pid, pn)$ we proceed in the following way. Let $vm = c_{\mathrm{CVM}}.up.procs(pid)$ denote the corresponding user process. At first, we blank the new part of the process' memory with zeros. For this purpose, we compute the first word address in this new part by $sa = ((vm.sprs!PTL) + 1) \cdot 1024$. The new memory of the user process is then given through

$$mm' := copy_{mm}(vm.mm, sa, (\lambda x.0), 0, pn \cdot 1024).$$

In a second step, we adjust the page table length register to the new value:

$$sprs' := vm.sprs(PTL := (vm.sprs!PTL) + pn).$$

The updated user process is then given through

$$vm' := vm \left[ \begin{array}{l} sprs := sprs', \\ mm := mm' \end{array} \right]$$

Allocation of new memory will fail, if one of the following conditions is violated:

- *pid* has to be valid, i.e. $pid \in \mathbb{P}$, and
- the new memory size after allocation does not exceed the total available virtual memory:

$$\sum_{i=1}^{pmax} size_{mm}(c_{\mathrm{CVM}}.up.procs(i)) + pn \cdot 1024 \leq TVM.$$

In the case of failure, we set $alloc(c_{\mathrm{CVM}}, pid, pn) = None$. Otherwise, we define

$$alloc(c_{\mathrm{CVM}}, pid, pn) = \left\lfloor \left( c_{\mathrm{CVM}} \left[ \begin{array}{l} up := c_{\mathrm{CVM}}.up[procs := c_{\mathrm{CVM}}.up.procs(pid := vm')], \\ kern := upd_{ret}(c_{\mathrm{CVM}}.kern, 0) \end{array} \right], [\,] \right) \right\rfloor$$

**Free**

The corresponding primitive to *alloc* for freeing memory is $free : confT_{\mathrm{CVM}} \times \mathbb{N} \times \mathbb{N} \to (confT_{\mathrm{CVM}} \times (\mathbb{D} \times Eifo)\ list)_{\perp}$. Similarly, *free* takes two inputs besides the CVM configuration: the id of the user process, of which the memory is to be downsized, and the number of pages by which the memory is to be reduced.

Let $free(c_{\mathrm{CVM}}, pid, pn)$ denote a primitive call to de-allocate *pn* pages from user process $vm = c_{\mathrm{CVM}}.up.proc(pid)$. If the process has less memory pages then *pn*, we set the page table length register to $-1$, i.e. *vm* has no more memory. Otherwise, we decrease the page table length register by *pn*:

$$sprs' := vm.sprs \left[ PTL := \begin{cases} -1 & \text{if } vm.sprs!PTL < pn \\ vm.sprs!PTL - pn & \text{otherwise} \end{cases} \right]$$

We abbreviate the user process with updated register file by $vm' = vm[sprs := sprs']$.

Execution of *free* fails, if the process id parameter is not valid, i.e. $pid \notin \mathbb{P}$. In this case we set $free(c_{\mathrm{CVM}}, pid, pn) = None$. Otherwise, we define

$$free(c_{\mathrm{CVM}}, pid, pn) = \left\lfloor \left( c_{\mathrm{CVM}} \left[ \begin{array}{l} up := c_{\mathrm{CVM}}.up[procs := c_{\mathrm{CVM}}.up.procs(pid := vm')], \\ kern := upd_{ret}(c_{\mathrm{CVM}}.kern, 0) \end{array} \right], [\,] \right) \right\rfloor$$

### 6.3.3 Inter-Process Communication

**Copy**

Since CVM does not offer support for shared memory, all communication between user processes has to be made explicitly via the primitive

$$copy : confT_{\mathrm{CVM}} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to (confT_{\mathrm{CVM}} \times (\mathbb{D} \times Eifo)\ list)_{\perp}.$$

The $copy(c_{\mathrm{CVM}}, pid_s, pid_d, sa_s, sa_d, am)$ primitive takes five arguments besides the CVM configuration:

- two user process ids, namely $pid_s$ for the source process and $pid_d$ for the destination process,

- two start addresses, $sa_s$ for the source process and $sa_d$ for the destination process, and
- the amount of words $am$ to be copied.

With $mm_s = c_{\mathrm{CVM}}.up(pid_s).mm$, we denote the source memory and the destination memory with $mm_d = c_{\mathrm{CVM}}.up.procs(pid_d).mm$. We define the new destination memory $mm'_d$ as

$$mm'_d = copy_{mm}(mm_s, sa_s, mm_d, sa_d, am)$$

and denote the user process with the updated memory by $vm'_d := vm_d[mm := mm'_d]$.

Execution of the primitive call will fail, if one or more of the following requirements are not met:

- both process ids have to be valid and not equal:

$$pid_s \in \mathbb{P} \wedge pid_d \in \mathbb{P} \wedge pid_s \neq pid_d;$$

- both end addresses lie within the memory of their processes:

$$?inmem(vm_s, sa_s + am - 1) \wedge ?inmem(vm_d, sa_d + am - 1).$$

In the case of failure, we set $copy(c_{\mathrm{CVM}}, pid_s, sa_s, pid_d, sa_d, am) = None$. Otherwise, we set

$$copy(pid_s, sa_s, pid_d, sa_d, am) =$$
$$\left\lfloor \left( c_{\mathrm{CVM}} \left[ \begin{array}{l} up := c_{\mathrm{CVM}}.up[procs := c_{\mathrm{CVM}}.up.procs(pid_d := vm'_d)], \\ kern := upd_{ret}(c_{\mathrm{CVM}}.kern, 0) \end{array} \right], [\,] \right) \right\rfloor$$

**GetGPR**

The

$$GetGPR : confT_{\mathrm{CVM}} \times \mathbb{N} \times \mathbb{N} \rightarrow (confT_{\mathrm{CVM}} \times (\mathbb{D} \times Eifo)\ list)_{\perp}$$

primitive can be used to read out a single general purpose register of a user process. Thus, $GetGPR(c_{\mathrm{CVM}}, pid, r)$ takes a process id $pid$ and a register number $r$ as input. Let $val = c_{\mathrm{CVM}}.up.procs(pid).gprs!r$ denote the content of the register specified. This value will then be used to update the return destination in the kernel.

Execution of the primitive call will fail, if one or more of the following conditions are violated:

- $pid$ is a valid process id, i.e. $pid \in \mathbb{P}$, and
- $r$ is a valid register number: $r < 32$.

In the failure case, we set $GetGPR(c_{\mathrm{CVM}}, pid, r) = None$. Otherwise we set

$$GetGPR(c_{\mathrm{CVM}}, pid, r) = \lfloor (c_{\mathrm{CVM}}[kern := upd_{ret}(c_{\mathrm{CVM}}.kern, val)], [\,]) \rfloor$$

### SetGPR

Correspondingly, we define the primitive

$$SetGPR : confT_{\mathrm{CVM}} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z} \to (confT_{\mathrm{CVM}} \times (\mathbb{D} \times Eifo)\, list)_{\perp},$$

which allows to write a value into a general purpose register of a user process. $SetGPR(c_{\mathrm{CVM}}, pid, r, val)$ takes parameters $pid$ to determine the user process involved in the call, $r$ to specify the register to be written into, and finally $val$ for the corresponding value. Let $vm = c_{\mathrm{CVM}}.up.procs(pid)$ be the user process. Then we define the new general purpose register file by

$$gprs' = vm.gprs[r := val].$$

We abbreviate the updated user process with $vm' = vm[gprs := gprs']$.

The primitive call must comply with two preconditions. First, $pid$ must be a valid process id, i.e. $pid \in \mathbb{P}$. Moreover, $r$ must be a valid register number, i.e. $r < 32$, and $-2^{31} \leq val < 2^{31}$. In the case of failure, we set $SetGPR(c_{\mathrm{CVM}}, pid, r, v) = None$. Otherwise, we set

$$SetGPR(c_{\mathrm{CVM}}, pid, r, val) = \left\lfloor \left( c_{\mathrm{CVM}} \left[ \begin{array}{l} up := c_{\mathrm{CVM}}.up[procs := c_{\mathrm{CVM}}.up.procs(pid := vm')], \\ kern := upd_{ret}(c_{\mathrm{CVM}}.kern, 0) \end{array} \right], [\,] \right) \right\rfloor$$

### SetMask

User processes can mask device interrupts, such that their execution won't be interrupted. This mechanism is handled via the mask field $mask$ of the user process component, which can be set using the primitive

$$SetMask : confT_{\mathrm{CVM}} \times \mathbb{N} \to (confT_{\mathrm{CVM}} \times (\mathbb{D} \times Eifo)\, list)_{\perp}.$$

$SetMask(c_{\mathrm{CVM}}, mask)$ takes a parameter $mask$, which defines the new mask to be set. Since the corresponding field of the CVM user process component stores a 32-bit number, we first have to convert the natural number into a bit vector $mask'$. There are two requirements on the successful execution of the primitive.

- The length of the bit string must be 32 bits, i.e. $|mask'| = 32$;
- since we only allow processes to mask device interrupts, the lower 13 bits have to be 0: $\bigvee_{i=0}^{12} mask'[i] = 0$.

As before, we set $SetMask(c_{\mathrm{CVM}}, mask) = None$ in the failure case and otherwise

$$SetMask(c_{\mathrm{CVM}}, mask) = \left\lfloor (c_{\mathrm{CVM}} \left[ \begin{array}{l} up := c_{\mathrm{CVM}}.up[mask := mask'], \\ kern := upd_{ret}(c_{\mathrm{CVM}}.kern, 0) \end{array} \right], [\,]) \right\rfloor$$

### GetWord

Similar to the $GetGPR$ primitive, the primitive

$$GetWord : confT_{\mathrm{CVM}} \times \mathbb{N} \times \mathbb{N} \to (confT_{\mathrm{CVM}} \times (\mathbb{D} \times Eifo)\, list)_{\perp}$$

allows to read a single word from a user process' memory. The obtained value is then used to update the return destination of the kernel.

For a primitive call $GetWord(c_{\mathrm{CVM}}, pid, ad)$, let $c_{\mathrm{CVM}}$ denote a CVM configuration, $pid$ a user process id, and $ad$ an address in the memory of this process. Then we denote the value stored at this address by $val$ and define

$$val = c_{\mathrm{CVM}}.up.proc(pid).mm(ad).$$

Execution of the primitive call will fail, if one of the following requirements is not met:

- the process id has to be valid: $pid \in \mathbb{P}$, and
- the address has to be in the address range of this process' memory: $?inmem(c_{\mathrm{CVM}}.up.procs(pid), ad) = True$.

In the failure case, we define $GetWord(c_{\mathrm{CVM}}, pid, ad) = None$. Otherwise, we set

$$GetWord(c_{\mathrm{CVM}}, pid, ad) = \lfloor (c_{\mathrm{CVM}}[kern := upd_{ret}(c_{\mathrm{CVM}}.kern, val)], [\,]) \rfloor$$

**SetWord**

CVM allows to write a single word into the memory of a user process using the primitive

$$SetWord : confT_{\mathrm{CVM}} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z} \to (confT_{\mathrm{CVM}} \times (\mathbb{D} \times Eifo)\ list)_{\perp}.$$

A call of this primitive $SetWord(c_{\mathrm{CVM}}, pid, ad, val)$ takes parameters $pid$ to specify the user process and $ad$ for the address, where the word $val$ is to be stored. With $mm'$, we denote the updated memory of the user process $c_{\mathrm{CVM}}.up.procs(pid)$. We define $mm' = write_{mm}(c_{\mathrm{CVM}}.up(pid), ad, val)$ and write $vm' = c_{\mathrm{CVM}}.up.procs(pid)[mm := mm']$ for the virtual machine with the updated memory.

The above primitive call will fail, if one of the following conditions is violated:

- $pid$ has to denote a valid user process, i.e. $pid \in \mathbb{P}$, and
- the address $ad$ has to lie within the address range of the user process' memory: $?inmem(c_{\mathrm{CVM}}.up.procs(pid), ad) = True$.
- $val$ has to be in the range of $-2^{31} \leq val < 2^{31}$.

If execution fails, we define $SetWord(c_{\mathrm{CVM}}, pid, ad, val) = None$. Otherwise, we set

$$SetWord(c_{\mathrm{CVM}}, pid, ad, val) = \left\lfloor \left( c_{\mathrm{CVM}} \left[ \begin{array}{l} up := c_{\mathrm{CVM}}.up[procs := c_{\mathrm{CVM}}.up.procs(pid := vm')], \\ kern := upd_{ret}(c_{\mathrm{CVM}}.kern, 0) \end{array} \right], [\,] \right) \right\rfloor$$

### 6.3.4 User Processes and Devices

In this subsection, we will deal with primitives that establish the communication between a user process and a device. Naming goes back to the fact that user processes work with *virtual* memory.

### VirtIOIn

The CVM primitive

$$\mathit{VirtIOIn} : \mathit{confT}_{\mathrm{CVM}} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to (\mathit{confT}_{\mathrm{CVM}} \times (\mathbb{D} \times \mathit{Eifo})\, \mathit{list})_{\perp}$$

can be used by user processes to read data word-wise from a single device port.

A primitive call $\mathit{VirtIOIn}(c_{\mathrm{CVM}}, \mathit{did}, \mathit{port}, \mathit{pid}, \mathit{sa}, \mathit{am})$ takes besides a CVM configuration the following arguments:

- *did* specifies the device id and
- *port* determines the corresponding port of the device, from which we plan to read.
- *pid* specifies the process id of the user process and
- *sa* the start address in its memory where we want to write to.
- Finally, *am* is the amount of words we are going to read and write, respectively.

At first, we construct the necessary input $\mathit{din}_{\mathit{int}} \in \mathit{Mifi}$ with

$$\mathit{din}_{\mathit{int}} = \begin{bmatrix} \mathit{rep} & = & \mathit{True}, \\ \mathit{wr} & = & \mathit{False}, \\ \mathit{port} & = & \mathit{port}, \\ \mathit{data} & = & \mathit{replicate}(\mathit{am}, 0), \\ \mathit{did} & = & \mathit{did} \end{bmatrix}$$

Note, that only the length of the fifth component of the generalized device input type *Mifi* matters, since we do not write to the device but merely read from it. Then, we use this device input in the generic internal step function for devices and denote with the tuple $(\mathit{devs}', \mathit{dout}_{\mathrm{int}}, \mathit{eifos})$ the output of the device:

$$(\mathit{devs}', \mathit{dout}_{\mathit{int}}, \mathit{eifos}) = \delta^{\mathrm{int}}_{\mathit{devs}}(\mathit{din}_{\mathit{int}}, c_{\mathrm{devs}}).$$

The $\mathit{dout}_{\mathit{int}}$ component of the device output is the data, which we have read. We write this data into the memory of the specified user process:

$$\mathit{vm}' = c_{\mathrm{CVM}}.\mathit{up}.\mathit{procs}(\mathit{pid})[\mathit{mm} := \mathit{write}_{\mathit{mm}}(\mathit{mm}, \mathit{sa}, \mathit{dout}_{\mathit{int}})]$$

and set

$$\mathit{up}' = c_{\mathrm{CVM}}.\mathit{up}[\mathit{procs} := c_{\mathrm{CVM}}.\mathit{up}.\mathit{procs}(\mathit{pid} := \mathit{vm}')].$$

Successful execution of the primitive call relies on the following requirements:

- *pid* has to be a valid user process id, i.e. $\mathit{pid} \in \mathbb{P}$,
- *did* has to be a valid device id, i.e. $\mathit{did} \in \mathbb{D}$,
- *port* has to be within the range of valid ports, i.e. $\mathit{port} < \mathit{port}_{\mathit{max}}$, and
- *sa* has to be within the memory of the user process, i.e.

$$?\mathit{inmem}(c_{\mathrm{CVM}}.\mathit{up}.\mathit{procs}(\mathit{pid}), \mathit{sa}).$$

Again, in the failure case, we define $VirtIOIn(c_{\mathrm{CVM}}, did, port, pid, sa, am) = None$. Otherwise, we set

$$VirtIOIn(c_{\mathrm{CVM}}, did, port, pid, sa, am) = \left\lfloor \left( c_{\mathrm{CVM}} \left[ \begin{array}{l} up := up', \\ kern := upd_{ret}(c_{\mathrm{CVM}}.kern, 0), \\ devs := devs' \end{array} \right], to_{\mathrm{eifo}}(did, eifos) \right) \right\rfloor$$

**VirtIOOut**

The correspondent primitive for user processes to write data to a single device port is

$$VirtIOOut : confT_{\mathrm{CVM}} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to (confT_{\mathrm{CVM}} \times (\mathbb{D} \times Eifo)\ list)_{\perp}.$$

Let $VirtIOOut(c_{\mathrm{CVM}}, did, port, pid, sa, am)$ denote a call of the primitive with these parameters:

- *did* specifies the device id and
- *port* determines the corresponding port of the device, to which we plan to write.
- *pid* specifies the process id of the user process, and
- *sa* the start address in its memory from where we want to read.
- Finally, *am* is the amount of words we are going to read and write, respectively.

First, we read *am* many words from the user process' memory:

$$data = read_{mm}(c_{\mathrm{CVM}}.up.procs(pid).mm, sa, am).$$

Subsequently, we create the corresponding device input $din_{int} \in Mifi$ using *data*:

$$din_{int} = \left[ \begin{array}{lcl} rep & = & True, \\ wr & = & True, \\ port & = & port, \\ data & = & data, \\ did & = & did \end{array} \right]$$

and obtain the device output using the device transition function:

$$(devs', dout_{int}, eifos) = \delta^{\mathrm{int}}_{devs}(din_{int}, c_{\mathrm{CVM}}.devs).$$

Execution of the primitive call will fail, if one or more of the following requirements are not met:

- *pid* has to be a valid user process id, i.e. $pid \in \mathbb{P}$,
- *did* has to be a valid device id, i.e. $did \in \mathbb{D}$, and
- *port* has to be port within the range of valid ports, i.e. $port < port_{max}$, and

- finally, $sa$ has to be in the memory of the user process:

$$?inmem(c_{\mathrm{CVM}}.up.procs(pid), sa).$$

In the failure case, we define $VirtIOOut(c_{\mathrm{CVM}}, did, port, pid, sa, am) = None$. Otherwise, we set

$$VirtIOOut(c_{\mathrm{CVM}}, did, port, pid, sa, am) = \left\lfloor \left( c_{\mathrm{CVM}} \left[ \begin{array}{l} kern := upd_{ret}(c_{\mathrm{CVM}}.kern, 0), \\ devs := devs' \end{array} \right], to_{\mathrm{eifo}}(did, eifos) \right) \right\rfloor$$

**WordIn**

The

$$WordIn : confT_{\mathrm{CVM}} \times \mathbb{N} \times \mathbb{N} \to (confT_{\mathrm{CVM}} \times (\mathbb{D} \times Eifo)\, list)_{\perp}$$

primitive reads a single word from a device and uses the value read as the return value of the function call.

A call of this primitive $WordIn(c_{\mathrm{CVM}}, did, port)$ takes a device id $did$ as input, specifying the device, and a port $port$ from which to read. We use the device input $din_{int} \in Mifi$ in order to use it in the generalized device transition function and define

$$din_{int} = \left[ \begin{array}{lcl} rep & = & False, \\ wr & = & False, \\ port & = & port, \\ data & = & replicate(1, 0), \\ did & = & did \end{array} \right].$$

The result of the device transition function is then given through

$$(devs', dout_{int}, eifos) = \delta^{\mathrm{int}}_{devs}(din_{int}, c_{\mathrm{CVM}}.devs).$$

The following conditions have to be fulfilled for a successful primitive call:

- $did$ has to be a valid device id, i.e. $did \in \mathbb{D}$, and
- $port$ has to be port within the range of valid ports, i.e. $port < port_{max}$.

Violating one or both of them leads to a run-time error and we define

$$WordIn(c_{\mathrm{CVM}}, did, port) = None.$$

Otherwise, we set

$$WordIn(c_{\mathrm{CVM}}, did, port) = \left\lfloor \left( c_{\mathrm{CVM}} \left[ \begin{array}{l} kern := upd_{ret}(c_{\mathrm{CVM}}.kern, dout_{int}), \\ devs := devs' \end{array} \right], to_{\mathrm{eifo}}(did, eifos) \right) \right\rfloor$$

### WordOut

Like the primitive in the section before, there is a corresponding one to write a single word to a device:

$$WordOut : confT_{\mathrm{CVM}} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to (confT_{\mathrm{CVM}} \times (\mathbb{D} \times Eifo)\, list)_{\perp}.$$

Let $WordOut(c_{\mathrm{CVM}}, did, port, val)$ denote a call of the primitive, where $did$ specifies the device and $port$ the port to which to write, and $val$ the value to be written. Then we define the device input $din_{int}$ by

$$din_{int} = \begin{bmatrix} rep & = & False, \\ wr & = & True, \\ port & = & port, \\ data & = & [val], \\ did & = & did \end{bmatrix}.$$

We can now use this device input in the device transition function and obtain

$$(devs', dout_{int}, eifos) = \delta^{\mathrm{int}}_{devs}(din_{int}, c_{\mathrm{CVM}}.devs).$$

The requirements for the successful execution of the primitive call are as follows:

- $did$ has to be a valid device id, i.e. $did \in \mathbb{D}$, and
- $port$ has to be port within the range of valid ports, i.e. $port < port_{max}$.

In the failure case, we define $WordOut(c_{\mathrm{CVM}}, did, port, val) = None$. Otherwise, we set

$$WordOut(c_{\mathrm{CVM}}, did, port, val) = \left\lfloor \left( c_{\mathrm{CVM}} \begin{bmatrix} kern := upd_{ret}(c_{\mathrm{CVM}}.kern, 0), \\ devs := devs' \end{bmatrix}, to_{\mathrm{eifo}}(did, eifos) \right) \right\rfloor$$

## 6.3.5 Dealing with Physical Memory

Originally, CVM was not meant to support primitives dealing explicitly with physical memory. Yet, while developing real-time operating system in Verisoft's automotive subproject [IK05, KP07b], it turned out necessary to introduce primitives that allow to

- copy data from user (virtual) memory into the kernel (physical) memory and vice versa, and
- copy data from devices into the kernel memory and vice versa.

Since the abstract kernel is given by a $C0$ machine configuration, the data transfer between it and user processes and/or devices becomes a little more tricky. Thus, we will have to introduce some more auxiliary functions that deal with the necessary data conversion.

First, we define a function $mem2int : (\mathbb{N} \to mcellT) \to \mathbb{Z}$ to convert the content of an integer memory cell (see Sect. 3.3.1) to an integer value. For a memory cell $mc = Int(i)$, we set $mem2int(mc) = i$. Observe, that this function

only produces meaningful results, if it is applied to memory cells with the corresponding type stored in them.

Then, we need a function that allows to convert the value of a $C0$ variable into a list of integer values as needed in the assembler layer. We name this function $c2i : (\mathbb{N} \to mcellT) \times \mathbb{N} \to \mathbb{Z}\ list$, which takes two parameters: the first one is the memory content to be converted and the second one determines the size of the type to be converted minus one, We define:

$$c2i(cont, 0) = [mem2int(cont(0))]$$

for the base case and for the inductive case

$$c2i(cont, i+1) = c2i(cont, i)@[mem2int(cont(i+1))]$$

Furthermore, we have to convert part of a user process' memory—i.e. an list of integers—into a $C0$ memory content. Thus, we define the function $i2c : \mathbb{Z}\ list \to (\mathbb{N} \to mcellT)$ as follows:

$$i2c(xs) = \lambda i. \begin{cases} Int(xs!i) & \text{if } i < |xs| \\ undef & \text{otherwise} \end{cases}$$

Due to readability in the following, we will abbreviate for a given CVM configuration $c_{\mathrm{CVM}}$

- the type name environment $c_{\mathrm{CVM}}.kern.tn$ with $tn$,
- the procedure table $c_{\mathrm{CVM}}.kern.pt$ with $pt$, and
- the actual $C0$ configuration of the abstract kernel $c_{\mathrm{CVM}}.kern.kconf$ with $c_{C0}$.

### P2Vcopy

The primitive

$$P2Vcopy : confT_{\mathrm{CVM}} \times stmt \times \mathbb{N} \times \mathbb{N} \to (confT_{\mathrm{CVM}} \times (\mathbb{D} \times Eifo)\ list)_{\perp}$$

allows to copy the value of a variable of the abstract kernel into the memory of a user process.

A call of this primitive $P2Vcopy(c_{\mathrm{CVM}}, prim, pid, sa)$ takes the following parameters:

- A CVM configuration $c_{\mathrm{CVM}}$,
- $prim$, the actual $C0$ statement of this call—i.e. a statement of the form $ESCall(vn, \texttt{P2Vcopy}, plist)$,
- the user process id $pid$, specifying the process, and
- $sa$ for the start address in its memory.

We assume that the variable, whose value is to be stored, is located in the first position of the parameter list of the statement $prim$, i.e. at $plist!0$, and

that it is a pointer to an array with elements of type *Int*. The value of the referenced array type variable is then given through

$$val = the(rval(tn, c_{C0}.m, Deref(plist!0)))$$

and the number of its elements is equal to its abstract type size, i.e.

$$am = size_T(the(type(tn, toplst(c_{C0}.m), gst(c_{C0}.m), Deref(plist!0)))).$$

Now, we can compute $data = c2i(val, am)$, i.e. the data which we want to store. We update the corresponding user process' main memory, where the new user process configuration is defined as

$$vm' = c_{\mathrm{CVM}}.up(pid)[mm := write_{mm}(c_{\mathrm{CVM}}.up(pid).mm, sa, data].$$

The execution of this primitive will fail, if one of the following conditions is violated:

- $pid$ has to be a valid process id, i.e. $pid \in \mathbb{P}$, and
- the end address $sa + am - 1$ has to be in the memory range of that user process: $?inmem(c_{\mathrm{CVM}}.up(pid), (sa + am - 1)) = True$.

In the failure case, we define $P2Vcopy(c_{\mathrm{CVM}}, prim, pid, sa) = None$. Otherwise, we set

$$P2Vcopy(c_{\mathrm{CVM}}, prim, pid, sa) = \left\lfloor \left( c_{\mathrm{CVM}} \left[ \begin{array}{l} up := c_{\mathrm{CVM}}.up[procs := c_{\mathrm{CVM}}.up.procs(pid := vm')], \\ kern := upd_{ret}(c_{\mathrm{CVM}}.kern, 0) \end{array} \right], [\,] \right) \right\rfloor$$

### V2Pcopy

The copying of data from user processes to the abstract kernel is handled by the primitive

$$V2Pcopy : confT_{\mathrm{CVM}} \times stmt \times \mathbb{N} \times \mathbb{N} \to (confT_{\mathrm{CVM}} \times Eifo\ list)_{\perp}.$$

Let $V2Pcopy(c_{\mathrm{CVM}}, prim, pid, sa)$ denote a call of this primitive with parameters

- $c_{\mathrm{CVM}}$, a CVM configuration,
- $prim$, the $C0$ statement of the call, i.e. a statement of form

$$ESCall(vn, \texttt{V2Pcopy}, plist),$$

- a process id $pid$ and a start address $sa$.

The $C0$ variable, whose value we want to update, is an integer array, given through a pointer in position three of the parameter list of $prim$, i.e. $plist!2$, while the first two parameters specify process id and start address. We access this variable via the left value of the corresponding expression evaluation:

$$var = the(lval(tn, c_{C0}.m, Deref(plist!2)))$$

and its abstract size by

$$am = size_T(the(type(tn, toplst(c_{C0}.m), gst(c_{C0}.m), Deref(plist!2)))).$$

We can now read the data from the user process' memory

$$data = read_{mm}(c_{\mathrm{CVM}}.up(pid), sa, am)$$

and convert it into a $C0$ memory content $val = i2c(data)$. Finally, we update the abstract kernel's $C0$ variable and obtain a new memory

$$upd_{mm}(c_{C0}.m, var, val) = \lfloor m' \rfloor.$$

We denote the new abstract kernel component of the CVM configuration by $kern'$ and define:

$$kern' = c_{\mathrm{CVM}}.kern[kconf := c_{C0}[m := m']].$$

There are certain requirements for a successful execution of the primitive:

- $pid$ has to be a valid process id, i.e. $pid \in \mathbb{P}$, and
- the end address $sa + am - 1$ has to be in the memory range of that user process: $?inmem(c_{\mathrm{CVM}}.up.procs(pid), (sa + am - 1)) = True$.

In the failure case, we define $V2Pcopy(c_{\mathrm{CVM}}, prim, pid, sa) = None$. Otherwise, we set

$$V2Pcopy(c_{\mathrm{CVM}}, prim, pid, sa) = \lfloor (c_{\mathrm{CVM}} \left[kern := upd_{ret}(kern', 0)\right], [\,]) \rfloor$$

**PhysIOIn**

In order to copy data read from a single device port into a variable of the abstract kernel, we introduce the primitive

$$PhysIOIn : confT_{\mathrm{CVM}} \times stmt \times \mathbb{N} \times \mathbb{N} \to (confT_{\mathrm{CVM}} \times (\mathbb{D} \times Eifo)\ list)_{\perp}.$$

Let $PhysIOIn(c_{\mathrm{CVM}}, prim, did, port)$ denote a call of this primitive with the following arguments:

- $c_{\mathrm{CVM}}$, a CVM configuration,
- $prim$, the $C0$ statement of the call, i.e. a statement of form

$$ESCall(vn, \texttt{PhysIOIn}, plist),$$

- a device id $did$ and a port $port$.

As for the other primitives dealing with physical memory, the amount of data in words to be copied is determined by the abstract size of the kernel variable, in which we want to copy the data. We assume that the third parameter of the actual statement $prim$—i.e. $plist!2$—is a pointer expression referencing the

corresponding variable of integer array type. We refer to this variable via the result of the corresponding expression left evaluation

$$var = the(lval(tn, c_{C0}.m, Deref(plist!2)))$$

and its abstract size is

$$am = size_T(the(type(tn, toplst(c_{C0}.m), gst(c_{C0}.m), Deref(plist!2)))).$$

The abstract size $am$ of this kernel variable determines, how many words we are going to read from the device. We use this information to build the appropriate device input

$$din_{int} = \begin{bmatrix} rep & = & True, \\ wr & = & False, \\ port & = & port, \\ data & = & replicate(am, 0), \\ did & = & did \end{bmatrix}$$

Let $(devs', dout_{int}, eifos)$ denote the output of the device transition function, i.e.

$$(devs', dout_{\text{int}}, eifos) = \delta^{\text{int}}_{devs}(din_{int}, c_{\text{CVM}}.devs),$$

where $dout_{\text{int}}$ is the data we have read. We now construct the $C0$ memory content $val = i2c(dout_{int})$ and use it to update the kernel memory:

$$m' = upd_{mm}(c_{C0}.m, var, val).$$

We denote the new abstract kernel component of the CVM configuration by $kern'$ and define:

$$kern' = c_{\text{CVM}}.kern[kconf := c_{C0}[m := m']].$$

Successful execution of the primitive call depends on the following requirements:

- $did$ has to denote a valid device id, that is $did \in \mathbb{D}$, and
- $port$ has to be a valid port, i.e. $port < port_{max}$.

In the failure case, we define $PhysIOIn(c_{\text{CVM}}, prim, did, port) = None$. Otherwise, we set

$$PhysIOIn(c_{\text{CVM}}, prim, did, port) = \left\lfloor \left( c_{\text{CVM}} \begin{bmatrix} devs := devs', \\ kern := upd_{ret}(kern', 0) \end{bmatrix}, to_{\text{eifo}}(did, eifos) \right) \right\rfloor$$

### PhysIOInRange

The primitive

$$PhysIOInRange : confT_{\text{CVM}} \times stmt \times \mathbb{N} \times \mathbb{N} \to (confT_{\text{CVM}} \times Eifo\ list)_{\perp}$$

does basically the same as the $PhysIOIn$ primitive, but instead of reading from a single device port, we read from a *range* of consecutive ports.

Let $PhysIOInRange(c_{\text{CVM}}, prim, did, port)$ denote a call of this primitive with the following arguments:

- $c_{\mathrm{CVM}}$, a CVM configuration,
- *prim*, the $C0$ statement of the call, i.e. a statement of form

$$ESCall(vn, \texttt{PhysIOInRange}, plist),$$

- a device id *did* and a port *port*.

As for the other primitives dealing with physical memory, the amount of data in words to be copied is determined by the abstract size of the kernel variable, in which we want to copy the data. We require that the third parameter of the actual statement *prim*—i.e. *plist*!2—is a pointer expression referencing the corresponding variable of integer array type (the first two parameters specify device id and port). We refer to this variable via the result of the corresponding expression left evaluation

$$var = the(lval(tn, c_{C0}.m, Deref(plist!2)))$$

and its abstract size is

$$am = size_T(the(type(tn, toplst(c_{C0}.m), gst(c_{C0}.m), Deref(plist!2)))).$$

The abstract size *am* of the kernel variables determines, how many words we are going to read from the device. We use this information to build the appropriate device input

$$din_{int} = \begin{bmatrix} rep & = & False, \\ wr & = & False, \\ port & = & port, \\ data & = & replicate(am, 0), \\ did & = & did \end{bmatrix}$$

Let $(devs', dout_{\mathrm{int}}, eifos)$ denote the output of the device transition function, i.e.

$$(devs', dout_{\mathrm{int}}, eifos) = \delta^{\mathrm{int}}_{devs}(din_{int}, c_{\mathrm{CVM}}.devs),$$

where $dout_{\mathrm{int}}$ is the data we have read. We now construct the $C0$ memory content $val = i2c(dout_{int})$ and use it to update the kernel memory: $m' = upd_{mm}(c_{C0}.m, var, val)$. We denote the new abstract kernel component of the CVM configuration by $kern'$ and define:

$$kern' = c_{\mathrm{CVM}}.kern[kconf := c_{C0}[m := m']].$$

Successful execution of the primitive call depends on the following requirements:

- *did* has to denote a valid device id, that is $did \in \mathbb{D}$, and
- the last port from which we are going to read has still to be a valid port, i.e. $port + am - 1 < port_{max}$.

In the failure case, we define $PhysIOIn(c_{\mathrm{CVM}}, prim, did, port) = None$. Otherwise, we set

$$PhysIOIn(c_{\mathrm{CVM}}, prim, did, port) = \left\lfloor \left( c_{\mathrm{CVM}} \begin{bmatrix} devs := devs', \\ kern := upd_{ret}(kern', 0) \end{bmatrix}, to_{\mathrm{eifo}}(did, eifos) \right) \right\rfloor$$

### PhysIOOut

Symmetrically to the two primitives defined before, we introduce the primitive

$$PhysIOOut : confT_{\mathrm{CVM}} \times stmt \times \mathbb{N} \times \mathbb{N} \rightarrow (confT_{\mathrm{CVM}} \times (\mathbb{D} \times Eifo)\, list)_{\perp},$$

which will take the value of an abstract kernel variable and write it to one of the devices. Here, data is written word-wise to a single port.

Let $PhysIOOut(c_{\mathrm{CVM}}, prim, did, port)$ denote a call of this primitive with the following arguments:

- $c_{\mathrm{CVM}}$, a CVM configuration,
- $prim$, the $C0$ statement of the call, i.e. a statement of form

$$ESCall(vn, \texttt{PhysIOOut}, plist),$$

- a device id $did$ and a port $port$.

We assume that the variable, whose content we are to copy, is of an integer array type and that it is given through a pointer at the first position of the parameter list in $prim$, i.e. $plist!0$. Thus, we obtain the value $val$ of this variable by

$$val = the(rval(tn, c.m, Deref(plist!0))),$$

where the corresponding abstract type size defines the numbers of words to write to the device:

$$am = size_T(the(type(tn, toplst(c_{C0}.m), gst(c_{C0}.m), Deref(plist!0)))).$$

We build the appropriate device input

$$din_{int} = \left[\begin{array}{lcl} rep & = & True, \\ wr & = & True, \\ port & = & port, \\ data & = & c2i(val, am), \\ did & = & did \end{array}\right]$$

and use it in the step function for devices to obtain the output

$$(devs', dout_{int}, eifos) = \delta^{\mathrm{int}}_{devs}(din_{int}, c_{\mathrm{CVM}}.devs).$$

The execution of the primitive will fail, if one or more of the following conditions are violated:

- $did$ has to denote a valid device id, that is $did \in \mathbb{D}$, and
- $port$ has to be a valid port, i.e. $port < port_{max}$.

In the failure case, we define $PhysIOOut(c_{\mathrm{CVM}}, prim, did, port) = None$. Otherwise, we set

$$PhysIOOut(c_{\mathrm{CVM}}, prim, did, port) = \left\lfloor \left( c_{\mathrm{CVM}} \left[\begin{array}{l} devs := devs', \\ kern := upd_{ret}(c_{\mathrm{CVM}}.kern, 0) \end{array}\right], to_{\mathrm{eifo}}(did, eifos) \right) \right\rfloor$$

### PhysIOOutRange

As with the *PhysIOIn* and *PhysIOInRange* primitive, we define

$$PhysIOOutRange : confT_{\mathrm{CVM}} \times stmt \times \mathbb{N} \times \mathbb{N} \to (confT_{\mathrm{CVM}} \times (\mathbb{D} \times Eifo)\ list)_{\perp},$$

which will take the value of an abstract kernel variable and write it to one of the devices. Here, data is written word-wise to a range of ports.

Let $PhysIOOutRange(c_{\mathrm{CVM}}, prim, did, port)$ denote a call of this primitive with the following arguments:

- $c_{\mathrm{CVM}}$, a CVM configuration,
- $prim$, the $C0$ statement of the call, i.e. a statement of form

$$ESCall(vn, \texttt{PhysIOOutRange}, plist),$$

- a device id $did$ and a port $port$.

We require that the variable whose content we are to copy, is of an integer array type and that it is given through a pointer at the first position of the parameter list in $prim$, i.e. $plist!0$. Thus, we obtain the value $val$ of this variable by

$$val = the(rval(tn, c_{C0}.m, Deref(plist!0))),$$

where the corresponding abstract type size defines the numbers of words to write to the device:

$$am = size_T(the(type(tn, toplst(c_{C0}.m), gst(c_{C0}.m), Deref(plist!0)))).$$

We build the appropriate device input

$$din_{int} = \left[\begin{array}{l} rep = True, \\ wr = True, \\ port = port, \\ data = c2i(val, am), \\ did = did \end{array}\right]$$

and use it in the step function for devices to obtain the output

$$(devs', dout_{int}, eifos) = \delta^{\mathrm{int}}_{devs}(din_{int}, c_{\mathrm{CVM}}.devs).$$

The execution of the primitive will fail, if one or more of the following conditions are violated:

- $did$ has to denote a valid device id, that is $did \in \mathbb{D}$, and
- the last port, to which we are going to write, has to be a valid port, i.e. $port + am - 1 < port_{max}$.

In the failure case, we define $PhysIOOutRange(c_{\mathrm{CVM}}, prim, did, port) = None$. Otherwise, we set

$$PhysIOOutRange(c_{\mathrm{CVM}}, prim, did, port) = \left\lfloor \left( c_{\mathrm{CVM}} \left[\begin{array}{l} devs := devs', \\ kern := upd_{ret}(c_{\mathrm{CVM}}.kern, 0) \end{array}\right], to_{\mathrm{eifo}}(did, eifos) \right) \right\rfloor$$

### 6.3.6 Special Primitives

#### LoadOS

Since CVM is conceived as a microkernel programmers framework, it lacks crucial features most operating systems offer, e.g., scheduling. Thus, microkernels usually provide a mechanism that allows for loading an operating system *after* the kernel itself is running and install it as one of the kernels user processes. This operating system is then using the microkernel interface, i.e. the primitives, and its virtualization to realize the services offered to user processes running on top of the operating system, thus making the microkernel invisible for them.

In CVM, we define for this purpose the primitive

$$\mathit{LoadOS} : \mathit{confT}_{\mathrm{CVM}} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to (\mathit{confT}_{\mathrm{CVM}} \times (\mathbb{D} \times \mathit{Eifo})\, \mathit{list})_{\perp}.$$

A concrete call of this primitive $\mathit{LoadOS}(c_{\mathrm{CVM}}, am, pid, did, sp)$ takes the following arguments:

- $c_{\mathrm{CVM}}$, a CVM configuration,
- the number of pages $am$ to load,
- the process id $pid$ to whose memory the operating system will be loaded,
- from device $did$, and
- starting with page $sp$.

[Alk09] has formally defined a hard disk model, which relies on the ATA/AT-API standard. A hard disk configuration in this model is given by a record type with fields for sectors, buffer, and other components of a hard disk. The technical details of such a configuration as well as the corresponding transition function are not required to understand the semantics of the *LoadOS* primitive. Most importantly, there exists a word-addressable sector memory, given by the component $sm : \mathbb{N} \to \mathbb{Z}$, which represents the content of the disk.

The relevant part of the device content—the image—starts at list position $sa = sp \cdot 2^{10}$ and comprises $am$ many words, i.e.

$$data = \mathit{read}_{mm}(c_{\mathrm{CVM}}.\mathit{devs}(did).sm, sa, am).$$

Now, we copy the image into the destination user process starting at address 0 and obtain a new machine configuration $up'$ for this process defined as follows:

$$vm' = c_{\mathrm{CVM}}.up.\mathit{procs}(pid)[mm := \mathit{write}_{mm}(c_{\mathrm{CVM}}.up.\mathit{procs}(pid).mm, 0, data)].$$

Successful execution of the primitive call depends on meeting with the following requirements:

- $pid$ has to denote a valid process id, i.e. $pid \in \mathbb{P}$,
- the last address to which we write has to be within the memory of the user process: $?\mathit{inmem}(c_{\mathrm{CVM}}.up.\mathit{procs}(pid), am \cdot 2^{10})$, and
- the device $did$ has to be a hard disk, i.e. $\mathit{type}_{\mathrm{dev}}(c_{\mathrm{CVM}}.\mathit{devs}(did)) = hd$,

- which is big enough to hold the image, that is

$$|c_{\mathrm{CVM}}.devs(did).sm| \geq (sa + am) \cdot 2^{10} - 1.$$

In the failure case, we define $LoadOS(c_{\mathrm{CVM}}, am, pid, did, sp) = None$. Otherwise, we set

$$LoadOS(c_{\mathrm{CVM}}, am, pid, did, sp) = \left\lfloor \left( c_{\mathrm{CVM}} \left[ \begin{array}{l} up := c_{\mathrm{CVM}}.up[procs := c_{\mathrm{CVM}}.up.procs(pid := vm')], \\ kern := upd_{ret}(c_{\mathrm{CVM}}.kern, 0) \end{array} \right], [\,] \right) \right\rfloor$$

## 6.4 Initial CVM Configuration and $n$-Step Transition Function

**Definition 6.1 (Initial CVM Configuration)** To create an initial configuration, we define a function $init_{\mathrm{CVM}} : c0prog \times confT_{\mathrm{devs}} \to confT_{\mathrm{CVM}}$, where both the abstract kernel program $\Pi_{\mathrm{abs}}$ and the initial device configurations $devs$ are parameters of the CVM model. Then, we obtain the corresponding initial CVM configuration $c^0_{\mathrm{CVM}} = init_{\mathrm{CVM}}(\Pi_{\mathrm{abs}}, devs)$ as follows:

- We set the current process field $c^0_{\mathrm{CVM}}.up.cp$ to $None$, since we start with a kernel step.
- The interrupt mask $c^0_{CVM}.up.mask$ disables all maskable interrupts but the one of the timer device.
- The type name environment and procedure table of the kernel component $c^0_{CVM}.kern$ are set to the ones provided by $\Pi_{\mathrm{abs}}$. The initial configuration of the abstract kernel is obtained as described in Sect. 3.5.1 with the difference that we do use the abstract kernel dispatcher function instead of the main function.
- The device configurations in $c^0_{\mathrm{CVM}}.devs$ are taken from $devs$. CVM assumes an already established memory virtualization. The hard disk, which is used by the page fault handler (cf. Sect. 6.5.1), is thus not visible any longer. Instead, we replace this device with a dummy (idle) device.
- The user processes are initialized as described above with the CVM primitive $reset$ (cf. Sect. 6.3).

**Definition 6.2 ($n$-Step CVM Transition Function)** For a whole run, i.e. several steps of the CVM model, we introduce the $n$-step transition function

$$\delta^n_{\mathrm{CVM}} : confT_{\mathrm{CVM}} \times (\mathbb{N} \to (\mathbb{D} \times Eifi)_\perp) \to (confT_{\mathrm{CVM}} \times (\mathbb{D} \times Eifo)\, list)_\perp.$$

It takes a start configuration and a sequence of events as inputs and returns the resulting state after $n$ steps together with the external output generated and accumulated during the computation. If any of the steps leads to a run-time error, the whole computation yields $None$.

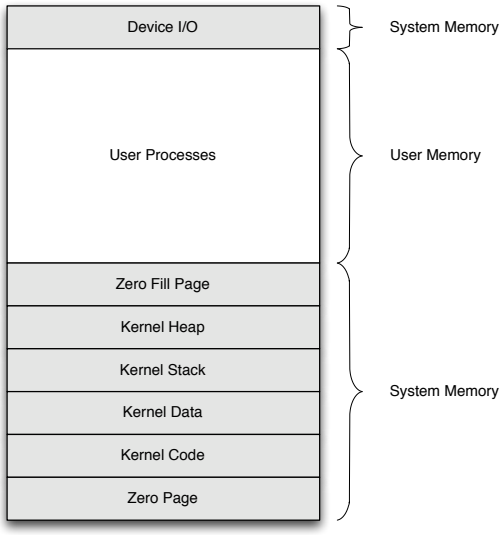


Figure 6.2: CVM Memory Map

## 6.5 CVM Implementation and Abstract Linking

As we have seen above, the abstract kernel is merely a 'crippled' C0 program, since it lacks all necessary low-level and hardware-dependent implementations. In order to obtain a compilable—and thus executable—kernel, we have to add code parts to a given abstract kernel program $\Pi_k$. We call these code parts the *CVM implementation* or short $\Pi_{\text{CVM}}$.

In the following subsection, we are going to present the necessary additions. A convenient way of merging the two programs is *abstract linking*. We describe this formalism in the second half of the section.

### 6.5.1 CVM Implementation

#### Data Structures

To simulate virtual machines and multi-processing, the CVM implementation has to maintain certain data structures (cf. Fig. 6.2):

1. In the kernel global memory (or *kernel data*), we store an array `pcb[]` of so-called *process control blocks (PCBs)*. For each user processes with id $pid \in \mathbb{P}$, the process control block `pcb[`$pid$`]` is a structure type with components to store the register files and the program counters of the corresponding user process.

2. The current process field $c_{\text{CVM}}.up.cp$ is stored in a global variable `cp` of type $Unsgnd_T$, the interrupt mask correspondingly in a global variable `mask` of the same type. Unlike $c_{\text{CVM}}.up.cp$, which is set to *None*, we leave

the `cp` variable unchanged on entering system mode[3].

3. Certain general purpose registers are used by the compiler to store the stack and heap pointers [Lei08, LP08]. When we leave kernel execution and start user execution, the corresponding registers of the kernel are overwritten and are thus lost, when we re-enter the kernel. This causes no problem for the stack pointer, since we re-initialize the kernel stack each time on entering kernel mode. Yet for the heap, we have to keep track of the already allocated data structures in order not to overwrite them. Thus, we declare a global variable `kheap`, in which we store the end address of the kernel heap when leaving kernel mode, and which we use to set the appropriate register on entering.

4. We have described the address translation mechanism of the VAMP in Sect. 5.1. The required page tables for each user process are stored in an array `ptspace[]` on the heap.

5. Some additional data structures required by the page fault handler to manage the physical and swap memory.

### Entering and Leaving System Mode

Entering system mode basically consists of three steps. Whenever the kernel starts to execute, we initialize its program rest with the function `init`. In all cases but *reset*, `init` will take the current user process and store its registers into the corresponding process control block `pcb[cp]` (*kernel save*).

In the next step, the CVM dispatcher `cvmdispatch` is called with parameters `pcb[cp].eca` (exception cause), `pcb[cp].edpc` (exception dpc), and `pcb[cp].edata` (exception data). Since our kernel is non-preemptive and thus should not be interrupted, we zero the status register of the hardware, i.e. mask all maskable interrupts.

In case of a page fault interrupt, `cvmdispatch` will call the page fault handler. Otherwise, we continue with a call of the abstract kernel dispatcher `abs_kernel_dispatch` with parameters `pcb[cp].eca` and `pcb[cp].edata`.

To leave system mode in order to start the user process with id $pid \in \mathbb{P}$, we set `cp=`$i$ and restore the process' register files and program counters using its process control block `pcb[`*pid*`]` (*kernel restore*). Finally, we leave kernel execution with a `rfe` (return from exception) instruction.

Note that a scheduler is not part of the low-level implementation. This means that the abstract kernel has to provide for an appropriate handling of timer interrupts.

### Page Fault Handler

In Sect. 5.1.1, we have introduced the hardware part of address translation. On the software side, page faults triggered by the hardware are handled by a piece of software called *page fault handler*.

[3]This is also the reason why do not relate these two values in the CVM correctness statement during kernel execution (cf. Sect. 7.2.6).

Whenever a user process step raises such an interrupt, the page fault handler is invoked. Depending on the actual reason for the interrupt, the page fault handler

- moves pages from physical to swap memory (if the physical memory is full),
- copies the accessed page from swap memory into physical memory,
- updates the translation function.

Besides the page faults during user steps, the page fault handler might also become involved in certain CVM primitives that access user memories, for instance *copy*.

Certain data structures are required by the page fault handler in order to implement these services.

1. The *page tables* (cf. Sect. 5.1.1) are both shared by the hardware and the page fault handler.
2. Swap memory is organized in larger chunks of memory of size $2^{20}$ words, called *big pages*. We maintain a *big page table* to organize these big pages and to retrieve data from the swap memory.
3. Since the page fault handler is invoked by the kernel, it also has to access the process control blocks in order to update register values of user processes (e.g., the page table length register on freeing or allocating of user memory).
4. Finally, there are several lists for the bookkeeping of free and used pages in user memory, exclusively used by the page fault handler for instance to implement page replacement strategies.

The detailed discussion of our page fault handler implementation and correctness is out of the scope of this work, but is presented in [ASS08, AHL+09, Sta09].

Basically, any implementation that satisfies the virtualization correctness criteria (cf. Sect. 7.2.1) can be used. The page fault handler we use in Verisoft is a *C0* implementation with small assembler parts used to access the swap hard disk [Con06].

### CVM Primitives Implementation

As we have seen in Sect. 6.3, some of these primitives manipulate just the registers of a process, that is the process control block of that process in the implementation.

Others, for instance those primitives that copy memory from one process to another or between processes and devices, are accessing data, which is not visible in *C0* structures of the kernel—though they might be visible in those of the corresponding user processes. In these cases, we work with assembler code in-lined into *C0* code using the *Asm* statement (cf. Sect. 3.2.3).

In [ST08, Tsy09], implementation and code verification of primitives with in-line assembler code is discussed in detail.

### 6.5.2 Abstract Linking

As we have seen in the section before, we have to put together the abstract kernel program and the CVM implementation to obtain a complete concrete kernel program, which can be compiled and executed. As we have seen in Sect. 3.2, a $C0$ program consists of a *type name environment*, a *procedure table*, and a *global symbol table*.

We denote the two programs with $\Pi_{\text{abs}} = (te_{\text{abs}}, pt_{\text{abs}}, gst_{\text{abs}})$ and $\Pi_{\text{CVM}} = (te_{\text{CVM}}, pt_{\text{CVM}}, gst_{\text{CVM}})$, abbreviating the type of a $C0$ program with $c0prog = tenv \times proctableT \times (\Sigma^+ \times ty)\, list$.

We will now define *abstract linking*, which is a formal way to combine the two implementations. In principle, we have do to the following: we merge the type name environments, $te_k$ and $te_{\text{CVM}}$, and the global symbol tables, $gst_k$ and $gst_{\text{CVM}}$, of the two programs. Then, we scan the abstract kernel implementation for external function calls to CVM primitives and replace them with ordinary function calls. Finally, we merge the two procedure tables.

Not all programs are linkable in the sense that the linked code is compilable or even runnable. The preconditions under which abstract linking is possible, will be introduced in Sect. 7.2.2 when we talk about CVM correctness.

**Definition 6.3 (Merging Type Name Environments)** Type name environments have been introduced in Sect. 3.2 as lists of pairs of a type name and a type, i.e. $tenv : (\Sigma^+ \times ty)\, list$. We merge two type name environments by appending them while taking care not to create redundant list elements.

We define a function $link_{\text{te}} : tenv \times tenv \to tenv$ and set:

$$\begin{aligned} link_{\text{te}}(ta, [\,]) &= ta \\ link_{\text{te}}(ta, x\#tb) &= \begin{cases} link_{\text{te}}(ta, tb) & \text{if } x \in ta \\ link_{\text{te}}(x\#ta, tb) & \text{otherwise} \end{cases} \end{aligned}$$

**Definition 6.4 (Merging Global Symbol Tables)** Information about the global variables of a program is stored in a symbol table, which is a list of variable names and their associated type: $(\Sigma^+ \times ty)\, list$. The function $link_{\text{gst}} : (\Sigma^+ \times ty)\, list \times (\Sigma^+ \times ty)\, list \to (\Sigma^+ \times ty)\, list$ merely appends one list to the other, so we set for two symbol tables $gst_a$ and $gst_b$:

$$link_{\text{gst}}(gst_a, gst_b) = gst_a @ gst_b.$$

#### Procedure Tables

We remember: CVM primitives in the abstract kernel program are *declared*, but not *defined*, that is the corresponding procedure body consists of a single *Skip* statement and there are no local variables declared. We call these declared-only procedures *external*.

The implementation for this procedures comes from the procedure table $pt_{\text{CVM}}$ of the CVM implementation. On the other hand, there are external procedures in $\Pi_{\text{CVM}}$, whose implementation comes from the abstract kernel—for instance the abstract kernel dispatcher.

The linking of two procedure tables therefore consists of several steps:

1. In order to avoid duplicate statement ids, we renumber all non-structural statements in both procedure tables to regain uniqueness.
2. Then, we split both procedure tables in two parts: one containing the 'normal' *defined* procedures, and one that contains the external ones.
3. The parts with the defined procedures are then simply concatenated.
4. The external procedures are scanned for entries that are defined in the other program to be linked. These entries will then be removed. We do this for both programs, respectively.
5. A potential rest of external functions is then appended to the already linked defined procedures.
6. All function bodies have to be searched for *ESCall* statements. If the functions called are now defined, we change the statement to a conventional *SCall* statement.

**Definition 6.5 (Renumbering)** The compiler correctness proof in [Lei08] requires a unique numbering for all statements in a $C0$ program. We introduce a very simple way to achieve this unique numbering for two procedure tables to be merged: in one procedure table, we multiply all statement ids with 2, hence obtaining only *even* numbered statements. In the other procedure table we multiply all ids with 2 and add 1, which yields only odd numbered statements.

For this purpose, we define two auxiliary functions $odd_{\text{stmt}} : stmt \to stmt$ and $even_{\text{stmt}} : stmt \to stmt$, and define recursively:

$$\begin{aligned}
even_{\text{stmt}}(Skip) &= Skip \\
even_{\text{stmt}}(Comp(s_1, s_2)) &= Comp(even_{\text{stmt}}(s_1), even_{\text{stmt}}(s_2)) \\
even_{\text{stmt}}(Ass(e_1, e_2, i)) &= Ass(e_1, e_2, 2 \cdot i) \\
even_{\text{stmt}}(Ass_c(e, l, i)) &= Ass_c(e, l, 2 \cdot i) \\
even_{\text{stmt}}(PAlloc(e, tn, i)) &= PAlloc(e, tn, 2 \cdot i) \\
even_{\text{stmt}}(SCall(vn, fn, pl, i)) &= SCall(vn, fn, pl, 2 \cdot i) \\
even_{\text{stmt}}(Return(e, i)) &= Return(e, 2 \cdot i) \\
even_{\text{stmt}}(Ifte(e, s_1, s_2, i)) &= Ifte(e, even_{\text{stmt}}(s_1), even_{\text{stmt}}(s_2), 2 \cdot i) \\
even_{\text{stmt}}(Loop(e, s, i)) &= Loop(e, even_{\text{stmt}}(s), 2 \cdot i) \\
even_{\text{stmt}}(Ass(il, i)) &= Ass(il, 2 \cdot i) \\
even_{\text{stmt}}(XCall(fn, rl, pl, i)) &= XCall(fn, rl, pl, 2 \cdot i) \\
even_{\text{stmt}}(ESCall(vn, fn, pl, i)) &= ESCall(vn, fn, pl, 2 \cdot i)
\end{aligned}$$

and for $odd_{\text{stmt}}$ likewise.

Then, we obtain a renumbered procedure table by applying one of the above functions to all of its function bodies, formally

$$map(even_{\text{stmt}}, pt),$$

and for odd statement ids

$$map(odd_{\text{stmt}}, pt),$$

**Definition 6.6 (External Procedures)** Given a $C0$ procedure table $pt \in proctableT$ (cf. Sect. 3.2.4). Such a procedure table consists of pairs of function names and their descriptors. We say a procedure table entry $(fn, fdesc)$ is *external* if

$$fdesc.lvars = [\,] \wedge fdesc.body = Skip$$

and denote this with the predicate $?ext : \Sigma^+ \times procT \to \mathbb{B}$.

The list of external procedures of $pt$ is then given through

$$pt^{\text{ext}} = filter(?ext, pt)$$

and, correspondingly, the list of internal procedures by

$$pt^{\text{def}} = filter(\neg ?ext, pt).$$

**Definition 6.7 (Removing External Procedures)** For two procedure tables $pta$ and $ptb$, we want to remove the external functions from $ptb$, which are defined in $pta$. The comparison has to rely on the function name, since the associated descriptors do obviously not match. We define the function $rem : proctableT \times proctableT \to proctableT$ that returns the remainder of the external procedures in $ptb$. We set:

$$rem(pta, ptb) = filter((\lambda x.\, fst(x) \notin pta^{\text{def}}), ptb^{\text{ext}}).$$

**Definition 6.8 (Merging Procedure Tables)** As mentioned further above, the merged procedure table of $pta$ and $ptb$ consists of the defined functions of both procedure tables and those procedures that remain external.

For this purpose, we define a function $link_{\text{pt}} : proctableT \times proctableT \to proctableT$. Given two procedure tables $pta$ and $ptb$, we first renumber the function bodies obtaining $pta_r = even(pta)$ and $ptb_r = odd(ptb)$. Then we merge the renumbered procedure tables as follows:

$$link_{\text{pt}}(pta, ptb) = pta_r^{\text{def}} @ ptb_r^{\text{def}} @ rem(pta_r, ptb_r) @ rem(ptb_r, pta_r).$$

### Updating Procedure Bodies

The last step in abstract linking is the update of the external function call statements in the procedure bodies of the linked procedure tables.

**Definition 6.9 (Updating a Statement Tree)** Procedure bodies are made up by statement trees, so we have to introduce a function that walks these trees recursively and replaces external function calls where necessary.

For a single external function call, we check if the function called is still external or if it is defined now. In the first case, we leave the statement as it is, in the latter case we replace it with a conventional function call statement.

For most statements, this function computes the identity. Thus, we omit these cases and define only the interesting ones here.

Given a procedure table $pt$. For $e2i : proctableT \times stmt \to stmt$, we set:

$$\begin{aligned} e2i(pt, Comp(s_1, s_2)) &= Comp(e2i(pt, s_1), e2i(pt, s_2)) \\ e2i(pt, Ifte(e, s_1, s_2)) &= Ifte(e, e2i(pt, s_1), e2i(pt, s_2)) \\ e2i(pt, Loop(e, s) &= Loop(e, e2i(pt, s)) \end{aligned}$$

and for external function calls

$$e2i(pt, ESCall(vn, fn, plist)) = \begin{cases} ESCall(vn, fn, plist) & \text{if } fn \in \{map(fst, pt^{\text{ext}})\} \\ SCall(vn, fn, plist) & \text{otherwise} \end{cases}$$

**Definition 6.10 (Updating a Procedure Table)** An element $(fn, f)$ of a procedure table can now be updated using the function introduced above. We define $repl : proctableT \times (\Sigma^+ \times procT) \to \Sigma^+ \times procT$ and set:

$$repl(pt, (fn, f)) = (fn, f[body := e2i(pt, f.body)]).$$

To replace all outdated external function calls in one procedure table, we simply apply this function with *map*:

$$repl_{\text{pt}}(pt) = map(repl\, pt).$$

**Definition 6.11 (Abstract Linking)** We combine all of the functions introduced in this section to define the overall abstract linking function

$$link : c0prog \times c0prog \to c0prog.$$

Given two programs $\Pi_a = (te_a, gst_a, pt_a)$ and $\Pi_b = (te_b, gst_b, pt_b)$, we set

$$link(\Pi_a, \Pi_b) = \begin{pmatrix} link_{\text{te}}(te_a, te_b), \\ link_{\text{gst}}(gst_a, gst_b), \\ repl_{\text{pt}}(link_{\text{pt}}(pt_a, pt_b)) \end{pmatrix}$$

# Part II

# CVM Correctness

Crash programs fail because they are based on theory that, with nine women pregnant, you can get a baby a month.

Werner von Braun

# Chapter 7

# CVM Implementation Correctness

**Contents**



Overall CVM correctness is a classical simulation theorem, where a VAMP instruction set architecture with devices (cf. Sect. 5.2) simulates CVM as introduced in Chapter 6. Single states of the two computational models are related by an abstraction relation, which subsumes several other abstraction relations for the single components of a CVM configuration (cf. Sect. 6.1).

The rest of this chapter is outlined as follows: at first, we introduce the major individual abstraction relations (Sect. 7.2). Particular attention will be given to the relation between the abstract and the concrete kernel (Sect. 7.2.2) and the user process relation (Sect. 7.2.1).

Most of them rely on other work, for instance compiler correctness [Lei08, Pet07], page fault handler correctness [Sta09, Alk09], and low-level CVM code correctness [Tsy09]. We will provide references for those as necessary, but will not discuss them in detail again.

Then, we combine the single relations to the overall abstraction relation, which we will use to formulate the overall CVM implementation correctness theorem (Sect. 7.3).

## 7.1 Auxiliary Functions

Frequently, we will access the memory regions, where variables of the concrete kernel are stored. For global variables, these addresses are fixed and can be statically computed using the compiler allocation function as introduced in [Lei08] (see also Fig. 6.2). We write short $ad_{c_{\mathrm{i\&d}}}(\sigma)$ for the corresponding base address of a $C0$ variable $\sigma$ in an ISA configuration $c_{\mathrm{i\&d}}$, which is assigned by the compiler.

For the corresponding integer and natural interpretation of the bit vector stored at the word address $ad_{c_{\mathrm{i\&d}}}(\sigma)$, we use the notation $\langle\sigma\rangle^{int}$ and $\langle\sigma\rangle^{nat}$,

respectively, and define:

$$\langle\sigma\rangle^{int}_{c_{\text{i\&d}}} = to_{\text{int}}(c_{\text{i\&d}}.proc.mm(ad_{c_{\text{i\&d}}}(\sigma)))$$
$$\langle\sigma\rangle^{nat}_{c_{\text{i\&d}}} = to_{\text{nat}}(c_{\text{i\&d}}.proc.mm(ad_{c_{\text{i\&d}}}(\sigma)))$$

## 7.2 Abstraction Relations

In this section, we will introduce the individual abstraction relations between the various components of CVM and the hardware, the devices, and the concrete kernel, respectively.

We start with $B$, a function relating the CVM user process component to a VAMP ISA configuration.

In a next step, we will treat the relations dealing with the CVM kernel component. Of particular relevance for the rest of this work is *konsis*, the abstraction relation between the abstract and the concrete kernel.

Then, we will shortly introduce the—rather trivial—relation between the devices in CVM and those on the hardware layer.

Before we will combine all individual abstraction relations into a single comprehensive one, we will introduce the relations for the current process identifier and the interrupt mask.

### 7.2.1 User Process Relation

We remember that the user process component *up* of a CVM configuration $c_{\text{CVM}}$ contains a mapping of process ids to assembly machines encoding user processes.

The user process relation

$$B : (\mathbb{P} \to confT_{\text{asm}}) \times confT_{\text{i\&d}} \to \mathbb{B}$$

relates this user process component to its representation in a VAMP ISA configuration.

To define the $B$ relation, we define auxiliary functions to extract a user process configuration for a process with id *pid* from a given VAMP ISA with devices configuration $c_{\text{i\&d}}$. Whenever a user process is not active—i.e. the kernel or another user process runs—, its registers are stored in the process control blocks of the concrete kernel (cf. Sect. 6.5.1). The memory of a user process is distributed over the physical memory $c_{\text{i\&d}}.proc.mm$ of the processor and the swap memory of the swap hard disk in $c_{\text{i\&d}}.devs$. Thus, we split this task into two functions, one to extract the process' memory, one to obtain its register file.

The so obtained virtual machine is then compared to the corresponding one stored in the user process component $c_{\text{CVM}}.up.procs(pid)$.

#### Extracting the Register Files

We consider two cases:

1. If the current process is not *pid* or if we are in system mode, we use the process control blocks to obtain the register files and the program counters.
2. Otherwise—i.e. process *pid* is running—, we take the registers and counters directly from the processor component of $c_{\text{i\&d}}$.

Let $x$ denote one of the program counters, that is either *pcp* or *dpc*. We define a function $get_x$ that takes a combined ISA and device configuration, a process id and returns the program counter $x$.

$$get_x(c_{\text{i\&d}}, pid) = \begin{cases} c_{\text{i\&d}}.proc.x & \text{if } pid = \langle \texttt{cp} \rangle^{nat}_{c_{\text{i\&d}}} \\ \langle pcb[pid].x \rangle^{nat}_{c_{\text{i\&d}}} & \text{otherwise} \end{cases}$$

Register files are represented in the assembly semantics as a list of integers. To construct a list of 32 values for a register file $x$ from a processor configuration, we define:

$$\begin{aligned} r2l_x(c_{\text{i\&d}}) \quad &= \quad to_{\text{int}}(c_{\text{i\&d}}.proc.x(0)) \\ &\# \quad to_{\text{int}}(c_{\text{i\&d}}.proc.x(1)) \\ &\# \quad \ldots \\ &\# \quad to_{\text{int}}(c_{\text{i\&d}}.proc.x(31)) \end{aligned}$$

Similarly, we define $p2l_x$ to extract the register file $x$ from the process control blocks of a process *pid* in a configuration $c_{\text{i\&d}}$.

$$\begin{aligned} p2l_x(c_{\text{i\&d}}, pid) \quad &= \quad \langle \texttt{pcb}[pid].x[0] \rangle^{int}_{c_{\text{i\&d}}} \\ &\# \quad \langle \texttt{pcb}[pid].x[1] \rangle^{int}_{c_{\text{i\&d}}} \\ &\# \quad \ldots \\ &\# \quad \langle \texttt{pcb}[pid].x[31] \rangle^{int}_{c_{\text{i\&d}}} \end{aligned}$$

We combine the above functions to extract the register set $x \in \{gprs, sprs\}$ and set:

$$get_x(c_{\text{i\&d}}, pid) = \begin{cases} r2l_x(c_{\text{i\&d}}) & \text{if } pid = \langle \texttt{cp} \rangle^{nat}_{c_{\text{i\&d}}} \\ p2l_x(c_{\text{i\&d}}, pid) & \text{otherwise} \end{cases}$$

**Definition 7.1 (Register Extraction)** We can now define the function

$$get_{\text{regs}} : confT_{\text{i\&d}} \times \mathbb{N} \to confT_{\text{asm}}$$

that takes a process id and a ISA and device configuration and returns an assembly configuration for the corresponding process as encoded in the hardware. We set:

$$get_{\text{regs}}(c_{\text{i\&d}}, pid) = \begin{bmatrix} pcp = get_{\text{pcp}}(c_{\text{i\&d}}, pid), \\ dpc = get_{\text{dpc}}(c_{\text{i\&d}}, pid), \\ gprs = get_{\text{gprs}}(c_{\text{i\&d}}, pid), \\ sprs = get_{\text{sprs}}(c_{\text{i\&d}}, pid), \\ mm = undef \end{bmatrix}$$

Note, that we leave the memory undefined for now. We will define the extraction of the virtual memory from physical and swap memory in the next paragraph.

### Extracting the Memory

As we have seen in Sect. 5.1.1, the *valid bit* of a page table entry determines, if the corresponding page is present in the physical memory or not. Note, that the memory at ISA level is byte-addressed, while it is word-addressed on assembly level. Therefore, the following definitions vary slightly from the ones introduced in Sect. 5.1.1, e.g., there is no byte index for a virtual address, but a word index.

**Definition 7.2 (Page Table Entry Extraction)** For a ISA and device configuration $c_{\text{i\&d}}$ and a process id $pid$, we extract the *page table entry* for a virtual address $va$ using the function $pte(c_{\text{i\&d}}, pid, va)$. In addition, let $va'$ denote the bit vector representation for $va$. Hence, we write $px(va') = va'[29:10]$ for the virtual page index and $wx(va') = va'[9:0]$ for the word index of $va'$.

We abbreviate $vm = get_{\text{regs}}(c_{\text{i\&d}}, pid)$ and compute the page table entry as

$$pte(c_{\text{i\&d}}, pid, va) = c_{\text{i\&d}}.proc.mm(vm.sprs!pto \cdot 2^{10} + px(va')).$$

The predicate $?val_{\text{pte}}$ takes an address $va$, a configuration $c_{\text{i\&d}}$, and a process id $pid$. It evaluates to *True*, if the corresponding page for $va$ is in physical memory and *False*, if it is in the swap memory. We set:

$$?val_{\text{pte}}(c_{\text{i\&d}}, pid, va) = \begin{cases} True & \text{if } pte(c_{\text{i\&d}}, pid, va)[11] = 1 \\ False & \text{otherwise} \end{cases}$$

For $?val_{\text{pte}}(c_{\text{i\&d}}, pid, va)$, we obtain the corresponding physical word address $pa(c_{\text{i\&d}}, pid, va) = [pte(c_{\text{i\&d}}, pid, va)[29:10]; wx(va')]$.

The swap memory is located on a hard disk present in the device part $c_{\text{i\&d}}.devs$ of the combined configuration. Address translation for the swap device—i.e. relating virtual addresses to sectors on the hard disk—works obviously differently from the address translation introduced in Sect. 5.1.1 and is not part of the hardware, but the page fault handler. We omit the details of address translation for the swap memory, which can be found in [ASS08] and [Sta09]. Instead, we use the function $sa(c_{\text{i\&d}}, pid, va)$, which yields the swap memory address for a virtual address with $\neg?val_{\text{pte}}(c_{\text{i\&d}}, va)$, in an uninterpreted way. Furthermore, we assume the existence of a swap device with id $did_{\text{swap}}$, a hard disk with a component $sm$, the swap memory.

**Definition 7.3 (Memory Extraction)** Now we can extract a virtual memory from the physical and swap memory. We define $get_{mm} : confT_{\text{i\&d}} \times \mathbb{N} \to (\mathbb{N} \to \mathbb{Z})$ and set

$$get_{mm}(c_{\text{i\&d}}, pid) = \lambda va.to_{\text{int}} \begin{cases} c_{\text{i\&d}}.proc.mm(pa(c_{\text{i\&d}}, pid, va)) & \text{if } ?val_{\text{pte}}(c_{\text{i\&d}}, pid, va) \\ c_{\text{i\&d}}.devs.did_{\text{swap}}.sm(sa(c_{\text{i\&d}}, pid, va)) & \text{otherwise} \end{cases}$$

**Definition 7.4 (Equivalence of Assembly Configurations)** We say two virtual machine configurations are equivalent, iff

- both program counters are equal,

- both register files are equal,
- the content of the two memories are equal.

We express this formally using the predicate $\cong$ and define for two assembly configurations $vm_1$ and $vm_2$:

$$\begin{aligned} vm_1 \cong vm_2 \quad = \quad & vm_1.dpc = vm_2.dpc \\ \wedge \quad & vm_1.pcp = vm_2.pcp \\ \wedge \quad & (0 \leq i \leq 31 \longrightarrow vm_1.gprs!i = vm_2, gprs!i) \\ \wedge \quad & (0 \leq i \leq 31 \longrightarrow vm_1.sprs!i = vm_2.sprs!i) \\ \wedge \quad & (i < (vm_1.sprs.ptl + 1) \cdot 2^{10} \longrightarrow vm_1.mm(i) = vm_2.mm(i)) \end{aligned}$$

**Definition 7.5 (*B*-Relation)** Combining both the extraction functions for registers and memory, we obtain a virtual machine $vm_{pid}$ of process *pid* from a given ISA and device configuration $c_{\text{i\&d}}$ as follows:

$$vm_{pid}(c_{\text{i\&d}}) = get_{\text{regs}}(c_{\text{i\&d}}, pid)[mm := get_{mm}(c_{\text{i\&d}}, pid)].$$

We say, the user process relation $B$ holds for a user process component $procs : \mathbb{P} \rightarrow confT_{\text{asm}}$ and a combined configuration $c_{\text{i\&d}} \in confT_{\text{i\&d}}$, iff all user process stored in *procs* are equal under the equivalence relation defined further above to the virtual machines extracted from the hardware, formally:

$$B(procs, c_{\text{i\&d}}) = (\forall pid \in \mathbb{P} : procs(pid) \cong vm_{pid}(c_{\text{i\&d}})).$$

### 7.2.2 Kernel Relations

#### Abstract Kernel to Concrete Kernel

The relation between the abstract kernel and the concrete kernel is the most complex one. In order to make it more understandable, we have split it up in several smaller relations, each relating specific parts of the abstract kernel to their counterparts in the concrete kernel. In the rest of this section, we will introduce these sub-relations and combine them finally to the *konsis* relation, which connects the abstract to the concrete kernel.

**Definition 7.6 (Type Name Environment Consistency)** Let $te_a$ denote a type name environment. Then, $te_a$ is *consistent* to another type name environment $te_b$, iff all elements in $te_a$ are also elements in $te_b$. We define $konsis_{\text{tenvs}} : tenv \times tenv \rightarrow \mathbb{B}$ and set

$$konsis_{\text{tenv}}(te_a, te_b) = \{te_a\} \subseteq \{te_b\}.$$

**Definition 7.7 (Relating Recursion Depths)** Compared to the abstract kernel, the concrete kernel has a certain computational overhead in the sense, that there have already happened some function calls before the call of the abstract kernel dispatcher.

For instance, the dispatcher function, which calls the page fault handler if necessary, is only present in the concrete kernel, but not in the abstract one, where page faults are not visible at all.

This means that the local memory stack of the concrete kernel has necessarily more stack frames than the one of the abstract kernel, in other words: the recursion depth of the first one is larger by a constant offset than the latter one.

We name this offset $rd_{\text{off}}$ and describe the consistency of two local memory recursion depths by a predicate $konsis_{\text{rd}} : memconfT \times memconfT \to \mathbb{B}$. Given two memory configurations $m_a$ and $m_b$, we define:

$$konsis_{\text{rd}}(m_a, m_b) = (|m_b.lm| = |m_a.lm| + rd_{\text{off}}).$$

We have to define a relation which connects statements in the abstract kernel with statements in the concrete kernel. This cannot just be ordinary equality, since the concrete kernel program has been created by abstract linking (cf. Sect. 6.5.2), that is:

- all statements have undergone renumbering, and
- external function calls have been replaced by conventional function calls.

**Definition 7.8 (Statement Equivalence)** We introduce the equivalence relation for statements $eq_{\text{stmt}} : stmt \times stmt \to \mathbb{B}$ and define recursively:

$$\begin{aligned}
eq_{\text{stmt}}(Skip, s) &= (s = Skip)\\
eq_{\text{stmt}}(Ass(e_1, e_2, i), s) &= (\exists j : s = Ass(e_1, e_2, j))\\
eq_{\text{stmt}}(PAlloc(e, tn, i), s) &= (\exists j : s = PAlloc(e, tn, j))\\
eq_{\text{stmt}}(SCall(vn, fn, plist, i), s) &= (\exists j : s = SCall(vn, fn, plist, j))\\
eq_{\text{stmt}}(ESCall(vn, fn, plist, i), s) &= (\exists j : s = SCall(vn, fn, plist, j))\\
eq_{\text{stmt}}(XCall(fn, elist_1, elist_2, i), s) &= (\exists j : s = XCall(fn, elist_1, elist_2, j))\\
eq_{\text{stmt}}(Asm(il, i), s) &= (\exists j : s = Asm(il, j))\\
eq_{\text{stmt}}(Return(e, i), s) &= (\exists j : s = Return(e, j))\\
eq_{\text{stmt}}(Loop(e, t, i), s) &= (\exists j, u : eq_{\text{stmt}}(t, u) \wedge (s = Loop(e, u, j)))\\
eq_{\text{stmt}}(Ifte(e, t_1, t_2, i), s) &= (\exists j, u_1, u_2 : eq_{\text{stmt}}(t_1, u_1)\\
&\quad \wedge eq_{\text{stmt}}(t_2, u_2) \wedge (s = Ifte(e, u_1, u_2, j)))\\
eq_{\text{stmt}}(Comp(t_1, t_2), s) &= (\exists s_1, s_2 : eq_{\text{stmt}}(t_1, s_1) \wedge eq_{\text{stmt}}(t_2, s_2)\\
&\quad \wedge (s = Comp(s_1, s_2)))
\end{aligned}$$

We will now use the $eq_{\text{stmt}}$ relation to reason about the defined functions in the abstract and the concrete kernel. For all *defined* functions in the abstract kernel's procedure table and the corresponding functions in the concrete kernel, we require that

- the statements in the function bodies are equal under the above relation, and
- the function parameters and local variables are the same.

**Definition 7.9 (Relating Procedure Tables)** Given two procedure tables $pt_a$ and $pt_b$. Let $(fn, f)$ denote an entry in the procedure table $pt_a$ and $(fn, f')$ its corresponding entry in $pt_b$ with $f' = the(mapof(pt_b, fn))$.

We say $(fn, f)$ and $(fn, f')$ are equivalent under a predicate $konsis_{\text{func}} : (\Sigma^+ \times procT) \times (\Sigma^+ \times procT) \to \mathbb{B}$, iff:

$$\begin{aligned} &eq_{\text{stmt}}(f.body, f'.body) \wedge \\ &(f.params = f'.params) \wedge \\ &(f.lvars = f'.lvars) \end{aligned}$$

We can now connect the two procedure tables $pt_a$ and $pt_b$ using the above predicate:

$$konsis_{\text{pt}}(pt_a, pt_b) = (\forall (fn, f) \in \{pt_a^{\text{def}}\}.\exists (fn, f') \in \{pt_b\}.konsis_{\text{func}}(f, f')).$$

The predicates defined above are dealing with $C0$ programs, thus dealing with static properties. We will now introduce several predicates connecting configurations of the abstract kernel with those of the concrete kernel. We start with the program rest of a $C0$ configuration.

**Definition 7.10 (Program Rest Consistency)** For two program rests $pr_a$ and $pr_b$, we define the equivalence predicate $konsis_{\text{prog}} : stmt \times stmt \to \mathbb{B}$ and set:

$$konsis_{\text{prog}} = (\forall i < |s2l(pr_a)| : eq_{\text{stmt}}(s2l(pr_a)!i), s2l(pr_b)!i).$$

We have to relate g-variables (cf. Sect. 3.3.1) of the abstract kernel to corresponding g-variables in the concrete kernel. This is pretty easy for global variables, which have identical names in the concrete kernel. For local variables of the abstract kernel, we find the corresponding local variables by adding the offset $rd_{\text{off}}$ to the local memory frame identifier.

With heap variables, things become a little more tricky. Since the concrete kernel can allocate new memory independently from the abstract kernel, we have to keep track, which abstract heap variables are connected to which concrete heap variables. We realize this bookkeeping by introducing a heap map function $hp : \mathbb{N} \to \mathbb{N}$ (cf. Fig. 7.1).

**Definition 7.11 (Corresponding g-Variables)** We introduce an allocation function $kalloc : gvar \times (\mathbb{N} \to \mathbb{N}) \to gvar$, which takes a g-variable of the abstract kernel and a heap map function, returning the corresponding g-variable in the concrete kernel. We define inductively:

$$\begin{aligned} kalloc(gvar_{gm}(\sigma), hp) &= gvar_{gm}(\sigma) \\ kalloc(gvar_{lm}(i, \sigma), hp) &= gvar_{lm}(i + rd_{\text{off}}, \sigma) \\ kalloc(gvar_{hm}(j), hp) &= gvar_{hm}(hp(j)) \\ kalloc(gvar_{arr}(a, i), hp) &= gvar_{arr}(kalloc(a, hp), i) \\ kalloc(gvar_{str}(st, cn), hp) &= gvar_{str}(kalloc(st, hp), cn) \end{aligned}$$

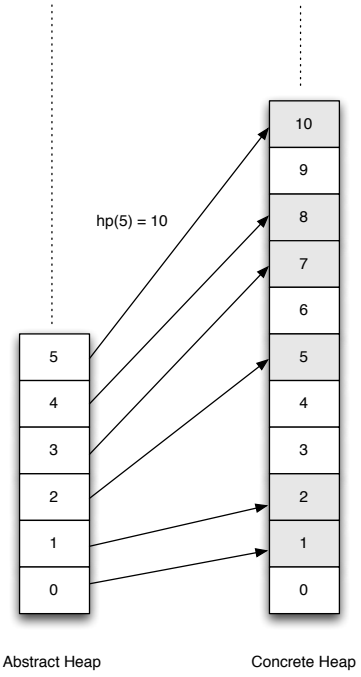


Figure 7.1: Mapping Abstract to Concrete Heap Variables

In the following sections, we will analyze and relate the initialization, type and values of g-variables of the abstract kernel and their counterparts in the concrete kernel. Yet, we will only consider *reachable* variables. We distinguish between reachable *named* g-variables, i.e. global and local variables, and reachable *nameless* g-variables, that is heap variables. Reachability relies on *validity* of g-variables, which we will therefore introduce first.

**Definition 7.12 (Validity of g-Variables)** Let $sc$ denote a symbol configuration. We define the set $gvars_{\surd} : symbolconft \rightarrow gvar\,set$ of all valid g-variables—i.e. those of well-formed structure—for this symbol configuration inductively. For the base case, we define:

$$\frac{\sigma \in map(fst, sc.gst)}{gvar_{gm}(\sigma) \in gvar_{\surd}(sc)}$$

$$\frac{\sigma \in map(fst, sc.lst!i) \qquad i < |sc.lst|}{gvar_{lm}(i, \sigma) \in gvar_{\surd}(sc)}$$

$$\frac{i < |sc.hst|}{gvar_{hm}(i) \in gvar_{\surd}(sc)}$$

Array variables are valid, if their parent g-variable (i) is valid and (ii) of array type, and the index is within the bounds of the array:

$$\frac{a \in gvar_{\surd}(sc) \qquad type_g(a, sc) = Arr_T(n, t) \qquad i < n}{gvar_{arr}(a, i) \in gvar_{\surd}(sc)}$$

We have similar requirements for the validity of struct variables: the parent g-variable (i) has to be valid itself and (ii) of structure type, and the component name has to be defined in the structure:

$$\frac{st \in gvar_{\surd}(sc) \qquad type_g(st, sc) = Str_T(cl) \qquad cn \in map(fst, cl)}{gvar_{str}(st, cn) \in gvar_{\surd}(sc)}$$

**Definition 7.13 (Reachability of g-Variables)** Basing on the definition of validity, we will now introduce the notion of reachability of g-variables formally. First, we will define the set of reachable named g-variables $reachable_{\text{named}} : memconfT \to gvar\ set$. This is pretty simple, since we only require that these variables are named and valid:

$$\frac{g \in gvar_{\surd}(sc(m)) \qquad ?named(g)}{g \in reachable_{\text{named}}(m)}$$

Second, we continue with the set of reachable nameless g-vars. We could define this set inductively, too, as presented in [Lei08]. Unfortunately, this notion has turned out to be too weak for the proof described in Chapter 8. Hence, we will introduce a stronger notion of reachability for nameless g-variables, which is defined inductively over a measure: $reachable_{\text{nameless}} : memconfT \times gvar \times \mathbb{N} \to \mathbb{B}$.

Let $h$ denote a valid nameless heap variable. For the base case, we look for a valid and initialized global or local variable $g$ of pointer type, whose value points to $h$. If this is the case, then $h$ is reachable in 0 steps.

$$\frac{\begin{array}{c} h \in gvar_{\surd}(m) \\ \neg named(h) \qquad named(g) \qquad g \in gvar_{\surd}(m) \qquad init_g(m, g) \\ value_g(m, g) = Ptr(h) \qquad \exists tn.\, ty_g(sc(m), g) = Ptr_T(tn) \vee Null_T \end{array}}{reachable_{\text{nameless}}(m, h, 0)}$$

If $h$ is reachable in $i$ steps, then it is also reachable in $i + 1$ steps:

$$\frac{reachable_{\text{nameless}}(m, h, i)}{reachable_{\text{nameless}}(m, h, i + 1)}$$

If there exists a pointer variable $g$, which is reachable in $i$ steps, and whose value points to $h$, then $h$ is also reachable in $i + 1$ steps:

$$\frac{\begin{array}{c} reachable_{\text{nameless}}(m, g, i) \qquad value_g(m, g) = Ptr(h) \\ h \in gvar_{\surd}(m) \qquad \neg ?named(h) \qquad \exists tn.\, ty_g(sc(m), g) = Ptr_T(tn) \vee Null_T \end{array}}{reachable_{\text{nameless}}(m, h, i + 1)}$$

Finally, $h$ is reachable in $i + 1$ steps, if it is the sub g-variable of $g$, which itself is reachable in $i$ steps:

$$\frac{reachable_{\text{nameless}}(m, g, i) \qquad \neg ?named(h) \qquad h \in gvar_{\surd}(m) \qquad h \in sub_g(g)}{reachable_{\text{nameless}}(m, h, i + 1)}$$

**Definition 7.14 (Content and Value Equality)** In Def. 7.11, we have defined how to obtain the corresponding concrete kernel g-variable for an abstract kernel g-variable. We will now define, when we consider the *values* of such a pair of variables to be equivalent. First, we introduce the predicate $eq_{\text{cont}} : ty \times (\mathbb{N} \to mcellT) \times (\mathbb{N} \to mcellT) \times (\mathbb{N} \to \mathbb{N}) \to \mathbb{B}$. $eq_{\text{cont}}$ takes a $C0$ type (cf. Sect. 3.2.1) and two memory frame contents (cf. Sect. 3.3.1) along with a heap map as arguments. In the following, let $hp$ be a heap map and $c_a$ and $c_b$ two memory contents.

For the basic types, i.e. $Bool_T$, $Unsgnd_T$, $Int_T$, and $Char_T$, we simply check, if the content at address 0 is equal in both contents:

$$\begin{aligned} eq_{\text{cont}}(Bool_T) &= \lambda\, c_a, c_b, hp.\, c_a(0) = c_b(0) \\ eq_{\text{cont}}(Unsgnd_T) &= \lambda\, c_a, c_b, hp.\, c_a(0) = c_b(0) \\ eq_{\text{cont}}(Int_T) &= \lambda\, c_a, c_b, hp.\, c_a(0) = c_b(0) \\ eq_{\text{cont}}(Char_T) &= \lambda\, c_a, c_b, hp.\, c_a(0) = c_b(0) \end{aligned}$$

For the null pointer type $Null_T$, we proceed similarly and check, if both memory cells at address 0 contain the null pointer literal:

$$eq_{\text{cont}}(Null_T) = \lambda\, c_a, c_b, hp.\, c_a(0) = NullPointer \wedge c_b(0) = NullPointer$$

For the pointer type $Ptr_T(\sigma)$, we need a case distinction:

- $c_a(0)$ contains a null pointer literal. In this case, this has also to be true for $c_b(0)$.
- Otherwise, $c_a(0)$ is of the form $Ptr(g)$, where $g$ is a g-variable of the abstract kernel. Then, $c_b(0)$ has to hold a pointer to the corresponding g-variable in the concrete kernel, that is $Ptr(kalloc(g, hp))$.

We write formally:

$$eq_{\text{cont}}(Ptr_T(\sigma)) = \lambda\, c_a, c_b, hp. \begin{cases} c_b(0) = NullPointer & \text{if } c_a(0) = NullPointer \\ c_b(0) = Ptr(kalloc(g, hp)) & \text{if } c_a(0) = Ptr(g) \end{cases}$$

For an array type $Arr_T(k, t)$, with $k$ being the number of elements and $t$ their type, we demand that for each array element the $eq_{\text{cont}}$ predicate holds. For such an element with index $i < k$, we compute its offset by $off(i, t) = size_T(t) \cdot i$. The relevant parts of the memory frame contents $c_a$ and $c_b$ are defined as follows:

$$c'_a(i, t) = \lambda\, j. \begin{cases} c_a(off(i, t) + j) & \text{if } j < size_T(t) \\ undef & \text{otherwise} \end{cases}$$

$$c'_b(j, t) = \lambda\, j. \begin{cases} c_b(off(i, t) + j) & \text{if } j < size_T(t) \\ undef & \text{otherwise} \end{cases}$$

We can now define the equivalence relation for a complete array type:

$$eq_{\text{cont}}(Arr_T(k, t)) = \lambda\, c_a, cb, hp. \forall i < k.\, eq_{\text{cont}}(t, c'_a(i, t), c'_b(i, t), hp)$$

Similarly, we proceed with an struct type $Str_T(cl)$, where $cl$ denotes a component list. For a component $(\sigma, t)$—a pair of component name and type—in this list, we obtain its offset by $off(\sigma, cl)$. The detailed definition of this function can be found in [Lei08]. Again, the relevant parts of the memory frame contents $c_a$ and $c_b$ for such a component are given through

$$c'_a((\sigma,t),cl) = \lambda\, j.\begin{cases} c_a(off(\sigma,cl)+j) & \text{if } j < size_T(t) \\ undef & \text{otherwise} \end{cases}$$

$$c'_b((\sigma,t),cl) = \lambda\, j.\begin{cases} c_b(off(\sigma,cl)+j) & \text{if } j < size_T(t) \\ undef & \text{otherwise} \end{cases}$$

The equivalence relation for a complete struct type and two contents is then defined as follows:

$$eq_{\text{cont}}(Str_T(cl)) = \lambda\, c_a, c_b, hp.\, \forall(\sigma,t) \in \{cl\}.\, eq_{\text{cont}}(t, c'_a(\sigma,cl), c'_b(\sigma,cl))$$

Since we do not want to deal with memory frame contents, but with g-variables and their values, we will now introduce the predicate

$$eq_{\text{val}} : memconfT \times gvar \times memconfT \times gvar \times (\mathbb{N} \to \mathbb{N}) \to \mathbb{B},$$

which bases on the above definitions. Given two g-variables $g_a$ and $g_b$ and two memory configurations $m_a$ and $m_b$. Then we define:

$$eq_{\text{val}}(g_a, m_a, g_b, m_b) = eq_{\text{cont}}(type_g(sc(m_a), g_a), value_g(m_a, g_a), value_g(m_b, g_b))$$

**Definition 7.15 (Type & Initialization Equality)** Given two g-variables $g_a$ and $g_b$, and two memory configurations $m_a$ and $m_b$. We introduce two predicates $eq_{\text{type}} : memconfT \times gvar \times memconfT \times gvar \to \mathbb{B}$ and $eq_{\text{init}} : memconfT \times gvar \times memconfT \times gvar \to \mathbb{B}$, which are valid, if the two variables have the same type and the same initialization, respectively:

$$\begin{aligned} eq_{\text{type}}(m_a, g_a, m_b, g_b) &= (type_g(sc(m_a), g_a) = type_g(sc(m_b), g_b)) \\ eq_{\text{init}}(m_a, g_a, m_b, g_b) &= (init_g(m_a, g_a) = init_g(m_b, g_b)) \end{aligned}$$

**Definition 7.16 (g-Variable Consistency)** We combine now the individual relations as introduced before to a single relation $konsis_{\text{gvars}} : memconfT \times memconfT \times (\mathbb{N} \to \mathbb{N}) \to \mathbb{B}$, which connects variables of the abstract kernel to variables in the concrete kernel.

Informally, we require the following for a variable $x$ in the abstract kernel:

- if $x$ is reachable, so is $kalloc(x, hp)$ reachable in the concrete kernel;
- the initialization of $x$ and $kalloc(x, hp)$ is equal;
- if we are dealing with an *elementary* g-variable, we additionally require
  - that the types are equal

– and in the case that $x$ is initialized, the values have to be equal, too.

We split $konsis_{\text{gvars}}$ in two smaller predicates, one dealing with named g-variables—$konsis_{\text{named}}$—and one for nameless ones—$konsis_{\text{nameless}}$. Let $m_a$ and $m_b$ denote two $C0$ memory configurations and $hp$ a heap map. Then, we define formally:

$$\begin{aligned}
&konsis_{\text{named}}(m_a, m_b, hp) = \forall g.g \in reachable_{\text{named}}(m_a) \implies \\
&\quad kalloc(g, hp) \in reachable_{\text{named}}(m_b) \wedge \\
&\quad eq_{\text{init}}(m_a, g, m_b, kalloc(g, hp)) \wedge \\
&\quad (?elem(type_g(sc(m_a), g)) \implies \\
&\qquad (eq_{\text{type}}(m_a, g, m_b, kalloc(g, hp)) \wedge \\
&\qquad (?init(m_a, g) \implies eq_{\text{val}}(g, m_a, kalloc(g, hp), m_b)))).
\end{aligned}$$

Correspondingly, we define for nameless g-variables:

$$\begin{aligned}
&konsis_{\text{nameless}}(m_a, m_b, hp) = \forall g, i.reachable_{\text{nameless}}(m_a, g, i) \implies \\
&\quad reachable_{\text{nameless}}(m_b, kalloc(g, hp), i) \wedge \\
&\quad (?elem(type_g(sc(m_a), g)) \implies \\
&\qquad eq_{\text{type}}(m_a, g, m_b, kalloc(g, hp)) \wedge \\
&\qquad eq_{\text{val}}(g, m_a, kalloc(g, hp), m_b))).
\end{aligned}$$

We now combine the above two predicates:

$$\begin{aligned}
&konsis_{\text{gvars}}(m_a, m_b, hp) = \\
&\quad konsis_{\text{named}}(m_a, m_b, hp) \wedge \\
&\quad konsis_{\text{nameless}}(m_a, m_b, hp)
\end{aligned}$$

**Definition 7.17 (Symbol Table Consistency)** Two global symbol tables are *consistent*, iff all elements in the first one are also elements in the second one. We define $konsis_{\text{gst}} : memconfT \times memconfT \to \mathbb{B}$ and set

$$konsis_{\text{gst}}(m_a, m_b) = \{gst(m_a)\} \subseteq \{gst(m_b)\}.$$

Similarly, we define for local symbol tables a predicate $konsis_{\text{lst}} : memconfT \times memconfT \to \mathbb{B}$. Here, the individual symbol tables of the abstract heap have to be equivalent to the corresponding ones in the concrete stack, which are shifted by the offset $rd_{\text{off}}$:

$$\begin{aligned}
&konsis_{\text{lst}}(m_a, m_b) = \\
&\qquad \forall i.\, i < |sc(m_a).lst| \implies sc(m_a).lst!i = sc(m_b).lst!(i + rd_{\text{off}})
\end{aligned}$$

Finally, we require regarding the heap symbol table that for an abstract heap variable with index $i$, the type has to be equivalent to the corresponding concrete heap variable with index $hp(i)$:

$$\begin{aligned}
&konsis_{\text{hst}}(m_a, m_b, hp) = \forall i.\, i < |hst(m_a)| \\
&\qquad\qquad \implies snd(hst(m_a)!i) = snd(hst(m_b)!(hp(i))
\end{aligned}$$

**Definition 7.18 (Return Destination Consistency)** On returning from a function call, the *return destination*—a g-variable—associated with that function's stack frame will be updated with the function's return value. We require, that the return destinations of corresponding stack frames are equivalent under the *kalloc* function from Def. 7.11:

$$konsis_{\text{return}}(m_a, m_b, hp) \equiv \forall i.i < |sc.lst(m_a)| \implies snd(m_b.lm!(i + rd_{\text{off}})) = kalloc(snd(m_a.lm!i), hp)$$

**Definition 7.19 (Heap Map Injectivity, Boundedness)** We have to apply certain restrictions on our heap map functions. We require that

- for each abstract heap variable, there is exactly one corresponding concrete heap variable, that is *injectivity*, and
- the result of the heap map function for an abstract heap variable returns an index within the concrete heap (*boundedness*).

We introduce two predicates that describe these properties formally. Given a heap map $hp$ and two memory configurations $m_a$ and $m_b$, we define:

$$hmap_{\text{inj}}(m_a, hp) \equiv \forall i, j.\, i \neq j \wedge i < |m_a.hm| \wedge j < |m_a.hm| \implies hp(i) \neq hp(j)$$

$$hmap_{\text{bound}}(m_a, m_b, hp) = \forall i.\, i < |m_a.hm| \implies hp(i) < |m_b.hm|$$

**Definition 7.20** As we have seen in Sect. 3.5.3, the result of a memory allocation depends on the availability of heap memory. If there is no more memory available for a given type $t$ in a configuration $c_a$, then the heap memory stays unchanged a null pointer will be assigned.

In order to keep the two kernels in sync, we need to state an invariant, which guarantees that if there is enough memory in the abstract kernel, then there is also enough heap memory available in the concrete kernel.

We define this invariant formally as:

$$heap_{\text{inv}}(c_a.m, c_b.m) \equiv ?heap(c_a.m, t) = True \implies ?heap(c_b.m, t) = True.$$

The justification of this invariant is not obvious, since the concrete kernel might use much more heap memory than the abstract kernel. Yet, our CVM low-level implementation allocates memory in a very restricted way, namely only for the page tables. In particular, there is now memory allocation in the CVM primitives or even in recursive functions. So, the total heap consumption of the CVM implementation is in fact statically computable.

Nevertheless, if the low-level implementation is changed or replaced, the corresponding heap memory consumption has to be reconsidered in order to discharge the $heap_{\text{inv}}$ invariant.

The above definition of the heap invariant is sufficient in the sense that the concrete kernel always uses at least as much heap memory as the abstract kernel. Thus, the scenario that abstract heap allocation fails while it succeeds in the concrete kernel can never happen.

**Definition 7.21 (Konsis)** We have now all necessary predicates to combine them into one definition describing how the abstract and concrete kernel are connected. We name this combining predicate $konsis : tenv \times proctableT \times confT \times tenv \times proctableT \times confT \to \mathbb{B}$ and describe it formally:

$$
\begin{aligned}
&konsis(te_a, pt_a, c_a, te_b, pt_b, c_b) = \\
&\exists hp.\ konsis_{\text{tenv}}(te_a, te_b) \\
&\quad \wedge\ konsis_{\text{pt}}(pt_a, pt_b) \\
&\quad \wedge\ konsis_{\text{prog}}(c_a.prog, c_b.prog) \\
&\quad \wedge\ konsis_{\text{rd}}(c_a.m, c_b.m) \\
&\quad \wedge\ konsis_{\text{gvars}}(c_a.m, c_b.m, hp) \\
&\quad \wedge\ konsis_{\text{gst}}(c_a.m, c_b.m) \\
&\quad \wedge\ konsis_{\text{lst}}(c_a.m, c_b.m) \\
&\quad \wedge\ konsis_{\text{hst}}(c_a.m, c_b.m, hp) \\
&\quad \wedge\ konsis_{\text{return}}(c_a.m, c_b.m, hp) \\
&\quad \wedge\ hmap_{\text{inv}}(c_a.m, hp) \\
&\quad \wedge\ hmap_{\text{bound}}(c_a.m, c_b.m, hp) \\
&\quad \wedge\ heap_{\text{inv}}(c_a.m, c_b.m)
\end{aligned}
$$

### Concrete Kernel to ISA and Concrete Kernel Invariants

As we have learned in Chapter 6, the abstract kernel is an incomplete program which cannot be compiled and thus cannot be directly related to the hardware. Instead, we do this with the concrete kernel using the compiler correctness relation.

In [Lei08], the original version of the compiler correctness theorem was formulated between the $C0$ small-step layer and the assembly layer. [Tsy09] has extended this work and has defined a relation *sim-c0-isa* connecting a $C0$ configuration with a VAMP ISA configuration.

By further adding requirements on the validity of the $C0$ configuration and the structure of the heap and global memory, [Tsy09] has obtained the relation

$$kernel\text{-}sim\text{-}c0\text{-}isa : confT_{C0} \times confT_{C0} \times confT_{\text{i\&d}} \times (gvar \to \mathbb{N}) \to \mathbb{B}.$$

Using this definition, the concrete kernel implementation, the $C0$ configuration encoding the page fault handler and the one encoding the concrete kernel to a ISA configuration can now be related to each other, using an allocation function mapping g-variables to their base address in the hardware.

This relation is not used in our proofs, but needed to present an overall correctness theorem sketch for CVM. [Tsy09] gives the full formal definitions of *kernel-sim-c0-isa* and uses it in her proofs.

Finally, the page fault handler configuration on small-step level has to be connected to the concrete kernel. In particular, the active and free lists used in the page fault handler for memory management as well as the process control blocks have to be well-formed and have corresponding values to those in the concrete kernel. [Sta09] has described this by the predicate

$$\mathit{concr\text{-}kernel\text{-}inv} : \mathit{confT}_{C0} \times \mathit{confT}_{C0} \to \mathbb{B}.$$

### Weak Relations

The abstraction relations, which we have introduced in the sections before, are too strong in certain cases. For instance, when a user process is making progress, the local memory stack and the program rest of the abstract kernel are empty.

Hence, we have to formulate a weaker relation between the abstract and the concrete kernel and name it $\mathit{konsis}_{\text{weak}}$. In particular, we omit

- the program rest consistency, since there is no abstract program rest during user execution, and
- the invariants on local memories and symbol tables as well as the recursion depth relation for the same reason.

The static properties, i.e. type name, procedure table and global symbol table consistency, still have to hold, as well as the properties of the heap and the g-variables.

**Definition 7.22 (Weak Konsis)** Let $te_a, te_b$ denote two type name environments, $pt_a, pt_b$ two procedure tables and $c_a, c_b$ two $C0$ configurations. The formal definition of the weaker konsis predicate

$$\mathit{konsis}_{\text{weak}} : \mathit{tenv} \times \mathit{proctableT} \times \mathit{confT}_{C0} \times \mathit{tenv} \times \mathit{proctableT} \times \mathit{confT}_{C0} \to \mathbb{B}$$

is then as follows:

$$\begin{aligned}
&\mathit{konsis}_{\text{weak}}(te_a, pt_a, c_a, te_b, pt_b, c_b) = \\
&\quad \exists hp.\, \mathit{konsis}_{\text{tenv}}(te_a, te_b) \\
&\quad \wedge\, \mathit{konsis}_{\text{pt}}(pt_a, pt_b) \\
&\quad \wedge\, \mathit{konsis}_{\text{gvars}}(m(c_a), m(c_b), hp) \\
&\quad \wedge\, \mathit{konsis}_{\text{gst}}(m(c_a), m(c_b)) \\
&\quad \wedge\, \mathit{konsis}_{\text{hst}}(m(c_a), m(c_b), hp) \\
&\quad \wedge\, \mathit{hmap}_{\text{inv}}(m(c_a), hp) \\
&\quad \wedge\, \mathit{hmap}_{\text{bound}}(m(c_a), m(c_b), hp)
\end{aligned}$$

Furthermore, the *kernel-sim-c0-isa* relation is also too strong during user execution. For instance, the hardware registers are those of the user process, which obviously affects control and register consistency in the overall compiler consistency (cf. [Lei08, Sect. 8.2]). Thus we do not relate the whole concrete kernel to the ISA, but consider only its global and heap memory content and its

heap symbol table (remember: there is no abstract local memory stack during user execution). A weaker form of this relation has been formulated by [Tsy09]:

$$\begin{aligned} \mathit{weak\text{-}sim\text{-}c0\text{-}isa} \quad : \quad & \mathit{confT}_{C0} \times (\mathbb{N} \to \mathit{mcellT}) \times (\mathbb{N} \to \mathit{mcellT}) \\ & \times (\Sigma \times \mathit{ty})\ \mathit{list} \times \mathit{confT}_{\mathrm{i\&d}} \times (\mathit{gvar} \to \mathbb{N}) \to \mathbb{B} \end{aligned}$$

### Combining the Kernel Relations

**Definition 7.23 ((Weak) Kernel Relation)** All of the relations defined before can now be combined into an overall kernel relation. Given a hardware configuration $c_{\mathrm{i\&d}}$ and an allocation function *alloc*. For a CVM configuration $c_{\mathrm{CVM}}$, we abbreviate as follows: $te_{\mathrm{abs}}$ for $c_{\mathrm{CVM}}.\mathit{kern.tn}$, $pt_{\mathrm{abs}}$ for $c_{\mathrm{CVM}}.\mathit{kern.pt}$, and $c_{C0\,\mathrm{abs}}$ for $c_{\mathrm{CVM}}.\mathit{kern.kconf}$. Furthermore we abbreviate the concrete kernel program with $\Pi_k = (te_k, pt_k, gst_k)$. Then there exist two $C0$ small-step configurations for the page fault handler and the concrete kernel, such that the concrete kernel to ISA and concrete kernel invariants as well as the relation between the abstract and the concrete kernel hold.

Formally, we define:

$$\begin{aligned} & \mathit{kernel\text{-}rel}(\Pi_k, c_{\mathrm{CVM}}.\mathit{kern}, c_{\mathrm{i\&d}}, \mathit{alloc}) = \\ & \exists \mathit{pfh}_{\mathrm{ss}}, c_{C0\,k}. \\ & \quad \mathit{kernel\text{-}sim\text{-}c0\text{-}isa}(\mathit{pfh}_{\mathrm{ss}}, c_{C0\,k}, c_{\mathrm{i\&d}}, \mathit{alloc}) \\ & \quad \wedge\ \mathit{concr\text{-}kernel\text{-}inv}(\mathit{pfh}_{\mathrm{ss}}, c_{C0\,k}) \\ & \quad \wedge\ \mathit{konsis}(te_{\mathrm{abs}}, pt_{\mathrm{abs}}, c_{C0\,\mathrm{abs}}, te_k, pt_k, c_{C0\,k}) \end{aligned}$$

For the weaker kernel relation *weak-kernel-rel*, we use the weaker forms of the corresponding predicates. Moreover, we omit the concrete kernel invariants defined in *concr-kernel-inv* and require that

- the first heap variable $\mathit{gvar}_{hm}(0)$, which represents the page tables, resides at address *heap-base*, and
- if the current process identifier denotes some process id, this process has some allocated memory corresponding the process control blocks as maintained by the page fault handler.

We define this weaker relation formally:

$$\begin{aligned} & \mathit{weak\text{-}kernel\text{-}rel}(\Pi_k, c_{\mathrm{CVM}}.\mathit{kern}, c_{\mathrm{i\&d}}, \mathit{alloc}, c_{\mathrm{CVM}}.\mathit{up.cp}) = \\ & \quad \exists \mathit{pfh}_{\mathrm{ss}}, c_{C0\,k}. \\ & \quad \mathit{weak\text{-}sim\text{-}c0\text{-}isa}(\mathit{pfh}_{\mathrm{ss}}, c_{C0\,k}.m.\mathit{gm.cont}, \\ & \qquad c_{C0\,k}.m.\mathit{hm.cont}, \mathit{hst}(c_{C0\,k}.m), c_{\mathrm{i\&d}}, \mathit{alloc}) \\ & \quad \wedge\ \mathit{konsis}_{\mathrm{weak}}(te_{\mathrm{abs}}, pt_{\mathrm{abs}}, c_{C0\,\mathrm{abs}}, te_k, pt_k, c_{C0\,k}) \\ & \quad \wedge\ \mathit{alloc}(\mathit{gvar}_{hm}(0)) = \mathit{heap\text{-}base} \\ & \quad \wedge\ c_{\mathrm{CVM}}.\mathit{up.cp} = \lfloor \mathit{pid} \rfloor \implies 0 \leq \mathit{pfh}_{\mathrm{ss}}.\mathit{abs\text{-}pcbs\text{-}ss}[\mathit{pid}].\mathit{ptl}) \end{aligned}$$

### Linkable Programs

We have described in Sect. 6.5.2 how to obtain a concrete kernel program by linking the low-level CVM implementation to the abstract kernel program.

Obviously, abstract linking does not produce meaningful programs for arbitrary input in the sense that the obtained code is runnable and/or compilable. Thus, we restrict the set of pairs of programs that are linkable by introducing a predicate

$$\mathit{linkable} : \mathit{c0prog} \times \mathit{c0prog} \to \mathbb{B},$$

which takes two programs for input and evaluates to *True*, if these two programs are linkable and *False* otherwise.

Most of our requirements base on the *validity* of $C0$ programs and configurations. We refer to [Lei08, Sect. 5.2] for the full definitions of the individual validity predicates and describe their intuitive meaning here instead.

Informally, we require that

- the two type name environments are *valid* and that no type name is declared twice with two different types,
- both symbol tables $gst_a$ and $gst_b$ are valid,
- after linking, no external function calls are left,
- all internal functions are valid functions, and finally
- only one program has in-line assembly parts.

A type name environment is valid, if all type names are pairwise distinct and their associated types are valid.

A type is valid, if (i) it is a basic type, that is $Bool_T$, $Int_T$, $Unsgnd_T$ or $Char_T$, (ii) it is a pointer $Ptr_T(tn)$ and $tn$ is defined in the type name environment, (iii) it is a struct and all component names are pairwise distinct and their types are valid, too, (iv) it is a non-empty array type with a valid element type. We abbreviate the set of all valid type name environments by $valid_{\mathrm{tenv}}$.

A symbol table is valid, if all variable names in it are pairwise distinct and each type associated with a variable name is valid, too. For a given type name environment $te$, we refer by $valid_{\mathrm{st}}(te)$ to the set of valid symbol tables.

Let $pt$ denote a procedure table, $te$ a type name environment and $gst$ a global symbol table. The set of valid functions $valid_{\mathrm{fun}}(te, pt, gst)$ is then defined as follows: A function $f$ is valid (i) if its body is a valid statement, (ii) the last and only the last statement in the function body of $f$ is a return statement, (iii) the type of the return expression matches the return type of the function, (iv) and the symbol table consisting of parameters and local variables of $f$ is a valid symbol table.

For valid statements, we basically require that for an assignment, the types on the right side and on the left side match, and that the types of a function call parameters match with those of the called function.

**Definition 7.24 (Linkable Programs)** Two programs $\Pi_a = (te_a, pt_a, gst_a)$ and $\Pi_b = (te_b, pt_b, gst_b)$ are *linkable*, if the following holds:

$$\begin{aligned}
linkable(\Pi_a, \Pi_b) \quad = \quad & te_a \in valid_{\text{tenv}} \wedge te_b \in valid_{\text{tenv}} \\
\wedge \quad & (\forall (t_a, tn_a) \in te_a, (t_b, tn_b) \in te_b.\, t_a = t_b \implies tn_a = tn_b) \\
\wedge \quad & gst_a \in valid_{\text{st}}(te_a) \wedge gst_b \in valid_{\text{st}}(te_b) \\
\wedge \quad & (\forall (fn, f) \in pt_a^{\text{ext}}.\, \exists f'.\, (fn, f') \in pt_b^{\text{def}}) \\
\wedge \quad & (\forall (fn, f) \in pt_b^{\text{ext}}.\, \exists f'.\, (fn, f') \in pt_a^{\text{def}}) \\
\wedge \quad & (\forall (fn, f) \in pt_a^{\text{def}}.\, f \in valid_{\text{fun}}(te_a, pt_a, gst_a)) \\
\wedge \quad & (\forall (fn, f) \in pt_b^{\text{def}}.\, f \in valid_{\text{fun}}(te_b, pt_b, gst_b)) \\
\wedge \quad & ((\forall (fn, f) \in pt_a.\, \forall s \in s2l(f.body).\, \neg is_Asm(s) \\
& \vee (\forall (fn, f) \in pt_b.\, \forall s \in s2l(f.body).\, \neg is_Asm(s)))
\end{aligned}$$

### 7.2.3 Device Relation

The relation between the CVM devices and those in the hardware is rather simple. The only difference is that in CVM the swap device with id $did_{\text{swap}}$ is not visible any longer, since page faults are not visible in the CVM model. Thus, we exclude this swap device from our device relation: We define $dev_{sim} : confT_{\text{devs}} \times confT_{\text{devs}} \to \mathbb{B}$ and set for two generalized device configurations $devs_1$ and $devs_2$:

$$dev_{\text{sim}}(devs_1, devs_2) = (\forall i \in \mathbb{D}, i \neq did_{\text{swap}} : devs_1(i) = devs_2(i))$$

### 7.2.4 Interrupt Mask and Current Process

As we have seen in Sect. 6.5.1, the concrete kernel has two variables to store the interrupt mask and the current user process identifier: `mask` and `cp`. The values of these variables have to be equal to the CVM components *cp* and *mask*. [Tsy09] has introduced two abstraction relations $SR_{\text{rel}} : confT_{\text{i\&d}} \times \mathbb{B}^{32} \to \mathbb{B}$ and $CP_{\text{rel}} : confT_{\text{i\&d}} \times \mathbb{N} \to \mathbb{B}$, formally defined as follows:

$$SR_{\text{rel}}(c_{\text{i\&d}}, mask) = \begin{cases} True & \text{if } to_{\text{int}}(mask) = \langle \texttt{mask} \rangle_{c_{\text{i\&d}}}^{int} \\ False & \text{otherwise} \end{cases}$$

$$CP_{\text{rel}}(c_{\text{i\&d}}, cp) = \begin{cases} True & \text{if } cp = \langle \texttt{cp} \rangle_{c_{\text{i\&d}}}^{nat} \\ False & \text{otherwise} \end{cases}$$

### 7.2.5 Other Invariants

Besides the abstraction relations, which we have introduced in the sections before, there are some minor invariants dealing with subtleties of the implementation. They are presented in detail by [Tsy09].

The relation *user-invariant* : $confT_{\text{i\&d}} \to \mathbb{B}$ for instance ensures that the hardware is in user mode ($sprs(mode) = 1$), the page table length register

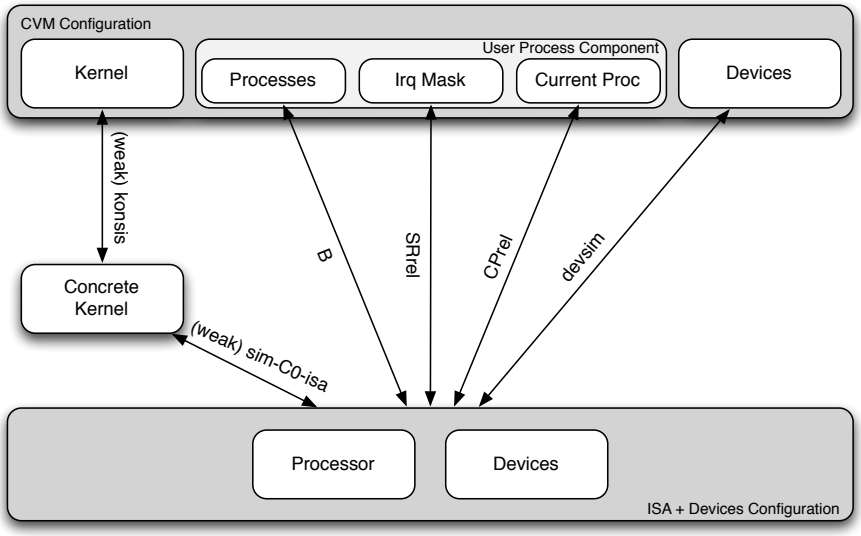


Figure 7.2: CVM Abstraction Relations

$sprs(ptl)$ and page table origin register $sprs(pto)$ in the instruction set architecture are equivalent to the values stored in the process control blocks, and that the interrupt mask $sprs.sr$ corresponds to the one stored in the variable `mask`.

The *reg-invariant* : $confT_{\mathrm{i\&d}} \to \mathbb{B}$ says that all internal interrupts are disabled, i.e., the bits $sprs(sr)[31:13]$ are zeroed. Furthermore, it states that the hardware is in system mode ($sprs(mode) = 0$).

There exists also another form of *reg-invariant* that describes the hardware registers, whenever we are in *kernel wait*: *weak-reg-invariant* : $confT_{\mathrm{i\&d}} \to \mathbb{B}$. This invariant guarantees that—unlike during 'normal' kernel execution—the external interrupts are enabled and that the program counters *dpc* and *pcp* establish the program loop that is used to implement kernel wait.

### 7.2.6 Putting It All Together

**Definition 7.25 (CVM Abstraction Relation)** We can now combine all of the above abstraction relations into one CVM relation (see Fig. 7.2).

$$CVMrelation : c0prog \times confT_{\mathrm{CVM}} \times confT_{\mathrm{i\&d}} \times (\mathbb{N} \to mcellT) \to \mathbb{B}.$$

Depending on the possible CVM transitions described in Sect. 6.2, we distinguish three different cases:

- In *kernel wait* (cf. Sect. 6.2.2), the weak kernel relation and the weak register invariant hold, and the current process variable has value 0.
- For *user steps* (cf. Sect. 6.2.3)—i.e. $c_{\mathrm{CVM}}.up.cp = \lfloor pid \rfloor$—, the weak kernel relation, the *CPrel* relation and the user invariant hold.

- For *kernel steps* different from kernel wait, the kernel relation and the register invariant hold.

In all cases, the user process relation and the device relation hold.

$$
\begin{aligned}
&\mathit{CVMrelation}(\Pi_k, c_{\mathrm{CVM}}, c_{\mathrm{i\&d}}, \mathit{alloc}) = \\
&\quad \mathit{dev}_{\mathrm{sim}}(c_{\mathrm{CVM}}.\mathit{devs}, c_{\mathrm{i\&d}}.\mathit{devs}) \wedge \\
&\quad B(c_{\mathrm{CVM}}.\mathit{up.procs}, c_{\mathrm{i\&d}}) \wedge \\
&\quad \mathit{SR}_{\mathrm{rel}}(c_{\mathrm{i\&d}}, c_{\mathrm{CVM}}.\mathit{up.mask}) \wedge \\
&\quad \textbf{if} \quad \text{(* Kernel Steps *)} \\
&\qquad c_{\mathrm{CVM}}.\mathit{up.cp} = \mathit{None} \\
&\quad \textbf{then} \\
&\qquad \textbf{if} \quad \text{(* Kernel Wait *)} \\
&\qquad\quad c_{\mathrm{CVM}}.\mathit{kern.kconf.prog} = \mathit{Asm}[\,] \\
&\qquad \textbf{then} \\
&\qquad\quad \mathit{weak\text{-}kernel\text{-}rel}(\Pi_k, c_{\mathrm{CVM}}.\mathit{kern}, c_{\mathrm{i\&d}}, \mathit{alloc}, \mathit{None}) \wedge \\
&\qquad\quad \mathit{CP}_{\mathrm{rel}}(c_{\mathrm{i\&d}}, 0) \wedge \\
&\qquad\quad \mathit{weak\text{-}reg\text{-}invariant}(c_{\mathrm{i\&d}}) \\
&\qquad \textbf{else} \quad \text{(* Other Kernel Step *)} \\
&\qquad\quad \mathit{kernel\text{-}rel}(\Pi_k, c_{\mathrm{CVM}}.\mathit{kern}, c_{\mathrm{i\&d}}, \mathit{alloc}) \wedge \\
&\qquad\quad \mathit{reg\text{-}invariant}(c_{\mathrm{i\&d}}) \\
&\quad \textbf{else} \quad \text{(* User Step *)} \\
&\qquad \mathit{weak\text{-}kernel\text{-}rel}(c_{\mathrm{CVM}}.\mathit{kern}, c_{\mathrm{i\&d}}, \mathit{alloc}, \mathit{the}(c_{\mathrm{CVM}}.\mathit{up.cp})) \wedge \\
&\qquad \mathit{CP}_{\mathrm{rel}}(c_{\mathrm{i\&d}}, \mathit{the}(c_{\mathrm{CVM}}.\mathit{up.cp})) \wedge \\
&\qquad \mathit{user\text{-}invariant}(c_{\mathrm{i\&d}})
\end{aligned}
$$

## 7.3 Sketch of a Correctness Theorem

The CVM top-level correctness theorem is a classical simulation theorem: a VAMP instruction set architecture with devices simulates the CVM model. It has been originally presented in [AHL+09]. We start with an initial combined ISA and device configuration $c^0_{i\&d} = (c_{\mathrm{isa}}, c_{\mathrm{devs}})$.

Furthermore, we assume that the concrete kernel program $\Pi_k$ has been loaded to the main memory $c_{\mathrm{isa}}.mm$. This hardware state is described by the predicate $\mathit{init\text{-}isa\text{-}conf}(\Pi_k, c_{\mathrm{isa}})$.

As we have seen in Sect. 5.2, the combined transition function of VAMP ISA and devices is parametrized over a sequence of *external device inputs* $\mathit{din}^{\mathrm{isa}}_{\mathrm{ext}} : \mathbb{N} \to (\mathbb{D} \times \mathit{Eifi})_\perp$ (cf. Sect. 4.2). Obviously, there have to be certain requirements that this sequence has to meet:

- the sequence has to be *fair* in the sense that the processor progresses infinitely often:

$$\forall i \in \mathbb{N} : \exists j > i : \mathit{din}^{\mathrm{isa}}_{\mathrm{ext}}(j) = \mathit{None}.$$

- the sequence has to be *well-typed*, that is that device type and external device input type match for each element of the sequence:

$$\forall i \in \mathbb{N} : \mathit{din}^{\mathrm{isa}}_{\mathrm{ext}}(i) = \lfloor (\mathit{did}, \mathit{ev}) \rfloor \wedge t = \mathit{type}_{\mathrm{dev}}(\mathit{devs}(\mathit{did})) \implies \mathit{ev} \in \mathit{Eifo}_t.$$

We combine these preconditions to one predicate *precond-seq-isa*($din^{\text{isa}}_{\text{ext}}, c_{\text{devs}}$).

In addition, we have some requirements that have to be satisfied by the swap hard disk $c_{\text{devs}}(did_{\text{swap}})$:

- requests—sector transfers—from or to the hard disk are handled in finite time, and
- the swap hard disk $c_{\text{devs}}(did_{\text{swap}})$ is large enough, i.e. it can store the kernel image and still offer enough space for swap memory as needed by the page fault handler.

We subsume these requirements as *invariant-hd*($c_{\text{devs}}(did_{\text{swap}})$).

As we have seen in Sect. 6.4, the swap device as present in the VAMP ISA and device configuration is replaced by an idle device for the CVM configuration. We denote this slightly updated device configuration by $c^{\text{CVM}}_{\text{devs}}$.

The corresponding external device input sequence $din^{\text{CVM}}_{\text{ext}}$ has to satisfy similar requirements as the one on VAMP ISA level, i.e., it has to be fair and well-typed. We denote this using the predicate *precond-seq-cvm*.

For a sub sequence of $din^{\text{CVM}}_{\text{ext}}$, we can easily count the number of processor steps in it. We define recursively

$$\textit{count-proc-steps}(din^{\text{CVM}}_{\text{ext}}, 0) \quad = \quad \begin{cases} 1 & \text{if } din^{\text{CVM}}_{\text{ext}}(0) = \textit{None} \\ 0 & \text{otherwise} \end{cases}$$

and

$$\textit{count-proc-steps}(din^{\text{CVM}}_{\text{ext}}, i+1) = \begin{cases} 1 + \textit{count-proc-steps}(din^{\text{CVM}}_{\text{ext}}, i) & \text{if } din^{\text{CVM}}_{\text{ext}}(i+1) = \textit{None} \\ \textit{count-proc-steps}(din^{\text{CVM}}_{\text{ext}}, i) & \text{otherwise} \end{cases}$$

In the Verisoft project, there are two ways regarding the treatment of heap overflows. In one scenario, the compiler returns a null pointer, when trying to allocate a new heap object on insufficient space. The other approach, which we use in the stack verification, guarantees correct translation only under the assumption that the compiled code does not allocate more heap memory than available.

Throughout the execution of CVM, the abstract kernel can exceed its maximal heap and/or stack size, hence create a run-time error. We therefore formulate two measure functions *asize-heap* and *asize-stack*, which return the current stack and heap size of the abstract kernel, and require that those values stay within the bounds given through the two constants *abs-heap-max-size* and *abs-stack-max-size*.

For the VAMOS personality, both preconditions can be discharged pretty easily: since the kernel only allocates two heap objects during initialization, but no further ones during execution, the heap size can be computed statically. For the stack size, the situation is similar. Since there are no recursive function calls in VAMOS, the maximal recursion depth can be computed statically and hence also the stack size. The necessary framework has been built and applied exemplarily by Leinenbach and Alkassar [Alk09].

Using all of the above definitions, we can now formulate the CVM top-level simulation theorem, which says that a CVM computation starting from an initial CVM configuration (cf. Sect. 6.4) and parametrized over an external device input sequence $din_{\text{ext}}^{\text{CVM}}$ can be simulated by a VAMP ISA with devices computation.

**Theorem 7.1 (CVM Top-Level Correctness)**
*For all $n \in \mathbb{N}$ denoting a number of non-device steps in the CVM model, which lead to some CVM configuration—that is the kernel has not produced a run-time fault—we have to show that there exists a step number $N'$ for the ISA and device model, after which the combined abstraction relation CVMrelation holds:*

$$
\begin{aligned}
&\mathit{linkable}(\Pi_{abs}, \Pi_{CVM}) \wedge \Pi_k = \mathit{link}(\Pi_{abs}, \Pi_{CVM}) \wedge \\
&\mathit{init\text{-}isa\text{-}conf}((\Pi_k, c_{isa}) \wedge \mathit{precond\text{-}seq\text{-}isa}(din_{ext}^{isa}, c_{devs}) \wedge \\
&\mathit{invariant\text{-}hd}(c_{devs}(did_{swap})) \implies \\
&\quad \exists din_{ext}^{CVM}.\mathit{precond\text{-}seq\text{-}cvm}(din_{ext}^{CVM}, c_{devs}) \wedge \\
&\quad (\forall n. \exists N. \mathit{count\text{-}proc\text{-}steps}(din_{ext}^{CVM}, N) = n \wedge \\
&\qquad (\forall c'_{CVM}, dout'_{ext}. \\
&\qquad\quad \delta_{cvm}^{N}(\mathit{init}_{CVM}(\Pi_{abs}, c_{devs}^{CVM}), din_{ext}^{CVM}) = \lfloor c'_{CVM}, dout'_{ext} \rfloor \wedge \\
&\qquad\quad \mathit{asize\text{-}heap}(\mathit{hst}(c'_{CVM}.\mathit{kern}.\mathit{kconf}.m)) \leq \mathit{abs\text{-}heap\text{-}max\text{-}size} \wedge \\
&\qquad\quad \mathit{asize\text{-}stack}(c'_{CVM}.\mathit{kern}.\mathit{kconf}.m.lm) \leq \mathit{abs\text{-}stack\text{-}max\text{-}size} \implies \\
&\qquad\qquad (\exists N', \mathit{alloc}'. \\
&\qquad\qquad\quad \mathit{CVMrelation}(\Pi_k, c'_{CVM}, \mathit{fst}(\delta_{i\&d}^{N'}((c_{isa}, c_{devs}), din_{ext}^{isa})), \mathit{alloc}'))))
\end{aligned}
$$

The proof of this theorem can be split according to the possible steps in the CVM model and as visible in the top-level abstraction relation *CVMrelation* (cf. Def. 7.25).

In the remainder of this work, we will deal with the case of a kernel step, which is not kernel wait nor a primitive call. More precisely, we will show how the relation of the abstract kernel and the concrete kernel *konsis* is preserved during such a step. User step correctness, including the page fault interrupt case, is summarized in [AHL+09].

We will now formulate inductively the correctness theorem for the case described above This means in particular:

- We start in a machine configuration $(c_{\text{isa}}, c_{\text{devs}}) \in \mathit{confT}_{\text{i\&d}}$, which is related to a CVM configuration $c_{\text{CVM}}$ and a concrete kernel program $\Pi_k$ using an allocation function *alloc*.
- Since the kernel is running, the current process identifier is *None*.
- We assume that we are not in kernel wait, i.e. the abstract kernel program rest does not consist of an *Asm* statement.
- We exclude the case of a primitive call, which would be denoted by an abstract kernel rest starting with an *ESCall* statement.
- Kernel execution has not terminated, which would be denoted by a program rest merely consisting of a single *Skip* statement.

Furthermore, we have requirements similar to those of the top-level correctness theorem (see Theorem 7.1):

- there exists a low-level CVM implementation that is linkable with the abstract kernel and hence yields the concrete kernel program;
- the external device input sequence satisfies well-typedness and fairness;
- the swap hard disk is big enough and its invariants hold.

Last but not least, there are additional assumptions on

- the well-formedness of the ISA configuration—*is-dlx-conft*($c_{\text{isa}}$)—;
- the boundedness of the abstract kernel heap and stack;
- that the concrete kernel program is present in the main memory of the ISA—*code-invariant-isa*($\Pi_k$, $c_{\text{isa}}$);

**Theorem 7.2 (Kernel Step, no Primitive, not Kernel Wait)**
*Given and abstract kernel program* $\Pi_{abs} = (te_{abs}, pt_{abs}, gst_{abs})$ *and an low-level CVM implementation* $\Pi_{CVM}$*, which are linkable to a concrete kernel implementation* $\Pi_k = (te_k, pt_k, gst_k)$*. Furthermore we assume that the kernel is to progress and that we do not have to deal with kernel wait, kernel termination or a primitive call and that we are in a configuration* $c_{CVM}$*, in which the combined abstraction relation holds for a processor and device configuration* $(c_{isa}, c_{devs})$*. Assuming that the next non-device step in the CVM model will not create a run-time fault but yield some successor configuration, there exists a step number* $N'$ *for the combined ISA and device model, after which the combined*

*abstraction relation holds again.*

$$
\begin{array}{l}
\mathit{linkable}(\Pi_{abs}, \Pi_{CVM}) \wedge \Pi_k = \mathit{link}(\Pi_{abs}, \Pi_{CVM}) \wedge \\
\mathit{precond\text{-}seq\text{-}isa}(\mathit{din}^{isa}_{ext}, c_{devs}) \wedge \\
\mathit{asize\text{-}heap}(\mathit{hst}(c'_{CVM}.\mathit{kern.kconf.m})) \leq \mathit{abs\text{-}heap\text{-}max\text{-}size} \wedge \\
\mathit{asize\text{-}stack}(c'_{CVM}.\mathit{kern.kconf.m.lm}) \leq \mathit{abs\text{-}stack\text{-}max\text{-}size} \wedge \\
\mathit{invariant\text{-}hd}(c_{devs}(\mathit{did}_{swap})) \wedge \mathit{code\text{-}invariant\text{-}isa}(\Pi_k, c_{isa}) \wedge \\
\mathit{is\text{-}dlx\text{-}conft}(c_{isa}) \wedge c_{CVM}.\mathit{up.cp} = \mathit{None} \wedge \\
\neg \mathit{is\_Asm}(\mathit{hd}(\mathit{s2}(c_{CVM}.\mathit{kern.kconf.prog}))) \wedge \\
\neg \mathit{is\_Skip}(c_{CVM}.\mathit{kern.kconf.prog}) \wedge \\
\neg \mathit{is\_ESCall}(\mathit{hd}(\mathit{s2l}(c_{CVM}.\mathit{kern.kconf.prog}))) \wedge \\
\mathit{CVMrelation}(\Pi_k, c_{CVM}, (c_{isa}, c_{devs}), \mathit{alloc}) \implies \\
\quad \exists \mathit{din}^{CVM}_{ext}.\mathit{precond\text{-}seq\text{-}cvm}(\mathit{din}^{CVM}_{ext}, c_{devs}) \wedge \\
\quad (\exists N.\mathit{count\text{-}proc\text{-}steps}(\mathit{din}^{CVM}_{ext}, N) = 1 \wedge \\
\qquad (\forall c'_{CVM}. \\
\qquad\quad \exists \mathit{dout}'_{ext}.\, \delta^{N}_{cvm}(\mathit{init}_{CVM}(\Pi_{abs}, c_{devs}), \mathit{din}^{CVM}_{ext}) = \lfloor c'_{CVM}, \mathit{dout}'_{ext} \rfloor \wedge \\
\qquad\quad \mathit{asize\text{-}heap}(\mathit{hst}(c'_{CVM}.\mathit{kern.kconf.m})) \leq \mathit{abs\text{-}heap\text{-}max\text{-}size} \wedge \\
\qquad\quad \mathit{asize\text{-}stack}(c'_{CVM}.\mathit{kern.kconf.m.lm}) \leq \mathit{abs\text{-}stack\text{-}max\text{-}size} \implies \\
\qquad\qquad (\exists N', c'_{isa}, c'_{devs}. \\
\qquad\qquad\quad \mathit{fst}(\delta^{N'}_{i\&d}((c_{isa}, c_{devs}), \mathit{din}^{isa}_{ext})) = (c'_{isa}, c'_{devs}) \wedge \\
\qquad\qquad\quad (\exists \mathit{alloc}'.\mathit{CVMrelation}(\Pi_k, c'_{CVM}, (c'_{isa}, c'_{devs}), \mathit{din}^{isa}_{ext}), \mathit{alloc}')))
\end{array}
$$

Do not consider it proof just because it is written in books, for a liar who will deceive with his tongue will not hesitate to do the same with his pen.

Maimonides, Spanish Philosopher, 1135–1204

Chapter 8

# Preservation of Kernel Consistency

**Contents**



In this chapter we will present the proof that the relation between the abstract and the concrete kernel, as introduced in Sect. 7.2.2, Def. 7.21, is preserved by a *C*0 kernel step within the CVM model (cf. Sect. 6.2.2).

We break down this proof in several parts. In Sect. 8.1, we will define and prove some auxiliary lemmas needed in the remainder of this chapter. Since the abstract kernel is a crippled program, its configurations cannot be valid in the sense of [Lei08]. We therefore define the notion of weak validity in Sect. 8.2.

Expression evaluation in the two kernels and the relation of its results is of crucial importance for the proof work. In Sect. 8.3, we will treat this topic in detail.

We continue with the static properties, that is type name environment, procedure table and global symbol table consistency. The corresponding lemmas and proofs are detailed in Sect. 8.4.

Finally, we will consider the dynamic properties, e.g., recursion depth, return destination and g-variable consistency (Sect. 8.5).

With Sect. 8.6, we conclude this chapter by putting together the various results into the comprising correctness theorem.

## 8.1 Auxiliary Lemmas

**Lemma 8.1 (Equal Types Invariant)** *Assuming that the symbol table relations and the heap invariants hold, and that $x$ is a valid g-variable in the abstract kernel, then its type is equal to the corresponding g-variable in the*

*concrete kernel.*

$$
\begin{aligned}
&konsis_{gst}(c_{abs}.m, c_k.m) \wedge konsis_{lst}(c_{abs}.m, c_k.m) \wedge \\
&konsis_{hst}(c_{abs}.m, c_k.m) \wedge c_k \in conf_{\surd}(te_k, pt_k) \wedge \\
&hmap_{inj}(c_{abs}.m, hp) \wedge konsis_{rd}(c_{abs}.m, c_k.m) \wedge \\
&x \in gvar_{\surd}(sc(c_{abs}.m)) \\
&\quad \Longrightarrow \\
&eq_{type}(c_{abs}.m, x, c_k.m, kalloc(x, hp))
\end{aligned}
$$

PROOF The proof is done by induction over the g-variable $x$. It is very straightforward; basically we just expand the invariants over the symbol tables for the base cases and instantiate the induction hypotheses for array and structure variables. q.e.d.

**Lemma 8.2 (Valid g-Variable Invariant)** *We assume that the symbol table relations, the recursion depth and the heap invariants hold, and that $x$ is a valid g-variable in the abstract kernel, then the corresponding g-variable $kalloc(x, hp)$ in the concrete kernel is also valid.*

$$
\begin{aligned}
&konsis_{gst}(c_{abs}.m, c_k.m) \wedge konsis_{lst}(c_{abs}.m, c_k.m) \wedge \\
&konsis_{hst}(c_{abs}.m, c_k.m) \wedge c_k \in conf_{\surd}(te_k, pt_k) \wedge \\
&hmap_{inj}(c_{abs}.m, hp) \wedge hmap_{bound}(c_{abs}.m, c_k.m, hp) \wedge \\
&konsis_{rd}(c_{abs}.m, c_k.m) \wedge x \in gvar_{\surd}(sc(c_{abs}.m)) \\
&\quad \Longrightarrow \\
&\quad kalloc(x, hp) \in gvar_{\surd}(sc(c_k.m))
\end{aligned}
$$

PROOF We prove this lemma by an induction over the structure of g-variable $x$.

**Case 1:** Given a global variable, i.e. $x = gvar_{gm}(\sigma)$. Due to validity of the variable, there exists a type $t$, such that $(\sigma, t) \in \{sc(c_{\text{abs}}.m.gm)\}$.

Expanding the invariant on the global symbol tables $konsis_{\text{gst}}$, this entry also exists in the linked global symbol configuration. Since *kalloc* is the identity for global variables, $kalloc(gvar_{gm}(\sigma), hp)$ is valid in the concrete kernel, too.

**Case 2:** In the local case $x = gvar_{lm}(\sigma, i)$, validity is defined by two criteria: (i) $\sigma$ is defined in the $i$-th symbol table of the abstract local memory, $\sigma \in \{map(fst, sc((c_{\text{abs}}.m.lm)!i))\}$, and (ii) $i$ is less than the recursion depth of the abstract kernel: $i < |c_{\text{abs}}.m.lm|$. Using the invariants on local symbol tables, $konsis_{\text{lst}}$, and on recursion depth, $konsis_{\text{rd}}$, we obtain that $\sigma$ is also obtained in the concrete local symbol table at position $i + rd_{\text{off}}$, which is still less than the concrete kernel's recursion depth. This implies that $gvar_{lm}(\sigma, i + rd_{\text{off}}) = kalloc(gvar_{lm}(\sigma, i), hp)$ is also valid in the concrete kernel.

**Case 3:** For a heap variable $x = gvar_{hm}(i)$ in the abstract kernel, the only validity criterion is that the heap index is within bounds, that is $i < |c_{\text{abs}}.m.hm|$.

$hp(i)$ is an index within the concrete heap, since the concrete heap is bounded by $hmap_{\text{bound}}(c_{\text{abs}}.m, c_k.m, hp)$.

So, $gvar_{hm}(hp(i)) = kalloc(gvar_{hm}(i), hp)$ is a valid concrete variable.

**Case 4:** We come to the inductive cases of array and structure variables. Both cases are very similar, hence we only elaborate on $x = gvar_{arr}(a, i)$.

Validity for array variables means that the root g-variable $a$ is valid and of array type, and that the index $i$ is less than the array size.

From Lemma 8.1 we obtain that $kalloc(a, hp)$ has the same type as $a$. From the induction hypothesis, we know that it is also valid. Hence, $gvar_{arr}(kalloc(a, hp), i) = kalloc(gvar_{arr}(a, i), hp)$ is a valid g-variable of the concrete kernel. q.e.d.

**Lemma 8.3 (Root g-Variable with *kalloc*)** *If $g$ is a root g-variable, then $kalloc(g)$ is a root variable, too:*

$$root_g(g) = g \implies root(kalloc(g, hp)) = kalloc(g, hp)$$

Proof The proof is straightforward, basing on a case distinction over the type of variables, using the definition of *kalloc*. q.e.d.

**Lemma 8.4 (Transitivity of Sub g-Variables)** *The relation $sub_g$ for sub g-variables is transitive under kalloc.*

$$\begin{aligned} &x \in gvar_{\surd}(sc(c_{abs}.m) \wedge y \in gvar_{\surd}(sc(c_{abs}.m) \\ &hmap_{inv}(c_{abs}.m, hp) \\ &\implies \\ &\quad (x \in sub_g(y) \iff kalloc(x, hp) \in sub_g(kalloc(y, hp))) \end{aligned}$$

Proof We split the proof into two cases.

**Case 1:** At first, we start with

$$x \in sub_g(y) \implies kalloc(x, hp) \in sub_g(kalloc(y, hp)).$$

We proceed with an induction over the g-variable $x$. For the base cases—global, local, and heap variables—the proof is trivial, since sub g-variable means identity in these cases (cf. Def. 3.4). Let us consider the global variable case exemplarily:

$$gvar_{gm}(\sigma) \in sub_g(y) \implies gvar_{gm}(\sigma) = y$$

So we have to show that

$$kalloc(gvar_{gm}(\sigma), hp) \in sub_g(kalloc(gvar_{gm}(\sigma), hp)),$$

which is covered by the base case in the definition of sub g-variables.

For the inductive cases, we concentrate on array variables, that is $x = gvar_{arr}(a, i)$.

There are now two possibilities: since $gvar_{arr}(a, i) \in sub_g(y)$, either $y$ is identical to $gvar_{arr}(a, i)$ or $a$ is a sub g-variable of $y$. The first possibility is covered again by the base case of sub g-variables. For the second one, we use the induction hypothesis to obtain that $kalloc(a, hp) \in sub_g(kalloc(y, hp))$, which by definition implies that

$$kalloc(gvar_{arr}(a, i), hp) \in sub_g(kalloc(y, hp)).$$

**Case 2:** Now we start with the assumption that

$$kalloc(x, hp) \in sub_g(kalloc(y, hp)).$$

We proceed again with an induction over $x$. Here, the base cases are very straightforward. For the heap variable case we additionally need heap map injectivity to obtain a unique index.

In the array case, we have again the two possibilities. Assuming that $gvar_{arr}(kalloc(a, hp), i) = kalloc(y, hp)$, we have to prove $gvar_{arr}(a, i) = y$—which is true due to the injectivity of *kalloc*. For the other possibility, we use the induction hypothesis to obtain $a \in sub_g(y)$, which trivially implies by definition that

$$gvar_{arr}(a, i) \in sub_g(y). \qquad \text{q.e.d.}$$

Nearly in all proofs in this chapter, we need to show that the concrete kernel's program rest starts with the same statement as the abstract kernel's program rest—modulo the statement id (due to renumbering, see Def. 6.5).

**Lemma 8.5 (Equivalence of Program Rest Heads)** *Let the consistency relation for program rests $konsis_{rd}$ hold for two $C0$ configurations $c_{abs}$ and $c_k$. Then the heads of the program rest of these two configurations are equivalent under the function $eq_{stmt}$:*

$$konsis_{prog}(c_{abs}.prog, c_k.prog) \wedge (s2l(c_{abs}.prog)) \neq [\,] \\ \implies eq_{stmt}(hd(s2l(c_{abs}.prog)), hd(s2l(c_k.prog)))$$

Proof From the definition of *s2l* (cf. Sect. 3.2.3), we know that its result is a list of length at least 1, hence

$$0 < |s2l(c_{\text{abs}}.prog)|$$

and

$$0 < |s2l(c_k.prog)|,$$

or in other words that there is at least one element in each of the two lists, i.e. there exists a head in each of them.

Using $konsis_{\text{prog}}(c_{\text{abs}}.prog, c_k.prog)$ and the definition of program rest consistency (cf. Def. 7.10), which says that for all elements in $s2l(c_{\text{abs}}.prog)$ the corresponding element in $s2l(c_k.prog)$ is equivalent under $eq_{\text{stmt}}$, we finally obtain.

$$eq_{\text{stmt}}(s2l(c_{\text{abs}}.prog)!0, s2l(c_k.prog)!0) \qquad \text{q.e.d.}$$

## 8.2 Weak Validity of $C0$ Configurations

The Isabelle/HOL versions of the proofs presented in this chapter make extensive use of existing lemmas about the $C0$ small-step semantics. These lemmas often have quite big assumptions, a lot of them dealing with validity of $C0$ configurations in the sense of [Lei08, Sect. 5.5, Def. 5.38] or at least parts of them.

The definition of valid $C0$ configurations is huge and the detailed definition does not contribute to a better understanding of this work, hence we will omit it here.

Nonetheless we have to mention that configurations encoding the abstract kernel cannot be valid in the sense of this definition, since not all of the functions in the abstract procedure table are *valid*, for which [Lei08, Def. 5.14] requires that

- its body is a valid statement,
- the local and parameter symbol tables are valid,
- the last and only the last statement of the function body is a *Return* statement, and
- the type of the return expression matches the return type of the function.

Since the abstract kernel necessarily has external functions, whose body merely consists of a *Skip* statement, the last two requirements cannot be met by all abstract kernel functions. Thus, we introduce the notion of *weak valid functions*, for which we only require that the body and the local and parameter symbol tables are valid.

The definition of valid procedure tables ([Lei08, Def. 5.17]) makes use of the notion of valid functions, as it requires that all functions in that procedure table have to be valid in the strong sense.

Hence we also have to come up with a weakened version for valid procedure tables, where we distinguish if a function is called via an *SCall* statement and has a body different from a single *Skip* statement. We start with the definition of two auxiliary functions, before we proceed with weak validity for procedure tables.

**Definition 8.1 (Exists Statement)** We define a predicate $exists_{\text{stmt}}$ taking two arguments, a statement $s$ and a predicate over statements $P$. $exists_{\text{stmt}}$

evaluates to *True*, iff at least one of the sub statements of $s$ satisfies $P$, formally:

$$\begin{aligned}
exists_{\text{stmt}}(Skip, P) &= P(Skip) \\
exists_{\text{stmt}}(Comp(s_1, s_2), P) &= P(Comp(s_1, s_2)) \\
&\quad \vee exists_{\text{stmt}}(s_1, P) \\
&\quad \vee exists_{\text{stmt}}(s_2, P) \\
exists_{\text{stmt}}(Ass(e), P) &= P(Ass(e)) \\
exists_{\text{stmt}}(PAlloc(e), P) &= P(PAlloc(e)) \\
exists_{\text{stmt}}(Return(e), P) &= P(Return(e)) \\
exists_{\text{stmt}}(SCall(e, fn, plist), P) &= P(SCall(e, fn, plist)) \\
exists_{\text{stmt}}(ESCall(e, fn, plist), P) &= P(XCall(e, fn, plist)) \\
exists_{\text{stmt}}(XCall(fn, elist, plist), P) &= P(ESCall(e, fn, plist)) \\
exists_{\text{stmt}}(Ifte(e, s_1, s_2), P) &= P(Ifte(s_1, s_2)) \vee exists_{\text{stmt}}(s_1, P) \\
&\quad \vee exists_{\text{stmt}}(s_2, P) \\
exists_{\text{stmt}}(Loop(e, s), P) &= P(Loop(e, s)) \vee exists_{\text{stmt}}(s, P)
\end{aligned}$$

**Definition 8.2 (Used Function)** We use the above definition to define the notion of a *used function*. A function with name *fn* is *used* in a procedure table, if there exists a statement $s$ in any of the function bodies of *pt* with $s$ being a function call of *fn*. We describe this property by $func_{\text{used}} : \Sigma^+ \times proctableT \to \mathbb{B}$ and define recursively:

$$\begin{aligned}
func_{\text{used}}(fn, [\,]) &= False \\
func_{\text{used}}(fn, (x, f)\#xs) &= (exists_{\text{stmt}}(x.body, \\
&\quad \lambda s.\, \exists e, plist.\, s = SCall(e, fn, plist))
\end{aligned}$$

This definition allows us to weaken the definition of valid procedure tables by a case split over its functions:

- For a *used function* and for all functions with a body different from a single *Skip* statement, we simply use the requirements of valid functions.
- For all others, we require that they only satisfy the conditions of weak validity of functions.

In simple words, this definition says that for defined and used functions, we leave the validity predicate as it is defined in [Lei08], while we use the weakened form for functions which are only declared and not called.

This definition of weak procedure table validity is finally used to define weak valid $C0$ configurations, which we denote with $conf_{\surd}^{\text{weak}}$ compared to $conf_{\surd}$. Here again, we leave all requirements of strong validity unchanged and merely replace the procedure table validity by its weakened form.

Our Isabelle/HOL proofs use approximately 50 lemmas from [Lei08] with most of them assume a given valid $C0$ configuration. For all of these lemmas we have shown, that weak $C0$ configuration validity is sufficient, so their use has been justified.

Similar to [Lei08, Lemma 5.17], there exists a lemma that weak validity is preserved by the $C0$ transition function.

**Lemma 8.6 (Preservation of Weak Validity)** *Given a weak valid C0 configuration $c_{C0}$ with respect to a page table pt and a type name environment te. If there exists a successor configuration $c_{C0}{}'$ obtained by applying the transition function, then $c_{C0}{}'$ is weak valid, too.*

$$c_{C0} \in conf_{\surd}^{weak}(te, pt) \wedge \delta_{C0}(te, pt, c_{C0}) = \lfloor c_{C0}{}' \rfloor$$
$$\Longrightarrow$$
$$c_{C0}{}' \in conf_{\surd}^{weak}(te, pt)$$

## 8.3 Expression Evaluation in the two Kernels

Expression evaluation plays a crucial role in the $C0$ small-step semantics—hence also in the correctness proofs presented in this chapter.

In this section, we will consider expressions in the abstract and concrete kernel. We will formulate and prove lemmas about their type, validity and the results of their evaluation.

**Lemma 8.7 (Type of Expressions Equality)** *Let $te_k$ be the type name environment the concrete kernel, which is encoded by a valid configuration $c_k$. Furthermore, let the consistency relations for recursion depth, global and local symbol tables hold, and let e denote an expression of type t, which is valid in the abstract kernel. Then it is also of type t in the concrete kernel.*

$$linkable(\Pi_{abs}, \Pi_{CVM}) \wedge \Pi_k = link(\Pi_{abs}, \Pi_{CVM}) \wedge$$
$$konsis_{rd}(c_{abs}.m, c_k.m) \wedge konsis_{lst}(c_{abs}.m, c_k.m) \wedge konsis_{tenv}(te_{abs}, t_k) \wedge$$
$$konsis_{gst}(c_{abs}.m, c_k.m) \wedge e \in valid_{exprs}(te_{abs}, toplst(c_{abs}.m), gst(c_{abs}.m)) \wedge$$
$$c_k \in conf_{\surd}(te_k, pt_k) \wedge type(te_{abs}, toplst(c_{abs}.m), gst(c_{abs}.m, ), e) = \lfloor t \rfloor$$
$$\Longrightarrow$$
$$type(te_k, toplst(c_k.m), gst(c_k.m, ), e) = \lfloor t \rfloor$$

Proof We will prove this lemma by induction over $e$. There are two base cases, namely *literals* and *variable accesses*. The literal case is trivial since purely syntactical, so we only present the variable access case.

**Case 1:** For a variable access $e = Var(\sigma)$, there are two possibilities: either we are dealing with a local variable or with a global variable (see also Sect. 3.4.3).

Assuming that $\sigma$ is the name of a local variable, we derive that $(\sigma, t) \in \{toplst(c_{\text{abs}}.m)\}$. By definition of *toplst* we know that this is the symbol table of the topmost stack frame, that is $sc.lst(c_{\text{abs}}.m)!(|c_{\text{abs}}.m.lm| - 1)$, with $|c_{\text{abs}}.m.lm| - 1$ denoting the recursion depth of the abstract kernel.

From $konsis_{\text{lst}}(c_{\text{abs}}.m, c_k.m)$ we obtain

$$sc(c_{\text{abs}}.m).lst!(|c_{\text{abs}}.m.lm| - 1) = sc(c_k.m).lst!(|c_{\text{abs}}.m.lm| - 1 + rd_{\text{off}}),$$

hence $(\sigma, t) \in \{sc(c_k.m).lst!(|c_{\text{abs}}.m.lm| - 1 + rd_{\text{off}})\}$.

Exploiting the recursion depth consistency relation, we know that this is exactly the recursion depth of the concrete kernel in configuration $c_k$ and can conclude:

$$\begin{aligned} & |c_{\text{abs}}.m.lm| - 1 + rd_{\text{off}} = |c_k.m.lm| - 1 \\ \implies & (\sigma, t) \in \{sc(c_k.m).lst!(|c_k.m.lm| - 1)\} \\ \implies & (\sigma, t) \in \{toplst(c_k.m)\} \end{aligned}$$

and—using the definition of *type*—this finally implies

$$type(te_k, toplst(c_k.m), gst(c_k.m, ), e) = \lfloor t \rfloor$$

For global variables, the proof is even simpler: knowing that $(\sigma, t) \in \{gst(c_{\text{abs}}.m)\}$, which is equivalent to $(\sigma, t) \in \{sc.gst(c_{\text{abs}}.m)\}$, we obtain that this pair also has to be an element of the linked global symbol table—since the abstract one is a subset of it as given by the relation $konsis_{\text{gst}}$:

$$\begin{aligned} & (\sigma, t) \in \{gst(c_{\text{abs}}.m)\} \\ \implies & (\sigma, t) \in \{gst(c_k.m)\} && \text{(using } konsis_{\text{gst}}\text{)} \\ \implies & (\sigma, t) \in \{gst(c_{\text{abs}}.m)\} && \text{(Def. of } gst\text{)} \\ \implies & type(te_k, toplst(c_k.m), gst(c_k.m, ), Var(\sigma)) = \lfloor t \rfloor && \text{(Def. of } type\text{)} \end{aligned}$$

**Case 2:** The proofs for the arithmetic operations are all very similar, since they merely rely on instantiating the induction hypotheses. Hence we present only the proof for binary arithmetic operations, that is $e = BinOp(op, e_1, e_2)$. From the fact that $e$ is a valid expression of the abstract kernel, we obtain that also the sub expressions are valid

$$\begin{aligned} e_1 &\in valid_{\text{exprs}}(te_{\text{abs}}, toplst(c_{\text{abs}}.m), gst(c_{\text{abs}}.m)) \\ e_2 &\in valid_{\text{exprs}}(te_{\text{abs}}, toplst(c_{\text{abs}}.m), gst(c_{\text{abs}}.m)) \end{aligned}$$

and there exist two types $t_1$ and $t_2$ for these sub expressions:

$$\begin{aligned} type(te_{\text{abs}}, toplst(c_{\text{abs}}.m), gst(c_{\text{abs}}.m, ), e_1) &= \lfloor t_1 \rfloor \\ type(te_{\text{abs}}, toplst(c_{\text{abs}}.m), gst(c_{\text{abs}}.m, ), e_2) &= \lfloor t_2 \rfloor. \end{aligned}$$

We can now use the induction hypothesis to show that the sub expression types are equivalent for the concrete kernel, that is

$$\begin{aligned} type(te_k, toplst(c_k.m), gst(c_k.m, ), e_1) &= \lfloor t_1 \rfloor \\ type(te_k, toplst(c_k.m), gst(c_k.m, ), e_2) &= \lfloor t_2 \rfloor \end{aligned}$$

which we then finally use to conclude

$$type(te_k, toplst(c_k.m), gst(c_k.m, ), BinOp(op, e_1, e_2)) = \lfloor t \rfloor,$$

since the type of *BinOp* only depends on the types of the sub-expressions.

**Case 3:** Let $e$ denote a pointer dereferencing expression, that is $e = \mathit{Deref}(ex)$. This means that

$$\mathit{type}(te_{\text{abs}}, \mathit{toplst}(c_{\text{abs}}.m), \mathit{gst}(c_{\text{abs}}.m), \mathit{Deref}(ex)) = \lfloor t \rfloor.$$

From the assumption that $e$ is a valid expression in the abstract kernel, we obtain that the sub expression $ex$ is valid and of a pointer type:

$$\begin{aligned} &ex \in \mathit{valid}_{\text{exprs}}(te_{\text{abs}}, \mathit{toplst}(c_{\text{abs}}.m), \mathit{gst}(c_{\text{abs}}.m)) \\ &\mathit{type}(te_{\text{abs}}, \mathit{toplst}(c_{\text{abs}}.m), \mathit{gst}(c_{\text{abs}}.m), ex) = \lfloor \mathit{Ptr}(tn) \rfloor \end{aligned}$$

and furthermore there is a corresponding element in the abstract type name environment, such that

$$(tn, t) \in \{te_{\text{abs}}\}.$$

Using the above facts together with the induction hypothesis, we obtain for the type of $ex$

$$\mathit{type}(te_k, \mathit{toplst}(c_k.m), \mathit{gst}(c_k.m), ex) = \lfloor \mathit{Ptr}(tn) \rfloor.$$

Due to type name environment consistency, we know that $(tn, t)$ is also an element in the concrete type name environment:

$$(tn, t) \in \{te_k\}. \tag{8.1}$$

By the definition, this implies for the type of the comprising dereferencing expression:

$$\mathit{type}(te_k, \mathit{toplst}(c_k.m), \mathit{gst}(c_k.m), \mathit{Deref}(ex)) = \lfloor t \rfloor.$$

**Case 4:** The proof works in a similar way for an Address-Of expression, that is $e = \mathit{AddrOf}(ex)$ with

$$\mathit{type}(te_{\text{abs}}, \mathit{toplst}(c_{\text{abs}}.m), \mathit{gst}(c_{\text{abs}}.m), \mathit{AddrOf}(ex)) = \lfloor \mathit{Ptr}(tn) \rfloor.$$

From the assumption that $\mathit{AddrOf}(ex)$ is valid we obtain for the sub expression $ex$ that there exists a type $t$ with

$$\mathit{type}(te_{\text{abs}}, \mathit{toplst}(c_{\text{abs}}.m), \mathit{gst}(c_{\text{abs}}.m), ex) = \lfloor t \rfloor$$

and

$$(tn, t) \in \{te_{\text{abs}}\}.$$

We use the induction hypothesis to obtain that this also holds in the concrete kernel:

$$\mathit{type}(te_k, \mathit{toplst}(c_k.m), \mathit{gst}(c_k.m), ex) = \lfloor t \rfloor.$$

From this, equation 4, and the fact that because of type name environment consistency $te_{\text{abs}}$ is a subset of $te_k$, we know that $(tn, t) \in \{te_k\}$ and finally

$$\mathit{type}(te_k, \mathit{toplst}(c_k.m), \mathit{gst}(c_k.m), \mathit{AddrOf}(ex)) = \lfloor \mathit{Ptr}(tn) \rfloor.$$

REMARK (ISABELLE/HOL) The above argumentation neglects the fact that there could be more type names in $te_k$ associated to the type $t$, hence the conclusion in the last step would not be justified.

In Isabelle, type name environments are modeled as lists and the function *mapof* returns the first witness in a list that satisfies the criterion.

By construction (cf. Def. 6.3) of $te_k$, the elements originally originating from the abstract type name environment always come before those of the low-level CVM implementation. So, in both cases the result of the *mapof* operation on the type name environments of the abstract and concrete kernel is $(tn, t)$.

**Case 5:** We will omit the detailed proofs for both the array element access case $e = Arr(e_a, e_i)$ and the structure access case $e = Str(e_s, cn)$. Both proofs are very similar to the one presented above for binary operations.

For the array case, we deduce the validity and types for both sub expressions and instantiate then the induction hypothesis adequately. For structure components its even simpler, since there is only one sub expression and the component name is static. q.e.d.

**Lemma 8.8 (Transitivity of Valid Expressions)** *Let $te_k$ denote the type name environment of the concrete kernel, which is given by a valid configuration $c_k$. Furthermore, let the consistency relations for recursion depth, global and local symbol tables hold, and let e denote an expression, which is valid in the abstract kernel. Then e is also valid in the concrete kernel.*

$$
\begin{aligned}
&linkable(\Pi_{abs}, \Pi_{CVM}) \wedge \Pi_k = link(\Pi_{abs}, \Pi_{CVM}) \wedge \\
&konsis_{rd}(c_{abs}.m, c_k.m) \wedge konsis_{lst}(c_{abs}.m, c_k.m) \wedge konsis_{tenv}(te_{abs}, t_k) \wedge \\
&konsis_{gst}(c_{abs}.m, c_k.m) \wedge e \in valid_{exprs}(te_{abs}, toplst(c_{abs}.m), gst(c_{abs}.m)) \wedge \\
&c_k \in conf_{\surd}(te_k, pt_k) \\
&\Longrightarrow \\
&\quad e \in valid_{exprs}(te_k, toplst(c_k.m), gst(c_k.m))
\end{aligned}
$$

PROOF We will prove this lemma by induction over $e$. For most cases, the proof is trivial and we will omit the details.

**Case 1:** For literals, that is $e = Lit(l)$, the definition says that $l$ is a valid literal. Since this $l$ is syntactically the same in the concrete kernel, $e$ is also valid there.

**Case 2:** Let $e = Var(\sigma)$ denote a local or global variable access. Due to similarities we will only consider the local variable case.

Expanding the definition of valid expressions, we obtain that $\sigma$ is defined in the topmost local symbol table:

$$(\sigma, t) \in \{toplst(c_{\text{abs}}.m)\}.$$

From $konsis_{\text{lst}}(c_{\text{abs}}.m, c_k.m)$ we know that the local symbol tables in both kernels are equal modulo the shift constant $rd_{\text{off}}$ regarding the recursion depth.

Using $konsis_{\text{rd}}$, we derive that $(\sigma, t)$ is also defined in the topmost local symbol table of the abstract kernel, which satisfies the criterion for a valid variable access expression.

**Case 3:** For $e = Deref(e_1)$, i.e. pointer dereferencing, we expand the definition of valid expressions and obtain:

- $e_1$ is valid in the abstract kernel,
- it is of a pointer type $t = Ptr_T(tn)$, that is $mapof(te_{\text{abs}}, tn) = \lfloor t \rfloor$, and
- the type name is defined in the type name environment of the abstract kernel.

We use the induction hypothesis to derive that $e_1$ is also valid in the concrete kernel. Due to the type name relation $konsis_{\text{tenv}}(te_{\text{abs}}, te_k)$, we know that $(tn, t) \in \{te_k\}$.

Finally, we use Lemma 8.7 to obtain that $e$ is of pointer type in the concrete kernel, too. These are all the requirements for a valid pointer dereferencing in the concrete kernel.

**Case 4:** Let $e = AddrOf(e_1)$ denote an Address-Of expression. Then, $e_1$ is valid, too, and its type $t$ is defined in the abstract type name environment, i.e. $t \in \{map(snd, te_{\text{abs}})\}$. Moreover, $e_1$ is a memory object.

From the induction hypothesis we obtain that $e_1$ is also valid in the concrete kernel. Using Lemma 8.7 yields that $e_1$ is of type $t$ in the concrete kernel, too, while the invariant $konsis_{\text{tenv}}$ guarantees that $t \in \{map(snd, te_k)\}$.

This is all we need to prove that $e$ is a valid expression in the concrete kernel.

**Case 5:** We will not present the proof for the structure case, since it is very similar to the array case $e = Arr(e_a, e_i)$.

From the assumption that $e$ is valid, we derive that both sub expressions are valid, too, and that $e_i$ is of a numerical type.

Using the induction hypothesis and expression type equality from Lemma 8.7, we prove this case. q.e.d.

**Lemma 8.9 (Relating Expression Evaluation)** *Given $c_{abs}$, a weak valid abstract kernel configuration, and $c_k$ a valid configuration of the concrete kernel, which has been obtained by abstract linking. Furthermore, let the invariants for the global and local symbol tables, the recursion depth and the type name environment as well as the invariant for g-variables hold. For any valid expression e in the abstract kernel of an elementary type, which can be right-hand evaluated, (i) there also exists a right-hand value in the concrete kernel, (ii) which is equivalent under the $eq_{cont}$ relation, if e is initialized. Furthermore, the initialization*

*of $e$ is equal in both kernels, and if $e$ is a memory object in the abstract kernel, then it is one in the concrete kernel, too.*

*Ultimately, if $e$ can be left evaluated to some g-variable $g$ in the abstract kernel, the left evaluation in the concrete kernel will return the corresponding g-variable $kalloc(g, hp)$.*

*We define this formally as follows:*

$$
\begin{aligned}
&c_k \in conf_{\surd}(te_k, pt_k) \wedge linkable(\Pi_{abs}, \Pi_{CVM}) \wedge \\
&c_{abs} \in conf_{\surd}^{weak} \wedge konsis_{tenv}(te_{abs}, t_k) \wedge \\
&\Pi_k = link(\Pi_{abs}, \Pi_{CVM}) \wedge konsis_{rd}(c_{abs}.m, c_k.m) \wedge konsis_{lst}(c_{abs}.m, c_k.m) \wedge \\
&konsis_{gst}(c_{abs}.m, c_k.m) \wedge e \in valid_{exprs}(te_{abs}, toplst(c_{abs}.m), gst(c_{abs}.m)) \wedge \\
&konsis_{gvars}(te_{abs}, c_{abs}.m, te_k, c_k.m, hp) \wedge \\
&type(te_{abs}, toplst(c_{abs}.m), gst(c_{abs}.m), e) = \lfloor t \rfloor \wedge \\
&rval(te_{abs}, c_{abs}.m, e) = \lfloor v \rfloor \\
&\Longrightarrow \\
&\quad rval(te_k, c_k.m, e) = \lfloor y \rfloor \wedge \\
&\quad ?init(te_{abs}, c_{abs}.m, e) = ?init(te_k, c_k.m, e) \wedge \\
&\quad ?inter(te_{abs}, c_{abs}.m, e) = ?inter(te_k, c_k.m, e) \wedge \\
&\quad (\forall g.\, lval(te_{abs}, c_{abs}.m, e) = \lfloor g \rfloor \implies lval(te_k, c_k.m, e) = \lfloor kalloc(g, hp) \rfloor) \wedge \\
&\quad (?init(te_{abs}, c_{abs}.m, e) \wedge ?elem_T(t) \implies eq_{cont}(t, v, y))
\end{aligned}
$$

Proof Proof by induction over the expression $e$. We will omit the trivial cases of arithmetic operations and literals.

**Case 1:** Let $e$ denote a variable access, that is $e = Var(\sigma)$. From the fact that this is a valid expression of the abstract kernel, we can derive that $\sigma$ is defined either in the global symbol table or in the topmost local symbol table:

$$(\sigma, t) \in \{toplst(c_{\text{abs}}.m)\} \vee (\sigma, t) \in \{gst(c_{\text{abs}}.m)\}$$

The proof for both the local and the global case are very similar, thus we will only work out on the first case, i.e. $\sigma$ is the name of a local variable. Hence, the left-hand value of $e$ is

$$lval(te_{\text{abs}}, c_{\text{abs}}.m, Var(\sigma)) = \lfloor gvar_{lm}(|c_{\text{abs}}.m.lm| - 1, \sigma) \rfloor.$$

From $konsis_{\text{lst}}(c_{\text{abs}}.m, c_k.m)$ we know that the abstract kernel's local symbol tables are equivalent to those in the concrete kernel—shifted by the offset $rd_{\text{off}}$. Together with the type equivalence for expression evaluation from Lemma 8.7 and the recursion depth consistency between the abstract and the concrete kernel we obtain:

$$(\sigma, t) \in \{toplst(c_k.m)\}$$

and hence from the definition of expression evaluation

$$
\begin{aligned}
lval(te_k, c_k.m, Var(\sigma)) \quad = \quad & \lfloor gvar_{lm}(|c_k.m.lm| - 1, \sigma) \rfloor \\
& \lfloor kalloc(gvar_{lm}(|c_{\text{abs}}.m.lm| - 1, \sigma), hp) \rfloor.
\end{aligned}
$$

This satisfies the proof goal regarding the left-hand evaluation of variable accesses.

In a (weak) valid configuration, any expression that can be left evaluated, i.e. *lval* is not equal to *None*, yields a reachable g-variable (cf. [Lei08, Theorem 8.1]), hence

$$gvar_{lm}(|c_{\text{abs}}.m.lm| - 1, \sigma) \in reachable_{\text{named}}(c_{\text{abs}}.m) \tag{8.2}$$

and for the concrete kernel

$$\begin{aligned} &gvar_{lm}(|c_k.m.lm| - 1, \sigma) \in reachable_{\text{named}}(c_k.m) \\ &\quad \Longrightarrow\ kalloc(gvar_{lm}(|c_{\text{abs}}.m.lm| - 1, \sigma), hp) \in reachable_{\text{named}}(c_k.m) \end{aligned} \tag{8.3}$$

Since we have assumed g-variable consistency, we can now unpack the definition of $konsis_{\text{gvars}}$ and obtain that the initialization of both variables is equal:

$$\begin{aligned} &?init_g(gvar_{lm}(|c_{\text{abs}}.m.lm| - 1, \sigma)) \\ &= ?init_g(kalloc(gvar_{lm}(|c_{\text{abs}}.m.lm| - 1, \sigma), hp)). \end{aligned}$$

From the definition of expression evaluation for variable access we know that

$$\begin{aligned} &?init(te_{\text{abs}}, c_{\text{abs}}.m,\ Var(\sigma)) \\ &= \sigma \in toplm(c_{\text{abs}}.m).init \\ &= ?init_g(gvar_{lm}(|c_{\text{abs}}.m.lm| - 1, \sigma)), \qquad \text{(Def. 3.11)} \end{aligned}$$

and hence

$$?init(te_{\text{abs}}, c_{\text{abs}}.m,\ Var(\sigma)) = ?init(te_k, c_k.m,\ Var(\sigma)).$$

For the rest of this case, let us assume that the variables, which are accessed, are initialized and of elementary type. We will now obtain the right-hand value *rval* for the variable access in the abstract kernel and relate it to the value of the g-variable given through the left-hand value.

Note that since $t$ is elementary, the corresponding type size $size_T$ equals one:

$$\begin{aligned} &rval(te_{\text{abs}}, c_{\text{abs}}.m,\ Var(\sigma)) \\ &\quad = \lfloor toplm(c_{\text{abs}}.m).ct[ba_v(toplst(c_{\text{abs}}.m), \sigma), 1] \rfloor \qquad \text{(cf. Sect. 3.4.3)} \\ &\quad = \lfloor toplm(c_{\text{abs}}.m).ct[ba_g(sc(c_{\text{abs}}.m), gvar_{lm}(|c_{\text{abs}}.m.lm| - 1, \sigma)), 1] \rfloor \\ &\qquad\qquad \text{(Def. 3.12)} \end{aligned}$$

which is the value of the g-variable as defined in Def. 3.14:

$$\lfloor value_g(c_{\text{abs}}.m, gvar_{lm}(|c_{\text{abs}}.m.lm| - 1, \sigma)) \rfloor \tag{8.4}$$

We proceed in a similar way for the concrete kernel and obtain

$$\begin{aligned} &rval(te_k, c_k.m,\ Var(\sigma)) \\ &\quad = \lfloor value_g(c_k.m, gvar_{lm}(|c_k.m.lm| - 1, \sigma)) \rfloor \\ &\quad = \lfloor value_g(c_k.m, kalloc(gvar_{lm}(|c_{\text{abs}}.m.lm| - 1, \sigma), hp)) \rfloor. \end{aligned} \tag{8.5}$$

We will now expand the $konsis_{\text{gvars}}$ invariant (cf. Def. 7.16). Using the fact that $gvar_{lm}(|c_{\text{abs}}.m.lm| - 1, \sigma)$ is reachable (8.2) and the assumptions on initialization and type, we obtain

$$eq_{val}(gvar_{lm}(|c_{\text{abs}}.m.lm| - 1, \sigma), c_{\text{abs}}.m, \\ kalloc(gvar_{lm}(|c_{\text{abs}}.m.lm| - 1, \sigma), hp), c_k.m)$$

which we expand to

$$eq_{\text{cont}}(t, value_g(c_{\text{abs}}.m, gvar_{lm}(|c_{\text{abs}}.m.lm| - 1, \sigma)), \\ value_g(c_k.m, kalloc(gvar_{lm}(|c_{\text{abs}}.m.lm| - 1, \sigma), hp))$$

and obtain finally using (8.4) and (8.5):

$$eq_{\text{cont}}(t, the(rval(te_{\text{abs}}, c_{\text{abs}}.m, Var(\sigma))), the(rval(te_k, c_k.m, Var(\sigma)))).$$

**Case 2:** We will now deal with pointer dereferencing, i.e. $e = Deref(e_1)$. Since $rval(te_{\text{abs}}, c_{\text{abs}}.m, e) = \lfloor v \rfloor$, we know that the right-hand expression evaluation of the sub expression $e_1$ yields a non-null pointer, that is

$$rval(te_{\text{abs}}, c_{\text{abs}}.m, e_1) = \lfloor Ptr(p) \rfloor \wedge \\ p \neq NullPointer$$

where the type of $e_1$ is given through

$$type(te_{\text{abs}}, toplst(c_{\text{abs}}.m), gst(c_{\text{abs}}.m), e_1) = \lfloor Ptr_T(tn) \rfloor \tag{8.6}$$

and

$$mapof(te_{\text{abs}}, tn) = \lfloor t \rfloor. \tag{8.7}$$

This means that the left-hand value of $Deref(e_1)$ in the abstract kernel is the g-variable that $e_1$ points to, that is

$$lval(te_{\text{abs}}, c_{\text{abs}}.m, Deref(e_1)) = \lfloor p \rfloor, \tag{8.8}$$

and the right-hand evaluation is equal to the value of this variable $p$:

$$rval(te_{\text{abs}}, c_{\text{abs}}.m, Deref(e_1)) = value_g(c_{\text{abs}}.m, p). \tag{8.9}$$

Using the induction hypothesis, we know that there exists a $y'$ such that

$$rval(te_k, c_k.m, e_1) = \lfloor y' \rfloor$$

and

$$eq_{\text{cont}}(Ptr_T(tn), Ptr(p), y').$$

Expanding the definition of $eq_{\text{cont}}$ for non-null pointer types, we obtain

$$y' = Ptr(kalloc(p, hp)). \tag{8.10}$$

Furthermore, we derive that since $e_1$ is initialized in the abstract kernel, so it is in the concrete kernel—$?init(te, m_k, e_1) = True$—, and that the types of $e_1$ are equal in both kernel configurations (see equation (8.6), using Lemma 8.7).

The invariant $konsis_{\text{tenv}}$ ensures that the type associated with the type name $tn$ in the concrete kernel is the same type as in the abstract kernel, namely $t$.

Finally, since $p$ is not a local variable, so $kalloc(p)$ is not local neither by definition of $kalloc$.

Since all of the above facts ensure also the successful expression evaluation for $Deref(e_1)$ in the concrete kernel, too, we can now compute the corresponding results. From (8.10) we can derive the left-hand value of $Deref(e_1)$ as

$$lval(te_k, c_k.m, Deref(e_1)) = \lfloor kalloc(p, hp) \rfloor. \tag{8.11}$$

and for the right-hand value

$$rval(te_k, c_k.m, Deref(e_1)) = value_g(c_k.m, kalloc(p, hp)). \tag{8.12}$$

Both the abstract kernel and the concrete kernel configurations are valid, which implies that the left-hand values obtained by expression evaluation yields reachable variables. Furthermore we know that $t$ is an elementary type by assumption. We can now again unfold the $konsis_{\text{gvars}}$ invariant (Def. 7.16), by which we can syntactically show our goal.

**Case 3:** Let $e$ be an Address-Of expression, i.e. $e = AddrOf(e_1)$. Since the evaluation of this expression in the abstract kernel was successful, we can conclude that the assumptions regarding right-hand evaluation and type on the sub expression $e_1$ have been satisfied, too, that is

$$lval(te_{\text{abs}}, c_{\text{abs}}.m, e_1) = \lfloor p \rfloor$$

and

$$mapof(map(flip, te_{\text{abs}}), type(te_{\text{abs}}, toplst(c_{\text{abs}}.m), gst(c_{\text{abs}}.m), e_1)) = \lfloor tn \rfloor.$$

From the definition of expression evaluation we obtain for the right-hand value a content list with exactly one element, namely a pointer to $p$:

$$rval(te_{\text{abs}}, c_{\text{abs}}.m, AddrOf(e_1)) = \lfloor [Ptr(p)] \rfloor \tag{8.13}$$

Using the induction hypothesis, we obtain for the concrete kernel

$$lval(te_k, c_k.m, e_1) = \lfloor kalloc(p, hp) \rfloor \tag{8.14}$$

and

$$?inter(te_k, toplst(c_k.m), gst(c_k.m), e_1) = False \tag{8.15}$$

In addition, we use the type name invariant $konsis_{\text{tenv}}$ to derive

$$mapof(map(flip, te_k), type(te_k, toplst(c_k.m), gst(c_k.m), e_1)) = \lfloor tn \rfloor.$$

These are all requirements for a successful evaluation of $AddrOf(e_1)$ in the concrete kernel. Hence, the right-hand evaluation returns

$$rval(te_k, c_k.m, AddrOf(e_1)) = \lfloor [Ptr(kalloc(p, hp))] \rfloor. \tag{8.16}$$

Furthermore, the type of the expression is identical in both the abstract and the concrete kernel, namely

$$\begin{aligned} &type(te_{\text{abs}}, toplst(c_{\text{abs}}.m), gst(c_{\text{abs}}.m), AddrOf(e_1)) = \\ &type(te_k, toplst(c_k.m), gst(c_k.m), AddrOf(e_1)) = \\ &\lfloor Ptr_T(tn) \rfloor \end{aligned}$$

and both expressions are initialized by definition.

Using these facts together with the definition of $konsis_{\text{gvars}}$, we have completed the proof for this case.

**Case 4:** The cases for an array element access and a structure component access are very similar. Hence, we will only discuss the case of an array element access, i.e. $Arr(e_a, e_i)$.

We know that the sub expression $e_i$—the index expression—is of a numerical type, which is by definition an elementary type. Hence we can use the induction hypothesis to obtain that the initialization of $e_i$ is equal in both kernel configurations.

For the initialization of the array element access the definition of expression evaluation says that

$$\begin{aligned} &?init(te_{\text{abs}}, c_{\text{abs}}.m, Arr(e_a, e_i)) = \\ &\quad ?init(te_{\text{abs}}, c_{\text{abs}}.m, e_a) \wedge ?init(te_{\text{abs}}, c_{\text{abs}}.m, e_i). \end{aligned}$$

This means that in the case of an uninitialized sub expression $e_i$, the proof for the array case is already complete[1]. So let us assume that both sub expressions $e_a$ and $e_i$ are initialized.

In addition, an array access is a memory object, iff the array expression $e_a$ is a memory object. By using the induction hypothesis, we obtain

$$?inter(te_{\text{abs}}, c_{\text{abs}}.m, e_a) = ?inter(te_k, c_k.m, e_a)$$

and hence

$$?inter(te_{\text{abs}}, c_{\text{abs}}.m, Arr(e_a, e_i)) = ?inter(te_k, c_k.m, Arr(e_a, e_i)).$$

Let $i$ denote the right-hand value of $e_i$ in the abstract kernel and $i'$ the one for the concrete kernel, respectively. Furthermore let $t_i$ denote the type of $e_i$. Then we derive from the induction hypothesis that for the right-hand values of $e_i$ holds

$$eq_{\text{cont}}(t_i, i, i')$$

[1] We would just use the induction hypothesis for $e_i$ and obtain that it is uninitialized in the concrete kernel, too.

and since $t_i$ is elementary, this equation can be simplified to $i = i'$.

Let us consider the array expression $e_a$ now. The left-hand evaluation in the abstract kernel returns a g-variable $g$ of an array type $Arr_T(n, t)$. For the corresponding right-hand value we use Def. 3.14 and obtain

$$\begin{aligned} &rval(te_{\text{abs}}, c_{\text{abs}}.m, e_a) \\ &\quad = \lfloor value_g(c_{\text{abs}}.m, g) \rfloor \end{aligned}$$

Using the induction hypothesis, we left evaluate $e_a$ in the concrete kernel to $kalloc(g, hp)$ with a right-hand value

$$\begin{aligned} &rval(te_k, c_k.m, e_a) \\ &\quad = \lfloor value_g(c_k.m, kalloc(g, hp)) \rfloor. \end{aligned}$$

If $t$ is not elementary, then the proof for the array access case is done now. So let us assume $?elem_T(t)$. Then we can expand the right-hand evaluation as follows:

$$\begin{aligned} &rval(te_{\text{abs}}, c_{\text{abs}}.m, e_a) \\ &\quad = \lfloor value_g(c_{\text{abs}}.m, g) \rfloor \\ &\quad = \lfloor m_{mem_g(g)}[ba_g(sc(c_{\text{abs}}.m), g), size_T(type_g(sc(m), g))] \rfloor \\ &\quad = \lfloor m_{mem_g(g)}[ba_g(sc(c_{\text{abs}}.m), g), n \cdot size_T(t)] \rfloor \\ &\quad = \lfloor m_{mem_g(g)}[ba_g(sc(c_{\text{abs}}.m), g), n] \rfloor \\ &\quad = \lfloor v_a \rfloor \end{aligned}$$

We proceed likewise with $kalloc(g, hp)$ in the concrete kernel and obtain its value $v'_a$.

From the preconditions for array element access evaluation we know that $i < n$, that is the index expression yields an index which is smaller than the array size.

This means that the left-hand value—$gvar_{arr}(g, i)$—is a sub g-variable of $g$ (see also Def. 3.4). Using Def. 3.12, we obtain the corresponding base address as

$$\begin{aligned} &ba_g(sc(c_{\text{abs}}.m), gvar_{arr}(g, i)) \\ &\quad = ba_g(sc(c_{\text{abs}}.m), g) + i \cdot size_T(t) \\ &\quad = ba_g(sc(c_{\text{abs}}.m), g) + i. && (\text{since } size_T(t) = 1) \end{aligned}$$

By definition of $value_g$, the value of this sub g-variable is given through

$$\begin{aligned} &value_g(c_{\text{abs}}.m, gvar_{arr}(g, i)) \\ &\quad = value_g(c_{\text{abs}}.m, g)!i \cdot size_T(t) \\ &\quad = value_g(c_{\text{abs}}.m, g)!i \\ &\quad = v_a!i \end{aligned}$$

For the concrete kernel we use $gvar_{arr}(kalloc(g, hp), i)$ and obtain a value $v'_a!i$.

$gvar_{arr}(g, i)$ has been obtained by evaluating a valid expression in a valid configuration, i.e. it is a *reachable* g-variable. Furthermore, it is initialized by assumption and of elementary type. As in the cases before, we unfold the definition of $konsis_{\text{gvars}}$ to obtain

$$\begin{aligned} & eq_{\text{val}}(t, value_g(c_{\text{abs}}.m, gvar_{arr}(g, i)), \\ & \quad value_g(c_k.m, kalloc(gvar_{arr}(g, i), hp)) \\ & = eq_{\text{val}}(t, value_g(c_{\text{abs}}.m, gvar_{arr}(g, i)), \\ & \quad value_g(c_k.m, gvar_{arr}(kalloc(g, hp), i)) \qquad \text{(using Def. of } kalloc\text{)} \end{aligned}$$

which is equivalent to

$$eq_{\text{cont}}(t, v_a!i, v'_a!i),$$

which is exactly the definition of the right-hand evaluation of the array element access. q.e.d.

## 8.4 Static Properties

In this section, we will deal with the static properties of the overall correctness relation between the abstract and the concrete kernel. Static means that these properties reason about the components of a $C0$ program, hence components that do not change during the kernel execution.

As there are three components for a $C0$ program (cf. Sect. 3.2)—a type name environment, a procedure table, and a global symbol table—, so there are three abstract relations that deal with these components:

1. $konsis_{\text{tenv}}$ is relating a type name environment to another,
2. $konsis_{\text{pt}}$ connects two procedure tables, and
3. $konsis_{\text{gst}}$ is talking about two global symbol tables.

**Lemma 8.10 (Preservation of $konsis_{\text{tenv}}$)** *Let $\Pi_{abs}$ and $\Pi_{CVM}$ denote an abstract kernel and a low-level CVM implementation, which are linkable with each other. Furthermore, let $\Pi_k$ denote the result of the abstract linking of these two programs.*

*Then the type name environment $te_{abs}$ of $\Pi_{abs}$ is consistent with the linked type name environment $te_k$:*

$$\begin{aligned} & linkable(\Pi_{abs}, \Pi_{CVM}) \wedge \\ & \Pi_k = link(\Pi_{abs}, \Pi_{CVM}) \implies \\ & \quad konsis_{tenv}(te_{abs}, te_k) \end{aligned}$$

Proof From $\Pi_k = link(\Pi_{\text{abs}}, \Pi_{\text{CVM}})$, and here in particular

$$te_k = link_{\text{te}}(te_{\text{abs}}, te_{\text{CVM}}),$$

we know by construction (cf. Def. 6.11) that $te_k$ is pairwise distinct and that all elements in $te_{\text{abs}}$ are also elements of $te_k$, and hence

$$\{te_{\text{abs}}\} \subseteq \{te_k\},$$

which is exactly the definition of $konsis_{\text{tenv}}$ as given in Def. 7.6:

$$konsis_{\text{tenv}}(te_{\text{abs}}, te_k). \qquad \text{q.e.d.}$$

**Lemma 8.11 (Preservation of $konsis_{\text{pt}}$)** *Let $\Pi_{abs}$ and $\Pi_{CVM}$ denote two linkable programs, with $\Pi_k$ representing the corresponding linked program.*

*Then, the procedure table $pt_{abs}$ is consistent with the linked procedure table $pt_k$ under the function $konsis_{pt}$.*

$$\begin{aligned} &linkable(\Pi_{abs}, \Pi_{CVM}) \wedge \\ &\Pi_k = link(\Pi_{abs}, \Pi_{CVM}) \implies \\ &\quad konsis_{pt}(pt_{abs}, pt_k) \end{aligned}$$

Proof Let $(fn, f)$ denote any defined procedure of the abstract kernel procedure table, i.e. $(fn, f) \in \{pt_{\text{abs}}\}$.

From $linkable(\Pi_{\text{abs}}, \Pi_{\text{CVM}})$ we know that the names of the defined functions of $pt_{\text{abs}}$ and $pt_{\text{CVM}}$ are pairwise disjoint, that is:

$$\{map(fst, pt^{\text{def}}_{\text{abs}})\} \cap \{map(fst, pt^{\text{def}}_{\text{CVM}})\} = \emptyset.$$

The procedure table of the concrete kernel $pt_k$ is the result of

$$repl_{\text{pt}}(link_{\text{pt}}(pt_{\text{abs}}, pt_{\text{CVM}})),$$

as defined in Def. 6.10. This means that $(fn, f') \in \{pt_k\}$ with $(fn, f') = repl(pt_k, (fn, f))$ or in words: $f'$ is equivalent to $f$ except that external function calls are replaced by 'normal' function calls and that the statements have been renumbered (cf. Def. 6.10).

This is exactly the equivalence as defined by the relation $eq_{\text{stmt}}$, which has been introduced in Def. 7.8, hence:

$$\begin{aligned} &konsis_{\text{func}}(f, f') = \\ &\quad eq_{\text{stmt}}(f.body, f'.body) \wedge \\ &\quad (|s2l(f.body)| = |s2l(f'.body)|) \wedge \\ &\quad (f.params = f'.params) \wedge \\ &\quad (f.lvars = f'.lvars) \end{aligned}$$

Since we have picked an arbitrary defined abstract kernel function, we can extend this property for all defined functions in $pt_{\text{abs}}$:

$$(\forall p \in \{pt^{\text{def}}_{\text{abs}}\}. \exists q \in \{pt_k\}. konsis_{\text{func}}(p, q))$$

which is in fact the expanded definition of $konsis_{\text{pt}}(pt_{\text{abs}}, pt_k)$ as introduced in Def. 7.9. q.e.d.

The global symbol table is not only a component of the $C0$ program, but is also part of the (dynamic) memory configuration. Hence, the correctness lemma has to include certain assumptions about the next state function, though we show that the global symbol table does not change at all during the execution of the program.

**Lemma 8.12 (Preservation of $konsis_{gst}$)** *Given $c_{abs}$ and $c_k$ denoting $C0$ configurations encoding the abstract kernel and the concrete kernel, where the global symbol table of the abstract kernel is consistent with the one of the concrete kernel under the relation $konsis_{gst}$.*

*Under the assumption that the $C0$ transition function does not yield None for any of the two configurations, the relation will also hold for the two successor configurations $c'_{abs}$ and $c'_k$:*

$$\begin{aligned}
&\delta_{C0}(te_{abs}, pt_{abs}, c_{abs}) = \lfloor c'_{abs} \rfloor \wedge \\
&\delta_{C0}(te_k, pt_k, c_k) = \lfloor c'_k \rfloor \wedge \\
&konsis_{gst}(gst(c_{abs}.m), gst(c_k.m)) \implies \\
&\quad konsis_{gst}(gst(c'_{abs}.m), gst(c'_k.m))
\end{aligned}$$

Proof The proof idea is to show that the global symbol table—though part of the $C0$ configuration—is not changed by any statement at all, that is it is *invariant*.

We start with the fact that $\delta_{C0}(te_{\text{abs}}, pt_{\text{abs}}, c_{\text{abs}}) \neq None$. In this case we know that the $C0$ transition function is defined over the head of the program rest[2] $hd(s2l(c_{\text{abs}}))$. By a case distinction over this statement and expanding the next state function we can show that the global symbol table is not altered by any $C0$ statement, that is it stays the same for each $C0$ step:

$$\begin{aligned}
&\delta_{C0}(te_{\text{abs}}, pt_{\text{abs}}, c_{\text{abs}}) = \lfloor c'_{\text{abs}} \rfloor \wedge \implies \\
&gst(c_{\text{abs}}.m) = gst(c'_{\text{abs}}.m)
\end{aligned} \tag{8.17}$$

We do the same for the concrete kernel and obtain

$$\begin{aligned}
&\delta_{C0}(te_{\text{k}}, pt_{\text{k}}, c_{\text{k}}) = \lfloor c'_{\text{k}} \rfloor \wedge \implies \\
&\quad gst(c_k.m) = gst(c'_k.m)
\end{aligned} \tag{8.18}$$

Finally, we combine the above two results (8.17) and (8.18) to show:

$$\begin{aligned}
&gst(c_{\text{abs}}.m) = gst(c'_{\text{abs}}.m) \wedge gst(c_k.m) = gst(c'_k.m) \wedge \\
&konsis_{\text{gst}}(gst(c_{\text{abs}}.m), gst(c_{\text{k}}.m)) \implies \\
&\quad konsis_{\text{gst}}(gst(c'_{\text{abs}}.m), gst(c'_{\text{k}}.m))
\end{aligned}$$

q.e.d.

## 8.5 Dynamic Properties

Some of the kernel relations talk about parts of the kernel configurations that might change during execution. We call this share *dynamic properties*.

Most of the proofs in this section rely on the fact that expression evaluation in the abstract and concrete kernel returns results, which are equivalent under certain relations.

In particular we want to assure that a successful expression evaluation in the abstract kernel—a property the implementer of an abstract kernel would

[2] Actually, $\delta_{C0}$ is defined inductively over the statement tree. Yet, there exists a lemma that shows equivalence in execution for a flattened statement tree, where the head of the list is the next statement to be executed.

probably want to show—also means that the evaluation in the concrete kernel will not fail.

The following lemmas will define these expectations formally.

**Lemma 8.13 (Absence of Run-Time Errors)** *Let $c_{abs}$ and $c_k$ denote two (weak) valid configurations encoding the abstract and concrete kernel, the latter one being obtained by abstract linking.*

*Moreover, let the invariants regarding the symbol tables, the recursion depth, the type name environments, the program rest and the g-variables in both kernels hold.*

*Given a program rest, which does not start with an assembly statement, an external function call, or an XCall, and assuming that the abstract kernel does not produce a run-time error, that is there exists a successor configuration $c'_{abs}$.*

*Then, concrete kernel execution will not produce a run-time error neither:*

$$
\begin{aligned}
&\mathit{linkable}(\Pi_{abs}, \Pi_{CVM}) \wedge \Pi_k = \mathit{link}(\Pi_{abs}, \Pi_{CVM}) \wedge \\
&c_{abs} \in \mathit{conf}^{\mathit{weak}}_{\surd}(te_{abs}, pt_{abs}) \wedge c_k \in \mathit{conf}_{\surd}(te_k, pt_k) \wedge \\
&\delta_{C0}(te_{abs}, pt_{abs}, c_{abs}) = \lfloor c'_{abs} \rfloor \wedge \\
&\mathit{konsis}_{prog}(c_{abs}, c_k) \wedge \mathit{konsis}_{tenv}(te_{abs}, te_k) \wedge \\
&\mathit{konsis}_{rd}(c_{abs}.m, c_k.m) \wedge \mathit{konsis}_{lst}(c_{abs}.m, c_k.m) \wedge \\
&\mathit{konsis}_{gst}(c_{abs}.m, c_k.m) \wedge \mathit{konsis}_{gvars}(te_{abs}, c_{abs}.m, te_k, c_k.m, hp) \wedge \\
&\neg \mathit{is\_Asm}(\mathit{hd}(\mathit{s2l}(c_{abs}.\mathit{prog}))) \wedge \neg \mathit{is\_Skip}(c_{abs}.\mathit{prog}) \wedge \\
&\neg \mathit{is\_ESCall}(\mathit{hd}(\mathit{s2l}(c_{abs}.\mathit{prog}))) \wedge \neg \mathit{is\_XCall}(\mathit{hd}(\mathit{s2l}(c_{abs}.\mathit{prog}))) \\
&\Longrightarrow \\
&\quad \exists c'_k.\, \delta_{C0}(te_k, pt_k, c_k) = \lfloor c'_k \rfloor
\end{aligned}
$$

Proof We will prove this lemma by case distinction over the head of the program rest of the abstract kernel. Due to readability, we will omit the proof steps showing that the first statements in the program rests of both kernels are equivalent under $eq_{\text{stmt}}$. This fact can be simply deduced using the $\mathit{konsis}_{\text{prog}}$ invariant and Lemma 8.5.

For similar reasons, we do not explicitly describe the deduction of valid expressions required by Lemma 8.9. This follows directly from the validity of the configurations and hence the validity of the statement at the head of the program rest (cf. [Lei08]).

Most proof cases are similar, since the existence of a successor configuration merely relies on a successful memory update. Thus we will not work out the proof for all statements, but concentrate on representative ones.

**Case 1:** Let the program rest start with an assignment, that is

$$\mathit{hd}(\mathit{s2l}(c_{\text{abs}}.\mathit{prog})) = \mathit{Ass}(e_l, e_r).$$

Since there is a successor configuration in the abstract kernel, expression evaluation has not failed there, that is

$$\mathit{lval}(te_{\text{abs}}, c_{\text{abs}}.m, e_l) = \lfloor g \rfloor$$

and

$$\mathit{rval}(te_{\text{abs}}, c_{\text{abs}}.m, e_r) = \lfloor v \rfloor.$$

Furthermore, the memory update has been successful, too, i.e. $g$ is initialized or a root g-variable.

From Lemma 8.9 we obtain that the expression evaluation in the concrete kernel will be successful, too:

$$lval(te_k, c_k.m, e_l) = \lfloor kalloc(g, hp) \rfloor$$

and for the right expression

$$rval(te_k, c_k.m, e_r) = \lfloor v' \rfloor.$$

Also, if $g$ has been initialized in the abstract kernel, so is $kalloc(g, hp)$ initialized in the concrete kernel. In the case that $g$ is a root g-variable, then this is also true for $kalloc(g, hp)$ (using Lemma 8.3).

Hence the memory update will succeed and by definition of the $C0$ transition function, there is a successor configuration in the concrete kernel, too.

**Case 2:** Given a program rest that starts with a memory allocation, that is $hd(s2l(c_{\text{abs}}.prog)) = PAlloc(e_l, tn)$.

Left evaluation has succeeded in the abstract kernel and has yielded a g-variable $g$. With Lemma 8.9 we obtain the left value $kalloc(g, hp)$ in the concrete kernel. Due to type name consistency, $tn$ is also defined in the linked type name environment $te_k$.

Depending on sufficient memory, the memory update in the concrete kernel is done either with a null pointer or a pointer to a newly created heap variable $gvar_{hm}(|hst(c_k.m)|)$.

$g$ is reachable in the abstract kernel, so we obtain initialization equality for $g$ and $kalloc(g, hp)$, the g-variable to be updated in the concrete kernel, by merely expanding the definition of $konsis_{\text{gvars}}$. Since the memory update has been successful in the abstract kernel, $g$ has to be initialized and hence $kalloc(g, hp)$ is initialized, so the concrete kernel memory update will succeed, too.

**Case 3:** Let the program rest start with a function call, that is

$$hd(s2l(c_{\text{abs}}.prog)) = SCall(\sigma, fn, plist).$$

Let us consider the concrete kernel. The execution of a function call basically fails, if either the left evaluation of the return destination fails, that is

$$lval(te_k, c_k.m, Var(\sigma)) = None$$

or if right evaluation of one or more parameters fails:

$$\exists p \in \{plist\} : \ rval(te_k, c_k.m, p) = None.$$

Using Lemma 8.9 and the fact that expression evaluation has not failed in the abstract kernel, we deduce that this will not happen in the concrete kernel neither, so there exists a successor configuration. q.e.d.

**Lemma 8.14 (Preservation of Recursion Depth Consistency)** *Let $c_{abs}$ and $c_k$ denote two configurations of the abstract and the concrete kernel, which are in the set of valid configurations.*

*Furthermore let the recursion depth consistency $konsis_{rd}$ and the program rest consistency hold for both configurations.*

*In addition, we assume that the program rest of the abstract kernel does not start with an assembly statement, an external function call, an XCall statement, or that kernel run has already terminated.*

*Then the recursion depths of the successor configurations $c'_{abs}$ and $c'_k$ will also be consistent:*

$$
\begin{aligned}
&c_{abs} \in conf_{\surd}^{weak}(te_{abs}, pt_{abs}) \wedge c_k \in conf_{\surd}(te_k, pt_k) \wedge \\
&\delta_{C0}(te_{abs}, pt_{abs}, c_{abs}) = \lfloor c'_{abs} \rfloor \wedge \delta_{C0}(te_k, pt_k, c_k) = \lfloor c'_k \rfloor \wedge \\
&konsis_{prog}(c_{abs}, c_k) \wedge konsis_{tenv}(te_{abs}, te_k) \wedge \\
&konsis_{rd}(c_{abs}.m, c_k.m) \wedge konsis_{lst}(c_{abs}.m, c_k.m) \wedge \\
&konsis_{gst}(c_{abs}.m, c_k.m) \wedge konsis_{gvars}(te_{abs}, c_{abs}.m, te_k, c_k.m, hp) \wedge \\
&\neg is\_Asm(hd(s2l(c_{abs}.prog))) \wedge \neg is\_Skip(c_{abs}.prog) \wedge \\
&\neg is\_ESCall(hd(s2l(c_{abs}.prog))) \wedge \neg is\_XCall(hd(s2l(c_{abs}.prog))) \\
&\Longrightarrow \\
&konsis_{rd}(c'_{abs}.m, c'_k.m)
\end{aligned}
$$

Proof We prove the above lemma by case distinction over the current statement of the abstract kernel $hd(s2l(c_{\text{abs}}.prog))$.

For all cases but function calls and returns this is trivial, since none of the other statements adds or removes stack frames. Thus, we concentrate on these two cases.

We assumption that there is a successor configuration for the concrete kernel, can easily been shown using Lemma 8.13.

**Case 1:** We start with function calls, i.e.

$$hd(s2l(c_{\text{abs}}.prog)) = SCall(vn, fn, pl)).$$

From $konsis_{\text{prog}}(c_{\text{abs}}.prog, c_k.prog)$, Lemma 8.5, and Def. 7.8 we obtain that also the program rest of the concrete kernel starts with a function call statement:

$$hd(s2l(c_k.prog)) = SCall(vn, fn, pl).$$

From $\delta_{C0}(te_{\text{abs}}, pt_{\text{abs}}, c_{\text{abs}}) = \lfloor c'_{\text{abs}} \rfloor$ we know that the function call succeeds. This means by definition of the transition function that a new stack frame has been added to the local memory, so we obtain that the recursion depth has been increased by 1:

$$|c'_{\text{abs}}.m.lm| = |c_{\text{abs}}.m.lm| + 1 \tag{8.19}$$

and likewise for the concrete kernel using $\delta_{C0}(te_{\text{k}}, pt_{\text{k}}, c_{\text{k}}) = \lfloor c'_{\text{k}} \rfloor$

$$|c'_k.m.lm| = |c_k.m.lm| + 1 \tag{8.20}$$

Using the induction hypothesis $konsis_{\text{rd}}(c_{\text{abs}}.m, c_k.m)$ along with (8.19) and (8.20), we show

$$\begin{aligned} |c'_k.m.lm| &= |c_k.m.lm| + 1 \\ &= |c_{\text{abs}}.m.lm| + rd_{\text{off}} + 1 \\ &= |c'_{\text{abs}}.m.lm| + rd_{\text{off}}, \end{aligned}$$

which is exactly the definition of $konsis_{\text{rd}}$ (cf. Def. 7.7) :

$$konsis_{\text{rd}}(c'_{\text{abs}}.m, c'_k.m).$$

**Case 2:** Similarly, we proceed with a Return statement at the head of the program rest, that is

$$hd(s2l(c_{\text{abs}}.prog)) = Return(e).$$

Again, we deduce that the head of the concrete kernel's program rest also starts with a *Return* statement using Lemma 8.5:

$$hd(s2l(c_k.prog)) = Return(e).$$

Since execution of the *Return* statement succeeds, we know that the topmost stack frame has been removed from the local memory, i.e. the recursion depth of the new abstract kernel configuration has been decreased by 1:

$$|c'_{\text{abs}}.m.lm| = |c_{\text{abs}}.m.lm| - 1 \tag{8.21}$$

and for the concrete kernel's recursion depth

$$|c'_k.m.lm| = |c_k.m.lm| - 1, \tag{8.22}$$

respectively.

Using $konsis_{\text{rd}}(c_{\text{abs}}.m, c_k.m)$, (8.21), and (8.22), we finally show by expanding the definition of $konsis_{\text{rd}}$:

$$\begin{aligned} |c'_k.m.lm| &= |c_k.m.lm| - 1 \\ &= |c_{\text{abs}}.m.lm| + rd_{\text{off}} - 1 \\ &= |c'_{\text{abs}}.m.lm| + rd_{\text{off}}, \end{aligned}$$

which is again equivalent to $konsis_{\text{rd}}(c'_{\text{abs}}.m, c'_k.m)$. q.e.d.

### 8.5.1 Program Rest Consistency

Informally, program rest consistency guarantees that the execution of the concrete kernel follows the same path as abstract kernel execution.

For most statements we rely on certain invariants, like return destination and procedure table consistency, to prove that program rest consistency holds. For conditional statements, we have to make sure that expression evaluation in the abstract kernel corresponds to the one in the concrete kernel (see Lemma 8.9). For others—like assignments—program rest consistency is obvious.

**Lemma 8.15 (Preservation of Program Rest Consistency)** *Given $c_{abs}$ and $c_k$, two (weak) valid C0 configurations encoding the abstract and concrete kernel, respectively. Furthermore, we assume that the program rest does not start with an assembler statement, an external function call or an XCall, and that execution has not terminated. Moreover, let the invariants on symbol tables, recursion depth, type name environment, procedure tables, g-variables, and program rest hold.*

*If there is no run-time error in the abstract kernel, then program rest consistency will also hold in the successor configurations of both kernels, formally:*

$$
\begin{aligned}
&\mathit{linkable}(\Pi_{abs}, \Pi_{CVM}) \wedge \Pi_k = \mathit{link}(\Pi_{abs}, \Pi_{CVM}) \wedge \\
&c_{abs} \in \mathit{conf}_{\surd}^{weak}(te_{abs}, pt_{abs}) \wedge c_k \in \mathit{conf}_{\surd}(te_k, pt_k) \wedge \\
&\delta_{C0}(te_{abs}, pt_{abs}, c_{abs}) = \lfloor c'_{abs} \rfloor \wedge \delta_{C0}(te_k, pt_k, c_k) = \lfloor c'_k \rfloor \wedge \\
&\mathit{konsis}_{prog}(c_{abs}, c_k) \wedge \mathit{konsis}_{tenv}(te_{abs}, te_k) \wedge \\
&\mathit{konsis}_{rd}(c_{abs}.m, c_k.m) \wedge \mathit{konsis}_{lst}(c_{abs}.m, c_k.m) \wedge \\
&\mathit{konsis}_{gst}(c_{abs}.m, c_k.m) \wedge \mathit{konsis}_{gvars}(te_{abs}, c_{abs}.m, te_k, c_k.m, hp) \wedge \\
&\mathit{konsis}_{pt}(pt_{abs}, pt_k) \wedge \\
&\neg \mathit{is\_Asm}(\mathit{hd}(\mathit{s2l}(c_{abs}.\mathit{prog}))) \wedge \neg \mathit{is\_Skip}(c_{abs}.\mathit{prog}) \wedge \\
&\neg \mathit{is\_ESCall}(\mathit{hd}(\mathit{s2l}(c_{abs}.\mathit{prog}))) \wedge \neg \mathit{is\_XCall}(c_{abs}.\mathit{prog}) \\
&\quad \Longrightarrow \\
&\quad \mathit{konsis}_{prog}(c'_{abs}, c'_k)
\end{aligned}
$$

Proof We will present proofs for the preservation of program rest consistency during kernel execution. The proof basically consists of a case distinction over the head statement of the program rest. We will omit the trivial cases and concentrate on one representative for statements with similar proofs.

**Case 1:** The next state function $\delta_{C0}$ defines a case distinction for a program rest starting with *Skip*: either the whole program rest consists of single statement—the termination case— or other statements follow the *Skip*.

For the abstract kernel we know by assumption that we are not in the termination case: $\neg \mathit{is\_Skip}(c_{\mathrm{abs}}.\mathit{prog})$. Hence, we know that the length of the flattened abstract program rest is greater 1, since there has to exist a second statement:

$$|\mathit{s2l}(c_{\mathrm{abs}}.\mathit{prog})| > 1.$$

From $\mathit{konsis}_{\mathrm{prog}}(c_{\mathrm{abs}}.\mathit{prog}, c_k.\mathit{prog})$ we obtain that the flattened concrete kernel program rest is at least as long as the abstract one, hence we can conclude $|\mathit{s2l}(c_k.\mathit{prog})| > 1$, which then implies

$$\neg \mathit{is\_Skip}(\mathit{s2l}(c_k.\mathit{prog})).$$

For this case, the $C0$ transition function defines the tail of the old program rest as the new configuration's program rest, that is

$$\mathit{s2l}(c'_{\mathrm{abs}}.\mathit{prog}) = \mathit{tl}(\mathit{s2l}(c_{\mathrm{abs}}.\mathit{prog}))$$

and likewise for the concrete kernel

$$s2l(c'_k.prog) = tl(s2l(c_k.prog))$$

This means that the length of the new program rest has been decreased by 1 and all statements but the consumed *Skip* have been shifted one position to the left in the new program rest:

$$|s2l(c'_{\text{abs}}.prog)| = |s2l(c_{\text{abs}}.prog)| - 1$$
$$\forall i < |s2l(c'_{\text{abs}}.prog)| : s2l(c'_{\text{abs}}.prog)!i = s2l(c_{\text{abs}}.prog)!(i+1)$$

and for the concrete kernel

$$|s2l(c'_k.prog)| = |s2l(c_k.prog)| - 1$$
$$\forall i < |s2l(c'_k.prog)| : s2l(c'_k.prog)!i = s2l(c_k.prog)!(i+1).$$

Together with the assumption $konsis_{\text{prog}}(c_{\text{abs}}.prog, c_k.prog)$, we can finally show the thesis by expanding the definition of $konsis_{\text{prog}}$.

**Case 2:** The proof for the *If-Then-Else* case relies on the fact that the conditional expression is evaluated consistently in both kernels, so that the program rests are updated equivalently.

Since $c_{\text{abs}}$ is valid, the head of the program rest is also valid. In the $Ifte(e, s_1, s_2)$ case this means that $e$ is a Boolean expression:

$$type(te_{\text{abs}}, toplst(c_{\text{abs}}.m), gst(c_{\text{abs}}.m), e) = Bool_T.$$

Since there is a successor configuration for $c_{\text{abs}}$, right-hand expression evaluation cannot have failed, that is

$$rval(te_{\text{abs}}, c_{\text{abs}}.m, e) = \lfloor v \rfloor.$$

Using Lemma 8.9, we obtain that there is also a right-hand value of $e$ in the concrete kernel

$$rval(te_k, c_k.m, e) = \lfloor v' \rfloor$$

and

$$eq_{\text{cont}}(Bool_T, v, v')$$

which we simplify—using the definition of $eq_{\text{cont}}$—to

$$v = v'.$$

Furthermore, depending on $v$, the program rest of the abstract kernel starts either with $s_1$ or $s_2$:

$$s2l(c'_{\text{abs}}.prog) = \begin{cases} s_1 \# tl(s2l(c_{\text{abs}}.prog)) & \text{if } v = True \\ s_2 \# tl(s2l(c_{\text{abs}}.prog)) & \text{otherwise} \end{cases}$$

Due to $konsis_{\text{prog}}$, the head of the concrete program rest is also a conditional statement:

$$hd(s2l(c_k.prog)) = Ifte(e, s_1', s_2'),$$

where $s_1$ and $s_1'$, as well as $s_2$ and $s_2'$ are equivalent under $eq_{\text{stmt}}$.

Since expression evaluation has returned equal values in both kernel, program rest consistency is also preserved in both kernel successor configurations.

**Case 3:** Let the program rest start with a function call, that is

$$hd(s2l(c_{\text{abs}}.prog)) = SCall(\sigma, fn, plist).$$

We abbreviate the called function with $f = the(mapof(pt_{\text{abs}}, fn))$.

Since there is a successor configuration $c'_{\text{abs}}$ for the abstract kernel, the program rest has been updated by replacing the function call statement with the body of $fn$, that is

$$s2l(c'_{\text{abs}}.prog) = s2l(f.body)\#tl(s2l(c_{\text{abs}}.prog))$$

Let $c_k'$ denote the successor configuration of the concrete kernel, which exists following Lemma 8.13. This means that there is an entry in the concrete procedure table $pt_k$ with name $fn$ and a descriptor $f'$, such that

$$s2l(c_k'.prog) = s2l(f'.body)\#tl(s2l(c_k.body)).$$

Since $fn$ is a defined function in the abstract kernel—$(fn, f) \in pt_{\text{abs}}^{\text{def}}$—, we can expand the invariant on procedure tables $konsis_{\text{pt}}(pt_{\text{abs}}, pt_k)$ (cf. Def. 7.9). In particular, this means that

$$eq_{\text{stmt}}(f.body, f'.body).$$

This is all we need to prove the function call case. q.e.d.

### 8.5.2 Symbol Table Consistency

Symbol table consistency guarantees that corresponding variables in the abstract and concrete kernel are of the same type. In this section, we will show that the consistency relations are preserved during $C0$ kernel steps. In Sect. 8.4, Lemma 8.12, we have already proven that for global symbol tables.

**Lemma 8.16 (Preserving Consistency of $konsis_{\text{lst}}$)** *Let $c_{abs}$ be a weak valid abstract kernel configuration and $c_k$ a valid concrete kernel configuration. Furthermore, let the invariants on symbol tables, program rest, type name environments, procedure table, recursion depth, and g-variables hold. If the abstract program rest does not start with an external function call or an assembler statement and if the abstract kernel has not terminated and the $C0$ transition*

*function yields an abstract successor configuration, then the consistency between the local symbol tables will be preserved.*

$$
\begin{aligned}
&\mathit{linkable}(\Pi_{abs}, \Pi_{CVM}) \wedge \Pi_k = \mathit{link}(\Pi_{abs}, \Pi_{CVM}) \wedge \\
&c_{abs} \in \mathit{conf}_{\surd}^{\mathit{weak}}(te_{abs}, pt_{abs}) \wedge c_k \in \mathit{conf}_{\surd}(te_k, pt_k) \wedge \\
&\delta_{C0}(te_{abs}, pt_{abs}, c_{abs}) = \lfloor c'_{abs} \rfloor \wedge \delta_{C0}(te_k, pt_k, c_k) = \lfloor c'_k \rfloor \wedge \\
&\mathit{konsis}_{prog}(c_{abs}, c_k) \wedge \mathit{konsis}_{tenv}(te_{abs}, te_k) \wedge \\
&\mathit{konsis}_{rd}(c_{abs}.m, c_k.m) \wedge \mathit{konsis}_{lst}(c_{abs}.m, c_k.m) \wedge \\
&\mathit{konsis}_{gst}(c_{abs}.m, c_k.m) \wedge \mathit{konsis}_{gvars}(te_{abs}, c_{abs}.m, te_k, c_k.m, hp) \wedge \\
&\mathit{konsis}_{pt}(pt_{abs}, pt_k) \wedge \\
&\neg \mathit{is\_Asm}(\mathit{hd}(\mathit{s2l}(c_{abs}.\mathit{prog}))) \wedge \neg \mathit{is\_Skip}(c_{abs}.\mathit{prog}) \wedge \\
&\neg \mathit{is\_ESCall}(\mathit{hd}(\mathit{s2l}(c_{abs}.\mathit{prog}))) \wedge \neg \mathit{is\_XCall}(\mathit{hd}(\mathit{s2l}(c_{abs}.\mathit{prog}))) \\
&\Longrightarrow \\
&\quad \mathit{konsis}_{lst}(c'_{abs}.m, c'_k.m)
\end{aligned}
$$

Proof This proof is done by case distinction over the head of the abstract program rest.

We remember: $\mathit{konsis}_{\mathrm{lst}}(c_{\mathrm{abs}}.m, c_k.m)$ says that for each local symbol table in the abstract kernel, there exists a corresponding, equal symbol table in the concrete kernel (cf. Def. 7.17). Here, corresponding means that the abstract kernel index is shifted by the constant recursion depth offset:

$$
\begin{aligned}
&\mathit{konsis}_{\mathrm{lst}}(c_{\mathrm{abs}}.m, c_k.m) = \\
&\quad \forall i.\, i < |\mathit{sc}(c_{\mathrm{abs}}.m).\mathit{lst}| \implies \mathit{sc}(c_{\mathrm{abs}}.m).\mathit{lst}!i = \mathit{sc}(c_k.m).\mathit{lst}!(i + rd_{\mathrm{off}}).
\end{aligned}
$$

In most cases, the proof is very straightforward, since most statements do not change the local symbol tables. In fact, only function calls and returns are relevant. We will concentrate on these two cases.

**Case 1:** We start with function calls, that is

$$\mathit{hd}(\mathit{s2l}(c_{\mathrm{abs}}.\mathit{prog})) = \mathit{SCall}(\sigma, \mathit{fn}, \mathit{plist})).$$

From the fact that execution of the abstract kernel has not failed, we know that for the called function *fn*, there exists a descriptor $f$ such that $(\mathit{fn}, f) \in \{pt_{\mathrm{abs}}^{\mathrm{def}}\}$.

Furthermore, there has been added a new frame to the existing local memory stack, whose symbol table is made up by the symbol table of the called function:

$$
\begin{aligned}
\mathit{toplst}(c'_{\mathrm{abs}}.m) &= \mathit{st}_{\mathrm{func}}(f) \\
&= f.\mathit{params}@f.\mathit{lvars}.
\end{aligned}
$$

Using Lemma 8.13, which implies that concrete kernel execution does not fail neither, together with the invariant on procedure tables, we obtain a function descriptor $f'$ for *fn* in the concrete kernel for which holds:

$$f'.\mathit{params} = f.\mathit{params} \wedge f'.\mathit{lvars} = f.\mathit{lvars}.$$

This means that the symbol table of the topmost local stack frame of the concrete kernel—which has just been added—is equal to the function symbol table of $f'$, which again is equal to the function symbol table of $f$:

$$\begin{aligned} toplst(c'_k.m) &= st_{\text{func}}(f') \\ &= st_{\text{func}}(f) \\ &= f.params@f.lvars. \end{aligned}$$

We take this intermediate result and the preservation of recursion depth consistency as defined in Lemma 8.14 together with the fact that all other local symbol tables stay untouched. By expanding the definition of $konsis_{\text{lst}}$, we prove the function call case.

**Case 2:** Let the abstract program rest start with a return statement, i.e.

$$hd(s2l(c_{\text{abs}}.prog)) = Return(e).$$

The successful execution of this statement in the abstract kernel has removed the topmost frame from the local memory stack. The same happens in the concrete kernel.

This means that in both successor configurations, the recursion depth has been decreased by 1:

$$\begin{aligned} |sc(c'_{\text{abs}}.m).lst| &= |sc(c_{\text{abs}}.m).lst| - 1 \\ |sc(c'_k.m).lst| &= |sc(c_k.m).lst| - 1 \end{aligned}$$

Together with the fact that all other frames stay unchanged and that the invariant on local symbol tables holds for them by assumption, we prove this case. q.e.d.

**Lemma 8.17 (Preserving Consistency of $konsis_{\text{hst}}$)** *Let $c_{abs}$ be a weak valid abstract kernel configuration and $c_k$ be a valid concrete kernel configuration. Furthermore, let the invariants on symbol tables, program rest, type name environments, procedure table, recursion depth, and g-variables hold, and in particular the invariants on the heap. If the abstract program rest does not start with an external function call or an assembler statement and if the abstract kernel has not terminated and the C0 transition function yields an abstract successor configuration, then the consistency between the heap symbol tables will*

*be preserved.*

$$
\begin{aligned}
&\mathit{linkable}(\Pi_{abs}, \Pi_{CVM}) \wedge \Pi_k = \mathit{link}(\Pi_{abs}, \Pi_{CVM}) \wedge \\
&c_{abs} \in \mathit{conf}^{\mathit{weak}}_{\surd}(te_{abs}, pt_{abs}) \wedge c_k \in \mathit{conf}_{\surd}(te_k, pt_k) \wedge \\
&\delta_{C0}(te_{abs}, pt_{abs}, c_{abs}) = \lfloor c'_{abs} \rfloor \wedge \delta_{C0}(te_k, pt_k, c_k) = \lfloor c'_k \rfloor \wedge \\
&\mathit{konsis}_{prog}(c_{abs}, c_k) \wedge \mathit{konsis}_{tenv}(te_{abs}, te_k) \wedge \\
&\mathit{konsis}_{rd}(c_{abs}.m, c_k.m) \wedge \mathit{konsis}_{lst}(c_{abs}.m, c_k.m) \wedge \\
&\mathit{konsis}_{gst}(c_{abs}.m, c_k.m) \wedge \mathit{konsis}_{gvars}(te_{abs}, c_{abs}.m, te_k, c_k.m, hp) \wedge \\
&\mathit{konsis}_{hst}(c_{abs}.m, c_k.m, hp) \mathit{konsis}_{pt}(pt_{abs}, pt_k) \wedge \\
&\mathit{heap}_{inv}(c_{abs}.m, c_k.m) \wedge \\
&\neg \mathit{is\_Asm}(\mathit{hd}(\mathit{s2l}(c_{abs}.\mathit{prog}))) \wedge \neg \mathit{is\_Skip}(c_{abs}.\mathit{prog}) \wedge \\
&\neg \mathit{is\_ESCall}(\mathit{hd}(\mathit{s2l}(c_{abs}.\mathit{prog}))) \wedge \neg \mathit{is\_XCall}(\mathit{hd}(\mathit{s2l}(c_{abs}.\mathit{prog}))) \\
&\quad \Longrightarrow \\
&\quad \exists hp'.\, \mathit{konsis}_{hst}(c'_{abs}, c'_k, hp')
\end{aligned}
$$

Proof We prove this lemma by case distinction over the head of the abstract program rest. All cases but memory allocation are trivial, since the other statements do not change the heap symbol table.

Given a program rest head $\mathit{hd}(\mathit{s2l}(c_{\mathrm{abs}}.\mathit{prog})) = \mathit{PAlloc}(e, tn)$. This means there is a type $t$ with $\mathit{mapof}(te_{,\mathrm{abs}}, tn) = \lfloor t \rfloor$.

There are two possible successful execution paths for memory allocation:

- either there is enough heap memory available to allocate a new object of type $t$—$?\mathit{heap}(c_{\mathrm{abs}}.m, t) = \mathit{True}$,
- or heap memory is used up, i.e. $?\mathit{heap}(c_{\mathrm{abs}}.m, t) = \mathit{False}$.

In the latter case, the heap memory stays unchanged, that is $c'_{\mathrm{abs}}.m.hm = c_{\mathrm{abs}}.m.hm$. Since we assume that the invariant on the heap symbol table holds for $c_{\mathrm{abs}}$ and $c_k$, we are done setting $hp' = hp$.

Let us assume that there is enough heap memory available in the abstract kernel. In this case, a new element has been appended to the old heap symbol table:

$$\mathit{hst}(c'_{\mathrm{abs}}.m) = \mathit{hst}(c_{\mathrm{abs}}.m) @ [(\mathit{undef}, t)]$$

Due to the invariant on heap memory, $\mathit{heap}_{\mathrm{inv}}(c_{\mathrm{abs}}.m, c_k.m)$, there is also enough memory available in the concrete kernel[3]. Furthermore, the type name $tn$ is associated with the same type $t$ in the concrete kernel, due to type name consistency. Hence we add the same element to the concrete heap symbol table as in the abstract kernel, thus

$$\mathit{snd}(\mathit{hst}(c'_{\mathrm{abs}}.m)!(|\mathit{hst}(c_{\mathrm{abs}}.m)|)) = \mathit{snd}(\mathit{hst}(c'_k.m)!(|\mathit{hst}(c_k.m)|)) \tag{8.23}$$

We update the heap map function as follows:

$$hp'(i) = \begin{cases} |\mathit{hst}(c_k.m)| & \text{if } i = |\mathit{hst}(c_{\mathrm{abs}}.m)| \\ \mathit{hp}(i) & \text{otherwise} \end{cases}$$

[3]The concrete kernel can have only more allocated memory, but never less than the abstract kernel

Using the assumption that the invariant on the heap symbol table has held before together with (8.23), we finally obtain $konsis_{\mathrm{hst}}(c'_{\mathrm{abs}}, c'_k, hp')$. q.e.d.

### 8.5.3 Return Destination Consistency

Return destination consistency states that the return destinations of corresponding stack frames are equivalent under *kalloc*.

**Lemma 8.18 (Preservation of $konsis_{\mathbf{return}}$)** *Let $c_{abs}$ and $c_k$ be two (weak) valid configurations encoding both kernels. Furthermore, let the invariants on symbol tables, program rest, type name environments, procedure table, recursion depth, and g-variables hold. If the abstract program rest does not start with an external function call or an assembler statement and if the abstract kernel has not terminated and the C0 transition function yields an abstract successor configuration, then the return destination consistency will be preserved.*

$$
\begin{aligned}
&linkable(\Pi_{abs}, \Pi_{CVM}) \wedge \Pi_k = link(\Pi_{abs}, \Pi_{CVM}) \wedge \\
&c_{abs} \in conf^{weak}_{\surd}(te_{abs}, pt_{abs}) \wedge c_k \in conf_{\surd}(te_k, pt_k) \wedge \\
&\delta_{C0}(te_{abs}, pt_{abs}, c_{abs}) = \lfloor c'_{abs} \rfloor \wedge \delta_{C0}(te_k, pt_k, c_k) = \lfloor c'_k \rfloor \wedge \\
&konsis_{prog}(c_{abs}, c_k) \wedge konsis_{tenv}(te_{abs}, te_k) \wedge \\
&konsis_{rd}(c_{abs}.m, c_k.m) \wedge konsis_{lst}(c_{abs}.m, c_k.m) \wedge \\
&konsis_{gst}(c_{abs}.m, c_k.m) \wedge konsis_{gvars}(te_{abs}, c_{abs}.m, te_k, c_k.m, hp) \wedge \\
&konsis_{pt}(pt_{abs}, pt_k) \wedge konsis_{hst}(c_{abs}.m, c_k.m) \\
&konsis_{return}(c_{abs}, c_k, hp) \wedge \\
&\neg is\_Asm(hd(s2l(c_{abs}.prog))) \wedge \neg is\_Skip(c_{abs}.prog) \wedge \\
&\neg is\_ESCall(hd(s2l(c_{abs}.prog))) \wedge \neg is\_XCall(hd(s2l(c_{abs}.prog))) \\
&\Longrightarrow \\
&konsis_{return}(c'_{abs}, c'_k, hp)
\end{aligned}
$$

Proof We prove this lemma by case distinction over the head of the abstract program rest. All cases but return and function call are trivial, hence we omit the former ones.

**Case 1:** We start with a function call, that is

$$hd(s2l(c_{\mathrm{abs}}.prog)) = SCall(\sigma, fn, plist).$$

The return destination of this function call is a g-variable which is given through the left-hand evaluation of the expression $Var(\sigma)$. We know that this left-hand value exists, since abstract kernel execution has yielded a successor configuration:

$$lval(te_{\mathrm{abs}}, c_{\mathrm{abs}}.m,\ Var(\sigma)) = \lfloor g \rfloor.$$

Corresponding to the definition of the $C0$ transition function, this g-variable is then used as the second component of the newly added frame in the local memory stack, that is:

$$snd(toplm(c'_{\mathrm{abs}}.m)) = g.$$

From Lemma 8.9 we obtain that the left-hand evaluation of the variable access in the concrete kernel is successful, too:

$$lval(te_k, c_k.m, Var(\sigma)) = \lfloor kalloc(g, hp) \rfloor.$$

Hence, the second component of the new topmost stack frame in the successor configuration of the concrete kernel is

$$snd(toplm(c'_{\text{abs}}.m)) = kalloc(g, hp).$$

Using the recursion depth invariant and Lemma 8.14, we know that the recursion depth—and thus the topmost stack frame— is shifted by a constant $rd_{\text{off}}$ to the recursion depth of the abstract kernel.

Hence, we obtain:

$$\begin{aligned} snd(toplm(c'_k.m)) &= snd(lst(c'_k.m)!(|lst(c'_{\text{abs}}.m)| + rd_{\text{off}})) \\ &= kalloc(g, hp) \\ &= kalloc(snd(toplm(c'_{\text{abs}}.m)), hp) \\ &= kalloc(snd(c'_{\text{abs}}.m.lm!(|lst(c'_{\text{abs}}.m)|)), hp)) \end{aligned}$$

Since all other stack frames of the successor configurations stay unchanged, we can exploit the assumption of $konsis_{\text{return}}(c_{\text{abs}}.m, c_k.m, hp)$ together with the equation above to prove this case.

**Case 2:** The return case $hd(s2l(c_{\text{abs}}.prog)) = Return(e)$ is much simpler.

Successful execution in the abstract kernel leads to a new local memory, which is equal to the old one without the topmost stack frame.

The same happens in the concrete kernel. This means that recursion depth has been decreased by one in both successor configurations. All other stack frames stay unchanged.

By simply using the assumption $konsis_{\text{return}}(c_{\text{abs}}.m, c_k.m, hp)$, we have completed the proof for this case. q.e.d.

### 8.5.4 Heap Invariants

The heap invariants capture two properties of the heap: the first is *injectivity* of the heap map, that is different arguments to this function deliver different results, the second one is boundedness, i.e. the values of the heap map—heap indices—stay within the boundaries of the concrete heap size.

**Lemma 8.19 (Preservation of Heap Invariants)** *Let $c_{abs}$ and $c_k$ be two (weak) valid configurations encoding both kernels. Furthermore, let the invariants on symbol tables, program rest, type name environments, procedure table, recursion depth and g-variables hold. If the abstract program rest does not start with an external function call or an assembler statement and if the abstract kernel has not terminated and the C0 transition function yields an abstract*

*successor configuration, then there exists an (updated) heap map such that the heap invariants hold in the successor configurations, too.*

$$
\begin{aligned}
&\mathit{linkable}(\Pi_{abs}, \Pi_{CVM}) \wedge \Pi_k = \mathit{link}(\Pi_{abs}, \Pi_{CVM}) \wedge \\
&c_{abs} \in \mathit{conf}_{\surd}^{\mathit{weak}}(te_{abs}, pt_{abs}) \wedge c_k \in \mathit{conf}_{\surd}(te_k, pt_k) \wedge \\
&\delta_{C0}(te_{abs}, pt_{abs}, c_{abs}) = \lfloor c'_{abs} \rfloor \wedge \delta_{C0}(te_k, pt_k, c_k) = \lfloor c'_k \rfloor \wedge \\
&\mathit{konsis}_{prog}(c_{abs}, c_k) \wedge \mathit{konsis}_{tenv}(te_{abs}, te_k) \wedge \\
&\mathit{konsis}_{rd}(c_{abs}.m, c_k.m) \wedge \mathit{konsis}_{lst}(c_{abs}.m, c_k.m) \wedge \\
&\mathit{konsis}_{gst}(c_{abs}.m, c_k.m) \wedge \mathit{konsis}_{gvars}(te_{abs}, c_{abs}.m, te_k, c_k.m, hp) \wedge \\
&\mathit{hmap}_{inj}(c_{abs}.m, hp) \wedge \mathit{hmap}_{bound}(c_{abs}.m, c_k.m, hp) \\
&\mathit{heap}_{inv}(c_{abs}.m, c_k.m) \wedge \\
&\neg \mathit{is\_Asm}(\mathit{hd}(\mathit{s2l}(c_{abs}.\mathit{prog}))) \wedge \neg \mathit{is\_Skip}(c_{abs}.\mathit{prog}) \wedge \\
&\neg \mathit{is\_ESCall}(\mathit{hd}(\mathit{s2l}(c_{abs}.\mathit{prog}))) \wedge \neg \mathit{is\_XCall}(\mathit{hd}(\mathit{s2l}(c_{abs}.\mathit{prog}))) \\
&\Longrightarrow \\
&\quad \exists hp'.\, \mathit{hmap}_{inj}(c'_{abs}.m, hp') \wedge \mathit{hmap}_{bound}(c'_{abs}.m, c'_k.m, hp')
\end{aligned}
$$

Proof We conduct this proof by case distinction over the head of the abstract program rest. All cases but memory allocation, i.e. $\mathit{hd}(\mathit{s2l}(c_{\mathrm{abs}}.\mathit{prog})) = \mathit{PAlloc}(e)$, are trivial and we omit them.

In $C0$, each newly allocated heap object gets an index larger than all other objects allocated before. This is due to the fact that there is no explicit deallocation of memory in $C0$.

We consider again the two possible execution paths depending on heap memory availability. The case when no memory is available is easy: since the new heap memory is equal to the old one, we leave the heap map as it is:

$$hp' = hp.$$

Since we have assumed the validity of both invariants, $\mathit{hmap}_{\mathrm{inj}}(c_{\mathrm{abs}}.m, hp)$ and $\mathit{hmap}_{\mathrm{bound}}(c_{\mathrm{abs}}.m, c_k.m, hp)$, the proof is done.

Let us consider the positive case, that is $?\mathit{heap}(c_{\mathrm{abs}}.m, t) = \mathit{True}$, with $\mathit{mapof}(te_{\mathrm{abs}}, tn) = \lfloor t \rfloor$. Then, the new heap has been extended by an element, both in the abstract and in the concrete kernel due to $\mathit{heap}_{\mathrm{inv}}(c_{\mathrm{abs}}.m, c_k.m)$, type name consistency, and absence of run-time errors (cf. Lemma 8.13).

In this case, we update the heap map function in the following way:

$$hp'(i) = \begin{cases} |c_k.m.hm| & \text{if } i = |c_{\mathrm{abs}}.m.hm| \\ hp(i) & \text{otherwise} \end{cases}$$

From $\mathit{hmap}_{\mathrm{bound}}(c_{\mathrm{abs}}.m, c_k.m, hp)$ we know that the value $|c_k.m.hm|$ has not been assigned to any abstract heap index before:

$$
\begin{aligned}
&\mathit{hmap}_{\mathrm{bound}}(c_{\mathrm{abs}}.m, c_k.m, hp) \equiv \\
&\quad \forall i.\, i < |c_{\mathrm{abs}}.m.hm| \implies hp(i) < |c_k.m.hm|
\end{aligned}
$$

Together with the assumption that the heap map has been injective before, we obtain the injectivity for $hp'$ in the successor configuration:

$$\mathit{hmap}_{\mathrm{inj}}(c'_{\mathrm{abs}}.m, hp').$$

This is also true for the boundedness of the heap map. Since it has been bounded before by assumption, we only have to consider the function value for the new abstract heap index $i = |c_{\text{abs}}.m.hm|$, which is:

$$\begin{aligned} hp'(i) &= |c_k.m.hm| \\ &= |c'_k.m.hm| - 1. \end{aligned}$$

Hence we obtain:

$$\begin{aligned} \forall i.i < |c'_{\text{abs}}.m.hm| &\implies hp'(i) < |c'_k.m.hm| \\ &\equiv hmap_{\text{bound}}(c'_{\text{abs}}.m, c'_k.m, hp'). \end{aligned}$$

q.e.d.

### 8.5.5 g-Variable Consistency

The consistency relation between g-variables of the abstract and concrete kernel is the most crucial of all kernel relations. Informally, it says: when a g-variable is reachable in the abstract kernel, so is its counterpart in the concrete kernel; plus, their initialization status is the same and their types are equal in the case that they are elementary. Finally, if the abstract kernel variable is elementary and initialized, then its value and the value of its concrete counterpart are equal under $eq_{\text{val}}$.

**Lemma 8.20 (Equal Content after Memory Update)** *The value of several g-variables might be changed by a memory update, so we have to make sure that the updates in the abstract and concrete kernel 'match' somehow.*

*Given two (weak) valid configurations for the two kernels and two values $v$ and $v'$ that are used in the successful memory update of the valid g-variables $g$ and $kalloc(g, hp)$. Moreover, let $v$ and $v'$ be equal under the $eq_{cont}$ function. If $x$ is valid, initialized, of elementary type, and value equal with its concrete counterpart, then this is also true after the memory update.*

$$\begin{aligned} &linkable(\Pi_{abs}, \Pi_{CVM}) \wedge \Pi_k = link(\Pi_{abs}, \Pi_{CVM}) \wedge \\ &c_{abs} \in conf_{\surd}^{weak}(te_{abs}, pt_{abs}) \wedge c_k \in conf_{\surd}(te_k, pt_k) \wedge \\ &upd_{mm}(c_{abs}.m, g, v) = \lfloor m'_{abs} \rfloor \wedge upd_{mm}(c_k.m, kalloc(g, hp), v') = \lfloor m'_k \rfloor \\ &konsis_{rd}(c_{abs}.m, c_k.m) \wedge konsis_{lst}(c_{abs}.m, c_k.m) \wedge \\ &konsis_{gst}(c_{abs}.m, c_k.m) \wedge konsis_{hst}(c_{abs}.m, c_k.m) \wedge \\ &hmap_{inj}(c_{abs}.m, hp) \wedge hmap_{bound}(c_{abs}.m, c_k.m, hp) \wedge \\ &x \in gvar_{\surd}(sc(c_{abs}.m) \wedge g \in gvar_{\surd}(sc(c_{abs}.m) \wedge \\ &(?init_g(c_{abs}.m, x) \longrightarrow eq_{val}(x, c_{abs}.m, kalloc(x, hp), c_k.m)) \wedge \\ &?elem_T(type_g(sc(c_{abs}.m), x)) \wedge eq_{cont}(type_g(sc(c_{abs}.m), g), v, v') \\ &\implies \\ &\quad ?init_g(m'_{abs}, x) \longrightarrow eq_{val}(m'_{abs}, x, m'_k, kalloc(x, hp)) \end{aligned}$$

Proof We consider an arbitrary but fixed g-variable $x$. We distinguish two cases: either $x$ is changed by the memory update or not. If $x$ is a sub g-variable of $g$, then its value will be updated. Otherwise, it stays unchanged.

**Case 1:** We start with the easier case, that is $x \notin sub_g(g)$. Using this along with the fact that $x$ is elementary yields that there is no overlap between the memory regions of $x$ and $g$:

$$\begin{aligned}&ba_g(sc(c_{\text{abs}}.m), x) \notin \\ &\{ba_g(sc(c_{\text{abs}}.m), g), \ldots, \\ &\quad ba_g(sc(c_{\text{abs}}.m), x) + (size_T(type_g(sc(c_{\text{abs}}.m), g)) - 1)\}.\end{aligned}$$

Due to the transitivity of sub g-variables (cf. Lemma 8.4) and type equality under *kalloc* (cf. Lemma 8.1), we deduce the corresponding property for $kalloc(x, hp)$ and $kalloc(g, hp)$ in the concrete kernel.

This means that the values of both $x$ and $kalloc(x, hp)$ are not changed by the memory update, hence

$$\begin{aligned}value_g(m'_{\text{abs}}, x) &= value_g(c_{\text{abs}}.m, x) \\ value_g(m'_k, kalloc(x, hp)) &= value_g(c_k.m, kalloc(x, hp)).\end{aligned}$$

Using the assumption that value equality for $x$ and $kalloc(x, hp)$ has been established before the memory update, we simply expand the definition of $eq_{\text{val}}$ to complete the proof for this case.

**Case 2:** Let $x$ be a sub g-variable of $g$, i.e. $x \in sub_g(g)$. So, $kalloc(x, hp)$ is in the set of sub g-variables of $kalloc(g, hp)$.

By definition, $g$ is definitely initialized after the memory update. This is also true for all its sub g-variables, hence

$$?init_g(m'_{\text{abs}}, x) = True. \tag{8.24}$$

As we have seen in Sect. 3.5.2, the value of $g$ after the memory update (see Def. 3.10 and Def. 3.14) is equal to $v$, that is

$$\begin{aligned}v &= value_g(m'_{\text{abs}}, g) \\ &= m'_{\text{abs}}.mem_g(g).ct[ba_g(sc(m'_{\text{abs}}), g), size_T(type_g(sc(m'_{\text{abs}}), g))]\end{aligned} \tag{8.25}$$

Similarly, we obtain for $v'$ and $kalloc(g, hp)$ in the concrete kernel:

$$\begin{aligned}v' &= value_g(m'_k, kalloc(g, hp)) \\ &= m'_k.mem_g(kalloc(g, hp)).ct[ba_g(sc(m'_k), kalloc(g, hp)), \\ &\quad size_T(type_g(sc(m'_k), kalloc(g, hp)))]\end{aligned} \tag{8.26}$$

In the case that $g$ is of an elementary type, its size is equal to 1 and $x$ and $g$ are trivially identical. Similarly, this is also true for $kalloc(g, hp)$ and $kalloc(x, hp)$, using the corresponding lemmas 8.1 and 8.4.

Formally we obtain:

$$\begin{aligned}v &= value_g(m'_{\text{abs}}, g) \\ &= m_{mem_g(g)}[ba_g(sc(m'_{\text{abs}}), g), 1)] \\ &= m_{mem_g(x)}[ba_g(sc(m'_{\text{abs}}), x), 1)] \\ &= value_g(m'_{\text{abs}}, x)\end{aligned}$$

and for the concrete kernel $v' = value_g(m'_k, kalloc(x, hp))$. Using the assumption $eq_{\text{cont}}(type_g(sc(c_{\text{abs}}.m), g), v, v')$ and the definition of $eq_{\text{val}}$, we finally obtain

$$eq_{\text{val}}(x, m'_{\text{abs}}, kalloc(x, hp), m'_k),$$

which together with equation (8.24) completes the proof for the case that $x = g$.

Let us consider the non-trivial case, i.e. $x \in sub_g(g) \wedge x \neq g$. We remember that the value of $g$ is equal to $v$ after the memory update (see eq. (8.25)).

We recall the structure of g-variables as presented in Sect. 3.3.1. Their tree-like structure can finally be broken down into a set of elementary sub g-variables (see Fig. 3.3). As we have seen in Sect. 3.3.1, our memory model is flat in the sense that we merely string together g-variables (see also Def. 3.12).

This results in a *compact* memory, which is free of holes, or in other words: for each memory cell within the memory range of a non-elementary g-variable, there exists a elementary g-variable, whose value is stored in this memory cell.

Hence, there exists an index $i$, such that

$$ba_g(sc(m'_{\text{abs}}), x) = ba_g(sc(m'_{\text{abs}}), g) + i$$

with $0 \leq i < size_T(type_g(sc(c_{\text{abs}}.m), g))$. Due to type equality, this index $i$ also exists for $kalloc(x, hp)$ in the concrete kernel with respect to $konsis(g, hp)$.

Using the fact that $x$ is elementary and thus has a type size of 1, we obtain for the value of $x$:

$$\begin{aligned} value_g(m'_{\text{abs}}, x) &= m'_{\text{abs}}.mem_g(x).ct[ba_g(sc(m'_{\text{abs}}), x), 1)] \\ &= m'_{\text{abs}}.mem_g(g).ct[ba_g(sc(m'_{\text{abs}}), g), \\ &\quad size_T(type_g(sc(m'_{\text{abs}}), g))](i) \\ &= v(i) \end{aligned}$$

Similarly, we proceed in the concrete kernel to obtain

$$value_g(m'_k, kalloc(x, hp)) = v'(i)$$

The definition of $eq_{\text{cont}}$ (cf. Def. 7.14) for the aggregate type case guarantees that the equality relation holds for all elements of that type. Using the assumption $eq_{\text{cont}}(type_g(sc(c_{\text{abs}}.m), g), v, v')$, this implies

$$eq_{\text{cont}}(type_g(sc(m'_{\text{abs}}), x), v(i), v'(i)),$$

which is by definition and using the fact that the abstract and concrete memory have been updated appropriately

$$eq_{\text{val}}(x, m'_{\text{abs}}, kalloc(x, hp), m'_k). \qquad \text{q.e.d.}$$

**Lemma 8.21 ($konsis_{\text{named}}$ on Memory Updates)** *We will now use the recent lemmas to prove that the g-variable consistency for named g-variables is preserved by a memory update.*

*Given two kernel configurations $c_{abs}$ and $c_k$. We assume that with two equal values, $v$ and $v'$, the memory update of the g-variables $g$ and $kalloc(g, hp)$ is successful. If the $konsis_{named}$ relation has held before, then it will be preserved by the memory updates.*

$$
\begin{aligned}
&linkable(\Pi_{abs}, \Pi_{CVM}) \wedge \Pi_k = link(\Pi_{abs}, \Pi_{CVM}) \wedge \\
&c_{abs} \in conf_{\surd}^{weak}(te_{abs}, pt_{abs}) \wedge c_k \in conf_{\surd}(te_k, pt_k) \wedge \\
&upd_{mm}(c_{abs}.m, g, v) = \lfloor m'_{abs} \rfloor \wedge upd_{mm}(c_k.m, kalloc(g, hp), v') = \lfloor m'_k \rfloor \\
&\wedge konsis_{pt}(pt_{abs}, pt_k) \wedge \\
&konsis_{prog}(c_{abs}, c_k) \wedge konsis_{tenv}(te_{abs}, te_k) \wedge \\
&konsis_{rd}(c_{abs}.m, c_k.m) \wedge konsis_{lst}(c_{abs}.m, c_k.m) \wedge \\
&konsis_{gst}(c_{abs}.m, c_k.m) \wedge konsis_{gvars}(c_{abs}.m, c_k.m, hp) \wedge \\
&konsis_{pt}(pt_{abs}, pt_k) \wedge konsis_{hst}(c_{abs}.m, c_k.m) \wedge \\
&konsis_{return}(c_{abs}, c_k, hp) \wedge eq_{cont}(type_g(sc(c_{abs}.m), g), v, v') \\
&\Longrightarrow \\
&\quad konsis_{named}(m'_{abs}, m_k, hp)
\end{aligned}
$$

Proof We remember: $konsis_{\text{named}}$ is an invariant over all named—hence reachable—g-variables in the abstract kernel. Without limitation of generality, let $x$ denote such a g-variable.

We distinguish two proof cases: either $x$ is a sub g-variable of the variable $g$ used in the memory update or it is not.

**Case 1:** We start with the sub g-variable case, that is $x \in sub_g(g)$. We use Lemma 8.4 to obtain $kalloc(x, hp) \in sub_g(kalloc(g, hp))$.

We assume that $x$ is a named g-variable in the new abstract memory configuration:

$$x \in reachable_{\text{named}}(m'_{\text{abs}}) \equiv x \in gvar_{\surd} \wedge ?named(x). \tag{8.27}$$

Using Lemma 8.2 on transitivity of g-variable validity and the trivial fact, that $kalloc(x, hp)$ is named, too, we derive

$$kalloc(x, hp) \in reachable_{\text{named}}(m'_k). \tag{8.28}$$

The variable used in a memory update is always initialized after the update has happened. So, all sub g-variables of $g$ are initialized, too (cf. Def. 3.11).

This means that

$$
\begin{aligned}
&?init_g(m'_{\text{abs}}, x) = True = ?init_g(m'_k, kalloc(x, hp)) \\
&\Longrightarrow eq_{\text{init}}(m'_{\text{abs}}, x, m'_k, kalloc(x, hp))
\end{aligned} \tag{8.29}
$$

Let us assume that $x$ is of an elementary type, i.e.

$$?elem(type_g(sc(m'_{\text{abs}}), x)). \tag{8.30}$$

Using Lemma 8.1 on type equality under *kalloc*, we deduce

$$eq_{\text{type}}(m'_{\text{abs}}, x, m'_k, kalloc(x, hp)). \tag{8.31}$$

Furthermore, from equation (8.29) we know that $x$ is initialized. So Let us consider the value of $x$ after the memory update. Using the auxiliary Lemma 8.20 for value equality after memory updates and the fact that symbol configurations are not altered by a memory update, we obtain

$$eq_{\text{val}}(x, m'_{\text{abs}}, kalloc(x, hp), m'_k). \tag{8.32}$$

Ultimately, we combine the equations (8.27), (8.28), (8.29), (8.30), (8.31), and (8.32) to complete the proof for this case.

**Case 2:** Now we will consider the case that $x$ is not a sub g-variable of the updated $g$: $x \notin sub_g(g)$. Using the transitivity of sub g-variables as specified in Lemma 8.4, we obtain $kalloc(x, hp) \notin sub_g(kalloc(g, hp))$.

The first steps of this case are identical to those in the sub g-variable case before: $x$ and $kalloc(x, hp)$ are also reachable in the succeeding memory configurations $m'_{\text{abs}}$ and $m'_k$ (see equations (8.27) and (8.28)).

Since the invariant was valid in the configurations $c_{\text{abs}}$ and $c_k$, the initialization of both $x$ and $kalloc(x, hp)$ was equal at this point in time. [Lei08] has shown that the initialization of variables not afflicted by a memory update (i.e. that are not sub g-variables of the updated variable) stays untouched. This gives us

$$eq_{\text{init}}(m'_{\text{abs}}, x, m'_k, kalloc(x, hp)) \tag{8.33}$$

Let us assume again that $x$ is elementary:

$$?elem(type_g(sc(m'_{\text{abs}}), x)). \tag{8.34}$$

The rest of this case's proof is equivalent to the case before. We use again Lemma 8.1 to establish type equality and—in case that $x$ is initialized—Lemma 8.20 to obtain value equality after the memory update. q.e.d.

### Reachability of Nameless g-Variables

Things are slightly different and more complicated when dealing with nameless variables. We have to strengthen our arguments slightly, in particular regarding the values used in the memory update. Thus, we will introduce the notion of *reachable* pointers first.

**Definition 8.3 (Memcells Containing Pointers to Reachable Objects)** We will define a predicate $mcell_{\text{reachable}} : memconfT \times mcellT \to \mathbb{B}$ for memory cells containing pointers to reachable objects. The predicate evaluates to *True*, if a pointer memory cell contains a pointer to a reachable memory object regarding a given memory configuration $m$ or if it is of non-pointer type, formally:

$$\frac{NullPointer}{mcell_{\text{reachable}}(m, Ptr(NullPointer))}$$

$$\frac{g \in reachable_{\text{named}}(m)}{mcell_{\text{reachable}}(m, Ptr(g))}$$

$$\frac{reachable_{\text{nameless}}(m, g, i)}{mcell_{\text{reachable}}(m, Ptr(g))}$$

$$\frac{c \in \{Int(i), Unsgnd(u), Bool(b), Char(z)\}}{mcell_{\text{reachable}}(m, c)}$$

We lift this definition to the level of *memory content* by defining a predicate $cont_{\text{reachable}} : (\mathbb{N} \to mcellT) \times \mathbb{N} \to \mathbb{B}$. Given a piece of memory content $ct$ and a bound $n \in \mathbb{N}$, the predicate evaluates to *True* if all memory cells of this content up to $n-1$ contain reachable pointers, formally:

$$cont_{\text{reachable}}(m, ct, n) = \forall i < n.\, mcell_{\text{reachable}}(m, ct(i))$$

**Lemma 8.22 ($konsis_{\text{nameless}}$ on Memory Update)** *For kernel configurations $c_{abs}$ and $c_k$, we assume that with two equal values, $v$ and $v'$, the memory update of the g-variables $g$ and $kalloc(g, hp)$ is successful. If the $konsis_{gvars}$ relation has held before, then its sub relation $konsis_{nameless}$ will be preserved by the memory updates.*

$$\begin{aligned}
&linkable(\Pi_{abs}, \Pi_{CVM}) \wedge \Pi_k = link(\Pi_{abs}, \Pi_{CVM}) \wedge \\
&c_{abs} \in conf^{weak}_{\surd}(te_{abs}, pt_{abs}) \wedge c_k \in conf_{\surd}(te_k, pt_k) \wedge \\
&upd_{mm}(c_{abs}.m, g, v) = \lfloor m'_{abs} \rfloor \wedge upd_{mm}(c_k.m, kalloc(g, hp), v') = \lfloor m'_k \rfloor \wedge \\
&konsis_{pt}(pt_{abs}, pt_k) \wedge konsis_{return}(c_{abs}, c_k, hp) \wedge \\
&konsis_{prog}(c_{abs}, c_k) \wedge konsis_{tenv}(te_{abs}, te_k) \wedge \\
&konsis_{rd}(c_{abs}.m, c_k.m) \wedge konsis_{lst}(c_{abs}.m, c_k.m) \wedge \\
&konsis_{gst}(c_{abs}.m, c_k.m) \wedge konsis_{gvars}(c_{abs}.m, c_k.m, hp) \wedge \\
&konsis_{hst}(c_{abs}.m, c_k.m) \wedge eq_{cont}(type_g(sc(c_{abs}.m), g), v, v') \\
&cont_{reachable}(c_{abs}.m, v, size_T(type_g(sc(c_{abs}.m, g)))) \\
&\Longrightarrow \\
&\quad konsis_{nameless}(m'_{abs}, m'_k, hp)
\end{aligned}$$

Proof We remember the definition of $konsis_{\text{nameless}}$:

$$\begin{aligned}
&konsis_{\text{nameless}}(c_{\text{abs}}.m, c_k.m, hp) \equiv \\
&\quad \forall x, i.reachable_{\text{nameless}}(c_{\text{abs}}.m, x, i) \Longrightarrow \\
&\quad reachable_{\text{nameless}}(c_k.m, kalloc(x, hp), i) \wedge \\
&\quad (?elem(type_g(sc(c_{\text{abs}}.m), x) \Longrightarrow \\
&\qquad eq_{\text{type}}(c_{\text{abs}}.m, x, c_k.m, kalloc(x, hp)) \wedge \\
&\qquad eq_{\text{val}}(x, c_{\text{abs}}.m, kalloc(x, hp), c_k.m)))
\end{aligned}$$

First, we fix an arbitrary nameless g-variable $x$. Then, the proof is conducted by induction over the reachability measure $i$ (cf. Def. 7.13).

**Case 1:** We begin with the induction start, i.e. $i = 0$. Hence, we assume that $x$ is reachable in 0 steps in the successor memory configuration $m'_{\text{abs}}$:

$$reachable_{\text{nameless}}(m'_{\text{abs}}, x, 0). \tag{8.35}$$

At first, we will show that $kalloc(x, hp)$ is reachable in 0 steps in the new memory configuration of the concrete kernel $m'_k$.

Expanding the definition of nameless reachability, we obtain that there exists a valid named pointer variable $p$, which points to $x$:

$$\begin{aligned} &?named(p) \wedge \\ &p \in gvar_{\surd}(m'_{\text{abs}}) \wedge \\ &value_g(m'_{\text{abs}}, p) = Ptr(x) \wedge \\ &ty_g(sc(m'_{\text{abs}}), p) = Ptr_T(tn) \wedge \\ &?init(m'_{\text{abs}}, p) = True \end{aligned} \tag{8.36}$$

Furthermore, $x$ itself is a valid nameless g-variable.

The validity of a named variable also implies its reachability. Together with the fact that the symbol configurations do not change during a memory update, we derive that $p$ was also reachable before the memory update.

Using the transitivity of g-variable validity yields that $kalloc(p, hp)$ was reachable in $c_k.m$ and hence is reachable in $m'_k$, too:

$$kalloc(p, hp) \in reachable_{\text{named}}(m'_k) \tag{8.37}$$

By expanding the consistency relation for named g-variables and using (8.36), we obtain the equivalence of initialization and type for $p$ and $kalloc(p, hp)$ in the old memory configurations. Both type and initialization are invariant regarding a memory update, so we know:

$$\begin{aligned} &?init(m'_{\text{abs}}, p) = ?init(m'_k, p) \implies \\ &\quad eq_{\text{init}}(m'_{\text{abs}}, p, m'_k, kalloc(p, hp)) \end{aligned} \tag{8.38}$$

and

$$eq_{\text{type}}(m'_{\text{abs}}, p, m'_k, kalloc(p, hp)) \tag{8.39}$$

In Lemma 8.21 we have shown that the consistency relation for named g-variables is preserved by a memory update: $konsis_{\text{named}}(m'_{\text{abs}}, m'_k, hp)$. Since $p$ is initialized, the values of $p$ and $kalloc(p, hp)$ are equivalent under $eq_{\text{val}}$:

$$eq_{\text{val}}(p, m'_{\text{abs}}, kalloc(p, hp), m'_k),$$

which we expand using the fact that $p$ is of pointer type:

$$value_g(m'_k, kalloc(p, hp)) = Ptr(kalloc(x, hp)). \tag{8.40}$$

Let us summarize what we have until now: we know that there exists a reachable and hence valid named g-variable $kalloc(p, hp)$ of pointer type $Ptr_T(tn)$, which points to $kalloc(x, hp)$, and is initialized (eq. (8.37), (8.38), (8.39), (8.40)).

There is only one requirement left to satisfy reachability in 0 steps, which is $kalloc(x, hp)$ being a valid nameless g-variable. This can easily be deduced using Lemma 8.2 and the definition of *kalloc*, thus

$$reachable_{\text{nameless}}(m'_k, kalloc(x, hp), 0). \tag{8.41}$$

We come now to type equality of $x$ and $kalloc(x, hp)$. Our theorem only makes a statement for variables of elementary type, hence we assume:

$$?elem(type_g(sc(m'_{\text{abs}}), x))$$

Using [Lei08, Theorem 8.1], which says that nameless heap g-variables being reachable after a memory update have already been reachable before, and the pointer reachability of $v$ we obtain from (8.35):

$$\exists j.\, reachable_{\text{nameless}}(c_{\text{abs}}.m, x, j).$$

This is the necessary precondition to expand the definition of the consistency relation for nameless g-variables. Due to symbol configuration invariance, we simply obtain type equality for the successor configurations:

$$eq_{\text{type}}(m'_{\text{abs}}, x, m'_k, kalloc(x, hp)). \tag{8.42}$$

Finally, there is still value equivalence to show for $x$ and $kalloc(x, hp)$. Again, we expand $konsis_{\text{nameless}}(c_{\text{abs}}.m, c_k.m, hp)$ to derive that this was the case before the memory update.

This—together with the assumptions and the fact that nameless (heap) variables are initialized by definition—is used to instantiate Lemma 8.20, hence obtaining that the values of $x$ and $kalloc(x, hp)$ are also equivalent after the memory update:

$$eq_{\text{val}}(m'_{\text{abs}}, x, m'_k, kalloc(x, hp)). \tag{8.43}$$

Combining equations (8.35), (8.41), (8.42), and (8.43) finish the proof for the induction start.

**Case 2:** Let us now consider the induction step, that is we assume

$$reachable_{\text{nameless}}(m'_{\text{abs}}, x, i + 1). \tag{8.44}$$

We start again with the proof for the reachability of $kalloc(x, hp)$ in the new concrete memory $m'_k$.

Corresponding the definition of $reachable_{\text{nameless}}$ (cf. Def. 7.13) in the inductive case, there are three different ways in which $x$ can be reachable in $m'_{\text{abs}}$.

1. The first, trivial case, is that $x$ is also reachable in $i$ steps. We just use the induction hypothesis to prove this case.

2. In the second case, there is a heap variable $p$, which is itself reachable in $i$ steps and points to a valid $x$:

$$\begin{aligned}&\mathit{reachable}_{\text{nameless}}(m'_{\text{abs}}, p, i) \wedge\\ &\mathit{type}_g(\mathit{sc}(m'_{\text{abs}}), p) = \mathit{Ptr}_T(tn) \wedge\\ &\mathit{value}_g(m'_{\text{abs}}, p) = \mathit{Ptr}(x) \wedge\\ &x \in \mathit{gvar}_{\surd}(m'_{\text{abs}}).\end{aligned}$$

   We instantiate the induction hypothesis for $p$ and $\mathit{kalloc}(p, hp)$, respectively, and obtain type and value equality. This and transitivity of g-variable validity satisfy the preconditions for

$$\mathit{reachable}_{\text{nameless}}(m'_k, \mathit{kalloc}(x, hp), i + 1).$$

3. In the third and last case, there exists a heap variable $p$, which is again reachable in $i$ steps, and $x$ is a sub g-variable of it:

$$\begin{aligned}&\mathit{reachable}_{\text{nameless}}(m'_{\text{abs}}, p, i) \wedge\\ &x \in \mathit{sub}_g(p) \wedge\\ &x \in \mathit{gvar}_{\surd}(m'_{\text{abs}}).\end{aligned}$$

   Here again, the induction hypothesis gives us the reachability for $\mathit{kalloc}(p, hp)$ in $m'_k$ in $j$ steps. In Lemma 8.4 we have shown that $x \in \mathit{sub}_g(p)$ implies $\mathit{kalloc}(x, hp) \in \mathit{sub}_g(\mathit{kalloc}(p, hp))$. So, $\mathit{kalloc}(x, hp)$ is reachable in $m'_k$ in $i + 1$ steps.

We still have to show type and value equality for $x$ and $\mathit{kalloc}(x, hp)$. Since the first one is quite trivial (Lemma 8.1 and symbol configuration invariance), we consider only the latter one.

We proceed similarly to the induction start: We consider the case that $x$ is elementary. Since it is reachable in $m'_{\text{abs}}$, so it must have been reachable in $c_{\text{abs}}.m$. This means that $x$ and $\mathit{kalloc}(x, hp)$ have had the same values before the memory update (expanding the definition of $\mathit{konsis}_{\text{nameless}}$).

Using Lemma 8.20 with these results, we finally obtain

$$\mathit{eq}_{\text{val}}(x, m'_{\text{abs}}, \mathit{kalloc}(x, hp), m'_k). \qquad \text{q.e.d.}$$

**Lemma 8.23 (Equal Content for g-Variables)** *We will now expand value equality of the abstract and concrete kernel variables to those of non-elementary types.*

*Let the invariant on g-variables hold for two (weak) valid kernel configurations $c_{abs}$ and $c_k$. Then, the content of a reachable g-variable $g$ is equal under*

*$eq_{cont}$ with the content of its concrete counterpart $kalloc(g, hp)$.*

$$
\begin{aligned}
&c_k \in conf_{\surd}(te_k, pt_k) \wedge linkable(\Pi_{abs}, \Pi_{CVM}) \wedge \\
&c_{abs} \in conf_{\surd}^{weak} \wedge konsis_{tenv}(te_{abs}, t_k) \wedge \\
&\Pi_k = link(\Pi_{abs}, \Pi_{CVM}) \wedge konsis_{rd}(c_{abs}.m, c_k.m) \wedge konsis_{lst}(c_{abs}.m, c_k.m) \wedge \\
&konsis_{gst}(c_{abs}.m, c_k.m) \wedge konsis_{gvars}(te_{abs}, c_{abs}.m, te_k, c_k.m, hp) \wedge \\
&?init(c_{abs}.m, g) \wedge g \in reachable_{gvars}(c_{abs}.m) \\
&\quad \Longrightarrow \\
&\quad eq_{cont}(type_g(sc(c_{abs}.m), g), value_g(c_{abs}.m, g), value_g(c_k.m, kalloc(g, hp)))
\end{aligned}
$$

Proof We distinguish between $g$ being elementary or not. The first case is trivial, since we simply expand the $konsis_{\text{gvars}}$ invariant.

So let us consider the non-elementary case. Then, the structure of $g$ is given through a set of elementary g-variables with base addresses within the range of $[ba_g(sc(c_{\text{abs}}.m), g), size_T(type_g(sc(c_{\text{abs}}.m), g))]$.

For an arbitrary sub g-variable $x$ let $0 \leq i < size_T(type_g(sc(c_{\text{abs}}.m), g))$ denote its offset in this range and let $v$ denote the value of $g$. The corresponding value of $x$ is then $v(i)$.

We proceed in a similar way for $kalloc(g, hp)$, whose value we abbreviate with $v'$, and $kalloc(x, hp)$, its sub g-variable with offset $i$.

Furthermore, since $g$ is reachable and initialized, so are all its sub g-variables, in particular $x$. By applying the definition of $konsis_{\text{gvars}}$, we obtain $eq_{\text{val}}(x, c_{\text{abs}}.m, kalloc(x, hp), c_k.m)$, which is equivalent to

$$eq_{\text{cont}}(type_g(sc(c_{\text{abs}}.m), x), v(i), v'(i)). \tag{8.45}$$

This holds for all sub g-variables of $g$, and hence for all offsets $i$ in the range mentioned above. With the same argumentation as in Lemma 8.20 regarding the structure of g-variables, we inductively show that eq. (8.45) holds for all array elements or struct components, respectively, comprised by $g$.

This is exactly the definition of $eq_{\text{cont}}$ for non-elementary variables as presented in Def. 7.14. q.e.d.

**Lemma 8.24 (g-Variable Consistency)** *We can now formulate the lemma for the preservation of g-variable consistency for an ordinary C0 kernel step. Given two kernel configurations $c_{abs}$ and $c_k$ and assuming that the usual kernel relations hold—in particular the one for g-variable consistency—and that there exists a successor configuration in the abstract kernel, then g-variable consistency*

*will be preserved.*

$$
\begin{aligned}
&\mathit{linkable}(\Pi_{abs}, \Pi_{CVM}) \wedge \Pi_k = \mathit{link}(\Pi_{abs}, \Pi_{CVM}) \wedge \\
&c_{abs} \in \mathit{conf}_{\surd}^{\mathit{weak}}(te_{abs}, pt_{abs}) \wedge c_k \in \mathit{conf}_{\surd}(te_k, pt_k) \wedge \\
&\delta_{C0}(te_{abs}, pt_{abs}, c_{abs}) = \lfloor c'_{abs} \rfloor \wedge \delta_{C0}(te_k, pt_k, c_k) = \lfloor c'_k \rfloor \wedge \\
&\mathit{konsis}_{prog}(c_{abs}, c_k) \wedge \mathit{konsis}_{tenv}(te_{abs}, te_k) \wedge \\
&\mathit{konsis}_{rd}(c_{abs}.m, c_k.m) \wedge \mathit{konsis}_{lst}(c_{abs}.m, c_k.m) \wedge \\
&\mathit{konsis}_{gst}(c_{abs}.m, c_k.m) \wedge \mathit{konsis}_{gvars}(te_{abs}, c_{abs}.m, te_k, c_k.m, hp) \wedge \\
&\mathit{konsis}_{pt}(pt_{abs}, pt_k) \wedge \mathit{konsis}_{hst}(c_{abs}.m, c_k.m) \wedge \\
&\mathit{konsis}_{return}(c_{abs}, c_k, hp) \wedge \\
&\mathit{hmap}_{inj}(c_{abs}.m, hp) \wedge \mathit{hmap}_{bound}(c_{abs}.m, c_k.m, hp) \wedge \\
&\mathit{heap}_{inv}(c_{abs}.m, c_k.m) \wedge \\
&\neg \mathit{is\_Asm}(\mathit{hd}(\mathit{s2l}(c_{abs}.\mathit{prog}))) \wedge \neg \mathit{is\_Skip}(c_{abs}.\mathit{prog}) \wedge \\
&\neg \mathit{is\_ESCall}(\mathit{hd}(\mathit{s2l}(c_{abs}.\mathit{prog}))) \wedge \neg \mathit{is\_XCall}(\mathit{hd}(\mathit{s2l}(c_{abs}.\mathit{prog}))) \\
&\quad \Longrightarrow \\
&\quad \exists hp'.\, \mathit{konsis}_{gvars}(te_{abs}, c'_{abs}.m, te_k, c'_k.m, hp')
\end{aligned}
$$

Proof The proof is done by case distinction over the head of the abstract program rest $\mathit{hd}(\mathit{s2l}(c_{\mathrm{abs}}.\mathit{prog}))$. For statements that do not update the memory, thus have no affect on the reachability of g-variables and their values, the proof is very straightforward. Hence, we will concentrate on the more interesting cases.

**Case 1:** We start with assignments, that is

$$\mathit{hd}(\mathit{s2l}(c_{\mathrm{abs}}.\mathit{prog})) = \mathit{Ass}(e_l, e_r).$$

The proof for the assignment case basically relies on the successful application of both lemmas 8.21 and 8.22, which state that in the case of a successful memory update the $\mathit{konsis}_{\mathrm{gvars}}$ relation will be preserved. Thus, we will concentrate on discharging the assumptions of these lemmas.

Since there is a successor configuration $c'_{\mathrm{abs}}$ for the abstract kernel, the memory update has been successful according to the transition function of $C0$. This means that left evaluation has yielded an initialized g-variable, right evaluation some value and that the right expression is of some type $t$, formally:

$$
\begin{aligned}
&\exists g, v, t. \\
&\mathit{lval}(te_{\mathrm{abs}}, c_{\mathrm{abs}}.m, e_l) = \lfloor g \rfloor \wedge \\
&\mathit{rval}(te_{\mathrm{abs}}, c_{\mathrm{abs}}.m, e_r) = \lfloor v \rfloor \wedge \\
&?\mathit{init}(te_{\mathrm{abs}}, c_{\mathrm{abs}}.m, e_r) = \mathit{True} \wedge \\
&\mathit{type}(te_{\mathrm{abs}}, c_{\mathrm{abs}}.m, e_l) = \lfloor t \rfloor \wedge \\
&\mathit{upd}_{mm}(c_{\mathrm{abs}}.m, g, v) = \lfloor c'_{\mathrm{abs}}.m' \rfloor
\end{aligned}
$$

We obtain an initialized $\mathit{kalloc}(g, hp)$ and $v'$ from the expression evaluation in the concrete kernel using Lemma 8.9. Hence, the memory update will

be successful in the concrete kernel, too:

$$upd_{mm}(c_k.m, kalloc(g, hp), v') = \lfloor c'_k.m \rfloor.$$

We still have to show $eq_{\text{init}}(type_g(sc(c_{\text{abs}}.m)), v, v')$. There are two cases to be considered: either $e_r$ is a memory object—$?inter(te_{\text{abs}}, c_{\text{abs}}, e_r) = False$—or not. In the latter case, $e_r$ is definitely of elementary type (which can be easily checked looking at the definitions of expression evaluation in Sect. 3.4) and we can apply Lemma 8.9. The same holds for $e_r$ being a memory object of elementary type.

Let us consider the case that $e_r$ is a memory object of non-elementary type. Then, there exists a g-variable $x$ such that $lval(te_{\text{abs}}, c_{\text{abs}}, e_r) = \lfloor x \rfloor$ and $value_g(c_{\text{abs}}.m, x) = v$. Furthermore, this variable is reachable due to [Lei08, Theorem 8.1] and initialized, since $e_r$ is initialized. So we can apply Lemma 8.23 to obtain $eq_{\text{cont}}(type_g(sc(c_{\text{abs}}.m)), v, v')$.

So there is only pointer reachability left to show. In [Lei08, Theorem 8.1], the proof has been done that the right value of an expression, given that it is initialized, in a valid configuration only contains reachable pointer memory cells, hence:

$$cont_{\text{reachable}}(c_{\text{abs}}.m, v, size_T(t)).$$

This is all we need to complete the proof for this case.

**Case 2:** Let us now consider memory allocation, i.e. $hd(s2l(c_{\text{abs}}.prog)) = PAlloc(e, tn)$. We obtain a g-variable $g$ by left evaluation of $e$ and a type $t$ by $mapof(te_{\text{abs}}, tn) = \lfloor t \rfloor$.

For the proof, we distinguish on the availability of heap memory in the abstract kernel.

We start with no memory available, that is $?heap(c_{\text{abs}}.m, t) = False$. In this case, the heap memory stays unchanged and the special null pointer value is assigned to $g$.

Due to the heap invariant $heap_{\text{inv}}(c_{\text{abs}}.m, c_k.m)$, the same happens in the concrete kernel. Both $g$ and $kalloc(g, hp)$ are reachable and—at latest after the memory update—initialized. Furthermore, they are of the same type, since type name environment consistency yields the same type $t$ associated with the type name $tn$ in the concrete kernel. Since the value of both $g$ and $kalloc(g, hp)$ is now $NullPointer$,

$$eq_{\text{val}}(g, c'_{\text{abs}}.m, kalloc(g, hp), c'_k.m)$$

holds, too. This is all we need to show $konsis_{\text{gvars}}(c'_{\text{abs}}.m, c'_k.m)$.

Now we examine the case in which enough heap memory is available. In an intermediate step (cf. Def. 3.25), we extend the heap symbol table of both kernels by appending an entry $(undef, t)$ and initialize the value of the new heap variable according to Def. 3.19. Let us denote these intermediate memories with $m'_{\text{abs}} = ext_{\text{heap}}(?heap, c_{\text{abs}}.m, t)$ and $m'_k = ext_{\text{heap}}(?heap, c_k.m, t)$, respectively.

We update our heap map function $hp$ in such a way that the new abstract heap element index is mapped to the new concrete element index . The

mapping stays unchanged for all other values and we denote the update heap map by $hp'$:

$$hp'(x) = \begin{cases} |hst(c_k.m)| & \text{if } x = |hst(c_{\text{abs}}.m)| \\ hp(x) & \text{otherwise} \end{cases}$$

Note that Lemma 8.19 guarantees that the heap invariants are preserved by such an update.

In a second step, we obtain the successor memory configuration by updating the value of $g$ with a pointer to the new heap variable:

$$c'_{\text{abs}}.m = upd_{mm}(m'_{\text{abs}}, g, Ptr(gvar_{hm}(|hst(c_{\text{abs}}.m)|)))$$

and similarly for the concrete kernel, using Lemma 8.13:

$$c'_k.m = upd_{mm}(m'_k, kalloc(g, hp), Ptr(gvar_{hm}(|hst(c_k.m)|))).$$

Since both $g$ and $kalloc(g, hp)$ are reachable, the new heap variables to which they point are reachable, too. Furthermore, we know that $gvar_{hm}(|hst(c_k.m)|) = kalloc(gvar_{hm}(|hst(c_{\text{abs}}.m)|), hp')$.

Obviously, both heap variables are of the same type, namely $t$. Let us assume that $t$ is elementary. Then the value of these variables is equal to the initial value of $t$, which is 0 for $Int_T$, $Unsgnd_T$ and $Char_T$, *False* for $Bool_T$, and *NullPointer* for pointer types. In any case, the value equivalence holds (see Def. 7.14 for elementary types).

So far for the new heap variables. We still have to have a look at $g$ and $kalloc(g, hp)$ (which is equal to $kalloc(g, hp')$). Both g-variables have been obtained by left evaluating an expression in a valid configuration, so they are reachable in this configuration.

In addition, reachability of the updated variable is invariant. Hence, both $g$ and $kalloc(g, hp')$ are reachable in $c'_{\text{abs}}.m$ and $c'_k.m$, respectively, and initialized.

Furthermore, they are of the same—elementary—type $Ptr_T(tn)$ Since the memory update was successful in the abstract kernel, the value of $g$ in the successor configuration is

$$value_g(c'_{\text{abs}}.m, g) = Ptr(gvar_{hm}(|hst(c_{\text{abs}}.m)|))$$

and for $kalloc(g, hp')$ in the concrete kernel

$$value_g(c'_k.m, g) = Ptr(kalloc(gvar_{hm}(|hst(c_{\text{abs}}.m)|), hp')).$$

Using the definition of $eq_{\text{cont}}$ for pointer types (cf. Def. 7.14), we finally deduce

$$eq_{\text{val}}(g, c'_{\text{abs}}.m, kalloc(g, hp'), c'_k.m).$$

**Case 3:** We will now consider function calls, that is $hd(s2l(c_{\text{abs}}.prog)) = SCall(\sigma, fn, plist)$.

The relevant parts of a function call regarding g-variable consistency are the stack extension and the parameter passing. The new stack frame being added by the call defines a return destination $g$, which comes from the expression evaluation of $Var(\sigma)$. Using Lemma 8.9, we know that the return destination in the concrete kernel is $kalloc(g, hp)$. Since these two variables are reachable in $c_{\text{abs}}$ and $c_k$ and since their values are not changed, the invariant on g-variables still holds in the successor configurations of both kernels.

Let us now deal with parameter passing. The called function $fn$ has a function descriptor $f$, such that $(fn, f) \in \{pt_{\text{abs}}\}$. Due to procedure table consistency, there exists a corresponding entry $(fn, f')$ in the concrete procedure table $pt_k$ with $f'.params = f.params$ and $f'.lvars = f.lvars$.

As we have seen in Sect. 3.5.3, at first the local memory stack is extended by a new stack frame $mf'$ with

$$mf' = \begin{bmatrix} st = st_{\text{func}}(f) \\ ct = undef \\ init = \emptyset \end{bmatrix}.$$

We denote the memories extended with $mf'$ with $m_{\text{abs}}$ and $m_k$, respectively. Subsequently, the parameter list $plist$ is right evaluated. In the next step, the parameter names of the called function are obtained. Finally, the memory is updated, e.g., given a parameter name $\pi$ and a value $v$:

$$upd_{mm}(m_{\text{abs}}, gvar_{lm}(|c_{\text{abs}}.m.lm|, \pi), v) = \lfloor m'_{\text{abs}} \rfloor.$$

In the concrete kernel, we proceed likewise. Since the parameters are the same, so are the names. Let $\pi$ and $v'$ denote such a parameter name and its corresponding value.

So, the corresponding memory update in the concrete kernel looks like this:

$$\begin{aligned} & upd_{mm}(m_k, gvar_{lm}(|c_k.m.lm|, \pi), v') \\ &= upd_{mm}(m_k, gvar_{lm}(|c_{\text{abs}}.m.lm| + rd_{\text{off}}, \pi), v') \\ &= upd_{mm}(m_k, kalloc(gvar_{lm}(|c_{\text{abs}}.m.lm|, \pi), hp), v') \\ &= \lfloor m'_k \rfloor. \end{aligned}$$

$gvar_{lm}(|c_{\text{abs}}.m.lm|, \pi)$ and $kalloc(gvar_{lm}(|c_{\text{abs}}.m.lm|, \pi), hp)$ are reachable in $m_{\text{abs}}$ and $m_k$, since they are valid and named. Furthermore, they are initialized due to the memory update and of equal type, since the function symbol tables are equal.

Finally, we obtain content equality for $v$ and $v'$ in the same manner as we did it for assignments further above, depending on the expression evaluation returning a memory object or not.

Using Lemma 8.9, we prove that the $konsis_{\text{named}}$ holds for these two variables in the two successor configurations.

We proceed likewise by induction over the parameter list of the function call. Since all other variables stay unchanged, both regarding reachability and value, the invariant on g-variable consistency also holds in the successor configurations $c'_{\text{abs}}$ and $c'_k$.

**Case 4:** The last case is an abstract program rest starting with a return statement, i.e. $hd(s2l(c_{\text{abs}}.prog)) = Return(e)$. The proof for this case is very similar to the one dealing with assignments.

The variable updated by a return statement is specified by the return destination of the current stack frame. For the abstract kernel, this is:

$$\begin{aligned} g &= snd(toplm(c_{\text{abs}}.m)) \\ &= snd(c_{\text{abs}}.m.lm!(|sc.lst(c_{\text{abs}}.m)| - 1)). \end{aligned}$$

Exploiting the invariants on return destinations and recursion depth, $konsis_{\text{return}}$ and $konsis_{\text{rd}}$, we obtain for the concrete kernel:

$$\begin{aligned} &snd(toplm(c_k.m)) \\ &= snd(c_k.m.lm!(|sc.lst(c_k.m)| - 1)) \\ &= snd(c_k.m.lm!(|sc.lst(c_{\text{abs}}.m)| + rd_{\text{off}} - 1)) \\ &= kalloc(g, hp). \end{aligned}$$

Both variables are reachable, since they are named and valid.

We right evaluate $e$ to obtain the return values: $v$ in the abstract kernel and $v'$ in the concrete kernel, where we obtain

$$eq_{\text{cont}}(type_g(sc(c_{\text{abs}}.m), g), v, v')$$

in a similar way as we did it in the assignment case.

Again, memory updates do not change the reachability of the updated variable, so both $g$ and $kalloc(g, hp)$ are reachable and initialized in the successor configurations. Applying Lemma 8.21 finally completes the proof for this case. q.e.d.

## 8.6 Preservation of Kernel Consistency by $C0$ Kernel Steps

In the previous sections, we have proven for ordinary $C0$ steps—no XCalls, no external function calls, no assembler—that the individual relations between the abstract and concrete kernel components are preserved under the assumption that the abstract kernel does not yield a run-time error.

Furthermore, we have shown that if the abstract kernel proceeds, i.e. yields some successor configuration using the $C0$ transition function, then concrete kernel execution will not fail neither.

We will now summarize these results into one theorem stating that the abstract and concrete kernel relation *konsis* will be preserved.

**Lemma 8.25 (Induction Step for *konsis*)** *Let $c_{abs}$ denote a weak valid abstract kernel configuration and $c_k$ a valid concrete kernel configuration, where the concrete kernel programs results from linking the abstract kernel to the low-level CVM implementation.*

*Further we assume that the kernel relation konsis holds and that the program rest of the abstract kernel neither starts with an external function call, an assembler or a XCall statement, nor that abstract kernel execution has terminated, and that there exists an abstract successor configuration obtained by applying the $C0$ transition function.*

*Then there exists also a concrete successor configuration, such that the kernel relation holds for these two new configurations.*

$$
\begin{aligned}
&\mathit{linkable}(\Pi_{abs}, \Pi_{CVM}) \wedge \Pi_k = \mathit{link}(\Pi_{abs}, \Pi_{CVM}) \wedge \\
&\mathit{konsis}(te_{abs}, pt_{abs}, c_{abs}, te_k, pt_k, c_k) \wedge \\
&c_{abs} \in \mathit{conf}^{weak}_{\surd}(te_{abs}, pt_{abs}) \wedge c_k \in \mathit{conf}_{\surd}(te_k, pt_k) \wedge \\
&\delta_{C0}(te_{abs}, pt_{abs}, c_{abs}) = \lfloor c'_{abs} \rfloor \wedge \\
&\neg \mathit{is\_Asm}(\mathit{hd}(\mathit{s2l}(c_{abs}.\mathit{prog}))) \wedge \neg \mathit{is\_Skip}(c_{abs}.\mathit{prog}) \wedge \\
&\neg \mathit{is\_ESCall}(\mathit{hd}(\mathit{s2l}(c_{abs}.\mathit{prog}))) \wedge \neg \mathit{is\_XCall}(\mathit{hd}(\mathit{s2l}(c_{abs}.\mathit{prog}))) \\
&\quad \Longrightarrow \\
&\quad \exists c'_k.\, \delta_{C0}(te_k, pt_k, c_k) = \lfloor c'_k \rfloor \wedge \mathit{konsis}(te_{abs}, pt_{abs}, c'_{abs}, te_k, pt_k, c'_k)
\end{aligned}
$$

A conclusion is simply the place where someone got tired of thinking.

Unknown

# Chapter 9

# Summary

**Contents**



In this Chapter, we will first summarize on the results presented in this thesis (Sect. 9.1). Subsequently, we discuss the work that has yet to be done in order to integrate our results into the overall correctness theorem (Sect. 9.2) We will then discuss some possible directions for future work in operating systems verification (Sect. 9.3).

## 9.1 Achievements

**CVM Computational Model** In this thesis, we have introduced the formal semantics of CVM, a hardware-abstracting framework for microkernel programmers. The CVM semantics definitions have not only been presented in this work, but they are also formalized within the Isabelle/HOL interactive theorem prover environment. Unlike the related work, CVM

- offers serious support for virtual memory and its management,
- integrates user processes and devices into the computational model,
- considers the low-level hardware in detail.

To the best of our knowledge, there exists no other project covering all of the above features.

The development of CVM semantics started with [GHLP05]. Though the current formal semantics has been largely specified by us, it obviously benefitted from contributions and demands from many people in the Verisoft project, who have been using CVM in one or the other way.

**Correctness and Pervasiveness** CVM correctness is formulated as a simulation theorem, where the hardware simulates the CVM model. Individual states of the computational models are related by an abstraction relation, which subsumes several other abstraction relations for the single components of a CVM configuration.

We have defined the relation between abstract and concrete kernel. The user process relations bases on our work and has been continued by Tsyban. [Tsy09] has formulated the other correctness relations and invariants, which are used in the top-level correctness theorem sketch as presented in [AHL+09], basing on an initial version of Tsyban.

For standard abstract kernel steps, that is those, in which no primitive is called or in which the kernel goes to kernel wait, we have presented the proof that the kernel relation is preserved. Particularly this involves that the concrete kernel does not produce any run-time faults if the abstract kernel does not. This proof has also been formalized to a very large extent in Isabelle/HOL.

Furthermore, CVM has been developed with pervasive system verification in mind. This means that for all layers in the system stack, we have developed computational models or have integrated models from other sources within the Verisoft project.

While most other OS verification projects aim to prove that the implementation satisfies a certain security policy, the CVM top-level correctness theorem is a statement on total functional correctness. Practically, this implies already more or less complex security policies as memory separation of user processes. Yet, it is also sufficient for much stronger correctness criteria, in particular if one aims to prove functional correctness of layers above CVM as it is done in the Verisoft project.

**Compositionality and Modularization** There are certainly different philosophies when verifying a system stack. One is to take the overall actual implementation of everything at hand—compiler, hardware, low-level CVM implementation—and disclose the full state of all components to the layers above.

There is nothing wrong with this if one merely aims to verify one specific given stack. It even makes verification life easier, since one can resort to all low-level detail at any time and does not have to worry about clean and concise interfaces.

Yet, this approach falls short whenever it comes to specification and/or implementation modifications on any layer: the proofs have to be re-checked and adapted, if possible. Moreover, the bigger the system stack gets, the more demanding it becomes for the verification engineer to handle this complexity—or even worse: become acquainted with an existing design and proof.

It is not a new idea to aim for encapsulation and modularization. In software technology and the design of programming languages, the same experiences with growing software have led already decades ago to the same principles. This is why we have chosen to stick with the credo *as much detail as necessary, as less as possible*.

In particular, we have introduced the notion of abstract linking and linkability of $C0$ programs. This allows to separate the low-level hardware-dependent implementation from the actual kernel implementation. Not for nothing, CVM

has been designed as a *hardware-independent framework for microkernel programmers*, where the abstract kernel is a parameter provided by the implementer.

This allows us to define a clean interface for the verification above CVM level: one can show code correctness for the abstract kernel (e.g., absence of run-time faults) using comfortable tool support and rely on the correctness of the CVM implementation, given that the resource restrictions and linkability requirements are met.

Furthermore, by proceeding like this, we were able to use and re-use plenty of results (and in particular lemmas) from [Lei08]. The proof we have presented demonstrates the feasibility of this approach: the correctness relation between the abstract and the concrete kernel—obtained by linking—is preserved.

## 9.2 Integrating Our Results

As a matter of fact, our work did not always proceed with the speed for which we have hoped. In order to be able to continue her work on the top-level correctness criterion, Tsyban has formulated her own version of a relation for the abstract and the concrete kernel.

Yet, the integration of our correctness proof requires some predictable and manageable effort. In the remainder of this section, we will sketch the changes and extra work that has to be done.

**The Dummy Function** The dummy function, whose existence we have assumed in Sect. 6.2.2, is a crutch to allow for the use of the existing *C*0 small-step semantics and its rich resources on lemmas when arguing about the abstract kernel. In the concrete kernel, the abstract kernel dispatcher is called by the low-level CVM dispatcher in the same way as the dummy function.

A meta-theorem that the concrete kernel obtained by linking behaves in the same way with or without this dummy function has to be shown. Yet, since this function is never called, the proof of such a theorem is supposedly rather simple.

For instance, our construction fits perfectly well with the $\mathit{kernel\text{-}rel}_{\mathrm{result}}$ relation from [Tsy09], which relates the `cup` variable of the concrete kernel with the `abs_kernel_res` variable of the abstract kernel.

**Induction Start** For the induction start, given through an abstract kernel configuration after reset, it has to be proven that the kernel relation holds. We would relate this abstract kernel configuration to a concrete kernel configuration where the head of the program rest also points to a call of the abstract kernel dispatcher.

In the abstract kernel, there has been no memory allocated, its global variables are disjoint from those of the low-level implementation (hence of initial value) and the stack has been freshly initialized. These facts make the proof for most kernel invariants rather easy. As Fig. 9.1 illustrates this for consistency of the program rest at induction start.

**Resource Restrictions** Currently we assume that memory allocation does not fail in the concrete kernel if it succeeds in the abstract kernel. This assumption has to be discharged.

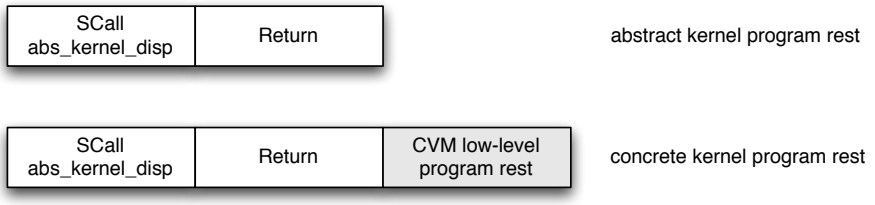


Figure 9.1: Abstract and Concrete Program Rest at Induction Start

Yet, the low-level CVM implementation in Verisoft makes currently very little use of memory allocation (and fortunately not in a recursive manner), basically only when creating the page tables for the (fixed number of) user processes and some data structures for the page fault handler. Computation of memory consumption can hence be done in the conventional way for the abstract kernel, with the constant offset for the CVM implementation being simply added.

**Kernel Wait and Primitives** For kernel wait and the primitives it still has to be shown that kernel consistency is respected (in the sense that it holds after a primitive call and after leaving kernel wait, not necessarily in between). The proof has to focus mainly on the consistency relation for g-variables, in particular heap variables (due to global variable disjointness). In the current CVM low-level implementation, primitives do not take pointer arguments, making things quite easier. This is not true when dealing with physical I/O, where data structures of the abstract kernel are directly changed. Yet, there have been no such primitives been verified in Verisoft up to now.

**Theorem Integration** [Tsy09] has used a correctness relation *kernel-rel* for the abstract and concrete kernel. Basically, this relation covers a subset of the sub relations of *konsis* as defined in the chapters before and uses concrete numbers instead of constants, as given through the current low-level CVM implementation.

For instance Tsyban's conversion function *abs2conc* [Tsy09, Def. 7.5] uses an offset of 2 for local variables, where we use the constant $rd_{\text{off}}$, and uses a constant shift for heap variables, where we prefer a mapping function between the abstract and the concrete heap (note, that this implies that the concrete kernel is only allowed to allocate memory at the very beginning, but never again during execution). The instantiation of *kalloc* with appropriate parameters is more or less trivial.

The relation for g-variables in the abstract and concrete kernel, $\textit{kernel-rel}_{\text{Gvar}}$ [Tsy09, Def. 7.7], is syntactically nearly the same as the one of $\textit{konsis}_{\text{gvars}}$ (cf. Def. 7.10). Generally, our correctness statement given by *konsis* is stronger, such that it should be feasible to prove that it implies the correctness statement of Tsyban—with a reasonable amount of work.

Regarding abstract linking, Tsyban prefers to swap the parameters compared to our use of *link* in this thesis. Since the compiler used in the Verisoft project

allocates $C0$ functions in the assembler memory in the order they are defined in the procedure table of the compiled program, she can use this to obtain the concrete address of the low-level `init()` and `dispatcher()` function, which she uses in her proofs.

Though this is a pretty strong assumption, since the use of a different compiler would render large parts of the proof useless, we do not require a certain order in the linking of procedure tables (unlike for type name environments, see also the remark at Lemma 8.7). Hence, we could easily adapt our linking process in the desired way.

Last but not least, our correctness theorem as presented in Lemma 8.25 has to be applied within the proof for the correctness of a kernel step (cf. Theorem 7.2). Besides the instantiation of the compiler correctness [Lei08], this would require to prove preservation of the other correctness relations.

Of all these relations, the user process relation $B$ is probably the most demanding one. Since in kernel mode, user process relation bases on the process control blocks, that is heap variables of the concrete kernel (cf. Sect. 6.5.1 and 7.2.1), we basically have to show that the abstract kernel does not alter these variables. The current CVM low-level implementation does not make use of pointers as function parameters, so we can be sure that this will not happen, making the proof seemingly not too tedious (though it is a strong assumption on the actual code, see also next section).

## 9.3 Operating Systems Verification

The academic system was not meant to be a high-performance system, but to show the feasibility of the Verisoft approach to a system stack of realistic complexity regarding size and functionality. This has been accomplished and we have the necessary theory at hand.

Yet, to play devil's advocate: the system has been built with formal verification in mind and neither the programming languages nor the hardware involved are commercially available or relevant.

To lift the quality of our results to industrial heights, we have to apply the methodology to an industrial system. In fact, this is part of the Verisoft XT project, where the Hypervisor and PikeOS verification benefits considerably from the experiences made in specifying and verifying an operating system.

Though the settings are by far more complex, the top-level correctness criteria are similar: memory separation and functional correctness of the kernel calls. The existing theory has of course be extended, in order to handle features like preemption, memory sharing and the fact that the kernel itself runs many threaded in a truly concurrent way.

An analysis, how our results and those of the L4.verified project [KEH+09] could be used to mutual improvement would be of great interest.

Last but not least, the current CVM low-level implementation does not make use of pointers as function parameters. This means we can be sure that the abstract kernel never gets access to concrete kernel only data structures (i.e. CVM data structures), a property that makes for instance proving of the $B$-relation much easier. It would be interesting to analyze the problems that arise when allowing for function pointers and its consequences regarding heap separation.

# Bibliography


[ABK08] Eyad Alkassar, Peter Boehm, and Steffen Knapp. Formal correctness of a gate-level automotive bus controller implementation. In Bernd Kleinjohann, Lisa Kleinjohann, and Wayne Wolf, editors, *6th IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES08)*, pages 57–68. Springer, 2008.

[AH08] Eyad Alkassar and Mark A. Hillebrand. Formal functional verification of device drivers. In Natarajan Shankar and Jim Woodcock, editors, *2nd IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08)*, volume 5295 of *LNCS*, pages 225–239. Springer, 2008.

[AHK+07] E. Alkassar, M. Hillebrand, S. Knapp, R. Rusev, and S. Tverdyshev. Formal device and programming model for a serial interface. In B. Beckert, editor, *Proceedings, 4th International Verification Workshop (VERIFY), Bremen, Germany*, pages 4–20. CEUR-WS Workshop Proceedings, 2007.

[AHL+09] Eyad Alkassar, Mark A. Hillebrand, Dirk C. Leinenbach, Norbert W. Schirmer, Artem Starostin, and Alexandra Tsyban. Balancing the load. Leveraging a semantics stack for systems verification. *Journal of Automated Reasoning*, 2009.

[Alk09] Eyad Alkassar. *OS Verification Extended - On the Formal Verification of Device Drivers and the Correctness of Client/Server Software*. PhD thesis, Saarland University, Computer Science Department, 2009.

[ASS08] Eyad Alkassar, Artem Starostin, and Norbert Schirmer. Formal pervasive verification of a paging mechanism. In C. R. Ramakrishnan and Jakob Rehof, editors, *Fourteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *LNCS*, pages 109–123. Springer, 2008.

[BB04] Bernhard Beckert and Gerd Beuster. Formal specification of security-relevant properties of user interfaces. In *Proceedings, 3rd International Workshop on Critical Systems Development with UML, Lisbon, Portugal*, Munich, Germany, 2004. TU Munich Technical Report TUM-I0415.

[BB06a] Bernhard Beckert and Gerd Beuster. Guaranteeing consistency in text-based human-computer interaction. In *Proceedings, International Workshop on Formal Methods for Interactive Systems (FMIS), Macao SAR China*, 2006.

[BB06b] Bernhard Beckert and Gerd Beuster. A method for formalizing, analyzing, and verifying secure user interfaces. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*, volume 4260 of *Lecture Notes in Computer Science*. Springer, 2006.

[BB09] Christoph Baumann and Thorsten Bormer. Verifying the PikeOS microkernel: An overview of the Verisoft XT avionics project. In *4th International Workshop on Systems Software Verification (SSV 2009)*, Electronic Notes in Theoretical Computer Science. Elsevier Science B.V., 2009. To appear.

[BBBB09a] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Better avionics software reliability by code verification – A glance at code verification methodology in the Verisoft XT project. In *Embedded World 2009 Conference*, Nuremberg, Germany, March 2009. Franzis Verlag. To appear.

[BBBB09b] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Formal verification of a microkernel used in dependable software systems. In Bettina Buth, Gerd Rabe, and Till Seyfarth, editors, *Computer Safety, Reliability, and Security (SAFECOMP 2009)*, Lecture Notes in Computer Science, Hamburg, Germany, 2009. Springer. To appear.

[BBG+05] Sven Beyer, Peter Böhm, Michael Gerke, Mark Hillebrand, Tom In der Rieden, Steffen Knapp, Dirk Leinenbach, and Wolfgang J. Paul. Towards the formal verification of lower system layers in automotive systems. In *ICCD '05*, pages 317–324. IEEE Computer Society, 2005.

[BBJ79] T.A. Berson and G.L. Barksdale Jr. KSOS: Development methodology for a secure operating system. In *AFIPS Conference Proceedings*, volume 48, pages 365–371, 1979.

[BDD+92] Manfred Broy, Frank Dederich, Claus Dendorfer, Max Fuchs, Thomas Gritzner, and Rainer Weber. The Design of Distributed Systems - An Introduction to FOCUS. Technical Report TUM-I9202, Technische Universität München, January 1992.

[Bev87] William R. Bevier. A verified operating system kernel. Technical Report 11, Computational Logic Inc., Austin, Texas, 1987.

[Bev88] William R. Bevier. Kit: A study in operating system verification. Technical Report 28, Computational Logic Inc., Austin, Texas, 1988.

[Bev89a] William R. Bevier. Kit: A study in operating system verification. *IEEE Trans. Softw. Eng.*, 15(11):1382–1396, 1989.

[Bev89b] William R. Bevier. Kit and the short stack. *J. Autom. Reasoning*, 5(4):519–530, 1989.

[Bey05] Sven Beyer. *Putting It All Together: Formal Verification of the VAMP*. PhD thesis, Saarland University, Computer Science Department, March 2005.

[BGH+06] Jewgenij Botaschanjan, Alexander Gruler, Alexander Harhurin, Leonid Kof, Maria Spichkova, and David Trachtenherz. Towards modularized verification of distributed time-triggered systems. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21–27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2006.

[BHMY89] William R. Bevier, Warren A. Hunt, Jr., J S. Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, 1989.

[BHW06] Gerd Beuster, Niklas Henrich, and Markus Wagner. Real world verification — experiences from the verisoft email client. In *Proceedings of the Workshop on Empirical Succesfully Computerized Reasoning (ESCoR 2006)*, 2006.

[BJK+03] Sven Beyer, Christian Jacobi, Daniel Kroening, Dirk Leinenbach, and Wolfgang Paul. Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP. In D. Geist and E. Tronci, editors, *Proc. of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2860 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 2003.

[BJK+06] Sven Beyer, Christian Jacobi, Daniel Kroening, Dirk Leinenbach, and Wolfgang Paul. Putting it all together: Formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer*, 8(4–5):411–430, August 2006.

[BKKS05] Jewgenij Botaschanjan, Leonid Kof, Christian Kühnel, and Maria Spichkova. Towards verified automotive software. In *SEAS '05: Proceedings of the Second International Workshop on Software Engineering for Automotive Systems*, pages 46–51. ACM, 2005.

[BKM95] R. S. Boyer, M. Kaufmann, and J. S. Moore. The boyer-moore theorem prover and its interactive enhancement. *Computers & Mathematics with Applications*, 29(2):27 – 62, 1995.

[BLW08] Sascha Böhme, Rustan Leino, and Burkhart Wolff. HOL-Boogie – An interactive prover for the Boogie program verifier. In *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170

of *Lecture Notes in Computer Science*, pages 150–166, Montréal, Québec, Canada, 2008. Springer.

[Boa96] Ariane 501 Inquiry Board. Ariane 5 - Flight 501 Failure. `http://homepages.inf.ed.ac.uk/perdita/Book/ariane5rep.html`, July 1996.

[Bog08] Sebastian Bogan. *Formal Specification of a Simple Operating System.* PhD thesis, Saarland University, Computer Science Department, August 2008.

[BS93a] William R. Bevier and Lawrence M. Smith. A mathematical model of the mach kernel: Atomic actions and locks. Technical Report 89, Computational Logic Inc., Austin, Texas, 1993.

[BS93b] William R. Bevier and Lawrence M. Smith. A mathematical model of the mach kernel: Entities and relations. Technical Report 88, Computational Logic Inc., Austin, Texas, 1993.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *In ACM Symp. on Principles of Programming Languages*, pages 266–277. ACM Press, 1997.

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lecture Notes in Computer Science*, Munich, Germany, 2009. Springer. Invited paper, to appear.

[CMST09] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. A precise yet efficient memory model for C. In *4th International Workshop on Systems Software Verification (SSV 2009)*, Electronic Notes in Theoretical Computer Science. Elsevier Science B.V., 2009. To appear.

[Con06] Cosmin Condea. Design and implementation of a page fault handler in C0. Master's thesis, Saarland University, 2006.

[Dal06] Iakov Dalinger. *Formal Verification of a Processor with Memory Management Units.* PhD thesis, Saarland University, July 2006.

[Dav04] David E. Hoffman. CIA slipped bugs to Soviets. `http://www.msnbc.msn.com/id/4394002`, 2004.

[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, *Proceedings, 5th International Verification Workshop (VERIFY), Sydney, Australia*, volume 372 of *CEUR Workshop Proceedings*, pages 56–70. CEUR-WS.org, August 2008.

[DDW09] Matthias Daum, Jan Dörrenbächer, and Burkhart Wolff. Proving fairness and implementation correctness of a microkernel scheduler. 42, Numbers 2-4:349–388, 2009.

[DDWS08] Matthias Daum, Jan Dörrenbächer, Burkhart Wolff, and Mareike Schmidt. A verification approach for system-level concurrent programs. In Jim Woodcock and Natarajan Shankar, editors, *Verified Software: Theories, Tools, Experiments Second International Conference, VSTTE 2008, Toronto, Canada, October 6–9, 2008. Proceedings*, volume 5295 of *Lecture Notes in Computer Science*, pages 161–176, Toronto, Canada, October 2008. Springer.

[DHP05] Iakov Dalinger, Mark Hillebrand, and Wolfgang Paul. On the verification of memory management mechanisms. In D. Borrione and W. Paul, editors, *Proceedings of the 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *Lecture Notes in Computer Science*, pages 301–316. Springer, 2005.

[EKD+07] Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, pages 117–122, San Diego, CA, USA, May 2007.

[EKE08] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified protection model of the seL4 microkernel. In Jim Woodcock and Natarajan Shankar, editors, *Proceedings of VSTTE 2008 — Verified Software: Theories, Tools and Experiments*, volume 5295 of *Lecture Notes in Computer Science*, pages 99–114, Toronto, Canada, Oct 2008. Springer.

[Elp04] Kevin Elphinstone. Future directions in the evolution of the L4 microkernel. In Gerwin Klein, editor, *Proceedings of the NICTA workshop on OS verification 2004, Technical Report 0401005T-1*, Sydney, Australia, Oct 2004. National ICT Australia.

[FSDG08] Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 170–182, New York, NY, USA, 2008. ACM.

[FSGD09] Xinyu Feng, Zhong Shao, Yu Guo, and Yuan Dong. Certifying low-level programs with hardware interrupts and preemptive threads. *Journal of Automated Reasoning*, 42(2-4):301–347, April 2009.

[GHLP05] Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang Paul. On the correctness of operating system kernels. In J. Hurd and T. F. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. *Security and Privacy, IEEE Symposium on*, 0:11, 1982.

[GM84] Joseph A. Goguen and Jose Meseguer. Unwinding and inference control. *Security and Privacy, IEEE Symposium on*, 0:75, 1984.

[Gro01] Network Working Group. Simple Mail Transport Protocol. `http://www.apps.ietf.org/rfc/rfc2821.html`, April 2001.

[Har85] Norman Hardy. Keykos architecture. *SIGOPS Oper. Syst. Rev.*, 19(4):8–25, 1985.

[HH01] Michael Hohmuth and Hermann Haertig. Pragmatic nonblocking synchronization for real-time systems. In *Proceedings of the 2001 Usenix Annual Technical Conference (USENIX '01)*, 2001.

[Hil05] Mark A. Hillebrand. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctness*. PhD thesis, Saarland University, 2005.

[HIP05] Mark Hillebrand, Tom In der Rieden, and Wolfgang Paul. Dealing with I/O devices in the context of pervasive system verification. In *ICCD '05*, pages 309–316. IEEE Computer Society, 2005.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[HP07] M. A. Hillebrand and W. J. Paul. On the architecture of system verification environments. In *Haifa Verification Conference 2007, October 23-25, 2007, Haifa, Israel*, LNCS. Springer, 2007.

[HPLR94] Robert Harper, Frank Pfenning, Peter Lee, and Eugene Rollins. A compilation manager for standard ml of new jersey. In *In 1994 ACM SIGPLAN Workshop on ML and its Applications*, pages 136–147, 1994.

[HSY06] David S. Hardin, Eric W. Smith, and William D. Young. A robust machine code proof framework for highly secure applications. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 11–20, New York, NY, USA, 2006. ACM.

[HT05] Michael Hohmuth and Hendrik Tews. The VFiasco approach for a verfied operating system. Technical Report TUD-FI05-15, Technical University Dresden, Dresden, Germany, December 2005.

[HW73] C. A. R. Hoare and Niklaus Wirth. An axiomatic definition of the programming language pascal. *Acta Inf.*, 2:335–355, 1973.

[Hyp09] Microsoft Hyper-V Server: Homepage. `http://www.microsoft.com/hyper-v-server/en/us/default.aspx`, 2009.

[IK05] Tom In der Rieden and Steffen Knapp. An approach to the pervasive formal specification and verification of an automotive system. In *FMICS '05*, pages 115–124. IEEE Computer Society, 2005.

[ILP05] Tom In der Rieden, Dirk Leinenbach, and Wolfgang Paul. Towards the pervasive verification of automotive systems. In Dominique Borrione and Wolfgang Paul, editors, *Proceedings of the 13th Advanced Research Working Conference on Correct Hardware Design*

*and Verification Methods (CHARME 2005)*, volume 3725 of *Lecture Notes in Computer Science*, pages 3–4. Springer, 2005.

[Inf81a] Information Sciences Institute. Internet Protocol, Darpa Internet Program, Protocol Specification. `http://www.apps.ietf.org/rfc/rfc791.html`, September 1981.

[Inf81b] Information Sciences Institute. Transmission Control Protocol - Protocol Specification. `http://www.apps.ietf.org/rfc/rfc793.html`, September 1981.

[IP08] Tom In der Rieden and Wolfgang J. Paul. Beweisen als Ingenieurwissenschaft: Verbundprojekt Verisoft (2003–2007). In Bernd Reuse and Roland Vollmar, editors, *Informatikforschung in Deutschland*, pages 321–326. Springer, 2008.

[ISO99] *ISO 9899:1999: Programming languages – C*. International Standardization Organization, 1999.

[IT08] Tom In der Rieden and Alexandra Tsyban. CVM - a verified framework for microkernel programmers. In *3rd intl Workshop on Systems Software Verification (SSV08), to appear*. Elsevier Science B. V., 2008.

[Jac02] Christian Jacobi. *Formal Verification of a Fully IEEE-compliant Floating-Point Unit*. PhD thesis, Saarland University, 2002.

[Jan95] Mark Janeba. The pentium problem. `http://www.willamette.edu/~mjaneba/pentprob.html`, 1995.

[Jon03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, Oct 2009. ACM. To appear.

[Kle09] Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, Feb 2009.

[Kna08] Steffen Knapp. *The Correctness of a Distributed Real-Time System*. PhD thesis, Universit"at des Saarlandes, 2008.

[KP07a] Steffen Knapp and Wolfgang Paul. Pervasive verification of distributed real-time systems. In Manfred Broy, Johannes Grünbauer, and Tony Hoare, editors, *Software System Reliability and Security*, volume 9 of *IOS Press, NATO Security Through Science Series. Sub-Series D: Information and Communication Security*, pages 239–297, 2007.

[KP07b] Steffen Knapp and Wolfgang J. Paul. Realistic worst case execution time analysis in the context of pervasive system verification. In *Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm*, volume 4444 of *Lecture Notes in Computer Science*, pages 53–81. Springer, 2007.

[Kro01] Daniel Kroening. *Formal Verification of Pipelined Microprocessors*. PhD thesis, Saarland University, Computer Science Department, 2001.

[KS06] Christian Kühnel and Maria Spichkova. Upcoming automotive standards for fault-tolerant communication: FlexRay and OSEK-time FTCom. In *International Workshop on Engineering of Fault-Tolerant Systems (EFTS 2006), 12–13 June 2006, Luxembourg*, pages 68–79, June 2006.

[Lei08] Dirk Carsten Leinenbach. *Compiler Verification in the Context of Pervasisve System Verification*. PhD thesis, Saarland University, Computer Science Department, 2008.

[Lev95] Nancy Leveson. *Medical Devices: The Therac-25*. Addison-Wesley, 1995.

[Lie95] Jochen Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating systems principles*, pages 237–250. ACM Press, 1995.

[LP08] Dirk Leinenbach and Elena Petrova. Pervasive compiler verification – from verified programs to verified systems. In *3rd intl Workshop on Systems Software Verification (SSV08), to appear*. Elsevier Science B. V., 2008.

[LS09] Dirk Leinenbach and Thomas Santen. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *16th International Symposium on Formal Methods (FM 2009)*, Lecture Notes in Computer Science, Eindhoven, the Netherlands, 2009. Springer. Invited paper, to appear.

[LWB06] Christina Lindenberg, Kai Wirt, and Johannes Buchmann. Formal proof for the correctness of RSA-PSS. Cryptology ePrint Archive, Report 2006/011, 2006.

[MD79] E.J. McCauley and P.J. Drongowski. KSOS: The design of a secure operating system. In *AFIPS Conference Proceedings*, volume 48, pages 345–353, 1979.

[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 S. Moore. System verification. *Journal of Automated Reasoning*, 5(4):409–410, 1989.

[Moo00] J. Strother Moore. Towards a mechanically checked theory of computation: the acl2 project. pages 547–574, 2000.

[MP00] Silvia M. Müller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[MWE03] David Greve Matthew, Matthew Wilding, and W. Mark Van Eet. A separation kernel formal security policy. In *Proc. Fourth International Workshop on the ACL2 Theorem Prover and Its Applications*, 2003.

[MWE05] David Greve Matthew, Matthew Wilding, and W. Mark Van Eet. High assurance formal security policy modeling. In *Proceedings of the 17th Systems and Software Technology Conference 2005 (SSTC'05)*, 2005.

[NBFR80] P.G. Neumann, R.J. Boyer, K.N. Feiertag, and L. Robinson. A Provably Secure Operating System: The system, its applications, and proofs. Report CSL-116, Computer Science Laborartory, SRI International, Menlo Park, California, May 1980.

[NF03] Peter G. Neumann and Richard J. Feiertag. PSOS revisited. *Computer Security Applications Conference, Annual*, 2003.

[Nor98] Michael Norrish. *C Formalised in HOL*. PhD thesis, University of Cambridge, Computer Laboratory, December 1998.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[NYS07] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Using XCAP to certify realistic systems code: Machine context management. In *Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 189–206. Springer Berlin / Heidelberg, 2007.

[OSR92] Sam Owre, Natarajan Shankar, and John M. Rushby. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[PCH84] T. Perrine, J. Codd, and B. Hardy. An overview of the kernalized secure operating system (KSOS). In *Proceedings of the Seventh DoD/NBS Computer Security Initiative Conference*, volume 48, pages 146–160, Gaithersburg, Maryland, September 1984.

[Pet07] Elena Petrova. *Verification of the C0 Compiler Implementation on the Source Code Level*. PhD thesis, Saarland University, Computer Science Department, 2007.

[Pik09] PikeOS RTOS Homepage. `http://www.sysgo.com/products/pikeos-rtos-technology/`, 2009.

[RBF+89] Richard Rashid, Robert Baron, Ro Forin, David Golub, and Michael Jones. Mach: A system software kernel. In *In Proceedings of the 1989 IEEE International Conference, COMPCON*, pages 176–178. Press, 1989.

[RL77] Lawrence Robinson and Karl N. Levitt. Proof techniques for hierarchically structured programs. *Commun. ACM*, 20(4):271–283, 1977.

[RLNS75] Lawrence Robinson, Karl N. Levitt, Peter G. Neumann, and Ashok R. Saxena. On attaining reliable software for a secure operating system. In *Proceedings of the international conference on Reliable software*, pages 267–284, New York, NY, USA, 1975. ACM.

[SA93] Zhong Shao and Andrew W. Appel. Smartest recompilation. In *In ACM Symp. on Principles of Programming Languages*, pages 439–450. ACM Press, 1993.

[Sch05] Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004*, volume 3452 of *Lecture Notes in Computer Science*, pages 398–414. Springer, 2005.

[Sch06] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technical University of Munich, 2006.

[SDNM04] Jonathan Shapiro, Michael Scott Doerrie, Eric Northup, and Mark Miller. Towards a verified, general-purpose operating system kernel. In *1st NICTA Workshop on Operating System Verification*, pages 1–19, October 2004.

[Spi98] Katharina Spies. *Eine Methode zur formalen Modellierung von Betriebssystemkonzepten*. PhD thesis, Technische Universität München, 1998.

[SS06] Swaroop Sridhar and Jonathan S. Shapiro. Type inference for unboxed types and first class mutability. In *PLOS '06: Proceedings of the 3rd workshop on Programming languages and operating systems*, page 7, New York, NY, USA, 2006. ACM.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros: a fast capability system. In *In Symposium on Operating Systems Principles*, pages 170–185, 1999.

[ST08] Artem Starostin and Alexandra Tsyban. Correct microkernel primitives. In G. Klein R. Huuck and B. Schlich, editors, *3rd intl Workshop on Systems Software Verification (SSV08)*, volume 217 of *ENTCS*, pages 169–185. Elsevier Science B. V., 2008.

[Sta09] Artem Starostin. *Formal Verification of Demand Paging*. PhD thesis, Saarland University, Computer Science Department, 2009. To appear.

[SW00] Jonathan S. Shapiro and Sam Weber. Verifying the EROS confinement mechanism. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 166, Washington, DC, USA, 2000. IEEE Computer Society.

[Tew07] Hendrik Tews. Microhypervisor verfication: possible approaches and relevant properties. In *Proceedings of the NLUUG Voorjaarsconferentie 2007*, 2007.

[The99] The Common Criteria Project Sponsoring Organisations. Common Criteria for Information Technology Security Evaluation version 2.1, Part I. `http://www.commoncriteriaportal.org/public/files/ccpart1v21.pdf`, 1999.

[Tic86] Walter F. Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):273–291, 1986.

[TS08] Sergey Tverdyhev and Andrey Shadrin. Formal verification of gate-level computer systems. In K.Y. Rozier, editor, *6th NASA Langley Formal Methods Workshop*, pages 56–58. NASA Scientific and Technical Information (STI), 2008.

[Tsy09] Alexandra Tsyban. *Formal Verification of a Framework for Microkernel Programmers*. PhD thesis, Saarland University, Computer Science Department, 2009. To appear.

[Tve05] Sergey Tverdyshev. Combination of Isabelle / HOL with automatic tools. In Bernhard Gramlich, editor, *Frontiers of Combining Systems: 5th International Workshop, FroCoS 2005, Vienna Austria, September 19–21, 2005. Proceedings*, volume 3717 of *Lecture Notes in Computer Science*, pages 302–309. Springer, 2005.

[Tve09] Sergey Tverdyshev. *Formal Verification of Gate-Level Computer Systems*. PhD thesis, Saarland University, 2009.

[Ver03] The Verisoft project. `http://www.verisoft.de/`, 2003.

[WKP80] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the ucla unix security kernel. *Commun. ACM*, 23(2):118–131, 1980.

# Index