Compiler Verification in the Context of Pervasive System Verification

Dissertation

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

Dirk Carsten Leinenbach

dirkl@cs.uni-sb.de

Saarbrücken, Juni 2008

Tag des Kolloqiums:	24.06.2008
Dekan:	Prof. Dr. Joachim Weickert
Vorsitzender des Prüfungsausschusses:	Prof. Dr. Gert Smolka
1. Berichterstatter:	Prof. Dr. Wolfgang J. Paul
2. Berichterstatter:	Prof. Dr. Tobias Nipkow
3. Berichterstatter:	Prof. Dr. Andreas Podelski
akademischer Mitarbeiter:	Dr. Mark A. Hillebrand

> Science: A way of finding things out and then making them work. There is a lot more Science than you think.
> —From A Scientific Encyclopedia for the Enquiring Young Nome by Angalo de Haberdasheri.
>
> *Wings*
> Terry Pratchett

Danksagung

An dieser Stelle möchte ich all jenen danken, die zum Gelingen der vorliegenden Arbeit beigetragen haben.

Zunächst gilt mein Dank meinen Eltern, die mich stets ermutigt haben meinen Weg zu gehen und mich während meiner gesamten Ausbildung unterstützt haben.

Herrn Prof. Paul danke ich für die Möglichkeit, meine Promotion im Rahmen eines so interessanten Projektes wie Verisoft durchführen zu können, sowie für die wissenschaftliche Betreuung insbesondere im Anfangsstadium der Arbeit.

Ganz besonders danke ich meiner Freundin Sandra Schäfer, ohne deren grenzenlose Geduld und Aufmunterungen diese Arbeit nicht zustandegekommen wäre.

Mein Dank gilt auch meinen Arbeitskollegen und Freunden am Lehrstuhl Paul für die gute Arbeitsatmosphäre und so manche lebhafte Party. Ganz besonders danke ich Mark Hillebrand, dessen schier endloses Wissen über LaTeX und Perl mir oft viele Stunden Arbeit erspart hat (außerdem hat er den Divisionscode entwickelt, der in dieser Arbeit benutzt wird), sowie Tom In der Rieden, ohne den das Verisoft-Projekt sicherlich ganz anders aussehen würde und der auf verschlungenen Pfaden eine grandiose Kaffeemaschine beschafft hat.

Des Weiteren danke ich Michael Backes für die tolle Zusammenarbeit während des Studiums und die Ermunterung, mich beim Graduiertenkolleg zu bewerben, sowie Norbert Schirmer, der so manche Stunde in die erfolgreiche Verbindung von Small-Step und Big-Step Semantik von *C*0 investiert hat.

Für ihre Hilfe bei der Verifikation der Codegenerierung für Ausdrucksauswertung danke ich außerdem Hristo Pentchev, Verena Kremer, und Kara Abdul Qadar.

Nicht zuletzt danke ich Sabine Nermerich, der guten Seele im Verisoft Projekt, die es schon so lange mit uns seltsamen Informatikern aushält.

This thesis work was funded by DFG Graduiertenkolleg “Leistungsgarantien für Rechnersysteme” and by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

Abstract

This thesis presents the formal verification of the compiling specification for a simple, non-optimizing compiler from the C-like programming language $C0$ to VAMP assembly code. The main result is a step-by-step simulation theorem between $C0$ programs and the compiled code (which is specified by the compiling specification). Additionally, a $C0$ small-step semantics and a verification methodology for VAMP assembly have been developed.

This work is part of the Verisoft project which aims at the pervasive formal verification of an entire computer system. The key concept in Verisoft's methodology is to prove properties of computer systems at the relatively abstract $C0$ layer and to transfer them via several intermediate layers down to the concrete hardware layer. After successful transfer of a property to the hardware layer, we can be sure that no oversimplifications have been done in the formalizations of the more abstract layers.

This context of pervasive system verification imposes several special requirements to our compiler correctness theorem. In particular, the simulation theorem had to be formulated based on small-step semantics to allow for reasoning about non-terminating and interleaving programs. Another important feature is that our result incorporates resource restrictions at the hardware layer and allows to discharge them at the $C0$ layer.

All results presented in this thesis have been formalized in the theorem prover Isabelle / HOL.

Kurzzusammenfassung

Die vorliegende Arbeit befasst sich mit der formalen Verifikation des Codegenerierungsalgorithmus eines nicht optimierenden Compilers von der C-ähnlichen Sprache $C0$ nach VAMP Assembler. Das Hauptergebnis ist ein Schritt-für-Schritt Simulationssatz zwischen $C0$ Programmen und dem compilierten Code. Die Arbeit umfasst zusätzlich die Entwicklung einer Small-Step Semantik für $C0$ sowie einer Verifikationsmethodik für VAMP Assemblerprogramme.

Diese Arbeit ist Teil des Verisoft Projekts, das auf die durchgängige formale Verifikation von Computersystemen abzielt. Die Methodik von Verisoft basiert auf der Verifikation von Eigenschaften eines Computersystems auf der relativ abstrakten $C0$ Ebene und deren anschließendem Transfer auf die konkrete Hardwareebene. Ein solcher erfolgreicher Eigenschaftstransfer garantiert, dass auf den abstrakten Ebenen keine zu starken Vereinfachungen vorgenommen worden sind.

Die Einbettung in die durchgängige Verifikation von Systemen stellt zahlreiche speziellen Anforderungen an den Compilerkorrektheitssatz. Insbesondere muss der Simulationssatz auf einer Small-Step Semantik basieren, um die Behandlung von nebenläufigen und von nicht terminierenden Programmen zu ermöglichen. Eine weitere Eigenschaft ist, dass unser Resultat Ressourcenbeschränkungen auf der Hardwareebene einbezieht und deren Entlastung auf der $C0$ Ebene erlaubt.

Alle Resultate dieser Arbeit sind im Theorembeweiser Isabelle / HOL formalisiert worden.

Extended Abstract

This thesis presents the formal verification of the compiling specification (or code generation algorithm) of a simple, non-optimizing compiler from the C-like programming language $C0$ to the assembly language of the formally verified VAMP processor. It covers the proof of a step-by-step simulation theorem between a $C0$ program and the compiled code as specified by the compiling specification; that is, it proves the correctness of the code generation algorithm. This result complements the verification of a $C0$ implementation of the code generation algorithm which has been done by E. Petrova.

As basis for the compiler verification, we have developed a $C0$ small-step semantics and proved several important invariants about its computations. On top of the semantics of the VAMP assembly language (which has been defined by Tsyban et al.) we have developed a verification methodology for the verification of VAMP assembly programs. This verification methodology has been used for the verification of assembly templates of the compiling specification.

The main result of our work is the formal proof of a step-by-step simulation theorem between execution of programs in the $C0$ small-step semantics and the execution of the compiled code in the VAMP assembly machine. This theorem states for every step of the $C0$ machine executing a program p that there exists a corresponding state in the computation of the VAMP assembly machine executing the compiled code of p. The meaning of correspondence in this theorem is mainly that the values of reachable variables of the $C0$ program are properly represented in the assembly machine.

This thesis is part of the Verisoft project which aims at the pervasive formal verification of entire computer systems. The goal of pervasive system verification is to have a *single* top-level correctness theorem for the complete system without abstracting from restrictions or properties of the underlying hardware or assembly language. The lower the language layer at which we formulate such an overall theorem is the better and more reliable is the result. One of the key concepts in Verisoft's methodology is to prove properties at a relatively abstract layer (e.g., the $C0$ layer) and to transfer them using meta theorems down to more concrete layers (e.g., an assembler language or even the processor hardware). After successful transfer of a property, we can be sure that no oversimplifications have been done in the formalization of the more abstract layers. However, abstractions of higher layers have to be justified during property transfer. This can sometimes be done once and for all in a meta theorem. Other justifications depend on the program being compiled.

The compiler correctness theorem presented in this thesis is such a meta theorem. To enable property transfer from the $C0$ to the assembly layer, it has to meet special requirements. In particular, we have verified a *complete* compilation chain from the $C0$ language down to assembly code instead of focusing on just a few parts of the compiler (like optimization patterns). Furthermore, the simulation theorem is formulated based on small-step semantics to allow for reasoning about non-terminating and interleaving programs. Additionally, the compiler correctness proof considers resource restrictions at the hardware layer and allows to discharge them at the $C0$ layer.

All results presented in this thesis have been formalized in the theorem prover Isabelle / HOL.

Zusammenfassung

Die vorliegende Arbeit befasst sich mit der formalen Verifikation des Codegenerierungsalgorithmus eines einfachen, nicht optimierenden Compilers aus der C-ähnlichen Programmiersprache $C0$ in die Assemblersprache des formal verifizierten VAMP Prozessors. Die Arbeit beinhaltet den Beweis eines Schritt-für-Schritt Simulationssatzes zwischen einem $C0$ Programm und dessen kompilierten Codes. Dieses Ergebnis vervollständigt die von E. Petrova durchgeführte Verifikation einer $C0$ Implementierung des Codegenerierungsalgorithmusses.

Als Basis für die Verifikation des Compilers haben wir eine $C0$ Small-Step Semantik entwickelt und mehrere wichtige Invarianten über deren Berechnungen bewiesen. Basierend auf der Semantik der VAMP Assemblersprache, die von A. Tsyban definiert wurde, haben wir eine Verifikationsmethodik für VAMP Assemblerprogramme entwickelt. Diese haben wir bei der Verifikation von kurzen Assemblercodestücken angewendet, die der Compiler als Vorlagen benutzt.

Das Hauptergebnis dieser Arbeit stellt der formale Beweis eines Schritt-für-Schritt Simulationssatzes zwischen der Ausführung von Programmen in der $C0$ Small-Step Semantik und der Ausführung des kompilierten Codes in der VAMP Assemblermaschine dar. Dieser Simulationssatz zeigt für jeden Schritt während der Ausführung eines Programms p in der $C0$ Maschine, dass es einen korrespondierenden Zustand in der Berechnung der VAMP Assemblermaschine gibt, die den kompilierten Code von Programm p ausführt. Korrespondierend bedeutet hierbei hauptsächlich, dass die Werte von erreichbaren Variablen der $C0$ Maschine im Zustand der entsprechenden Assemblermaschine kodiert sind.

Die vorliegende Arbeit ist Teil des Verisoft Projektes, das die durchgängige formale Verifikation ganzer Computersystem anstrebt. Das Ziel von durchgängiger Verifikation ist der Beweis eines einzigen Korrektheitssatzes für das komplette System, ohne von Einschränkungen oder Eigenschaften der zugrunde liegenden Schichten zu abstrahieren. Je tiefer die Sprachebene ist, auf der solch ein Gesamtkorrektheitssatz bewiesen wird, desto vertrauenswürdiger ist das Ergebnis. In Verisoft versucht man, Eigenschaften auf einer möglichst abstrakten Ebene (z.B. auf der $C0$ Ebene) zu verifizieren und die Ergebnisse danach mittels Meta-Theoremen auf konkretere Schichten (z.B. Assemblersprache oder sogar die Prozessorhardware) zu transferieren. Nachdem dies für eine Eigenschaft gelungen ist, kann man sicher sein, dass keine zu starke Vereinfachung in der Formalisierung der abstrakten Ebene gemacht worden ist. Trotzdem müssen Abstraktionen der höheren Ebenen beim Transfer von Eigenschaften gerechtfertigt werden. Für einige Abstraktionen kann dies ein für alle Mal im Rahmen des Meta-Theorems gezeigt werden; für andere hängt es vom kompilierten Programm ab.

Der Compilerkorrektheitssatz der vorliegenden Arbeit ist solch ein Meta-Theorem. Um den Eigenschaftstransfer von der $C0$ auf die Assemblerebene zu erlauben, muss der Satz spezielle Anforderungen erfüllen. Insbesondere umfasst er einen kompletten Übersetzer von $C0$ nach Assemblercode und konzentriert sich nicht bloß auf einige Teile (wie einzelne Optimierungsstufen). Weiterhin ist der Simulationssatz auf Basis einer Small-Step Semantik formuliert, um Aussagen über nicht terminierende oder nebenläufige Programme zu ermöglichen. Der Korrektheitssatz berücksichtigt ferner auch Ressourcenbeschränkungen auf der Hardwareebene und erlaubt deren Entlastung auf der $C0$ Ebene.

Alle Resultate dieser Arbeit wurden in Isabelle / HOL formalisiert.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

Nowadays, computer systems are used more and more in security and safety critical applications. They are ubiquitous not only in the obvious places, i.e., personal computers which we use for on-line banking or confidential communication, but also hidden, e.g., in machine controls, automobiles, or aircrafts. Computer errors in these application areas can cause tremendous costs, both in money and human life.

Developers and engineers try to reduce the risk of failure in many ways. They use special development and controlling principles [The99], use redundant systems (both for hardware and software), do extensive and systematic tests, and – especially in highly critical systems – prove the correctness of hardware and software both using traditional paper-and-pencil mathematics and formal methods (where 'formal methods' means that computer systems check or completely conclude correctness proofs).

However, even the formal verification of single parts of a computer system does *not* ensure the correctness of the overall system. For example the failure of the Ariane 5 maiden flight 1996 was caused by the Inertial Reference System (SRI) which is responsible for the position monitoring system of the rocket. About 40 seconds after take-off, the SRI software produced an uncaught exception while trying to convert a 64-bit floating point number to a 16-bit integer; the software component was reused from Ariane 4 where metered values have been in a smaller domain. However, "the supplier of the SRI followed the specifications given to it, which stipulated that in the event of any detected exception the processor was to be stopped." [Boa96] The backup system which was activated automatically after shutting down the first SRI produced the same error within micro seconds and fate took its course.

The value whose computation generated the exception was for Ariane 5 only needed during take-off whereas for Ariane 4 it was also needed later; thus, there was no reason for the component in Ariane 5 to be still active 40 seconds after take-off. Additionally, because of a different trajectory of Ariane 5 compared

with Ariane 4, the problem did only occur for the new model. What can we learn from this disaster?

1. Redundant hardware system are a good idea in general, but they do not help against software failures or design errors.

2. Even the correctness of all components of a system does not imply the correctness of the overall system if the specifications do not fit together or external influences are not considered correctly.

3. We can only trust in the correctness of specifications of a component or even a complete system if it allows us to prove specific top-level properties of the system.

In addition to these qualitative arguments, King [KHCP00] stresses the fact that formal verification does often require less effort than normal testing (given that good and highly-automated tools are available). Beyer et al. [BBM+07] come to similar conclusions: they report that normal testing of a given complex processor has required twice the effort than the formal verification of a similar design while it did not deliver as high a quality.

The Verisoft project [Ver03] worked on eliminating the above-mentioned three problems by pervasively verifying the correctness of entire computer systems, including gate level hardware [BJK+06], system software [GHLP05], compiler [LPP05, LP08], and communicating applications [BGH+06], with a special focus on industrial applications [IK05]. In particular, the seamless verification of large parts of the *academic system* has been attempted and was successfully finished. This system consists of hardware (processor and devices) on top of which runs a micro kernel, a simple operating system, and applications. The software of the academic system is implemented in the C-like language $C0$ together with few lines of inline assembly code.

In the context of pervasive verification, it is not sufficient to prove application correctness only at the source level. A faulty compiler may introduce bugs into the application during translation to machine language. Additionally, trustworthy execution of programs on a concrete processor with restricted resources (memory etc.) imposes additional requirements which are often abstracted in programming language semantics. Thus, the verification of a $C0$ compiler including treatment of restricted resources is a central part of the Verisoft project.

This thesis presents a formal small-step semantics of the $C0$ language and a formal correctness proof of the code generation algorithm (also called compiling specification) of a simple, non-optimizing $C0$ compiler. These formalizations of the $C0$ semantics, the compiling specification, and the compiler correctness proof have been inspired by paper-and-pencil definitions and a compiler proof sketch which W. J. Paul has given in two internal technical reports of the Verisoft project [LP04, BKLP04].

To integrate the work presented in this thesis smoothly into the rest of the Verisoft project we had to meet several special requirements.

- In multitasking systems like the academic system, programs do not run in isolation: execution of (many) user programs and of the operating systems is *interleaved*. In big-step semantics computations cannot be interleaved in a simple way as it is hard to argue about intermediate

states. Additionally, non-terminating programs have to be considered; this is also non-trivial in big-step semantics. Thus, big-step semantics (and classical Hoare logics built on top of them) *alone* do not suffice to conveniently model interleaving programs. Therefore, we model source language semantics by small-step semantics. Of course this does not entirely preclude the use of Hoare logics for pervasive verification.

Indeed both a big-step semantics and classical Hoare logics for $C0$ were formally defined [Sch05, Sch06]. Program analysis in Hoare logics necessarily concerns the execution of terminating portions of programs. In order to handle interleaving, these results then have to be 'imported' into the small-step semantics. This import is based on classical results relating Hoare logics and small-step semantics in the style of Theorem 6.16 in [NN99]. In a pervasive verification effort these results have to be shown formally; this has been done in the Verisoft project [ASS08].

- The compiler should make it possible to discharge most of the low-level resource restrictions already on the $C0$ level. This allows to show the *total* correctness of a particular $C0$ program on an abstract level without arguing about the compiled assembly code.

- To solve the bootstrap problem [GH98] it is not sufficient to verify the code generation algorithm. E. Petrova has verified a $C0$ implementation of the $C0$ compiler [Pet07] using a $C0$ verification environment [Sch06] for Isabelle / HOL [NPW02]. Her results are briefly sketched in Chapter 11. There are several ways to get a trustworthy executable from the compiler implementation in $C0$. One of them – using Isabelle's built-in code generation feature – is described in more detail in [Tzi07].

- A big effort went into the integration of this work into the large project. Harmonization of the big-step and small-step semantics was surprisingly hard, but had to be done in order to allow formal equivalence proofs. The compiler simulation theorem had to be enriched with several special properties to make it useful for the correctness proofs of the operating system kernel. Additional properties, e.g. the absence of self-modifying code, are needed to allow for the transfer of results to the hardware layer.

1.1 Outline

In the last section of this first chapter we introduce some basic notation. A detailed requirements analysis and related work are discussed in Chapter 2. The remainder of the thesis is divided into two parts.

Part I defines the syntax and semantics of the source and target language of the compiler.

- In Chapter 3 we introduce the C-like programming language $C0$. In particular, we introduce a preprocessor tool which translates $C0$ source code into different representations (e.g., for the formal small-step semantics in Isabelle / HOL) and provides a front end for the verified compiler implementation. Additionally, we sketch the language stack which is used in the Verisoft project.

- In Chapter 4 we formally define the *C*0 small-step semantics. We start with a definition of the data types which represent *C*0 programs in Isabelle and continue with definitions of the expression evaluation and of the transition function.

- In Chapter 5 we introduce several properties and invariants of the *C*0 small-step semantics, which are later needed for compiler correct proof. In particular, we define the set of *valid* *C*0 programs, introduce the notion of type correctness and show that the *C*0 expression evaluation is type correct. Finally, we define an invariant about the structure of program rests and prove that it is preserved by the *C*0 transition function (this theorem is called 'folklore theorem' in [LPP05]).

- In Chapter 6 we describe the instruction set architecture and assembly language of the verified architecture microprocessor (VAMP) which is the target architecture of the verified *C*0 compiler. Amongst others, we list requirements which allow to prove that the VAMP ISA simulates the VAMP assembly language and introduce a verification methodology for VAMP assembly programs. This verification methodology is used to prove the low-level correctness of the generated assembly code.

Part II handles the *C*0 compiler itself and presents a correctness proof.

- In Chapter 7 we define the code generation algorithm of the *C*0 compiler. Additionally, we investigate restrictions on input programs which are caused by restrictions of the target machine and define the set of *translatable* *C*0 programs. We conclude this chapter with a discussion on how the Isabelle specification of the code generation algorithm can be executed; this is a possible solution of the bootstrap problem.

- In Chapter 8 we present the simulation theorem which states the correctness of the *compiling specification* of the *C*0 compiler. This includes defining the simulation relation for the compiler correctness theorem, the introduction of reachability and allocation functions, and, finally, the formalization of the main correctness theorem for the compiling specification and a sketch of its correctness proof.

- In Chapter 9 we sketch the proof that the code generation algorithm for expressions produces correct code.

- In Chapter 10 we prove the correct compilation of statements. We start with a theorem which allows us to separate the correctness of control flow instructions from the correctness proofs for individual statements. In the remaining sections of the chapter we prove theorems about the code generation for the different *C*0 statements. This proves the different induction cases of the proof in Chapter 8.

- In Chapter 11 we briefly sketch Petrova's top-level correctness theorem for the *C*0 implementation of our compiler (cf. [Pet07]).

We conclude in Chapter 12 with summary and future work.

1.2 Notation

In this section we introduce some basic notation which is used in the remainder of this thesis.

1.2.1 Basics

We denote by $\mathbb{S}$ the set of identifiers (e.g., variable or type names), by $\mathbb{Z}$ the set of integers, by $\mathbb{N}$ the set of natural numbers including zero, and by $\{a, \ldots, b\}$ the integer interval from a to b (inclusive). By $\mathbb{Z}_w$ and $\mathbb{N}_w$, we denote the set of integers and natural numbers which fit into w bits; that is, $\mathbb{Z}_w = \{-2^{w-1}, \ldots, 2^{w-1} - 1\}$ and $\mathbb{N}_w = \{0, \ldots, 2^w - 1\}$. For natural numbers n and $d > 0$ we use the notation $\lceil n \rceil_d = \lceil n/d \rceil \cdot d$. This is the smallest multiple m of d such that m is greater or equal than n.

We use $a \div b$ to denote integer division. Unfortunately, the semantics of modulo computation for integers differs between Isabelle / HOL and the standard gcc C-compiler (observe, that the C standard from [ISO99] does not require a specific behavior). Thus, we define our own version of a signed modulo operator $a \bmod_s b = ((a + b) \mod (2 \cdot b)) - b$ which is used to define the modulo semantics of binary operators (cf. Section 4.3.3). Unsigned modulo $a \bmod_u b = a \mod b$ is computed using the usual mathematical definition. The predicate $a \mid b$ checks whether a divides b.

We use the notation $a +_{32} b = a + b \bmod_u 2^{32}$ and $a -_{32} b = a - b \bmod_u 2^{32}$ to denote bounded addition and subtraction, respectively. Functions are anonymously introduced using lambda expressions. For example, $\lambda\, x, y.\ (x + y)$ is a function which adds its two operands.

We use the notation $r.c$ to access record component c of a record r. Update of records is written as $r[c_1 := 5, c_2 := 6, \ldots, c_n := 42]$ where r is the original record and c_1 to c_n are its components; record components which are not mentioned in the update remain unchanged. Records may also be composed on the fly writing $[c_1 = 5, c_2 = 6, \ldots, c_n = 42]$.

The ternary operator, ? :, is used for case distinctions in formulas. We write $c?e_1 : e_2$ to denote an expression which evaluates to e_1 if the condition is fulfilled, i.e., $c = \mathit{true}$, and to e_2 otherwise.

Predicates are functions from a given domain into the set $\mathbb{B}$ of boolean values. We say that a predicate p holds or is fulfilled for some value x if $p(x) = \mathit{true}$. In formulas we often abbreviate $p(x) = \mathit{true}$ with $p(x)$ and $p(x) = \mathit{false}$ with $\neg p(x)$.

We use inference rules of the form

$$\frac{a \qquad b}{c}$$

to denote logical implication $(a \wedge b) \Longrightarrow c$. Inference rules are often used for definitions of predicates by case distinction or pattern matching, giving one inference rule for every case. Observe, that everything which is not covered explicitly by rules does by default *not* fulfill the predicate.

Observe also, that we often omit quantifiers like $\forall$ and $\exists$ for free variables

in definitions and lemmas to keep formulas concise. For example, the definition

$$\frac{a \cdot i = b \qquad i \in \mathbb{Z}}{a \mid b}$$

should be read as: a divides b if there *exists* an integer i s.t. $a \cdot i = b$.

We use $fst :: (t_1, t_2) \mapsto t_1$ and $snd :: (t_1, t_2) \mapsto t_2$ to extract the first and second element of a pair. The function $flip :: (t_1, t_2) \mapsto (t_2, t_1)$ exchanges the elements of a pair: $flip(a, b) = (b, a)$.

In general we model finite sequences of elements from set A as mappings $s : [0 : n-1] \mapsto A$. For natural numbers a, b with $0 \leq a \leq b \leq n-1$ we denote by $s[a : b]$ the subsequence $(s[a], \ldots, s[b])$. Formally, this is the sequence $s' : [0 : b - a + 1] \mapsto A$ with $s'[i] = s[a + i]$ for all i. We denote by $s[a, l]$ the subsequence $s[a : a + l - 1]$ of s of length l starting at index a. Note that we will sometimes just write s for the only element of a singleton sequence instead of $s(0)$. A constant sequence of elements a, b, c, and d may be written in the form $[a, b, c, d]$; correspondingly, $[a]$ denotes a sequence with only one element. Updating a sequence s starting a position a with l values from sequence s' is denoted by $s\,([a, l] := s')$ and defined as follows:

$$s\,([a, l] := s')\,[i] = \begin{cases} s'[i - a] & \text{if } a \leq i < a + l \\ s[i] & \text{otherwise} \end{cases}$$

We denote intervals of natural numbers as *ranges* in this thesis. A range $(a : b)$ is represented by start and end address and contains all numbers i with $a \leq i \leq b$. We define several predicates on ranges. We use the notation $(i \in (a, b)) = (a \leq i \leq b)$ to denote that a given number i is *inside* a range $(a : b)$. We denote by $b \subseteq a$ that range b is *completely contained* in range a.

Sometimes we denote ranges also by pairs of start address and length. Such ranges a and b are said to be *disjoint* iff

$$a \asymp b = (fst(a) \geq fst(b) + snd(b) \vee fst(b) \geq fst(a) + snd(a))\,;$$

they overlap iff

$$a \not\asymp b = (fst(a) < fst(b) + snd(b) \wedge fst(b) < fst(a) + snd(a))\,.$$

1.2.2 Abstract Data Types

In this section we introduce the concept of abstract data types. They are a way to formalize terms which are built by a set of so-called *constructors*. The set of terms which can be built for an abstract data type t is identified with the *type* t. Each constructor is a function with an arbitrary number of parameters and image t. We demonstrate the concept by an abstract data type for lists whose elements are of type t.

$$\begin{aligned} t\ list\ &= nil \\ &\mid cons(t, t\ list) \end{aligned}$$

Thus, lists have constructors $nil :: list$ and $cons :: \mathbb{Z} \times list \mapsto list$. A list is either the empty list nil or concatenation of a single element and a list. Recognizers

are predicates over elements of a given data type which check whether the element is built by a specific constructor. For example, the recognizer

$$is\_cons(l) = \begin{cases} true & \text{if } \exists h, t : l = cons(h, t) \\ false & \text{otherwise} \end{cases}$$

checks whether l is built using the *cons* constructor. Observe, that we often omit the existential quantifiers in such formulas, when the meaning is clear from the context.

Isabelle / HOL only supports total functions. Partial functions can be simulated by using an uninterpreted constant for undefined values. In this thesis we use '*undef*' as such a constant.

Another way to simulate partial functions is the so-called *option type*. The option type is a polymorphic data type with two constructors. For type t it is defined by

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

The constructor *None* models an undefined value and $Some(x)$ a defined value x. We use the abbreviations $t_\perp$ for $t\ option$ and $\lfloor x \rfloor$ for $Some(x)$. The function $the :: t\ option \mapsto t$ converts option types back into the base type.

$$the(y) = \begin{cases} x & \text{if } y = \lfloor x \rfloor \\ undef & \text{if } y = None \end{cases}$$

1.2.3 Lists

In this section we introduce some basic notation and functions regarding lists.

In addition to the definition of the list data type from above we use the following pretty printing for the constructors. We denote the empty list *nil* by $[\,]$, and $cons(h, t)$ by $h\#t$. Additionally, we use the abbreviation $[a, b, c] = a\#b\#c\#[\,]$ to denote concrete instances of list.

Inductive definitions on data types can be easily done via pattern matching. As example we define the function $map :: (t_1 \mapsto t_2) \times t_1\ list \mapsto t_2\ list$ which maps a function $f :: t_1 \mapsto t_2$ on a list of type $t_1\ list$.

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

We define the length of lists by the following notation.

$$\begin{aligned} |[\,]| &= 0 \\ |h\#t| &= |t| + 1 \end{aligned}$$

Functions $hd :: t\ list \mapsto t$ and $tl :: t\ list \mapsto t\ list$ compute the head and tail of lists, respectively.

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

Similarly, *last* returns the last element of a list and *butlast* all elements *but* the last one. Function $l[i, n]$ returns the n elements of list l starting at the i-th element.

Concatenation of two lists l_1 and l_2 is defined by induction on the first list.

$$[\,] \circ l_2 = l_2$$
$$(h\#t) \circ l_2 = h\#(t \circ l_2)$$

We overload the notation $[l_1, l_2, \ldots, l_n]$ to denote concatenation of lists l_1, l_2, up to l_n.

We turn lists l into sets by just writing $\{l\}$. With $l!n$ we denote the n-th element of list n; if $n \geq |l|$, the result is undefined. Updating the n-th element of a list with some value x is denoted by $l\,[n := x]$.

We reuse the notation $x \in l$ to denote that x is element of list l and $x \notin l$ to denote that x is not an element of l.The predicate $\mathit{distinct}(l)$ checks whether the elements of list l are pairwise distinct. The function $\mathit{replicate}(n, x)$ generates a list which consists of n copies of element x.

Finally, we define two additional functions on lists. The first one is $\mathit{map\text{-}of} :: (t_1, t_2)\ \mathit{list} \times t_1 \mapsto t_{2\,\bot}$. This function takes a list l of pairs and finds the first element whose first component equals the second parameter x; it returns the second component of this pair. If no such element exists, the function returns *None*. Formally, we define the function by list induction.

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

The second function is $\max :: \mathbb{N}\ \mathit{list} \mapsto \mathbb{N}$ which finds the maximum element in a given list of natural numbers.

1.2.4 Bits and Bit Vectors

We use standard notation for bits $a, b :: \mathit{bit}$ and bit vectors (bit lists) $x, y :: bv_n$ of length n. We denote with x_i the i-th bit of bit vector x and with $x[15 : 8]$ bits 15 to 8. The notation b^l produces a bit vector of length l consisting only of bits b.

Bit vectors can be interpreted as natural numbers by $\langle x \rangle = \sum_{i=0}^{n-1} x_i \cdot 2^i$ and as integers by $[x] = -x_{n-1} \cdot 2^{n-1} + \langle x[n-2:0] \rangle$. For conversion in the other direction we define functions $\mathit{bin}_n(j)$ and $\mathit{two}_n(i)$ which compute the n-bit binary and two's complement representation of j and i, respectively. For details see also [MP00].

The highest bit of a two's complement number is also called *sign bit*. Sign extension $\mathit{sext}_n(x)$ extends a bit vector x to length n by cloning the sign bit. A simple computation shows that this does not alter the integer value of the bit vector.

We use the basic operations on single bits: negation ($\neg_\mathrm{b}$), or ($\vee_\mathrm{b}$), and ($\wedge_\mathrm{b}$), and exclusive or ($\otimes_\mathrm{b}$). Observe that these operations are also used for bit vectors; there, they are applied to each bit.

We use two kinds of shift operations on bit vectors. Logical left shift $\ll_\mathrm{l}$ and logical right shift $\gg_\mathrm{l}$ pad with zeroes; arithmetic right shifts $\gg_\mathrm{a}$ pad with the sign bit of the original bit vector.

1.2.5 Miscellaneous

In the remainder of this thesis we will sometimes refer to details of the corresponding formalization in Isabelle / HOL. These remarks are denoted by the keyword Isabelle. More general remarks are marked by Remark.

Chapter 2

Requirements Analysis and Related Work

Programming language semantics and compiler verification are active fields of research and a lot of impressive results have been achieved. However, the context of pervasive system verification imposes many special requirements. Without meeting *all* of them, our results would not be suitable to prove the desired overall correctness theorems for whole computer systems.

We start this chapter in Section 2.1 with a detailed discussion of the special requirements. Then, we present in Section 2.2 the most relevant related work and check whether their results meet our needs.

2.1 Requirements Analysis

The goal of pervasive system verification is to have a *single* top-level correctness theorem for complete systems without abstracting from restrictions or properties of the underlying hardware or assembly language. In the following, we will elaborate on the requirements imposed by this kind of pervasiveness.

2.1.1 Language Layers

The lower the implementation layer at which we formulate an overall correctness theorem is the better and more reliable is the result. For example, it is better to prove the correctness of some program by showing that its binary computes the correct result when executed at the hardware layer than just showing that the result at the C layer is correct; in particular, the latter does not ensure that the program has been correctly compiled. There are basically two ways how to obtain such an overall theorem at lower language layers.

The naive approach is to verify the top-level theorem directly arguing at the lowest considered language layer, e.g., directly at the level of an instruction set architecture or assembly language or even directly at the hardware layer. Obviously, this approach does not scale very well for non-trivial applications because we have to consider all (possibly irrelevant) details of the architecture in every single proof step and because assembly implementations of programs tend to hide or obscure a lot of useful structure which is present in implementations using high-level languages.

The better approach is to introduce additional and more abstract language layers. This increases the level of abstraction and by that also productivity.

However, to achieve the ultimate goal of having a correctness theorem at the lowest layer of abstraction, we have to transfer results from the higher layers to the lower layers. In Verisoft, this is handled using meta theorems which allow to transfer individual correctness results all the way down to the lowest layer and combine them into a single overall system correctness theorem.

The lower the layer at which we argue is the more restrictions and side conditions have to be considered. The meta theorems justify the abstractions on the different layers and ensure that no oversimplifications have been done.

Obviously, a high-level programming language which is supposed to be useful in pervasive verification needs to be expressive enough to handle the bulk of all implementations while at the same time remaining 'neat' to allow for efficient formal verification of medium sized applications. Simultaneously, in order to prevent the results of such effort from being academic toys, the stack needs to be based on some realistic target architecture.

Figure 2.1 depicts the semantics stack used in the Verisoft project. The lowest layer represents the gate-level hardware of the verified VAMP processor [BJK+06]. On top of it we have the VAMP instruction set architecture (ISA) and its assembly language. In the upper half of the figure we have three different $C0$ semantics (cf. below for more details). Observe that the semantics stack is orthogonal to the stack of system components (hardware with devices, micro kernel, operating system, ...) in the Verisoft project.

The justification of the $C0$ language layers on top of the assembly layer is the compiler correctness theorem presented in this thesis. This meta theorem allows to transfer properties of concrete $C0$ programs down to the assembly layer. There are also meta theorems which relate all other layers of the semantics stack. Exemplary, this is detailed in [ASS08]; additional details about the simulation theorems between the $C0$ Hoare logic and big-step semantics are given in [Sch06].

In order to prove the correctness of programs which invoke – directly or indirectly – inline assembly code using the $C0$ Hoare logic or the big-step semantics we need a way to formalize the effects of the inline assembly parts. Besides performance optimizations, inline assembly is usually used when the intended effect of a piece of code is not expressible in the pure $C0$ state but changes machine components which are not visible in $C0$. Nevertheless, we want to use the Hoare logic verification environment also for programs with inline assembly. Thus, the $C0$ state in the Hoare logic has been extended by an *extended* state component for the essential parts of the environment and so-called *XCalls* have been introduced which model the inline assembly parts [ASS08].

XCalls can arbitrarily change the additional state component and have limited access to the Hoare logic state. In the $C0$ semantics, the effect of XCalls is defined axiomatically. Their implementation, usually containing inline assembly, can be plugged in at the level of the assembly semantics. Here, the axiomatic specification which is used in the higher layers has to be justified.

2.1.2 Small-Step Semantics and Step-By-Step Simulation Theorem

An important decision regarding the pervasive verification of systems is whether high-level languages should be modeled using big-step or small-step semantics.

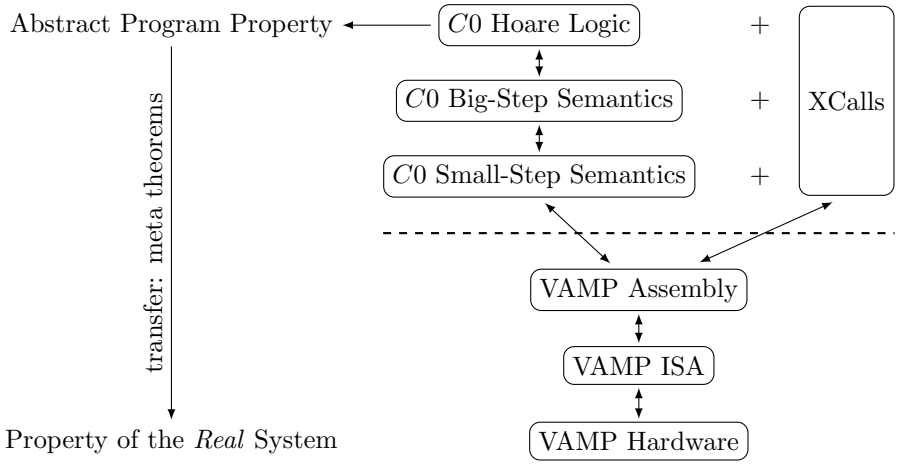


Figure 2.1: Semantics Layers in Verisoft

At first glance, big-step semantics seems to be adequate for the scenario in the Verisoft project: $C0$ is a sequential language and the target system consists only of a *single* processor, i.e., there is no *real* parallel execution of several programs on the hardware. Big-step semantics are more abstract (e.g., they do not need to model a stack explicitly) and they allow proofs by rule induction on the depth of the syntax tree of the program being verified. Both properties speed up verification of concrete programs and make the use of big-step semantics desirable to achieve optimal verification productivity.

However, the processor is not the only hardware unit present. Devices like hard disk, timer, network interface, and serial interface execute concurrently and may interrupt execution of user programs. Additionally, execution of $C0$ applications may be interrupted when they communicate with an operating system or a kernel via trap interrupts (these are activated using inline assembly code). A good example is inter process communication (IPC) which is implemented via special kernel calls. In big-step semantics computations cannot be interleaved in a simple way as it is hard to argue about intermediate states.

Moreover, big-step semantics do not distinguish between non-terminating and stuck computations in a natural way. This limits the use of big-step semantics for the verification of non-terminating programs like operating system kernels or user applications implementing server processes. In contrast, small-step semantics easily handle proofs about non-terminating computations.

However, a simulation theorem for small-step semantics also needs to be formulated in a small-step manner relating *every* state of the high-level machine to some state of the target machine (step-by-step simulation). A big-step style simulation theorem would only argue about the final state of terminating computations and would render the use of small-step semantics unnecessary.

Thus, big-step semantics (and classical Hoare logics build on top of them) *alone* do not suffice to conveniently model interleaving or non-terminating programs. The use of small-step semantics is appropriate. However, it is still possible (and reasonable) to use big-step semantics and Hoare logics for the verification of uninterrupted and terminating parts of programs. These partial

results can then be transfered to (via the meta theorems) and combined at the small-step layer.

In the same way, the combination of pure $C0$ parts and inline assembly can be handled. The step-by-step simulation theorem allows for switching from $C0$ to assembly layer when entering inline assembly parts and switching back to a consistent $C0$ small-step state when the inline assembly portion has been executed (some more details regarding the verification of inline assembly code in Verisoft are given on page 66 of this thesis and in [ASS08]).

In a similar way as combining the inline assembly portions with the $C0$ parts of programs, small-step semantics are needed to combine verified parts of both applications and system software. Execution of these parts starts and ends in *intermediate* states. Thus, small-step semantics and a small-step style compiler correctness theorem are the best choice in handling this kind of arguments. Examples are the combination of application programs issuing system calls and the isolated correctness statements about the execution of the corresponding parts of the operating system kernel [IT08], the integration of driver correctness into the micro kernel [HIP05, ASS08], and the concatenation of *single* executions of the main kernel loop to argue about the correctness of this kind of non-terminating code [GHLP05].

2.1.3 Implementation Correctness and Bootstrapping

For pervasive verification, it is not sufficient to have a verified compiling specification: we additionally need to have a trustworthy executable version of the compiler (cf. [Moo88, CM86]). The question how to obtain such an executable is known as the *bootstrap* problem and was stressed in [GH98].

One simple way to solve this problem is to have an executable compiling specification (as the one presented in Chapter 7). However, as the runtime system (e.g., the underlying hardware, a ML interpreter, or Isabelle's code generation package [Ber03]) is not completely reliable, simply executing the compiling specification still leaves a certain probability for errors in the generated code.

Another solution is to generate – once and for all – a trustworthy binary of the verified compiler and to execute this binary on the pervasively verified computer system. This requires us to have – in addition to the compiling specification – an implementation of the compiler in its own source language. Not only is it more comfortable to have an executable binary instead of having to apply the runtime system for the executable compiling specification for each compilation, it also decreases the probability of errors because only a single program (the compiler) is compiled by the (not completely trusted) execution of the compiling specification.

2.1.4 Property Transfer

The lower the language layer at which we formulate correctness statements of a system the more additional side conditions show up. To successfully transfer correctness statements from the more abstract to the more concrete layers these conditions need to be discharged. Depending on the kind of conditions, this can be done in several ways.

Some conditions can be discharged once and for all as part of the meta theorem between the layers concerned. For a compiler from $C0$ to VAMP assembly code, this set of conditions comprises the correct alignment of data and instructions, the absence of most kinds of interrupts, the immutability of special purpose registers and of memory outside the application's address range, and the guarantee that no illegal instructions or self modifying code are generated.

Other conditions require additional proof effort for each individual program. However, a proper formulation of the meta theorems can greatly simplify the verification of such side conditions. This set of conditions comprises resource restrictions on the target machine (e.g., the bounded memory size for heap or stack) and real-time requirements as in Verisoft's automotive sub project [KP07, BGH+06].

Our compiler correctness theorem incorporates resource restrictions and allows to discharge them at the $C0$ rather than at the assembly level. This simplifies arguing that the restrictions are not violated and thereby increases productivity. The small-step character of our simulation theorem allows to easily argue about resource restrictions also for intermediate states.

Even more, a properly designed compiler correctness theorem is an inevitable requisite for discharging some of the conditions at all. An important example is the question whether the theorem states total or partial correctness of the compiled programs. A partial correctness statement which shows that after termination the compiled program has computed the same result as the original high-level program, does only allow to transfer safety properties, i.e., to show that the compiled program does nothing bad. Functional correctness statements could only be transferred assuming the termination of the compiled code. To completely transfer functional correctness properties we need to know that every result (also intermediate ones) being computed by the high-level program is also computed by the compiled code. This includes the fact that termination of the high-level program implies termination of the compiled code.

Finally, at least in Verisoft, system level software relies on some special low-level properties of the compiled programs. For example, the operating system kernel allocates the page tables in the heap memory and depends on them being allocated at a given (fixed) address and staying there. Without detailed knowledge about the compiler used, such properties cannot be deduced.

2.2 Related Work

2.2.1 Related Work in the Context of System Verification

There have been different approaches to system verification. In the late 1980s, the CLI stack project [BHMY89] developed the idea of building a system of formally verified and hierarchically stacked components. This project achieved impressive results including a formal correctness proof for a compiler from the high-level language Micro-Gypsy to the Piton assembly language [You89, Moo88]. Micro-Gypsy is a subset of the Gypsy programming language; however, it does not support the dynamic data types and concurrency features of Gypsy. As this project was – like Verisoft – aiming at the *pervasive* verification of systems, their compiler correctness proof also considers resource restrictions on time and space. Similar to our result, these restrictions have to be proved at the

Micro-Gypsy layer for each individual program. In contrast to our work, the compiler correctness statement in the CLI stack does only argue about the *final* state of Micro-Gypsy computations. However, this is not as bad as it would be in the Verisoft context as the CLI system stack does not contain devices.

More recently, the projects L4.verified [HEK+07] and VFiasco [HTS02] started working on the verification of micro kernels. L4.verified uses a larger subset of C than $C0$, including pointer arithmetic, for the verification of the L4 micro kernel [Lie95]. For code verification they use a special instantiation of Verisoft's Hoare environment [Sch06]. VFiasco aims at the verification of the Fiasco micro kernel implemented in a subset of C++. They embed the C++ code into the PVS theorem prover [OSR92]. Both projects stop at the source code level: there is no attempt to map results down to lower system or language layers. Consequently, the projects do not involve any compiler verification.

In the FLINT project, an assembly code verification framework has been developed and used for the formal verification of code for context switching on x86 machines [NYS07]. However, this project does – so far – work on smaller code examples and does not yet comprise any results regarding high-level languages. Accordingly, the project does not address formal verification of a compilation.

2.2.2 Detached Related Work

There exist several formalizations of semantics for large subsets of the C programming language [GH93, Pap98, Nor98]; among these, the work of Norrish is most interesting for us as it is formalized in Isabelle / HOL – the theorem prover which is being used in Verisoft. None of these works addresses the integration of inline assembly code. However, semantics of the (almost) full C language are complex [ISO99] and the use of all features of C leads to an error-prone programming style [MIR04]. For example, expression evaluation in C is non-deterministic in the evaluation order; in [ZD03] the author describes a framework which is able to model this.

As stated in the previous section, when trying to verify properties of non-toy programs, such complicated languages are not desirable as they make correctness proofs of larger programs infeasible. It seems worthwhile to restrict implementation languages to a small set of features as has been done for the above-mentioned Micro-Gypsy programming language.

In [dRHH+01, OG76] Hoare logics for concurrent programs have been developed but the languages considered there are not powerful enough for our purposes, e.g., they do not support function calls. N. Schirmer has developed a generic Hoare logic environment for imperative programming languages in [Sch05]. An $C0$ instantiation of this environment is being used in the Verisoft project to accomplish code verification [Sch06].

Compiler verification is a well established field in computer science and a lot of remarkable work has been done. Some overview is given in [Dav03]. In the following we discuss related work with respect to compiler verification. The discussion is segmented according to the verification approach; recent big efforts in compiler verification are presented separately. In general, we distinguish several approaches for the reliable compilation of computer programs. We briefly sketch these approaches before actually examining the related work.

- Verification of the Compiling Algorithm and its Implementation
  This approach tries to verify the translation algorithm once and for all. To achieve this, one proves that for any given program p the translated program $c(p)$ behaves *equivalently*. As mentioned in the previous section, there are several ways how to define equivalence between the source program and the compiled code. In contrast to the work in Verisoft, most early work following this approach only argues about the compiling algorithm and ignores the correctness of the implementation.

- Compiler Synthesis
  This approach tries to automatically generate compilers for a given source and target language. One variety of the approach uses formalizations of the semantics of both languages to generate the compiler. This compiler is then correct 'by definition' although the algorithm which generates code from the semantical description of the source language is usually not verified.

  Another variety uses domain specific languages to specify the code transformations. However, most work using the latter approach do not formally verify the correctness of their transformations. And even when verifying the transformations, they do not show formally that the generation of the compiler out of the description is being done correctly.

- Certifying Compilers
  In addition to the compiled code, a certifying compiler generates an easily checkable proof that the compiled code and the source program behave equivalently (this is sometimes also called *credible compilation*). The approach of certifying compilers is related to the work on proof-carrying code [Nec97]. In contrast to the previous approaches, certifying compilers do not guarantee the correctness of translation once and for all but the check has to be repeated for each translated program.

  A big advantage of certifying compilers (and proof carrying code in general) is that the user does not have to trust the compiled binary but can easily check the provided proof prior to execution of the program. This guarantees that the binary, even if modified by an attacker (before the check), still fulfills the promised (safety) properties. However, this advantage is not relevant for the Verisoft scenario where we verify a complete stack including the applications (which means that we do not have to trust in the safety of unknown code).

- Translation Validation
  Similar to the previous approach, translation validation has to be repeated for each source program being compiled. However, not the compiler itself is in charge of generating a proof that source program and compiled code behave equivalently, but an external tool tries to find this proof. This external tool, the so-called *analyzer*, either establishes the equivalence between inputs or produces a counter-example. Such a counter-example is produced not only when the compiler introduces a bug during translation but also when the analyzer cannot decide whether the translation has been done correctly or not. A big advantage of this approach is that it can be applied to (industrial) compilers, for which no implementation or specification details are available.

Miscellaneous and Early Work. Early work of McCarthy [MP67] proves the correct translation of simple arithmetic expressions into a very simple target language. Some years later, Samet [Sam75, Sam78] used a technique similar to translation validation to check the correctness of optimizations steps in compilers. In [GB03], the authors present correctness proofs for constant folding; these proofs have been formalized in Isabelle / HOL.

Curzon presents in [Cur92, Cur94] the verification of a compiler from a significant subset of the structured assembly language Vista to the assembly language of the VIPER microprocessor. The proof has been verified in the HOL theorem prover and the compiler specification is reliably executable using formal proof within HOL. Curzon is very interested in ensuring that properties proved on the source program also hold for the compiled code. His work allows to transfer properties from high-level to low-level and he addresses the boot strap problem in a similar way as we propose in Section 7.7. However, his proofs are carried out big-step style and are only applicable to terminating programs.

In [ORW95], the authors describe a verified compiler for PreScheme, the implementation language for the VLISP run-time system. The compiler as well as the proof were divided into three parts: a front end which translates source text into a core language, a syntax-directed compiler which translates the core language into a combinator-based tree-manipulation language, and a linearizer which translates combinator code into the target language. The work has not been formalized in a theorem prover. However, the authors believe that a simplification-based theorem prover could automatically handle most cases of the syntax-directed translation while the back end proofs may be amenable to mechanization by interactive theorem provers. During tests of a compiler implementation which was manually derived from the specification, the authors found bugs in the assembly code sequences generated for the so-called stored-program machine instructions. These errors were below the grain of the presented proofs (e.g., registers not being saved across routine calls). According to the authors, extending the proof to reach this level – which is obviously required in our scenario – would require an extremely detailed model of the behavior of the machine and operating system.

[BZ07] presents a semantic type soundness result, formalized in the Coq proof assistant, for a compiler from a simple imperative language with heap-allocated data into an idealized assembly language.

Chirica and Martin [CM86] address the problem of implementation correctness of compilers. They compare, for each source program, the corresponding object program produced by the compiler implementation to the object program dictated by the compiler specification.

Refinement Techniques. In the ProCoS project, refinement techniques have been used to show compiler correctness. Hoare [HJS93] and Müller-Olm [MO97] verify translation to the machine language of an actual processor: the Inmos Transputer. Börger et al. apply refinement techniques to the translation from Occam, a non-toy imperative programming language with non-determinism and parallelism, to Transputer code [BD96]. Later, Stärk, Börger et al. used refinement techniques to prove a compiler from Java to Java byte code correct [SSB01]. All this work does not include correctness proofs for the compiler implementation.

Translation Validation. Despite the earlier work in [Sam75, Sam78] which uses a similar technique, the term 'translation validation' was only introduced by Pnueli in [PSS98a, PSS98b]. In this work, the authors examine translation from finite automata to C. Using a rather general approach, the paper presents theoretical insights into the constructions of translation validation tools.

In [PSS98c] Pnueli et al. present a first tool which implements their translation validation approach. This tool, called CVT, validates the translation of State-Mate / Sildex mixed specifications to C. It was developed in the scope of the ESPRIT project SACRES which dealt with the formal development of 'Safety Critical Real-time Embedded Systems'. Before actually proving equivalence between the original and the translated program they have been translated into a common semantic domain: synchronous transition systems. The tool was successfully evaluated on an industrial-sized program of a few thousand lines of code.

More recently, in [ZPFG02, ZPF+02, ZPFG03] the authors present an improved translation validation methodology called VOC which has been implemented in the prototype translation validator VOC-64 that automatically produces verification conditions for the global optimizations (including some structure modifying optimizations like loop unrolling) of the SGI Pro-64 compiler; these optimizations are performed on the basis of the compiler's intermediate language WHIRL.

As already hinted in [ZPF+02], Barret et al. present in [BGZ03] an approach to deal with speculative loop optimizations. Speculative loop optimization is an aggressive form of compiler optimizations which is only correct under certain conditions which cannot be validated at compile time. The paper suggests using run-time verification to ensure that the conditions are fulfilled. For each application of speculative loop optimization a runtime test is automatically generated using the automatic theorem prover CVC. This run-time test checks whether the condition is true. If it is not, the optimization is reverted without destroying the program's semantics.

In [GZB05], the authors describe an approach for improving the tool Tvoc which handles many optimizations of Intel's ORC compiler but is limited when dealing with loop reordering transformations. Leviathan applies translation to loop optimizations and to several compiler back ends [Lev04]; in particular, he demonstrates his method on a commercial compiler which translates synchronous C programs to PowerPC binary code.

In [Eme05], the author uses a translation validation approach to show absence of interrupts for a given assembly program.

Necula [Nec00] presents the tool 'prototype translation validation infrastructure' (TVI) for the GNU C Compiler gcc. TVI compares programs in gcc's intermediate language and handles most optimizations applied by the compiler. The tool has been tested on large software systems as gcc itself and the Linux kernel. However, TVI produced a relatively large number of false alarms.

In general, a great advantage of translation validation techniques is the high degree of automation and that they are applicable to optimizing industrial compilers. However, it produces no general, pervasive compiler correctness theorem (even more, most work in this area focuses only on specific parts of the translation process) and a bug in the compiler can only be discovered when the compiler is run on a program which triggers the bug. In contrast, our approach guarantees correct compilation for each and every legal $C0$ program.

Certifying Compilers. Necula introduced in [Nec98, NL98] the notion of certifying compilers whose realization has later been declared a great challenge by Hoare in [Hoa03]. In his work Necula presents a certifying compiler which translates a type-safe subset of C to highly optimized DEC Alpha assembly code. The certifier checks automatically type safety and memory safety of the generated programs. If certification fails, the system generates a counterexample pointing to a potential violation of the type system. In [CLN+00], Colby et al. extend this approach to a certifying compiler which translates a large subset of Java to x86 code. However, the work of Necula and Colby concentrates completely on safety properties. They do not address any functional correctness properties.

Rinard presents in [Rin99] an approach called 'credible compilation' for standard imperative languages which is basically equivalent to the concept of certifying compilers. However, Rinard does not stop at safety properties. His approach allows to show that both programs produce the same final result, i.e., that they are semantically equivalent. The paper contains several examples illustrating how standard optimizations (dead assignment elimination, branch movement, induction variable elimination, loop unrolling, dead code elimination) could be handled. In [RM99] he extends this approach to programs with pointers.

Blech and Poetzsch-Heffter also extend the approach of certifying compilers to functional correctness [BPH07]. They study a compiler from a C subset to MIPS. However, their verification is limited to the compiler back end which translates a simple intermediate language (no function calls, no dynamic memory allocation) to MIPS. Their work was formalized in the theorem prover Isabelle / HOL. A prominent result of their work is that they do not only show that both programs produce the same result. Instead, they model language via small-step semantics and prove them equivalent by showing that they produce the same output traces. Thus, they are not limited to terminating programs. Internally, they use a simulation relation similar to ours (although much simpler, due to the simple intermediate language) which finally implies the observational equivalence. The run-time of their proofs is dependent on the number of variables in the source program (counting array elements as single variables). Unfortunately, the performance of their certifier is still very bad for non-trivial programs. For example, for a program that computes the first two hundred Fibonacci numbers the correctness proof runs about 1 500 seconds. For a program sorting an array of fifty elements, the run-time is over 6 000 seconds.

In [Tch04], the author uses a so-called phase semantics. Phase semantics model the semantics of real-life programming languages through a series of equivalence relations between languages $L_0, \ldots, L_n$ where for each i the languages L_i and L_{i+1} are very similar. Using this concept, the author developed a self-validating compiler called Vitamin for a subset of C. This compiler translates from language to language, finally translating L_0 (the C subset) to L_n (the machine language). In addition to the compiled code, the compiler generates expressions in all intermediate languages which are used as a certificate in the sense of certifying compilers.

In [Riv04, Riv05], Rival presents a new framework for reasoning about compilation and of compiled programs. He uses a symbolic representation of source and assembly for a new approach of invariant translation based on translation validation. The framework supports pointer aliasing and dynamic memory allocation. The properties which he was able to prove include the

absence of runtime errors in the compiled program. He implemented and tested his approach on real examples of highly critical embedded software (targeting mainly at safety properties) based on the GNU C compiler generating code for the PowerPC processor. Rival successfully applied his tools to a large real-world example (75 000 lines of C code) with a run-time of approximately one hour.

Compiler Synthesis. Early work regarding the synthesis of compilers from semantical descriptions of source and target language has been done by Polak [Pol81]. Paulson describes in [Pau82] a system that accepts definitions of programming languages given via attribute grammars and denotational semantics and produces an interpreter for the language. These classical systems have the problem that the performance of the generated code is by several orders of magnitude slower than the code generated by standard compilers.

Later, Palsberg used the technique to compile a non-trivial subset of ADA to RISC code [Pal92a, Pal92b]. He reports that the generated code is an order of magnitude better than that produced by compilers generated by the classical systems. Although the algorithm was proved correct manually, the implementation of the generation tool (written in Perl) is not verified and hence cannot be completely trusted.

Brown et al. developed the ACTRESS compiler generator [BMW92] which is, as the previous work of Palsberg, based on action semantics. Later, Diehl further improved performance of code generated via the compiler synthesis approach [Die96].

The approach to specify code transformations using domain specific languages goes back to [TH92]. In [LJWF02, LMC03, KSK06], temporal logic is used for the specification and correctness proofs of optimizations. This restricts the analysis to properties which are expressible in the used temporal logic. Of these, only the Cobalt system [LMC03] allows for completely automatic verification but it is restricted to structure preserving transformations. The Rhodium system enhances the Cobalt system by replacing temporal logic by local propagation rules [LMRC05]. These allow to formulate more program properties while still guaranteeing fully automatic correctness proofs for the transformations.

[BGL04] reports on the correctness proofs of compiler optimizations based on data-flow analysis. The optimizations and analyzes are formulated as instances of a general framework for data-flow analyzes and transformations. Proof obligations for specific optimizations are generated automatically and are to be proved in the Coq proof assistant.

μJava. Strecker presents in [Str02] the formal verification of a compiler for a subset of Java (called μJava) in the theorem prover Isabelle. The semantics of μJava and of the generated Java byte code are defined using big-step semantics. The correctness theorem states (via the usual commuting diagram) that (complete) execution of some Java statement and of the generated byte code have corresponding effect. Due to the big-step nature of the semantic definitions, the theorem is restricted to terminating programs. The compiler correctness proof presented in this paper is part of a more comprehensive research effort aiming at formalizing and verifying key aspects of a substantial subset of the Java programming language. This includes type system and operational semantics of

Java with a proof of type soundness [Ohe01a], an axiomatic semantics with a proof of its equivalence to the operational semantics [Ohe01b], and an abstract data flow analyzer instantiated for Java byte code, with a proof of correctness of Java byte code verification [KN03].

Verifix. The goal of the DFG project Verifix (1995–2003) was to develop methods for the construction of correct compilers for realistic source and target languages [GZ99]. The correctness property (preservation of observable behavior up to resource restrictions [DV01, Section 4]) allows support of all common optimizations; however, no optimizations have been verified within Verifix. The project partitioned compiler correctness into three sub tasks: correctness of the compiling specification, correct high-level implementation of the compiling specification, and a correct binary for the compiler (cf. Section 7.7 of this thesis).

In [GGZ99, Gle03] an approach similar to translation validation was used to prove correctness of the compiler implementation. The latter combines this with ideas from certifying compilers. Program checking was also applied to compiler front ends [HGG+99].

The correctness of the compiling specification was shown using simulation theorems based on small-step semantics [DV01]. However, only partial correctness of the compiled programs is proved. That is, a compiled program behaves equivalently to the source program *only if* the compiled code terminates. As stated before, partial correctness of the compiled code it not sufficient for our needs regarding pervasive verification.

In [ZG97] and [GZ00], the authors present an elegant theory using abstract state machines for the translation across the various intermediate languages used in compilers; the work was partially formalized in the PVS theorem prover. The results on the correctness of the transformations in compiler back ends have been summarized in [Zim06]. This work sums up earlier work and presents results for the translation from intermediate language to DEC-Alpha binary code. However, no integration of these results has been done in Verifix.

Nevertheless, a non-optimizing compiler from ComLisp (a Lisp subset) to binary machine code of the Inmos Transputer has been pervasively verified in the scope of Verifix. The verification has been done at the binary layer in the theorem prover PVS [GL01, DV01]. In [GH98], the authors discuss the problem of how to get an initial correct binary for this compiler. They use manual code inspection to close the last remaining gaps in the argumentation.

Clight. Recently, Leroy et al. have developed and verified a lightly optimizing compiler which translates the C subset Clight via the intermediate language Cminor to PowerPC assembly code [BDL06, Ler06]. The results of this effort are for several reasons very impressive. Clight [BDL06] supports more features of C than $C0$; in particular, pointer arithmetic is supported. However, similar to $C0$ pointers to local variables are not supported. Additionally, the back-end of the compiler is lightly-optimizing and very well structured using four intermediate languages. There exist optimization modules for constant propagation and common sub expression elimination, although, according to [Ler06], only constant propagation has been integrated in the certified compiler so far. The most prominent feature of Leroy's work for the purpose of pervasive verification is that he has verified a *complete* optimizing compilation chain from a structured

imperative language down to assembly code instead of focusing on just a few parts of the compiler (like optimization patterns) like recent other work tends to do.

The Clight compiler has been specified and verified in the Coq proof assistant. An executable version has been obtained by automatic translation from Coq specifications to Caml code (cf. Section 2.1.3).

Similar to us, Leroy targets the translation of critical embedded software. However, Clight and Cminor semantics as well as the compiler correctness theorem are currently formulated in big-step style. That is, he shows that source program and compiled code behave observationally equivalent only if the source program terminates. In [Ler06], he points out the need for small-step simulations theorems to allow reasoning over non-terminating executions.

Another drawback of his work in the context of pervasive verification is that his memory model assumes an infinite memory: allocation requests always succeed. This prevents from transferring properties from the abstract layer (in our case $C0$) to the machine layer without an expensive proof at the machine layer that the compiled code respects memory resources. Although Leroy states that it is hopeless to prove a stack memory bound on the source program and expect this resource certification to carry out to compiled code, we have integrated exactly this property in our compiler correctness theorem. However, to transfer such a property into his much more complex translation scheme may actually turn out to be infeasible.

2.2.3 Integration of Solutions

As pointed out in Section 2.1, there are many challenges for compiler verification in the context of pervasive verification. Some of them have been solved (in isolation) in a similar or even more general way in other work. Most recent work with respect to compiler verification has concentrated on just a few parts of compilers, mostly optimization patterns. However, an essential part of the verification effort has to be invested in the combination of the individual solutions into a single framework to gain a completely verified compilation chain from a high-level source language to machine code.[1] In addition to the impressive work of the CLI stack project [BHMY89, Moo03], early work from Joyce discusses problems imposed by the formal combination of a verified compiler with verified hardware [Joy89]. Leroy [Ler06] also stresses these special requirements (although his work presently does not address an important one: non-terminating computations).

To deal with these requirements, the $C0$ language has been designed to be powerful enough to implement system software while remaining 'neat' to allow for efficient formal verification of medium sized applications. Additionally, both the $C0$ semantics and the compiler correctness proof have been done in a small-step manner, allowing for the application of the compiler correctness theorem also for non-terminating and interleaving programs [ASS08].

Furthermore, our compiler correctness theorem shows *total* correctness of the compiled code. That is, for every state of the source program's computation, the target machine will eventually reach a corresponding state, in which the

[1] Confer the work of Beyer et al. in [BJK+06] for an example of the efforts needed to combine several formal results.

values of all reachable *C*0 variables are properly represented. Without a total correctness theorem, it would not be possible to transfer functional correctness properties of source program's down to the compiled code without proving the termination of the compiled code. The correctness theorem also includes lifting resource restrictions from the target layer to the *C*0 layer. If these resource restrictions have been discharged at the *C*0 layer, we can be sure that they also hold for the compiled code.

To the best of our knowledge, our work is the first compiler correctness result which *simultaneously*

1. proves a small-step simulation theorem between source programs in a non-toy input language and compiled code for a realistic target architecture,
2. considers resource restrictions and other special requirements of pervasive verification, and
3. argues about the complete compilation chain from source to target language (both represented by abstract data types in the theorem prover) instead of focusing on just a few parts of the compiler (like optimization patterns).

Our results have been heavily used in other parts of the Verisoft project. Most prominently, the compiler correctness theorem has been used to transfer correctness properties from the *C*0 layer to the lower layers according to the Verisoft approach to system verification. The formal pervasive verification of a paging mechanism gives a concrete example of this approach [ASS08]. This result provides strong confidence that our notion of compiler correctness is indeed appropriate for the targeted field of application. Furthermore, the small-step compiler correctness theorem has been used for the integration of (verified) inline assembly code into (verified) *C*0 applications. Without such a 'semantics' for inline assembly code, the system layer software in Verisoft (and the libraries which allow invocation of operating system services from user applications) could not have been implemented in a high-level programming language as *C*0 [ST08].

Part I

Languages

Chapter 3

The Language C0

Contents

In this chapter we introduce the C-like programming language $C0$. Most applications and the system layer software of the Verisoft project are implemented in $C0$ with only small parts being written using inline assembly. $C0$ is the source language of the compiler which has been formally verified in this thesis.

In Section 3.1 we describe the main features of $C0$ in an informal way and highlight some design choices. In Section 3.2 we present the concrete syntax of $C0$ programs. In Section 3.3 we describe a tool that does some basic syntax and type correctness checks on $C0$ programs, acts as front end for the implementation of the formally verified $C0$ compiler, and allows to translate concrete $C0$ programs into different representations (e.g., for the formal $C0$ semantics in Isabelle / HOL and a translation validation tool). We conclude this chapter with an introduction of a stack of three different $C0$ semantics: (i) a deep embedded small-step semantics, (ii) a deep embedded big-step semantics, and (iii) a shallow embedded big-step semantic together with a Hoare logic and a $C0$ verification environment.

3.1 Informal Description

One of the goals of the Verisoft project is the formal verification of implementations of an operating system micro kernel, a simple operating system, and several user programs [GHLP05]. Operating systems and especially operating system kernels, are usually not written in functional languages or Java (for an exception see [HJLT05]). Most operating system kernel are implemented in C while some hardware dependent parts have to be implemented with inline assembly code. Even the kernel of the JavaOS operating system [Jav96] is written in C with inline assembly.

Semantics of the full C language are complex [GH93, Pap98, Nor98] and the use of all features of C leads to an error prone programming style [MIR04].

In contrast, formal verification of programs is easier and more efficient for programming languages with concise and well-defined semantics.

In this thesis we introduce the C-like imperative programming language $C0$ which meets both requirements: (i) it has sufficient features to implement all system and application software in the Verisoft project and (ii) has a concise semantics which allows for efficient verification of $C0$ programs with several thousand lines of code.

Compared with standard C [ISO99] the language $C0$ has several limitations. We list the most important differences. Side effects in expressions are not allowed; this forbids in particular function calls a sub expressions and requires a special function call statement in $C0$ (cf. below). $C0$ Pointers are typed and must not point to local variables or to functions; void pointers and pointer arithmetic are not supported. There is a separate type for arrays in $C0$ and the size of arrays has to be statically fixed at compile time. Unions or bit fields are not supported. Switch and goto statements as well as long jumps are also not allowed in $C0$.

In contrast to normal C, $C0$ does not support manual deallocation of memory via some kind of 'free' statement. Instead, a garbage collector is planned to be used to free unreachable parts of the heap automatically. So far, no garbage collector has been integrated into the $C0$ compiler. However, the $C0$ implementation of a copying garbage collector has been verified recently by E. Petrova. Its integration into our results remains as future work (cf. Chapter 12).

One of the main reasons to use a garbage collector instead of explicit deallocation and to forbid pointers to local variables is to avoid the problem of dangling pointers. A pointer is called dangling if it is different from the null pointer but does not point to a *valid* memory object. To make sure that $C0$ expressions always refer to valid memory object, we have to make sure that no dangling pointers are dereferenced. This would be an additional proof obligation for languages which do not preclude dangling pointers by design.

A dangling pointer can occur by 'deallocating' a memory object while there still exist references to that object; observe, that this includes the case that we have a pointer to some local variable and return from the stack frame to which that variable belongs. In $C0$, both causes are impossible; thus, dangling pointers cannot occur in this language.

3.1.1 Types

$C0$ supports four *basic* types:

- Boolean: *bool*, $\{true, false\}$
- 32-bit signed integers: *int*, $\{-2^{31}, \ldots, 2^{31}-1\}$
- 32-bit natural numbers: *unsigned int*, $\{0, \ldots, 2^{32}-1\}$
- 8-bit signed integers: *char*, $\{-2^{7}, \ldots, 2^{7}-1\}$

We call *int* and *unsigned int* arithmetic types. Arithmetic operations with 8-bit signed integers are not supported.

If t, t_1, ..., t_s are types, n is a number, and n_1, ..., n_s are names, then pointers $*t$, arrays of fixed size $t[n]$, and structures $struct\{n_1 : t_1, \ldots, n_s : t_s\}$ are *aggregate* types. Thus, pointers are typed in $C0$; there exists a special type

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 null pointer constants. Void pointers, function pointers, or pointers to local variables are not allowed. Unions or bit fields are not supported.

We combine the basic types and pointer into *elementary* types.

3.1.2 Expressions

Variable names and literals of a basic type are expressions.

If e and i are expressions, and n is a name, then array access $e[i]$, access to structure components $e.n$, dereferencing of pointers $*e$, and the 'address-of' operator $\&e$ are also expressions given that some preconditions on the types of e and i are fulfilled. We permit the setting of pointers to sub variables of aggregate variables;[1] pointer arithmetic however is forbidden. Expressions in $C0$ do not have side effects; thus, we do not permit function calls as part of expressions.

If $\circ_1$ is one of the unary operators from Table 3.1 and if $\circ_2$ is one of the binary operators from Table 3.2 then $\circ_1 e_1$ and $e_1 \circ_2 e_2$ are also expressions.

The evaluation of the logical *and* and *or* operators is done using short circuit evaluation.[2] Thus, for the logical and operator the evaluation of the right operand is skipped in case that the left operand evaluates to *false*. In this case the result of the logical and is *false*. Similarly for the logical or operator the evaluation of the second operand is skipped if the first one evaluates to *true*. In this case the result of the logical or is *true*. The lazy operators of $C0$ are listed in Table 3.3.

Finally, if e is an expression then (e) is also an expression.

For a detailed definition of the additional requirements regarding the expressions see Section 5.5.

3.1.3 Statements

If s_1 and s_2 are statements, e and e_i are expressions, t is a type specifier, and f is the name of a function, $C0$ supports the following standard statements:

[1] This is only allowed for the $C0$ small-step semantics and will be forbidden when we integrate the garbage collector.

[2] This is sometimes also called lazy evaluation.

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_{eq}	$t' = bool$
`!=`	not equal	c_{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_{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

- assignments: $e_1 = e_2$
  Observe that in $C0$ not only assignments of elementary types or structures are allowed, but also assignments of array values, which is not possible in C.

- dynamic heap memory allocation: $e_1 = new(t)$
  There is no statement for deallocation of memory. Instead, a garbage collector will be used to free unreachable parts of the heap. The correctness of the garbage collector is not part of this thesis, but is currently being verified in the Verisoft project.

- function calls: $e = f(e_1, \ldots, e_n)$
  In order to guarantee that $C0$ expressions do not have side effects, there is a separate statement for function calls. Function calls may be recursive. Similar to assignments, the arguments and return value of the function call may be of arbitrary type.

- return from a function: *return* e
  Each function contains a single return statement at the end of the function body.

- while loops: *while* (e) s_1

- for loops: *for* $(e_1 = e_2;\ e_3;\ e_4 = e_5)$ s_1

- if statements: *if* e s_1 or *if* e s_1 *else* s_2

Statements can be concatenated by ; or grouped by braces. Switch or goto statements and long jumps are not allowed in $C0$.

Aggregate Literals

In addition to these standard statements $C0$ supports assignments of so-called *aggregate literals*. If e is an $C0$ expression and c is an aggregate literal of matching type then $e = c$ is a $C0$ statement.

In addition to normal literals, aggregate literals comprise literals of structure and array types. They are used for the initialization of non-elementary local variables which, in the $C0$ Hoare logic [Sch06], has to be done in one atomic step and cannot be split into several assignments of normal literals. Observe that aggregate literals must not be used inside expressions.

Normal literals of an elementary type are the base case of aggregate literals. If c_i are aggregate literals of type t then $\{c_0, c_1, \ldots, c_n\}$ is an aggregate literal of array type $t[n]$.[3] If c_i are aggregate literals of types t_i and n_i are component names then $\{.n_0 = c_0, .n_1 = c_1, \ldots, .n_n = c_n\}$ is an aggregate literal of structure type $struct\{n_0 : t_0, \ldots, n_n : t_n\}$.

[3] Do not confuse this notation with mathematical sets. This informal section uses the concrete $C0$ syntax (cf. Appendix A) to give the programmer a consistent introduction. In the (formal) remainder of the thesis, we will use unambiguous data type notation to represent $C0$ language elements.

3.1.4 Initialization of Variables

Global variables and dynamically allocated heap memory are automatically zero initialized in $C0$. However, local variables of a function – except for the function parameters – are *not* initialized automatically. The programmer has to ensure to initialize them before the first read access.

3.1.5 Inline Assembly

Some low-level parts of system software cannot be implemented in $C0$ alone. For these parts we extend $C0$ with inline assembly statements and call the resulting language $C0_{\mathrm{A}}$. If u is a list of assembly instructions then the inline assembly statement has the form $asm\ \{u\}$.

In $C0_{\mathrm{A}}$ the use of inline assembly code is restricted.

- Only a certain subset of DLX instructions is allowed (e.g., no load or store of bytes or half words, only *relative* jumps).
- The target address of store word instructions must be outside the code and data regions of the $C0_{\mathrm{A}}$ program or it must be equal to the allocated base address of a sub variable of the $C0_{\mathrm{A}}$ program with type *int* or *unsigned int*. This implies that inline assembly code cannot change the stack layout of the $C0_{\mathrm{A}}$ program.
- Certain registers (e.g., the stack pointer) must not be updated.
- The last assembly instruction in u must not be a jump or branch instruction and the target of jump and branch instructions must not be outside the code of u.
- The execution of u must terminate and must not generate misalignment or illegal instruction interrupts.

3.2 Concrete Syntax

The concrete $C0$ syntax resembles the corresponding parts of the standard C syntax (cf. [ISO99]). Its grammar in Backus-Naur form [NBB+97] is given in appendix A.

3.3 Front End Tool

The formal work described in this thesis starts from syntax trees of $C0$ programs. These syntax trees are represented by an instance of an abstract data type in Isabelle / HOL (cf. Section 4.1). Of course, $C0$ programs are not written in this form. Instead, they have to be translated from $C0$ text files into different intermediate formats, one of them being the abstract data types as they are used by the formal $C0$ small-step semantics in Chapter 4 and by the compiler specification from Chapter 7. This translation is done by a preprocessor tool called `c0_check`.

The structure of `c0_check` is depicted in Figure 3.1. The tool has several responsibilities. First, it checks $C0$ text files for syntactical errors and type

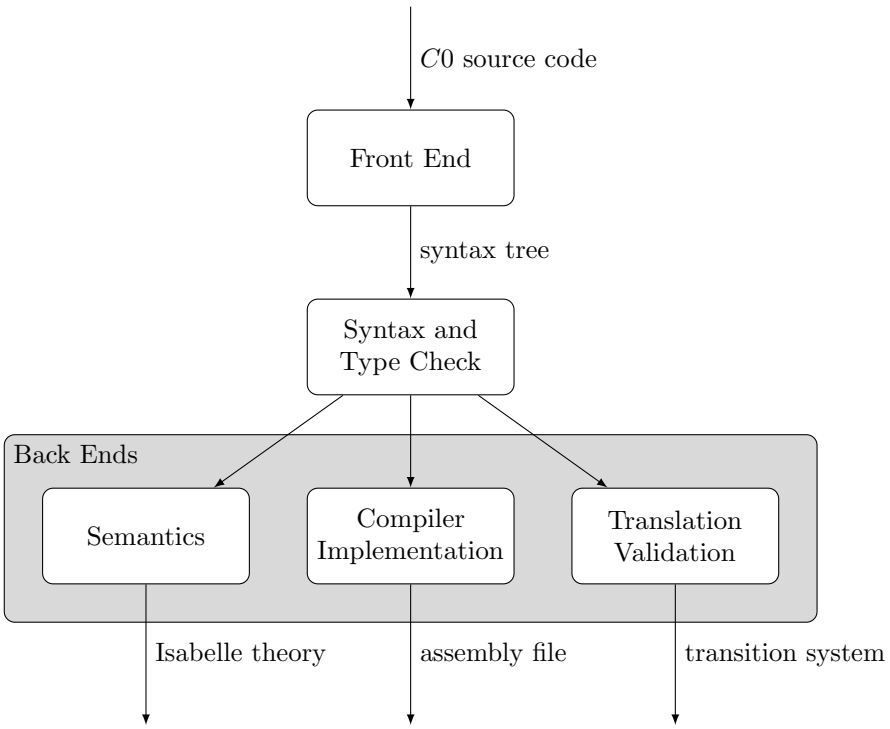


Figure 3.1: Structure of `c0_check`

checks them.[4] If these tests have been successful, several back ends are available to translate the programs into different forms. The first back end translates into representations suitable for the different formal $C0$ semantics; its output are Isabelle / HOL theory files. A second back end generates transition systems which are suitable for the Verisoft translation validation tool. The third back end is the formally verified $C0$ compiler implementation (cf. Chapter 11) which translates a given $C0$ program to VAMP assembly code.

[4] During these tests the tool also does some simple transformations. Constant arithmetic expressions like `4*1024` are being evaluated. Additionally, 'for' loops – which are not directly supported by $C0$ – are transformed into equivalent 'while' loops.

Chapter 4

Formal C0 Small-Step Semantics

Contents

We start this chapter in Section 4.1 with a description of how we formally represent $C0$ programs for the small-step semantics in Isabelle / HOL. In Section 4.2 we extend this to the representation of entire $C0$ configurations including memory and program rest. We introduce expression evaluation in Section 4.3 and conclude the Chapter in Section 4.4 with a definition of the $C0$ transition function.

4.1 Representation of C0 Programs

A $C0$ program is identified by the functions of the program, including information about the list of parameters and local variables of the functions, by the list of global variables, and by a type name environment which maps type names to types. In this Section we show how we formally represent $C0$ programs, i.e., their function table and the symbol table for the global variables, in the $C0$ small-step semantics.

4.1.1 Types and Type Name Environment

There are four *basic* types in $C0$: $Bool_\mathrm{T}$ for booleans, Int_T for signed 32-bit integers, $Char_\mathrm{T}$ for signed 8-bit integers, and $Unsigned_\mathrm{T}$ for 32-bit naturals. If t_i are types and n_i are component names then $Str_\mathrm{T}([(n_0, t_0), \ldots, (n_l, t_l)])$ is a structure type with l components. If t is a type then $Arr_\mathrm{T}(l, t)$ is an array with l elements of type t. We call a type t elementary if it is neither a structure nor an array type:

$$elem?_\mathrm{t}(t) = t \in \{Bool_\mathrm{T}, Int_\mathrm{T}, Unsigned_\mathrm{T}, Char_\mathrm{T}, Ptr_\mathrm{T}, Null_\mathrm{T}\}.$$

```
struct list_element{
  int data;
  struct list_element *next;
};
```

Listing 4.1: Example of a Fully Recursive Type in $C0$

In $C0$ pointers can be used inside the definition of the type to which they point (cf. Listing 4.1). Such loops in the type structure cannot be modeled directly by an abstract data type. Thus, we introduce an additional level of indirection for pointer types. They do not directly contain the type of the target but a name for the type. Using a so-called *type name environment* these type names can be mapped to a type. A pointer to a type name tn is represented by $Ptr_\mathrm{T}(tn)$.

Null pointer literals may be assigned to or compared with pointers of arbitrary type. Thus, we introduce a special representation *Null* of the type of null pointer literals to simplify the formalization of type correctness conditions.

We represent $C0$ types formally in Isabelle by the abstract data type ty.

$$\begin{aligned} ty \ = \ & Bool_\mathrm{T} \mid Int_\mathrm{T} \mid Char_\mathrm{T} \mid Unsigned_\mathrm{T} \\ & \mid Str_\mathrm{T}(\mathbb{S} \times ty\ list) \\ & \mid Arr_\mathrm{T}(\mathbb{N}, ty) \\ & \mid Ptr_\mathrm{T}(\mathbb{S}) \mid Null_\mathrm{T} \end{aligned}$$

Type name environments are represented in Isabelle / HOL by the type $tenv : (\mathbb{S}, ty)\ list$, i.e., they map type names to types.

Definition 4.1 (Abstract size of types) We define the abstract size $size_\mathrm{t}$ of $C0$ types by induction. The size of elementary types is set to 1; the size of aggregate types is the sum of the size of their components.

$$size_\mathrm{t} :: ty \mapsto \mathbb{N}$$

$$size_\mathrm{t}(t) = \begin{cases} 1 & \text{if } elem?_\mathrm{t}(t) \\ 0 & \text{if } t = Str_\mathrm{T}([\,]) \\ n \cdot size_\mathrm{t}(t') & \text{if } t = Arr_\mathrm{T}(n, t') \\ size_\mathrm{t}(t) + size_\mathrm{t}(Str_\mathrm{T}(xs)) & \text{if } t = Str_\mathrm{T}((cn, t)\#xs) \end{cases}$$

4.1.2 Expressions

The most basic element of $C0$ expressions are literals. They are modeled by the abstract data type *lit* with one constructor for each of the four basic types and for null pointer literals. Except for null pointers they all have a single parameter of the corresponding type which stores their value.

$$lit \ = Bool(\mathbb{B}) \mid Int(\mathbb{Z}) \mid Unsigned(\mathbb{N}) \mid Char(\mathbb{Z}) \mid Null$$

In addition to these literals of elementary types, aggregate literals allow to

Table 4.1: Unary Operators

Operator	Meaning
minus	unary minus
neg	bitwise negation
not	logical not
to_{int}	conversion to Int_{T}
to_{unsgnd}	conversion to $Unsigned_{\mathrm{T}}$
to_{char}	conversion to $Char_{\mathrm{T}}$

build literal values for structures and arrays.

$$\begin{aligned} lit_{\mathrm{a}} \; = \; & ALPrim(lit) \\ & \mid ALStruct(\mathbb{S} \times lit_{\mathrm{a}}\ list) \\ & \mid ALArr(lit_{\mathrm{a}}\ list) \end{aligned}$$

Expressions are defined inductively by the data type *expr*.

$$\begin{aligned} expr \; = \; & Lit(lit) \\ & \mid Var(\mathbb{S}) \\ & \mid Arr(expr, expr) \\ & \mid Str(expr, \mathbb{S}) \\ & \mid UnOp(unop, expr) \\ & \mid BinOp(binop, expr, expr) \\ & \mid LazyBinOp(lazyop, expr, expr) \\ & \mid AddrOf(expr) \\ & \mid Deref(expr) \end{aligned}$$

The basic cases are literal expressions $Lit(l)$ with a literal l and variable access $Var(n)$ to a local or global variable of name n.

Let n be a name and e, e_1, e_2, and i expressions. Then, array access $Arr(e, i)$ to the i-th element of an array expression e, access to structure components $Str(e, n)$, addr-of computation $AddrOf(e)$ and dereferencing of pointers $Deref(e)$ are also expressions.

Arithmetic and logical computations can be done by unary, binary, and lazy operators. The operators which are supported by $C0$ are listed in Tables 4.1, 4.2, 4.3, and 4.4.

4.1.3 Statements

Statements of the $C0$ small-step semantics are tagged with unique statement identifiers of type *sid*. Theses identifiers are used to determine during the execution of a program where a certain statement in the program rest was originally placed in the program and to examine the relationship between statements. This is mainly used in the compiler correctness proof. For simplicity we omit the statement identifiers in the text when they are not explicitely used.

Table 4.2: Binary Operators

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

Table 4.3: 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?
$comp_{\mathrm{neq}}$	different?

Table 4.4: Lazy Binary Operators

Operator	Meaning
and_{l}	logical and
or_{l}	logical or

In the $C0$ small-step semantics statements are modeled by the following data type.

$$
\begin{aligned}
stmt \;=\; & \mathit{Skip} \\
& \mid \mathit{Comp}(stmt, stmt) \\
& \mid \mathit{Ass}(expr, expr, sid) \\
& \mid \mathit{Ass}_{\mathrm{AL}}(expr, lit_{\mathrm{a}}, sid) \\
& \mid \mathit{PAlloc}(expr, \mathbb{S}, sid) \\
& \mid \mathit{SCall}(\mathbb{S}, \mathbb{S}, expr\ list, sid) \\
& \mid \mathit{Return}(expr, sid) \\
& \mid \mathit{Ifte}(expr, stmt, stmt, sid) \\
& \mid \mathit{Loop}(expr, stmt, sid) \\
& \mid \mathit{Asm}(asm\ list, sid) \\
& \mid \mathit{XCall}(\mathbb{S}, expr\ list, expr\ list, sid)
\end{aligned}
$$

Skip represents the empty statement and $Comp(s_1, s_2)$ the sequential composition of two statements s_1 and s_2. These two statements are not tagged with a statement identifier and also called *structural* statements in the rest of this thesis. We use the predicate $stmt_{\text{structural}} :: stmt \mapsto \mathbb{B}$ to distinguish structural from non-structural statements: $stmt_{\text{structural}}(s) = (is\_Skip(s) \vee is\_Comp(s))$.

Assignments come in two flavors. We have normal assignments of an expression e_r to an expression e_l, which is modeled by $Ass(e_l, e_r)$. Additionally, we allow assignments of aggregate literals. Allocation of dynamic memory for a type with name t_n is done by the statement $PAlloc(e, t_n)$. The pointer to the newly allocated memory will be assigned to expression e.

Function calls to a function of name f with parameters e_1 to e_n are represented by $SCall(vn, f, [e_1, \ldots, e_n])$. After termination, the return value of the function will be stored in variable vn. Return statements with return expression e are modeled by $Return(e)$.

Conditional execution is supported via $Ifte(e, s_1, s_2)$ which executes one of the statements s_1 and s_2 depending on condition e. 'While' loops with condition e and body lb are represented by $Loop(e, lb)$. There is no special statement for 'for' loops; however, they are supported in the concrete syntax and converted to equivalent 'while' loops by the preprocessor (cf. Section 3.3).

In $C0_{\text{A}}$ a sequence a of VAMP assembly instructions can be inlined by the statement $Asm(a)$. However, inline assembly is not covered by the compiler which is considered in this thesis. The semantics of $C0$ programs with inline assembly are defined by alternating execution of the $C0$ small-step semantics and of the VAMP assembly semantics (cf. on page 66). For the transitions from the $C0$ to the assembly layer, the compiler simulation relation from Section 8.2 is used. The big-step semantics and Hoare logic do not support inline assembly code.

XCalls are represented by the statement $XCall(f, [e_1, \ldots, e_n], [p_1, \ldots, p_m])$, where f is the name of the XCall, e_i are left-expressions denoting the parts of the $C0$ state that can be changed by the XCall, and p_i are right-expressions which are used as input parameters for the XCall. In addition to the parts of the normal $C0$ state (defined by the left-expressions e_i), the XCall may arbitrary change the extended state component.

Isabelle For better readability, we have two separate constructors for normal assignments and assignments of aggregate literals in this thesis. In Isabelle, both cases are covered by the same constructor.

Miscellaneous Functions on Statements

We define several functions on statements.

Definition 4.2 (Sub statements) First we define the set of sub statements of a given statement.

$$sub_{\text{s}} :: stmt \mapsto stmt\ set$$

$$\begin{aligned} sub_{\text{s}}(Comp(s_1, s_2)) &= \{Comp(s_1, s_2)\} \cup sub_{\text{s}}(s_1) \cup sub_{\text{s}}(s_2) \\ sub_{\text{s}}(Ifte(e, s_1, s_2)) &= \{Ifte(e, s_1, s_2)\} \cup sub_{\text{s}}(s_1) \cup sub_{\text{s}}(s_2) \\ sub_{\text{s}}(Loop(e, lb)) &= \{Loop(e, lb)\} \cup sub_{\text{s}}(lb) \end{aligned}$$

For all other statements s we simply define $sub_{\text{s}}(s) = \{s\}$.

Definition 4.3 (Number of returns) We define a function $\#ret :: stmt \mapsto \mathbb{N}$ which computes the number of return statements in the statement tree spanned by a given statement. For compound statements and returns we set

$$\begin{aligned} \#ret(Comp(s_1, s_2)) &= \#ret(s_1) + \#ret(s_2) \\ \#ret(Return(e)) &= 1. \end{aligned}$$

For all other statements s we define $\#ret(s) = 0$.

Definition 4.4 (Number of top-level returns) We also define the number of top-level return statements in a given statement list.

$$\#ret_{\text{top}} :: stmt\ list \mapsto \mathbb{N}$$

$$\begin{aligned} \#ret_{\text{top}}([\,]) &= 0 \\ \#ret_{\text{top}}(h\#t) &= \begin{cases} \#ret_{\text{top}}(t) + 1 & \text{if } is\_Return(h) \\ \#ret_{\text{top}}(t) & \text{otherwise} \end{cases} \end{aligned}$$

Compound statements can be regarded as a special encoding of lists of *top-level* statements (cf. Fig. 5.1). To convert a statement tree to such a list of top-level statements we introduce the two functions *s2l* and $s2l_{\text{ns}}$.

Definition 4.5 (List of top-level statements) To convert a list of statements encoded by compound statements into a *real* list of statements we introduce the function $s2l :: stmt \mapsto stmt\ list$.[1] For compound statements we define

$$s2l(Comp(s_1, s_2)) = s2l(s_1) \circ s2l(s_2).$$

For all other statements s we set $s2l(s) = [s]$.

Definition 4.6 (Top-level statements no skips) We introduce the function $s2l_{\text{ns}} :: stmt \mapsto stmt\ list$ which is similar to *s2l* except that it removes *Skip* statements from the statement list. We set

$$\begin{aligned} s2l_{\text{ns}}(Skip) &= [\,] \\ s2l_{\text{ns}}(Comp(s_1, s_2)) &= s2l_{\text{ns}}(s_1) \circ s2l_{\text{ns}}(s_2) \end{aligned}$$

For all other statements s we define: $s2l_{\text{ns}}(s) = [s]$.

Definition 4.7 (Code behind first return) We define a function $drop_{\text{ret}} :: stmt\ list \mapsto stmt\ list$ which removes the prefix of a given statement list up to the first return statement.

$$\begin{aligned} drop_{\text{ret}}([\,]) &= [\,] \\ drop_{\text{ret}}(h\#t) &= \begin{cases} t & \text{if } is\_Return(h) \\ drop_{\text{ret}}(t) & \text{otherwise} \end{cases} \end{aligned}$$

[1] Observe that this function only converts the *top-level* of a statement tree. For example, the body of loop statements will stay unchanged.

Definition 4.8 (Successor of the *i*-th return) We introduce the function $succ_{\text{ithret}} :: stmt\ list \times \mathbb{N} \mapsto stmt_\perp$ which returns the statement behind the i-th return statement of a statement list. Observe that we count the returns starting from 0, s.t. for $i = 0$ we return the statement behind the *first* return. Formally, we define the function by a double induction on i and the statement list

$$succ_{\text{ithret}}(sl, 0) = \begin{cases} \lfloor h \rfloor & \text{if } drop_{\text{ret}}(sl) = h\#t \\ None & \text{if } drop_{\text{ret}}(sl) = [\,] \end{cases}$$
$$succ_{\text{ithret}}(sl, i+1) = succ_{\text{ithret}}(drop_{\text{ret}}(sl), i)$$

Definition 4.9 (Remove last statement) To define later the initial program rest of a $C0$ program (cf. Section 4.4.1) we will need to remove the last statement from the body of the main function (this is the return statement). Thus, we define the function $remlast :: stmt \mapsto stmt$ which replaces the last statement in a statement tree by *Skip*, i.e., which effectively removes the last statement. For compound statements its definition is recursive.

$$remlast(Comp(s_1, s_2)) = Comp(s_1, remlast(s_2))$$

For all other statements s we simply set $remlast(s) = Skip$.

4.1.4 Function Table

The function table stores information about the functions of a $C0$ program. Information for a *single* function is stored in the record $funcT$. Elements p of this type have four components.

- $p.body :: stmt$: The body of the function.
- $p.params :: (\mathbb{S} \times ty)\ list$: A list of function parameters. For every parameter we store its name and its type.
- $p.rtype :: ty$: The return type of the function.
- $p.lvars :: (\mathbb{S} \times ty)\ list$: The list of local variables of the function. This list has the same format as the list of function parameters but does not include those.

The complete function table is a list of pairs of the functions name and a $funcT$ record which contains the information about the function. Formally, it is represented by the type $functableT = (\mathbb{S} \times funcT)\ list$.

4.2 Configuration of the C0 Small-Step Semantics

A configuration of the $C0$ small-step semantics stores the run-time information about the execution of a program in the small-step semantics. The configuration has two components.

- The *memory configuration* stores information about the variables of a $C0$ program and their values. This includes the values of the dynamically allocated variables on the heap.
- The *program rest* keeps track of the statements which still have to be executed.

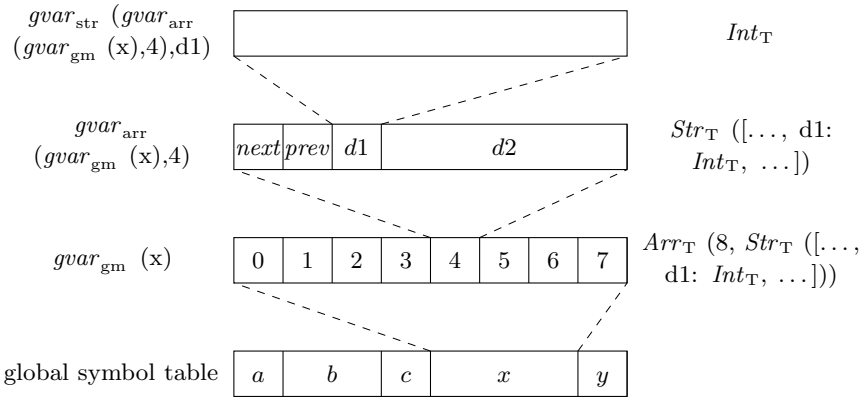


Figure 4.1: Hierarchy of G-Variables

4.2.1 Memory Configuration

The memory configuration of the $C0$ small-step semantics stores information about the variables of a $C0$ program and their values. It consists of three parts: a memory frame for the global variables, a memory frame for the heap variables, and a list of memory frames and return destinations for the local variables.

Generalized Variables

We represent pointers in the small-step semantics in a structural way using so-called generalized variables or short g-variables. G-variables are defined inductively and structurally similar to left-expressions.

There are three base cases for g-variables. Global variables of name x are represented by $gvar_{\text{gm}}(x)$, local variables x in the i-th local memory frame by $gvar_{\text{lm}}(i, x)$, and nameless heap variables by $gvar_{\text{hm}}(i)$. Variables in the global memory or in one of the local memories are referenced by their name; heap variables are nameless, so they are referenced by an index i.

The inductive case defines g-variables for structure and array access. If g is a g-variable of structure type then a component $gvar_{\text{str}}(g, n)$ of name n is also a g-variable. If g is a g-variable of array type then the i-th array element $gvar_{\text{arr}}(g, i)$ is also a g-variable.

Formally, we represent g-variables by the following data type.

$$\begin{aligned} gvar \;= \; & gvar_{\text{gm}}(\mathbb{S}) \\ & \mid gvar_{\text{lm}}(\mathbb{N}, \mathbb{S}) \\ & \mid gvar_{\text{hm}}(\mathbb{N}) \\ & \mid gvar_{\text{arr}}(gvar, \mathbb{N}) \\ & \mid gvar_{\text{str}}(gvar, \mathbb{S}) \end{aligned}$$

We define several functions which allow to reason about the hierarchy of g-variables (cf. Figure 4.1). The figure illustrates the structure of a g-variable in the global symbol table. In every level we look deeper into global variable x; the structure of the corresponding g-variable is denoted to the left of the figure. Observe that while the structure of the g-variables gets more complex

the structure of the corresponding type (denoted to the right of the figure) gets simpler.

Definition 4.10 (Parent g-variable) For g-variables $g = gvar_{\text{arr}}(h, n)$ or $g = gvar_{\text{str}}(h, c)$ we call h the parent g-variable of g. We formalize this by the function $parent_{\text{g}} : gvar \mapsto gvar_{\perp}$.

$$\begin{aligned} parent_{\text{g}}(gvar_{\text{gm}}(x)) &= None \\ parent_{\text{g}}(gvar_{\text{lm}}(i, x)) &= None \\ parent_{\text{g}}(gvar_{\text{hm}}(i)) &= None \\ parent_{\text{g}}(gvar_{\text{arr}}(h, i)) &= \lfloor h \rfloor \\ parent_{\text{g}}(gvar_{\text{str}}(h, c)) &= \lfloor h \rfloor \end{aligned}$$

Definition 4.11 (Root g-variable) If we would apply the function $parent_{\text{g}}$ multiple times to a g-variable g we would finally reach the last (or highest) ancestor r of g. We call r the root of g-variable g. Formally, we define the function $root_{\text{g}} : gvar \mapsto gvar$ which computes the root of a given g-variable.

$$\begin{aligned} root_{\text{g}}(gvar_{\text{gm}}(x)) &= gvar_{\text{gm}}(x) \\ root_{\text{g}}(gvar_{\text{lm}}(i, x)) &= gvar_{\text{lm}}(i, x) \\ root_{\text{g}}(gvar_{\text{hm}}(i)) &= gvar_{\text{hm}}(i) \\ root_{\text{g}}(gvar_{\text{arr}}(g, i)) &= root_{\text{g}}(g) \\ root_{\text{g}}(gvar_{\text{str}}(g, c)) &= root_{\text{g}}(g) \end{aligned}$$

Definition 4.12 (Sub variables of a g-variable) We define by induction the set $sub_{\text{g}} :: gvar \mapsto gvar\ set$ of sub variables of a given g-variable. Initally, this set contains the g-variable itself.

$$\frac{}{g \in sub_{\text{g}}(g)}$$

For the inductive cases we define

$$\frac{h \in sub_{\text{g}}(g)}{gvar_{\text{arr}}(h, i) \in sub_{\text{g}}(g)}$$

$$\frac{h \in sub_{\text{g}}(g)}{gvar_{\text{str}}(h, c) \in sub_{\text{g}}(g)}$$

Memory Frames

We use a relatively explicit, flat memory model in the style of [Nor98], which stores the memory content as a mapping from addresses (natural numbers) to memory cells.

A single memory cell can store the value of a variable of an elementary type. Values of aggregate types are stored as a consecutive sequence of memory cells. Memory cells are modeled by an abstract data type with one constructor for each elementary type. Thus, the memory model of the $C0$ small-step semantics

is typed. Pointers are represented by a generalized variable or by a special null pointer value $\perp$.

$$mcell_{\text{C0}} = Int(\mathbb{Z}) \mid Nat(\mathbb{N}) \mid Char(\mathbb{Z}) \mid Bool(\mathbb{B}) \mid Ptr(gvar \cup \{\perp\})$$

In addition to the content of the memory, a memory frame contains a list of variables of the memory frame together with their types and keeps track of the set of variables which are already initialized. Formally, we represent memory frames m by the record *mframe* which has three components.

- $m.ct :: \mathbb{N} \mapsto mcell_{\text{C0}}$: The content of the memory frame. Althoguh technically infinite, we use this mapping only for addresses smaller than the size of the memory (which changes dynamically for the heap memory); the values for other addresses are not defined.
- $m.st :: (\mathbb{S} \times ty)\ list$: A list of all variables of the memory frame together with their types. We call this list a symbol table.
- $m.init :: \mathbb{S}\ set$: The set of variables which are already initialized.

Definition 4.13 (Base address of variables) We define the base address of variables x in a symbol table by induction on the symbol table. For non-empty symbol tables the base address is zero if x is the first variable in the symbol table. Otherwise, it is defined inductively as the sum of the size of the first variable and the base address of x in the tail of the symbol table. If the variable is not present in the symbol table we return *None*.

$$ba_{\text{v}} :: (\mathbb{S} \times ty)\ list \times \mathbb{S} \mapsto \mathbb{N}_{\perp}$$

$$ba_{\text{v}}([\,], x) = None$$

$$ba_{\text{v}}((n,t)\#xs, x) = \begin{cases} \lfloor 0 \rfloor & \text{if } n = x \\ \lfloor size_{\text{t}}(t) + b \rfloor & \text{if } n \neq x \wedge ba_{\text{v}}(xs, x) = \lfloor b \rfloor \\ None & \text{if } n \neq x \wedge ba_{\text{v}}(xs, x) = None \end{cases}$$

Definition 4.14 (Type of variables) We define the type of variable x in a given symbol table st by

$$type_{\text{v}} :: (\mathbb{S} \times ty)\ list \times \mathbb{S} \mapsto ty_{\perp}$$
$$type_{\text{v}}(st, x) = map\text{-}of(st, x).$$

Memory Configuration

Based on single memory frames we can now give a formal definition of the memory configuration of the *C*0 small-step semantics. A memory configuration *mc* is formalized by the record *memconf* which has three components.

- $mc.gm :: mframe$: The memory frame for the global variables.
- $mc.lm :: (mframe \times gvar)\ list$: A list of memory frames and g-variables which model the stack of local memories. The memory frames store the values of the local variables whereas the g-variables store the so-called

return destination of a stack frame. During the execution of a return statement, the value of its return expression is stored at the memory location specified by the return destination of the currently topmost stack frame.

G-variables refer to local memories by their index. To keep the (list) index of stack frames – and thus also the names of g-variables which refer to them – constant during their lifetime, we store the current stack frame at the *end* of the list of local memories. Thus, the top most (or current) local memory frame has index $|mc.lm| - 1$. We abbreviate it with $lm_{\text{top}}(mc) = mc.lm!(|mc.lm| - 1)$.

- $mc.hm :: mframe$: The memory frame for the dynamically allocated heap variables. Observe that the variable names of heap variables in $mc.hm.st$ are not used because heap variables are nameless.

Isabelle Observe that in Isabelle the local memories are stored in reverse order. This simplifies the definitions for adding and removing new local memory frames because it corresponds to the direction in which lists grow due to the datatype constructors. However, indices of stack frames are still constant because we index in Isabelle starting from the end of the local memory list.

We introduce abbreviations for the symbol tables of the global memory, of the current local memory frame, and of the heap memory.

$$\begin{aligned} lst_{\text{top}} &:: memconf \mapsto (\mathbb{S} \times ty)\ list \\ gst &:: memconf \mapsto (\mathbb{S} \times ty)\ list \\ hst &:: memconf \mapsto (\mathbb{S} \times ty)\ list \end{aligned}$$

$$\begin{aligned} lst_{\text{top}}(mc) &= lm_{\text{top}}(mc).st \\ gst(mc) &= mc.gm.st \\ hst(mc) &= mc.hm.st \end{aligned}$$

We name memories in the obvious way. A memory name is either gm for the global memory, $lm(i)$ for the i-th local memory, or hm for the heap memory. Formally, we define memory names by the data type $memname$.

$$memname \ = gm \mid lm(\mathbb{N}) \mid hm$$

Symbol Configuration

Frequently in this thesis we are not interested in the memory content and do not need the complete memory configuration. It suffices to have only the symbol tables of all memory frames available. For these situations we introduce similar to the type for memory configurations a record type for *symbol configurations*. Symbol configurations sc of the $C0$ small-step semantics are formalized by the record $symbolconf$ which has three components.

- $sc.gst :: (\mathbb{S} \times ty)\ list$: The symbol table of the global memory frame.
- $sc.lst :: (\mathbb{S} \times ty)\ list\ list$: A list of symbol tables for the stack of local memories.

- *sc.hst* :: $(\mathbb{S} \times ty)$ *list*: The symbol table of the heap memory frame.

We will use the notation $sc(mc)$ to refer to the symbol configuration of a given memory configuration mc.

4.2.2 Program Rest

The program rest keeps track of the statements which still have to be executed.[2] It is initialized with the body of the main function and grows or shrinks depending on the execution of the program. Formally, the program rest is represented by a $C0$ statement of type *stmt*.

4.2.3 Configuration

Now, we can introduce configurations of the $C0$ small-step semantics. A configuration c of the $C0$ small-step semantics is formalized by the record $conf_{\text{C0}}$ consisting of two components:

- the memory configuration *c.mem* :: *memconf* and
- the program rest *c.prog* :: *stmt*.

4.3 Expression Evaluation

In this section we will define the evaluation of expressions in the $C0$ small-step semantics. This definition will slightly differ from the formal definition of expression evaluation in Isabelle / HOL.

Isabelle In Isabelle, we have defined a single function `eval` for the evaluation of expressions. This function takes a type name environment, a memory configuration, and an expression and returns a (option of a) record *ds* of type `data_slice` which has five components: (i) the left value of the expression, (ii) its type, (iii) a flag which determines whether the expression represents a memory object or if it is a literal or a unary or binary operator,[3] (iv) a flag which indicates if the expression is initialized, and (v) the value of the expression. Additionally, we have defined two functions which compute the same result as the components ii and iii of the *eval* function. The main advantage of these additional functions is that they do not take a complete memory configuration as parameter but just the symbol configuration; this allows to conclude automatically that their result is independent of the current memory content whereas this would be a proof obligation for the corresponding components of the *eval* function.

In this thesis, we define five *separate* functions, each computing one component of the `eval` function, instead of one monolithic function. The definitions of these functions are mutually recursive, which is easy to handle on paper but makes induction proofs in Isabelle much harder. The formalization of

[2] In [NN99] the program rest is called *code component* of the configuration.

[3] In Isabelle, this component is called `intermediate` and is set to *false* if the expression is a memory object and to *true* otherwise. In this thesis, we represent it by a predicate *mobj* which checks whether the expression represents a memory object or not. The value of *mobj* corresponds to the inverted value of the component `intermediate`.

expression evaluation via the monolithic `eval` function, however, is possible without mutual recursion.

4.3.1 Address of G-Variables

In this section we define several functions and predicates which compute among others type and address of g-variables. Let te be a type name environment and sc a symbol configuration in the remainder of this section.

Remark For simplicity we will often omit constant parameters like te and sc in the following formulas when the meaning is clear from the context.

Definition 4.15 (Type of g-variables) We define the type of a g-variable by the inductive function $ty_{\mathrm{g}} : \mathit{symbolconf} \times \mathit{gvarT} \mapsto \mathit{ty}$. For the base cases we get the type directly from the corresponding symbol table.

$$\begin{aligned} ty_{\mathrm{g}}(sc, gvar_{\mathrm{gm}}(x)) &= type_{\mathrm{v}}(sc.gst, x) \\ ty_{\mathrm{g}}(sc, gvar_{\mathrm{lm}}(i, x)) &= type_{\mathrm{v}}(sc.lst!i, x) \\ ty_{\mathrm{g}}(sc, gvar_{\mathrm{hm}}(i)) &= snd(sc.hst!i) \end{aligned}$$

For array and structure accesses we use underspecified definitions.

$$ty_{\mathrm{g}}(sc, gvar_{\mathrm{arr}}(g, i)) = \begin{cases} t & \text{if } ty_{\mathrm{g}}(g) = Arr_{\mathrm{T}}(t, j) \\ \mathit{undef} & \text{otherwise} \end{cases}$$

$$ty_{\mathrm{g}}(sc, gvar_{\mathrm{str}}(g, x)) = \begin{cases} the(\mathit{map\text{-}of}(c, x)) & \text{if } ty_{\mathrm{g}}(g) = Str_{\mathrm{T}}(c) \\ \mathit{undef} & \text{otherwise} \end{cases}$$

Definition 4.16 (Memory name of a g-variable) We define the memory name of a g-variable by the function $mem_{\mathrm{g}} : \mathit{gvar} \mapsto \mathit{memname}$.

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

Definition 4.17 (Named g-variables) We define the predicate $named_{\mathrm{g}} :: \mathit{gvar} \mapsto \mathbb{B}$ to refer to g-variables which are not located in the heap memory.

$$named_{\mathrm{g}}(g) = (mem_{\mathrm{g}}(g) = gm \vee \exists i : mem_{\mathrm{g}}(g) = lm(i))$$

Definition 4.18 (Base address of g-variables) We define the base address of a g-variable by the function $ba_{\mathrm{g}} : \mathit{symbolconf} \times \mathit{gvarT} \mapsto \mathbb{N}$.

$$\begin{aligned} ba_{\mathrm{g}}(sc, gvar_{\mathrm{gm}}(x)) &= ba_{\mathrm{v}}(sc.gst, x) \\ ba_{\mathrm{g}}(sc, gvar_{\mathrm{lm}}(i, x)) &= ba_{\mathrm{v}}(sc.lst!i, x) \end{aligned}$$

Heap variables are not named, so we cannot use the function ba_{v} to compute their base address. Instead, we add the size of all preceding heap variables.

$$ba_{\mathrm{g}}(sc, gvar_{\mathrm{hm}}(i)) = \sum_{j=0}^{i-1} size_{\mathrm{t}}(snd(sc.hst!j))$$

For array and structure access we have again incomplete definitions.

$$ba_{\mathrm{g}}(sc, gvar_{\mathrm{arr}}(g,i)) = \begin{cases} ba_{\mathrm{g}}(g) + i \cdot size_{\mathrm{t}}(t) & \text{if } ty_{\mathrm{g}}(g) = Arr_{\mathrm{T}}(t,j) \\ undef & \text{otherwise} \end{cases}$$

For structure access, we exploit that the list of structure components looks exactly like a symbol table. Thus, we can use the function ba_{v}, which computes the base address of a variable in a symbol table, also for the offset of a component inside a structure.

$$ba_{\mathrm{g}}(sc, gvar_{\mathrm{str}}(g,x)) = \begin{cases} ba_{\mathrm{g}}(g) + the(ba_{\mathrm{v}}(scl,x)) & \text{if } ty_{\mathrm{g}}(g) = Str_{\mathrm{T}}(scl) \\ undef & \text{otherwise} \end{cases}$$

Definition 4.19 (Overlapping g-variables) We say that two g-variables g and g' overlap if the following formula is fulfilled.

$$\begin{aligned} overlap_{\mathrm{g}}(sc,g,g') = & mem_{\mathrm{g}}(g) = mem_{\mathrm{g}}(g') \\ & \wedge \quad (ba_{\mathrm{g}}(sc,g), size_{\mathrm{t}}(ty_{\mathrm{g}}(sc,g))) \not\asymp \\ & \quad\; (ba_{\mathrm{g}}(sc,g'), size_{\mathrm{t}}(ty_{\mathrm{g}}(sc,g'))) \end{aligned}$$

Definition 4.20 (Initialized g-variables) We call a g-variable initialized if its root g-variable is in the set of initialized variables of the corresponding memory frame; g-variables in the heap are initialized by definition. We define inductively the predicate $initialized_{\mathrm{g}} : memconf \times gvar \mapsto \mathbb{B}$.

$$\begin{aligned} initialized_{\mathrm{g}}(mc, gvar_{\mathrm{gm}}(x)) &= x \in mc.gm.init \\ initialized_{\mathrm{g}}(mc, gvar_{\mathrm{lm}}(i,x)) &= x \in mc.lm!i.init \\ initialized_{\mathrm{g}}(mc, gvar_{\mathrm{hm}}(i)) &= true \\ initialized_{\mathrm{g}}(mc, gvar_{\mathrm{arr}}(g,i)) &= initialized_{\mathrm{g}}(mc, g) \\ initialized_{\mathrm{g}}(mc, gvar_{\mathrm{str}}(g,x)) &= initialized_{\mathrm{g}}(mc, g) \end{aligned}$$

4.3.2 Reading from the Memory

To denote reading of memory cells from the memory of the $C0$ configuration we use the following abbreviation.

Definition 4.21 (Reading memory cells) Let mc be a memory configuration, n a memory name, and a and b natural numbers. We define the content of the memory n in the range from a to b by case distinction on the memories name:

$$mc_n\,[a:b] = \begin{cases} mc.gm.ct[a:b] & \text{if } n = gm \\ mc.lm!i.ct[a:b] & \text{if } n = lm(i) \\ mc.hm.ct[a:b] & \text{if } n = hm \end{cases}$$

For convenience, we also define the following syntax for the content of the memory n in the range from a to $a + l$:

$$mc_n\,[a, l] = mc_n\,[a : a + l - 1]$$

Definition 4.22 (Value of a g-variable) Let mc a memory configuration, and g a g-variable. We define the value of g-variable g by

$$value_{\mathrm{g}}(mc, g) = mc_{mem_{\mathrm{g}}(g)}\left[ba_{\mathrm{g}}(sc(mc), g), size_{\mathrm{t}}(ty_{\mathrm{g}}(sc(mc), g))\right]$$

4.3.3 Semantics of Operators

In this section we will define the semantics of the unary and binary operators of $C0$. For operators, only the value and type of the result are of interest, so we will not define the left value, the memory object flag, or the initialized flag in this section.

For unary operators we will define

$$\begin{aligned} value_{\circ_1} &: unop \times mcell_{\mathrm{C0}\perp} \mapsto mcell_{\mathrm{C0}\perp} \\ type_{\circ_1} &: unop \times ty \mapsto ty \end{aligned}$$

and for binary operators

$$\begin{aligned} value_{\circ_2} &: binop \times mcell_{\mathrm{C0}\perp} \times mcell_{\mathrm{C0}\perp} \mapsto mcell_{\mathrm{C0}\perp} \\ type_{\circ_2} &: binop \times ty \times ty \mapsto ty. \end{aligned}$$

The functions which compute the values of operator application use option types to support operators like division or modulo which compute partial functions. If one of the inputs of the functions is *None* the result will also be *None* by default (the only exception are the lazy binary operators from Section 4.3.3). In the following we only define the remaining cases.

Arithmetic Operators

$C0$ supports arithmetic operators for integer and unsigned integer operands. Arithmetic operations on character operands are not supported.

Unary minus is only supported for integer operands and defined by

$$\begin{aligned} value_{\circ_1}(minus, \lfloor Int(a)\rfloor) &= \begin{cases} \lfloor Int(a)\rfloor & \text{if } a = -2^{31} \\ \lfloor Int(-a)\rfloor & \text{otherwise} \end{cases} \\ type_{\circ_1}(minus, Int_{\mathrm{T}}) &= Int_{\mathrm{T}} \end{aligned}$$

Addition, subtraction, and multiplication for integer and unsigned integer operands are defined by

$$\begin{aligned} value_{\circ_2}(add, \lfloor Int(a)\rfloor, \lfloor Int(b)\rfloor) &= \lfloor Int(a + b \bmod_{\mathrm{s}} 2^{31})\rfloor \\ value_{\circ_2}(add, \lfloor Nat(a)\rfloor, \lfloor Nat(b)\rfloor) &= \lfloor Nat(a + b \bmod_{\mathrm{u}} 2^{32})\rfloor \\ value_{\circ_2}(sub, \lfloor Int(a)\rfloor, \lfloor Int(b)\rfloor) &= \lfloor Int(a - b \bmod_{\mathrm{s}} 2^{31})\rfloor \\ value_{\circ_2}(sub, \lfloor Nat(a)\rfloor, \lfloor Nat(b)\rfloor) &= \lfloor Nat(a - b \bmod_{\mathrm{u}} 2^{32})\rfloor \\ value_{\circ_2}(mult, \lfloor Int(a)\rfloor, \lfloor Int(b)\rfloor) &= \lfloor Int(a \cdot b \bmod_{\mathrm{s}} 2^{31})\rfloor \\ value_{\circ_2}(mult, \lfloor Nat(a)\rfloor, \lfloor Nat(b)\rfloor) &= \lfloor Nat(a \cdot b \bmod_{\mathrm{u}} 2^{32})\rfloor \end{aligned}$$

Division and modulo are only defined for unsigned operands. If the divisor is zero we define the result of division and modulo to be *None*.

$$value_{\circ_2}(div, \lfloor Nat(a) \rfloor, \lfloor Nat(b) \rfloor) = \begin{cases} None & \text{if } b = 0 \\ \lfloor Nat(a \text{ div } b \bmod_u 2^{32}) \rfloor & \text{otherwise} \end{cases}$$

$$value_{\circ_2}(mod, \lfloor Nat(a) \rfloor, \lfloor Nat(b) \rfloor) = \begin{cases} None & \text{if } b = 0 \\ \lfloor Nat(a \bmod b \bmod_u 2^{32}) \rfloor & \text{otherwise} \end{cases}$$

For all binary arithmetic operators b the type of the result equals the type of the first operand: $type_{\circ_2}(b, t_1, t_2) = t_1$.

Comparison Operators

We support the usual comparison operators (cf. Table 4.3) for operands of basic types. Additionally, we support tests for equality and inequality for pointers. The tests have the usual semantics; thus, we omit formal definitions here. For all comparision operators c we define the type of the result by $type_{\circ_2}(c, t_1, t_2) = Bool_T$.

Bitwise Operators

For the formalization of the $C0$ semantics, we had the choice to model the basic types Int_T, $Unsigned_T$, and $Char_T$ as *bit vectors*, which is close to the definitions in the VAMP processor [BJK+03, BJK+06] and thus would simplify the compiler verification, or as *numbers*, which is closer to the usual usage of such types in high-level languages as C and thus would simplify the verification of $C0$ programs. Obviously, if enough $C0$ programs have to be verified the additional one-time effort for compiler correctness proof pays off. Verification of non-trivial $C0$ programs plays a major role in the Verisoft project thus we decided to represent the basic types as numbers. This decision makes the definition of bitwise operations more complicated: we have to convert the numbers to bit vectors before applying the bitwise operations and back to numbers afterwards.

For bitwise negation and for all bitwise binary operators $\circ$ we define the result type as the type of the first operand. We set $type_{\circ_1}(not, t) = t$ and $type_{\circ_2}(\circ, t_1, t_2) = t_1$. We define the value of bitwise negation by

$$\begin{aligned} value_{\circ_1}(not, \lfloor Int(a) \rfloor) &= \lfloor Int([\neg_b(two_{32}(a))]) \rfloor \\ value_{\circ_1}(not, \lfloor Char(a) \rfloor) &= \lfloor Char([\neg_b(two_8(a))]) \rfloor \\ value_{\circ_1}(not, \lfloor Nat(a) \rfloor) &= \lfloor Nat(\langle \neg_b(bin_{32}(a)) \rangle) \rfloor \end{aligned}$$

The values of applying the binary operators bitwise *and*, bitwise *or*, and bitwise *exclusive or* are defined by

$$\begin{aligned} value_{\circ_2}(or_b, \lfloor Int(a) \rfloor, \lfloor Int(b) \rfloor) &= \lfloor Int([two_{32}(a) \vee_b two_{32}(b)]) \rfloor \\ value_{\circ_2}(or_b, \lfloor Char(a) \rfloor, \lfloor Char(b) \rfloor) &= \lfloor Char([two_8(a) \vee_b two_8(b)]) \rfloor \\ value_{\circ_2}(or_b, \lfloor Nat(a) \rfloor, \lfloor Nat(b) \rfloor) &= \lfloor Nat(\langle bin_{32}(a) \vee_b bin_{32}(b) \rangle) \rfloor \end{aligned}$$

$$\begin{aligned}
value_{\circ_2}(and_{\mathrm{b}}, \lfloor Int(a) \rfloor, \lfloor Int(b) \rfloor) &= \lfloor Int([two_{32}(a) \wedge_{\mathrm{b}} two_{32}(b)]) \rfloor \\
value_{\circ_2}(and_{\mathrm{b}}, \lfloor Char(a) \rfloor, \lfloor Char(b) \rfloor) &= \lfloor Char([two_{8}(a) \wedge_{\mathrm{b}} two_{8}(b)]) \rfloor \\
value_{\circ_2}(and_{\mathrm{b}}, \lfloor Nat(a) \rfloor, \lfloor Nat(b) \rfloor) &= \lfloor Nat(\langle bin_{32}(a) \wedge_{\mathrm{b}} bin_{32}(b) \rangle) \rfloor
\end{aligned}$$

$$\begin{aligned}
value_{\circ_2}(xor_{\mathrm{b}}, \lfloor Int(a) \rfloor, \lfloor Int(b) \rfloor) &= \lfloor Int([two_{32}(a) \otimes_{\mathrm{b}} two_{32}(b)]) \rfloor \\
value_{\circ_2}(xor_{\mathrm{b}}, \lfloor Char(a) \rfloor, \lfloor Char(b) \rfloor) &= \lfloor Char([two_{8}(a) \otimes_{\mathrm{b}} two_{8}(b)]) \rfloor \\
value_{\circ_2}(xor_{\mathrm{b}}, \lfloor Nat(a) \rfloor, \lfloor Nat(b) \rfloor) &= \lfloor Nat(\langle bin_{32}(a) \otimes_{\mathrm{b}} bin_{32}(b) \rangle) \rfloor
\end{aligned}$$

Logical Shift Operators

$C0$ does not support arithmetic shifts; only logical shifts are possible. For shift operators we do not require the types for left and right operands to be equal. Thus, let $x \in \{Int(b), Char(b), Nat(b)\}$ be the shift distance. We define the semantics of the logical shift operators in $C0$ by

$$\begin{aligned}
value_{\circ_2}(shift_{\mathrm{l}}, \lfloor Int(a) \rfloor, \lfloor x \rfloor) &= \lfloor Int([two_{32}(a) \ll_{\mathrm{l}} (b \bmod_{\mathrm{u}} 2^5)]) \rfloor \\
value_{\circ_2}(shift_{\mathrm{l}}, \lfloor Char(a) \rfloor, \lfloor x \rfloor) &= \lfloor Char([two_{8}(a) \ll_{\mathrm{l}} (b \bmod_{\mathrm{u}} 2^5)]) \rfloor \\
value_{\circ_2}(shift_{\mathrm{l}}, \lfloor Nat(a) \rfloor, \lfloor x \rfloor) &= \lfloor Nat(\langle bin_{32}(a) \ll_{\mathrm{l}} (b \bmod_{\mathrm{u}} 2^5) \rangle) \rfloor
\end{aligned}$$

$$\begin{aligned}
value_{\circ_2}(shift_{\mathrm{r}}, \lfloor Int(a) \rfloor, \lfloor x \rfloor) &= \lfloor Int([two_{32}(a) \gg_{\mathrm{l}} (b \bmod_{\mathrm{u}} 2^5)]) \rfloor \\
value_{\circ_2}(shift_{\mathrm{r}}, \lfloor Char(a) \rfloor, \lfloor x \rfloor) &= \lfloor Char([two_{8}(a) \gg_{\mathrm{l}} (b \bmod_{\mathrm{u}} 2^5)]) \rfloor \\
value_{\circ_2}(shift_{\mathrm{r}}, \lfloor Nat(a) \rfloor, \lfloor x \rfloor) &= \lfloor Nat(\langle bin_{32}(a) \gg_{\mathrm{l}} (b \bmod_{\mathrm{u}} 2^5) \rangle) \rfloor
\end{aligned}$$

The type of the result is determined by the type of the left operand. We define for logical shift operators l the result type by $type_{\circ_2}(l, t_1, t_2) = t_1$.

Logical Operators

$C0$ supports logical operators only for operands of boolean type. The result type of logical negation and binary logical operators $\circ$ is always boolean: $type_{\circ_1}(neg, Bool_{\mathrm{T}}) = Bool_{\mathrm{T}}$ and $type_{\circ_2}(\circ, Bool_{\mathrm{T}}, Bool_{\mathrm{T}}) = Bool_{\mathrm{T}}$. We define the value of logical negation by

$$value_{\circ_1}(neg, \lfloor Bool(a) \rfloor) = \lfloor Bool(\neg a) \rfloor$$

The binary logical operators are defined using lazy evaluation. Thus, if the result is already determined by the first operand the second operand will not be evaluated; in this case the second operand cannot generate run time errors.[4] We define

$$value_{\circ_2}(and_{\mathrm{l}}, \lfloor Bool(a) \rfloor, \lfloor Bool(b) \rfloor) = \begin{cases} \lfloor Bool(false) \rfloor & \text{if } a = false \\ \lfloor Bool(b) \rfloor & \text{if } a = true \end{cases}$$

$$value_{\circ_2}(or_{\mathrm{l}}, \lfloor Bool(a) \rfloor, \lfloor Bool(b) \rfloor) = \begin{cases} \lfloor Bool(true) \rfloor & \text{if } a = true \\ \lfloor Bool(b) \rfloor & \text{if } a = false \end{cases}$$

[4] This is often useful to simplify conditions, as in the following code snippet where the value of p is only read if p is not the null pointer: '**if** ((p != NULL) && (p->value==5))'

Type Conversion

We define three function for conversion between $C0$ types

$$\begin{aligned} value_{\circ_1}(to_{int}, \lfloor Int(a) \rfloor) &= \lfloor a \rfloor \\ value_{\circ_1}(to_{int}, \lfloor Char(a) \rfloor) &= \lfloor a \rfloor \\ value_{\circ_1}(to_{int}, \lfloor Nat(a) \rfloor) &= \lfloor a \bmod_{\mathrm{s}} 2^{31} \rfloor \end{aligned}$$

$$\begin{aligned} value_{\circ_1}(to_{unsgnd}, \lfloor Int(a) \rfloor) &= \lfloor a \bmod_{\mathrm{u}} 2^{32} \rfloor \\ value_{\circ_1}(to_{unsgnd}, \lfloor Char(a) \rfloor) &= \lfloor a \bmod_{\mathrm{u}} 2^{32} \rfloor \\ value_{\circ_1}(to_{unsgnd}, \lfloor Nat(a) \rfloor) &= \lfloor a \rfloor \end{aligned}$$

$$\begin{aligned} value_{\circ_1}(to_{char}, \lfloor Int(a) \rfloor) &= \lfloor a \bmod_{\mathrm{s}} 2^{7} \rfloor \\ value_{\circ_1}(to_{char}, \lfloor Char(a) \rfloor) &= \lfloor a \rfloor \\ value_{\circ_1}(to_{char}, \lfloor Nat(a) \rfloor) &= \lfloor a \bmod_{\mathrm{s}} 2^{7} \rfloor \end{aligned}$$

For type conversion the definition of the result type is obvious

$$\begin{aligned} type_{\circ_1}(to_{int}, t) &= Int_{\mathrm{T}} \\ type_{\circ_1}(to_{unsgnd}, t) &= Unsigned_{\mathrm{T}} \\ type_{\circ_1}(to_{char}, t) &= Char_{\mathrm{T}} \end{aligned}$$

4.3.4 Evaluation Functions for Expressions

In this section we define the evaluation functions for expressions in the $C0$ small-step semantics using the definitions from the previous sections. We define five functions for expression evaluation:

The function *type* computes the type of an expression. If the memory of the $C0$ configuration fulfills certain type correctness requirements (cf. Section 5.3) this function corresponds to the field `ds.type` for a data slice in Isabelle.

The function *leval* computes the address of an expression. It corresponds to the field `ds.lval` for a data slice in Isabelle. If the expression has no address, e.g., for literals, *leval* is undefined.

The function *reval* computes the value of an expression. It corresponds to the field `ds.data` for a data slice in Isabelle. If *reval* is applied to an uninitialized expression it returns None.

The function *mobj* tests whether an expression represents a memory object ($mobj = true$) or not ($mobj = false$). It corresponds to the (negated) field `ds.intermediate` for a data slice in Isabelle.

The function *initialized* tests whether an expression is initialized. It corresponds to the field `ds.initialized` for a data slice in Isabelle.

The signature of these functions is as follows. Observe that instead of the complete memory configuration the functions *type* and *mobj* take only two symbol tables – for the global and the local variables – as parameters. These symbol tables are usually the global and topmost local symbol table of the

current memory configuration.

$$\begin{aligned} \mathit{leval} &: \mathit{tenv} \times \mathit{memconf} \times \mathit{expr} \mapsto \mathit{gvar}_{\perp} \\ \mathit{type} &: \mathit{tenv} \times (\mathbb{S} \times \mathit{ty})\ \mathit{list} \times (\mathbb{S} \times \mathit{ty})\ \mathit{list} \times \mathit{expr} \mapsto \mathit{ty}_{\perp} \\ \mathit{mobj} &: \mathit{tenv} \times (\mathbb{S} \times \mathit{ty})\ \mathit{list} \times (\mathbb{S} \times \mathit{ty})\ \mathit{list} \times \mathit{expr} \mapsto \mathbb{B} \\ \mathit{initialized} &: \mathit{tenv} \times \mathit{memconf} \times \mathit{expr} \mapsto \mathbb{B} \\ \mathit{reval} &: \mathit{tenv} \times \mathit{memconf} \times \mathit{expr} \mapsto (\mathbb{N} \mapsto \mathit{mcell}_{\mathrm{C0}})_{\perp} \end{aligned}$$

Remark For the first memory cell of a value $v :: \mathbb{N} \mapsto \mathit{mcell}_{\mathrm{C0}}$ we will sometimes just write v instead of $v(0)$ when the context is clear, i.e., when it is obvious that we just refer to a single memory cell.

For the remainder of this section we denote by te a type name environment, mc a memory configuration, and by gst and lst global and local symbol tables, respectively.

Literals

We start with the evaluation of literals which also includes aggregate literals (cf. Section 3.1.3). The definition of memory object and initialized flags and of the left value is simple for expressions of primitive literals l.

$$\begin{aligned} \mathit{mobj}(te, gst, lst, \mathit{Lit}(l)) &= \mathit{false} \\ \mathit{initialized}(te, mc, \mathit{Lit}(l)) &= \mathit{true} \\ \mathit{leval}(te, mc, \mathit{Lit}(l)) &= \mathit{undef} \end{aligned}$$

We will not define these three functions for aggregate literals because they are only used in the special statement for assignments of aggregate literals (cf. Section 3.1.3). There, these functions are not needed.

The type of literal expressions and of aggregate literals is defined in two steps.

Definition 4.23 (Type of literals) First we define the function $\mathit{type}_{\mathrm{lit}} :: \mathit{lit} \mapsto \mathit{ty}$ which computes the type of primitive literals.

$$\begin{aligned} \mathit{type}_{\mathrm{lit}}(\mathit{Bool}(b)) &= \mathit{Bool}_{\mathrm{T}} \\ \mathit{type}_{\mathrm{lit}}(\mathit{Int}(i)) &= \mathit{Int}_{\mathrm{T}} \\ \mathit{type}_{\mathrm{lit}}(\mathit{Unsigned}(u)) &= \mathit{Unsigned}_{\mathrm{T}} \\ \mathit{type}_{\mathrm{lit}}(\mathit{Char}(c)) &= \mathit{Char}_{\mathrm{T}} \\ \mathit{type}_{\mathrm{lit}}(\mathit{Null}) &= \mathit{Null}_{\mathrm{T}} \end{aligned}$$

Definition 4.24 (Type of aggregate literals) The type of aggregate literals is defined by the recursive function $\mathit{type}_{\mathrm{alit}} :: \mathit{lit}_{\mathrm{a}} \mapsto \mathit{ty}$.

$$\begin{aligned} \mathit{type}_{\mathrm{alit}}(\mathit{ALPrim}(l)) &= \mathit{type}_{\mathrm{lit}}(l) \\ \mathit{type}_{\mathrm{alit}}(\mathit{ALStruct}(xs)) &= \mathit{Str}_{\mathrm{T}}(\mathit{map}(\mathit{type}_{\mathrm{alit}}, \mathit{map}(\mathit{snd}, xs))) \\ \mathit{type}_{\mathrm{alit}}(\mathit{ALArr}(x\#xs)) &= \mathit{Arr}_{\mathrm{T}}(|x\#xs|, \mathit{type}_{\mathrm{alit}}(x)) \end{aligned}$$

The definitions of the value of literal expressions and aggregate literals follow the same structure.

Definition 4.25 (Value of literal expressions) To compute the value of primitive literals we define the function $reval_{\text{lit}} :: lit \mapsto mcell_{\text{C0}}$.

$$\begin{aligned} reval_{\text{lit}}(Bool(b)) &= Bool(b) \\ reval_{\text{lit}}(Int(i)) &= Int(i) \\ reval_{\text{lit}}(Unsigned(u)) &= Nat(u) \\ reval_{\text{lit}}(Char(c)) &= Char(c) \\ reval_{\text{lit}}(Null) &= Ptr(\bot) \end{aligned}$$

Definition 4.26 (Value of aggregate literals) We define the value of aggregate literals by the recursive function $reval_{\text{alit}} :: lit_{\text{a}} \mapsto mcell_{\text{C0}}\ list$.

$$\begin{aligned} reval_{\text{alit}}(ALPrim(l)) &= [reval_{\text{lit}}(l)] \\ reval_{\text{alit}}(ALStruct([])) &= [] \\ reval_{\text{alit}}(ALStruct(x\#xs)) &= reval_{\text{alit}}(snd(x)) \circ reval_{\text{alit}}(ALStruct(xs)) \\ reval_{\text{alit}}(ALArr([])) &= [] \\ reval_{\text{alit}}(ALArr(x\#xs)) &= reval_{\text{alit}}(x) \circ reval_{\text{alit}}(ALArr(xs)) \end{aligned}$$

The definition of *type* and *reval* for literal expressions is now trivial:

$$\begin{aligned} type(te, gst, lst, Lit(l)) &= \lfloor type_{\text{lit}}(l) \rfloor \\ reval(te, mc, Lit(l)) &= \lfloor reval_{\text{lit}}(l) \rfloor \end{aligned}$$

Variable Access

The *mobj* flag for access to a variable v is obviously defined as

$$mobj(te, gst, lst, Var(v)) = true.$$

In $C0$, local variables shadow / hide global variables of the same name. Thus, for an access to variable v we first have to test whether a local variable of this name exists. If it does we access the local variable. Otherwise, we access the corresponding global variable. If the variable is defined neither in the local nor in the global memory, evaluation fails and we set

$$\begin{aligned} initialized(te, mc, Var(v)) &= false \\ type(te, gst, lst, Var(v)) &= None \\ leval(te, mc, Var(v)) &= None \\ reval(te, mc, Var(v)) &= None \end{aligned}$$

Otherwise, we define type and left evaluation by

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

$$leval(te, mc, Var(v)) = \begin{cases} \lfloor gvar_{\text{lm}}(|mc.lm| - 1, v) \rfloor & \text{if } v \in map(fst, lst_{\text{top}}(mc)) \\ \lfloor gvar_{\text{gm}}(v) \rfloor & \text{otherwise} \end{cases}$$

For the remaining evaluation functions we abbreviate with m the memory which is used for the variable access

$$m = \begin{cases} lm_{\text{top}}(mc) & \text{if } v \in map(fst, lst_{\text{top}}(mc)) \\ mc.gm & \text{otherwise} \end{cases}$$

and define

$$initialized(te, mc, Var(v)) = v \in m.init$$
$$reval(te, mc, Var(v)) = \lfloor m.ct[ba_{\text{v}}(m.st, v), size_{\text{t}}(type_{\text{v}}(m.st, v))] \rfloor$$

Pointer Dereferencing

Now, we define the evaluation of pointer dereferencing $Deref(e)$. By definition, pointers refer to memory object; thus, we set

$$mobj(te, gst, lst, Deref(e)) = true.$$

For the remaing definitions we require

- $initialized(te, mc, e)$, i.e., that e is initialized,
- $type(te, gst(mc), lst_{\text{top}}(mc), e) = \lfloor Ptr_{\text{T}}(tn) \rfloor$, i.e., that e is of pointer type,
- $map\text{-}of(te, tn) = \lfloor t \rfloor$, i.e., that the type name tn is present in the type name environment,
- $reval(te, mc, e) = \lfloor Ptr(p) \rfloor \wedge p \neq \bot$, i.e., that the value of e is a non-null pointer, and finally,
- $mem_{\text{g}}(p) \in \{gm, hm\}$, i.e., that the pointer points to the global memory or to the heap.

If one of these conditions is violated evaluation fails and we set *initialized* to *false* and *type*, *leval*, and *reval* to None. Otherwise, we define:

$$initialized(te, mc, Deref(e)) = initialized_{\text{g}}(mc, p)$$
$$type(te, gst, lst, Deref(e)) = map\text{-}of(te, tn)$$
$$leval(te, mc, Deref(e)) = \lfloor p \rfloor$$
$$reval(te, mc, Deref(e)) = \lfloor value_{\text{g}}(mc, p) \rfloor$$

Address-Of Operator

Now, we define the evaluation of the address-of operator $AddrOf(e)$. The address of an object is not a memory object itself but it is always initialized. Thus, we define

$$mobj(te, gst, lst, AddrOf(e)) = false$$
$$initialized(te, mc, AddrOf(e)) = true$$

As for pointer derefering we have some requirements for the successful evaluation of an address-of operator. We require

- $map\text{-}of(map(flip, te), type(te, gst(mc), lst_{\text{top}}(mc), e)) = \lfloor tn \rfloor$, i.e., that there exists a type name for the type of expression e,

- $mobj(te, gst, lst, e) = true$, i.e., that e is a memory object, and
- $leval(te, mc, e) = \lfloor g \rfloor$, i.e., that left evaluation of e gives some g-variable g.

If one of these requirements is violated we set *type*, *leval*, and *reval* to None. Otherwise, we define

$$\begin{aligned} type(te, gst, lst, AddrOf(e)) &= \lfloor Ptr_{\mathrm{T}}(tn) \rfloor \\ leval(te, mc, AddrOf(e)) &= undef \\ reval(te, mc, AddrOf(e)) &= \lfloor [Ptr(g)] \rfloor \end{aligned}$$

Access to Structure Components

In this section we define the evaluation of an access to a structure component. The memory object and initialized flags of a structure access $Str(e, cn)$ equal the corresponding flags for e.

$$\begin{aligned} mobj(te, gst, lst, Str(e, cn)) &= mobj(te, gst, lst, e) \\ initialized(te, mc, Str(e, cn)) &= initialized(te, mc, e) \end{aligned}$$

For the remaining evaluation functions we require

- that e is a structure, i.e., $type(te, gst, lst, e) = \lfloor Str_{\mathrm{T}}(cl) \rfloor$
- that the component name cn occurs in the compont list cl of this structure, i.e., $map\text{-}of(cl, cn) = \lfloor t \rfloor$, and
- that e can be evaluated, i.e., $leval(te, mc, e) = \lfloor g \rfloor$ and $reval(te, mc, e) = \lfloor v \rfloor$.

If one of these requirements is violated we set *type*, *leval*, and *reval* to None. Otherwise, the relative base address of component cn in the structure is defined, i.e., $ba_{\mathrm{v}}(cl, cn) = \lfloor b \rfloor$, and we define

$$\begin{aligned} type(te, gst(mc), lst_{\mathrm{top}}(mc), Str(e, cn)) &= \lfloor t \rfloor \\ leval(te, mc, Str(e, cn)) &= \lfloor gvar_{\mathrm{str}}(g, cn) \rfloor \\ reval(te, mc, Str(e, cn)) &= \lfloor v[b, size_{\mathrm{t}}(t)] \rfloor \end{aligned}$$

Access to Array Elements

For array access we have instead of a component name cn a second sub expression e_{i} which specifies the index of the requested array element. The definition of the memory object and initialized flags is similar to structure access.

$$\begin{aligned} mobj(te, gst, lst, Arr(e_{\mathrm{a}}, e_{\mathrm{i}})) &= mobj(te, gst, lst, e_{\mathrm{a}}) \\ initialized(te, mc, Arr(e_{\mathrm{a}}, e_{\mathrm{i}})) &= initialized(te, mc, e_{\mathrm{a}}) \wedge initialized(te, mc, e_{\mathrm{i}}) \end{aligned}$$

For the remaining evaluation functions we require

- that the array expression e_{a} can be evaluated, i.e., $leval(te, mc, e_{\mathrm{a}}) = \lfloor g_{\mathrm{a}} \rfloor$ and $reval(te, mc, e_{\mathrm{a}}) = \lfloor v_{\mathrm{a}} \rfloor$,

- that the index expression e_i can be evaluated to some numerical value

$$\begin{aligned} & \mathit{reval}(te, mc, e_\mathrm{i}) = \lfloor \mathit{Int}(i) \rfloor \\ \vee\ & \mathit{reval}(te, mc, e_\mathrm{i}) = \lfloor \mathit{Nat}(i) \rfloor \\ \vee\ & \mathit{reval}(te, mc, e_\mathrm{i}) = \lfloor \mathit{Char}(i) \rfloor \end{aligned}$$

- that $\mathit{type}(te, \mathit{gst}(mc), \mathit{lst}_\mathrm{top}(mc), e_\mathrm{a}) = \lfloor \mathit{Arr}_\mathrm{T}(l, t) \rfloor$, i.e., that e_a is an array with l elements of type t,
- and that the index is in range, i.e., $i < l$.

If one of these requirements is violated we set *type*, *leval*, and *reval* to None. Otherwise, we define:

$$\begin{aligned} \mathit{type}(te, \mathit{gst}, \mathit{lst}, \mathit{Arr}(e_\mathrm{a}, e_\mathrm{i})) &= \lfloor t \rfloor \\ \mathit{leval}(te, mc, \mathit{Arr}(e_\mathrm{a}, e_\mathrm{i})) &= \lfloor \mathit{gvar}_\mathrm{arr}(g_\mathrm{a}, i) \rfloor \\ \mathit{reval}(te, mc, \mathit{Arr}(e_\mathrm{a}, e_\mathrm{i})) &= \lfloor v_\mathrm{a}[i \cdot \mathit{size}_\mathrm{t}(t), \mathit{size}_\mathrm{t}(t)] \rfloor \end{aligned}$$

Unary Operators

For the definition of expression evaluation for unary operators we use the definitions from Section 4.3.3. For an expression $\mathit{UnOp}(\circ_1, e)$ which applies a unary operator $\circ_1$ to an expression e we define

$$\begin{aligned} \mathit{mobj}(te, \mathit{gst}, \mathit{lst}, \mathit{UnOp}(\circ_1, e)) &= \mathit{false} \\ \mathit{initialized}(te, mc, \mathit{UnOp}(\circ_1, e)) &= \mathit{initialized}(te, mc, e) \end{aligned}$$

For the definition of the remaining evaluation functions we require

- $\mathit{reval}(te, mc, e) = \lfloor v_\mathrm{e} \rfloor$, i.e., that e can be evaluated and
- $\mathit{value}_{\circ_1}(\circ_1, v_\mathrm{e}) = \lfloor v \rfloor$, i.e., that applying the unary operator to this value returns a meaningful result.

If one of these requirements is violated we set *type*, *leval*, and *reval* to *None*. Otherwise we define

$$\begin{aligned} \mathit{type}(te, \mathit{gst}, \mathit{lst}, \mathit{UnOp}(\circ_1, e)) &= \left\lfloor \mathit{type}_{\circ_1}(\circ_1, \mathit{the}(\mathit{type}(te, \mathit{gst}, \mathit{lst}, e))) \right\rfloor \\ \mathit{leval}(te, mc, \mathit{UnOp}(\circ_1, e)) &= \mathit{undef} \\ \mathit{reval}(te, mc, \mathit{UnOp}(\circ_1, e)) &= \lfloor [v] \rfloor \end{aligned}$$

Binary and Lazy Binary Operators

The definition of the evaluation functions for binary and lazy operators follows the scheme from the unary operators. Let e be a binary or a lazy binary expression, i.e., $e = \mathit{BinOp}(\circ_2, e_1, e_2)$ or $e = \mathit{LazyBinOp}(\circ_2, e_1, e_2)$. We define the memory object and initialized flags by

$$\begin{aligned} \mathit{mobj}(te, \mathit{gst}, \mathit{lst}, e) &= \mathit{false} \\ \mathit{initialized}(te, mc, e) &= \mathit{initialized}(te, mc, e_1) \wedge \mathit{initialized}(te, mc, e_2) \end{aligned}$$

For the definition of the remaining evaluation functions we require that

- e_1 and e_2 can be evaluated to values, i.e., $reval(te, mc, e_1) = \lfloor v_1 \rfloor$ and $reval(te, mc, e_2) = \lfloor v_2 \rfloor$,
- that applying the binary operator to these values returns a meaningful result, i.e., $value_{\circ_2}(\circ_2, v_1, v_2) = \lfloor v \rfloor$,
- and that for both sub expressions we can compute a type

$$type(te, gst(mc), lst_{\text{top}}(mc), e_1) = \lfloor t_1 \rfloor$$
$$type(te, gst(mc), lst_{\text{top}}(mc), e_2) = \lfloor t_2 \rfloor$$

If one of these requirements is violated we set *type*, *leval*, and *reval* to *None*. Otherwise we define

$$type(te, gst, lst, e) = \lfloor type_{\circ_2}(\circ_2, t_1, t_2) \rfloor$$
$$leval(te, mc, e) = undef$$
$$reval(te, mc, e) = \lfloor [v] \rfloor$$

4.4 Execution of C0 Programs

In this section we define the execution of $C0$ programs. We start in Section 4.4.1 with the definition of the initial configuration for a given $C0$ program. In Section 4.4.2 we formalize a function for memory updates which is used in Section 4.4.3 to define the transition function of the $C0$ small-step semantics.

4.4.1 Initial Configuration

In $C0$ the values of global and heap variables are initialized with *default* values. In this section we define these default values (sometimes also called *initial* values) for all $C0$ types. Then, we use the initial values to define the initial configuration for a given $C0$ program. This initial configuration depends only on the function table and the global symbol table.

Definition 4.27 (Initial memory frame) Before its initialization with default values, the initial content of a memory frame for a given symbol table st is defined as follows.

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

Definition 4.28 (Initial values) We introduce the function $init_{\text{val}} :: ty \mapsto mcell_{\text{C0}}\ list$ which defines default values for all $C0$ types by structural induction.

$$init_{\text{val}}(Bool_{\text{T}}) = [Bool(false)]$$
$$init_{\text{val}}(Int_{\text{T}}) = [Int(0)]$$
$$init_{\text{val}}(Char_{\text{T}}) = [Char(0)]$$
$$init_{\text{val}}(Unsigned_{\text{T}}) = [Nat(0)]$$

$$
\begin{aligned}
init_{\text{val}}(Ptr_{\text{T}}(tn)) &= [Ptr(\bot)] \\
init_{\text{val}}(Null_{\text{T}}) &= [Ptr(\bot)] \\
init_{\text{val}}(Arr_{\text{T}}(n,t)) &= replicate(n, init_{\text{val}}(t)) \\
init_{\text{val}}(Str_{\text{T}}([\,])) &= [\,] \\
init_{\text{val}}(Str_{\text{T}}((cn,t)\#xs)) &= init_{\text{val}}(t) \circ init_{\text{val}}(Str_{\text{T}}(xs))
\end{aligned}
$$

Definition 4.29 (Initial symbol tables) We introduce the function $init_{\text{st}} :: (\mathbb{S} \times ty)\ list \mapsto mcell_{\text{C0}}\ list$. The function computes the initialized content for a given symbol table by concatenation of the initial values of all its variables.

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

Definition 4.30 (Initializing variables) We define the function $init_{\text{vars}} :: mframe \mapsto mframe$ which initializes all variables of a given memory frame. The function initializes the content of a memory and additionally adds all variables of the memory to the set of initialized variables.

$$
init_{\text{vars}}(m) = m \left[\begin{array}{l} ct := \lambda\, i.\ init_{\text{st}}(m.st)!i \\ init := \{map(fst, m.st)\} \end{array} \right]
$$

Next, we define the initial memory configuration of a $C0$ machine. The initial memory depends on the function table and the symbol table for the global variables. The global variables are zero-initialized using the function $init_{\text{vars}}$, the local memory stack is initialized with a single frame for the main function, and the heap memory is initially empty.

Definition 4.31 (Initial memory configuration) The initial memory configuration of a $C0$ program is defined by the function $init_{\text{mc}} :: functableT \times (\mathbb{S} \times ty)\ list \mapsto memconf_{\,\bot}$. Let ft be a function table and gst a symbol table with the global variables. Further, let mf be the main function of the program, i.e., $map\text{-}of(ft, "main") = \lfloor mf \rfloor$. If there is no main function, i.e., $map\text{-}of(ft, "main") = None$, we set $init_{\text{mc}}(ft, gst) = None$. Otherwise, we define

$$
init_{\text{mc}}(ft, gst) = \left\lfloor \left[\begin{array}{l} gm = init_{\text{vars}}(init_{\text{mem}}(gst)) \\ lm = [(init_{\text{mem}}(mf.params \circ mf.lvars), undef)] \\ hm = [init_{\text{mem}}([\,])] \end{array} \right] \right\rfloor
$$

Now, we can define the initial configuration of a $C0$ machine.

Definition 4.32 (Initial configuration) Let ft be a function table and gst the symbol table of the global variables. We define the function $init_{\text{conf}} :: functableT \times (\mathbb{S} \times ty)\ list \mapsto conf_{\,\text{C0}\,\bot}$ which generates the initial configuration of a $C0$ machine. If there is no main function, i.e., $map\text{-}of(ft, "main") = None$, we set $init_{\text{conf}}(ft, gst) = None$. Otherwise, we abbreviate with $mf = the(map\text{-}of(ft, "main"))$ the main function of the given function table and initialize the program rest of the $C0$ machine with the body of the main function without the final return statement.

$$
init_{\text{conf}}(ft, gst) = \left\lfloor \left[\begin{array}{l} mem = the(init_{\text{mc}}(ft, gst)) \\ prog = remlast(mf.body) \end{array} \right] \right\rfloor
$$

4.4.2 Updating Memory

Before we start defining the $C0$ transition function we introduce a function which updates the memory of a $C0$ configuration. This function will be used in Section 4.4.3 to model the semantics of assignments, function calls, returns, and memory allocation.

Let mc a memory configuration, g a g-variable, and $v :: \mathbb{N} \mapsto mcell_{\mathrm{C0}}$ a value. We define the function $memupd :: memconf \times gvar \times (\mathbb{N} \mapsto mcell_{\mathrm{C0}}) \mapsto memconf_{\perp}$ which updates g-variable g with value v.

In $C0$, partial updates of uninitialized variables are not allowed; thus, we require that g is initialized or it is a root g-variable: $initialized_{\mathrm{g}}(mc, g) \vee root_{\mathrm{g}}(g) = g$. If this requirement is violated, we set $memupd(mc, g, v) = None$.

Otherwise, we define the memory update by case distinction on g's root g-variable. For this, abbreviate the base address of g by $b = ba_{\mathrm{g}}(sc(mc), g)$ and its size by $s = size_{\mathrm{t}}(ty_{\mathrm{g}}(sc(mc), g))$.

- $root_{\mathrm{g}}(g) = gvar_{\mathrm{gm}}(x)$:

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

- $root_{\mathrm{g}}(g) = gvar_{\mathrm{lm}}(i, x)$:

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

- $root_{\mathrm{g}}(g) = gvar_{\mathrm{hm}}(j)$:

$$memupd(mc, g, v) := \lfloor mc \left[\; hm.ct := hm.ct\,([b, s] := v[0, s]) \; \right] \rfloor$$

Observe that we do not update the set of initialized variables for heap variables, because all heap variables are initialized by definition. Thus, the set of initialized variables for the heap memory frame is never used.

Isabelle Observe that for the formal counterpart `mem_update` of this function in Isabelle the value is represented by a data slice. The data slice contains a flag which specifies whether the value of the data slice is initialized and this flag is checked by `mem_update`: if the data slice is not initialized it returns *None*. In this thesis the function *memupd* cannot check if the value is initialized; rather, this test is done explicitely for each use of *memupd*.

4.4.3 Transition Function

Now, we define the transition function δ_{C0} of the $C0$ small-step semantics which models the execution of $C0$ programs. Given a type name environment te and a function table ft, the transition function maps the current $C0$ configuration c to its successor configuration $\delta_{\mathrm{C0}}(te, ft, c)$ or, in case of an error, to *None*.

$$\delta_{\mathrm{C0}} :: tenv \times functableT \times conf_{\mathrm{C0}} \mapsto conf_{\mathrm{C0}\perp}$$

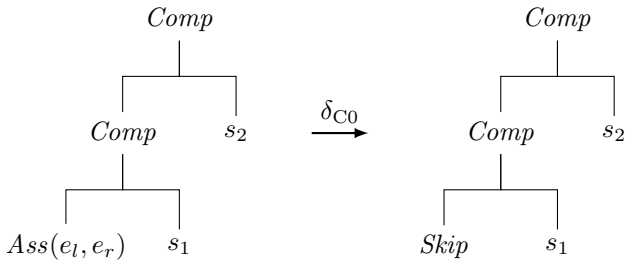


Figure 4.2: Execution of Compound Statements: Real Statement

We define the transition function by induction on the program rest. If the program rest is a compound statement $Comp(s_1, s_2)$, we apply the transition function recursively to s_1. Otherwise, it consists of a single statement and we execute it without recursive execution of δ_{C0}.

Skip

If $c.prog = Skip$ the $C0$ program has terminated. We model termination by a fix point and define $\delta_{C0}(te, ft, c) = \lfloor c \rfloor$.

Statement Composition

For $c.prog = Comp(s_1, s_2)$ we define the execution of the $C0$ program by recursion on the left sub statement s_1. We consider two cases.

If $s_1 = Skip$ the execution of the left sub tree has finished and we set $\delta_{C0}(te, ft, c) = \lfloor c[prog := s_2] \rfloor$, i.e., we execute the skip statement. Otherwise, we define

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

The effect of this recursive definition is that the transition function executes the first *real* (i.e., different from *Comp*) statement in the program rest and replaces it by a skip statement. The structure of the program rest changes only locally at the executed statement. This is illustrated in Figures 4.2 and 4.3.

Conditional

If the program rest is built by some conditional statement, i.e., $c.prog = Ifte(e, s_1, s_2)$, the new program rest will either be s_1 or s_2 depending on the condition e. We require $type(te, gst(c.mem), lst_{top}(c.mem), e) = \lfloor Bool_T \rfloor$ and $initialized(te, c.mem, e)$, i.e., that the condition expression is of boolean type and initialized, and that in can be evaluated, i.e., $reval(te, c.mem, e) = \lfloor v_e \rfloor$. If one of these requirements is violated we set $\delta_{C0}(te, ft, c) = None$. Otherwise

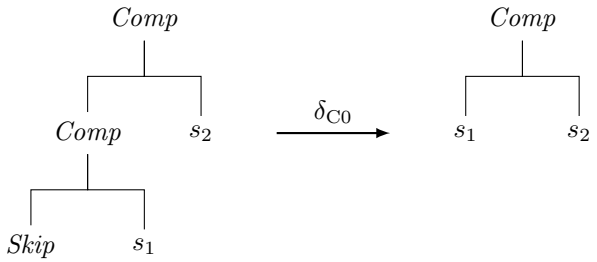


Figure 4.3: Execution of Compound Statements: Skip Statement

we define

$$\delta_{C0}(te, ft, c) = \begin{cases} \lfloor c[prog := s_1] \rfloor & \text{if } v_e = Bool(true) \\ \lfloor c[prog := s_2] \rfloor & \text{otherwise} \end{cases}$$

While Loop

If $c.prog = Loop(e, lb)$ the new program rest will either be *Skip*, if the loop condition evaluates to *false*, or $Comp(lb, Loop(e, lb))$, if the condition is *true*. In the latter case, the loop body *lb* will be executed once before testing the loop condition again. For loops we require as for conditionals $initialized(te, c.mem, e)$ and $type(te, gst(c.mem), lst_{top}(c.mem), e) = \lfloor Bool_T \rfloor$, i.e., that the loop condition is initialized and of boolean type, and that it can be evaluated to some value, i.e., $reval(te, c.mem, e) = \lfloor v_e \rfloor$. If one of these requirements is violated we set $\delta_{C0}(te, ft, c) = None$. Otherwise we define

$$\delta_{C0}(te, ft, c) = \begin{cases} \lfloor c[prog := Comp(lb, Loop(e, lb))] \rfloor & \text{if } v_e = Bool(true) \\ \lfloor c[prog := Skip] \rfloor & \text{otherwise} \end{cases}$$

Assignment

As described in Section 4.1.3, the *C*0 small-step semantics distinguishes two kinds of assignments: normal assignments and assignments of aggregate literals. The latter form is used for the initialization of non-elementary variables in a single step which is necessary for the equivalence proof between the *C*0 small-step semantics and the Hoare logic from [Sch06].

Normal Assignment. Let the program rest consist of a normal assignment statement: $c.prog = Ass(e_l, e_r)$. We require that the left expression e_l can be evaluated to some g-variable g, i.e., $leval(te, c.mem, e_l) = \lfloor g \rfloor$, that the right expression e_r can be evaluated to some value v and is initialized, i.e., $reval(te, c.mem, e_r) = \lfloor v \rfloor$ and $initialized(te, c.mem, e_r)$, and that updating the memory succeeds, i.e., $memupd(c.mem, g, v) = \lfloor mc' \rfloor$.

If one of these conditions is violated we set $\delta_{C0}(te, ft, c) = None$. Otherwise we define

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

Assignment of Aggregate Literals. The semantics of aggregate literal assignments are similar to normal assignments. Let the program rest be $c.prog = Ass_{\mathrm{AL}}(e_l, l)$. We require that the left expression e_l can be evaluated to some g-variable g, i.e., $leval(te, c.mem, e_l) = \lfloor g \rfloor$, that the aggregate literal l can be evaluated to some value v, i.e., $reval_{\mathrm{alit}}(l) = \lfloor v \rfloor$, and that updating the memory succeeds, i.e., $memupd(c.mem, g, v) = \lfloor mc' \rfloor$.

If one of these requirements is violated we set $\delta_{\mathrm{C0}}(te, ft, c) = None$. Otherwise we define as before for normal assignments

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

Memory Allocation

Let the program rest consist of a statement for allocation of new heap memory, i.e., $c.prog = PAlloc(e_l, tn)$.

We introduce the predicate $avail_{\mathrm{heap}} :: memconf \times ty \mapsto \mathbb{B}$ which specifies if there is enough heap memory available to allocate a new object of a given type. For the $C0$ semantics, this predicate remains an uninterpreted parameter. This keeps the $C0$ semantics independent of the amount of available memory in a concrete machine. However, in Section 8.3 we will give a concrete instance of this predicate which decides based on the available memory on the target machine and the set of already allocated objects. It is possible to replace that simple instance with a more involved version which supports garbage collection without changing the $C0$ semantics.

If there is enough free heap memory left the execution of the allocation statement can be divided into two steps.

1. In a first step, we extend the heap memory by a new (unnamed) variable of correct type and initialize it.
2. In a second step, we create a pointer to the newly allocated variable and assign it to the left expression e_l.

If there is not enough heap memory left we skip the first step and assign a null pointer in the second.

For both cases we require that the left expression e_l can be evaluated to some g-variable g, i.e., $leval(te, c.mem, e_l) = \lfloor g \rfloor$, and that the type name tn is defined in the type name environment, i.e., $map\text{-}of(te, tn) = \lfloor t \rfloor$.

Definition 4.33 (Heap extension) We introduce the function $extend_{\mathrm{heap}} :: (memconf \times ty \times \mathbb{B}) \times memconf \times ty \mapsto memconf$ which handles step 1 of the above list, i.e., it adds a new initialized variable to the heap. Let m the old memory configuration, and t the type of the new heap variable. Then the new memory configuration $m' = extend_{\mathrm{heap}}(avail_{\mathrm{heap}}, m, t)$ is defined as follows.

We abbreviate the base address of the new heap variable (which equals the abstract size of the old heap memory) by $b = \sum_{j=0}^{|hst(m)|-1} size_{\mathrm{t}}(snd(hst(m)!j))$. If there is enough heap memory available, i.e., $avail_{\mathrm{heap}}(m, t) = true$, the new heap memory is defined by

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

Otherwise, if $avail_{\text{heap}}(m,t) = \mathit{false}$, we set $hm' = m.hm$.

For the second step, let p be a pointer which is either the null pointer, if there is not enough heap memory, or points to the newly allocated heap variable.

$$p = \begin{cases} Ptr(gvar_{\text{hm}}(|hst(c.mem)|)) & \text{if } avail_{\text{heap}}(c.mem,t) \\ Ptr(\bot) & \text{otherwise} \end{cases}$$

If one of the requirements is violated we set $\delta_{\text{C0}}(te,ft,c) = \mathit{None}$. Otherwise, updating g with p succeeds (because p is by definition properly initialized) and gives a new memory configuration mc', i.e.,

$$memupd(extend_{\text{heap}}(avail_{\text{heap}}, c.mem, t), g, [p]) = \lfloor mc' \rfloor .$$

Then, we define the new configuration by

$$\delta_{\text{C0}}(te,ft,c) = \left\lfloor c \left[\begin{array}{l} prog := \mathit{Skip} \\ mem := mc' \end{array} \right] \right\rfloor$$

Function Call

Let the program rest consist of a function call to a function with name fn and parameters pl, i.e., $c.prog = SCall(vn, fn, pl)$. The return value of this function call with later be stored in variable vn. Such a function call to is executed in several steps.

1. First, we extend the local memory stack by a new stack frame. The symbol table of this new stack frame consists of the parameters and local variables of the called function.
2. Then, we evaluate the parameters of the function call and copy their values into the new stack frame.
3. As a last step, we set the new program rest to the body of the called function.

We start with the definition of a function which copies the parameters.

Definition 4.34 (Parameter passing) We define the semantics of parameter passing via the function $copy_{\text{para}} :: memconf \times (\mathbb{N} \mapsto mcell_{\text{C0}})\ list \times \mathbb{S}\ list \mapsto memconf_{\bot}$. An application $copy_{\text{para}}(mc, vl, pnl)$ of this function copies the values from the value list vl to variables in the current local memory frame of memory configuration mc. Which variables are updated is specified by the list pnl of parameter names. The function $copy_{\text{para}}$ is defined by recursion on the list of values. We define the base case by

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

For the recursive case, i.e., $copy_{\text{para}}(mc, v\#vl, pnl)$, assume that copying the first parameter value to the local variable $hd(pnl)$ in the current local memory frame succeeds. If this is not the case we set $copy_{\text{para}}(mc, v\#vl, pnl) = \mathit{None}$. If it succeeds there exists a new memory configuration mc' such that $memupd(mc, gvar_{\text{lm}}(|mc.lm| - 1, hd(pnl)), v) = \lfloor mc' \rfloor$ and we define

$$copy_{\text{para}}(mc, v\#vl, pnl) = copy_{\text{para}}(mc', vl, tl(pnl)).$$

Now, we start with the definition of the transition function for function calls. We combine steps 1 and 2 in the function $extend_{\text{stack}}$. First, we introduce the notation $st_{\text{fun}}(f) = f.params \circ f.lvars$ to refer to the symbol table of a function $f :: funcT$. This symbol table is composed of the parameters $f.params$ and of the local variables $f.lvars$.

Definition 4.35 (Stack extension) We introduce the function $extend_{\text{stack}} :: tenv \times memconf \times \mathbb{S} \times funcT \times expr\ list \mapsto memconf_{\perp}$. An application $extend_{\text{stack}}(te, mc, vn, f, pl)$ of this function adds a new memory frame for function f to the local memory stack of configuration mc and copies parameters pl to it. The return destination of the new memory frame – the address / g-variable where the result of the function will be written – is obtained from variable vn.

Let m' be the new local memory frame and mc' an intermediate extended memory configuration:

$$m' = \begin{bmatrix} ct & = undef \\ st & = st_{\text{fun}}(f) \\ init & = \emptyset \end{bmatrix}$$
$$mc' = mc\,[lm := mc.lm \circ [(m', leval(te, mc, Var(vn)))]]$$

Observe that the local variables of the function are not automatically initialized ($m'.init = \emptyset$). Only the function's parameters are being initialized during the function call. For all other local variables, the programmer is in charge of proper initialization before usage. Given that all parameters can be evaluated and are initialized, we obtain the final extended memory configuration from mc' by copying the values of the parameters

$$extend_{\text{stack}}(te, mc, vn, f, pl) = \\ copy_{\text{para}}(mc', map(the(reval(te, mc)), pl), map(fst, f.params)).$$

Otherwise, i.e., if not all parameters can be evaluated, stack extension fails, and we set $extend_{\text{stack}}(te, mc, vn, f, pl) = None$.

Now, it is easy to define step 3 and by that the new configuration of the $C0$ machine after executing the statement $SCall(vn, fn, pl)$. Abbreviating with $f = the(map\text{-}of(ft, fn))$ the function which is being called we define

$$\delta_{\text{C0}}(te, ft, c) = \left\lfloor c \begin{bmatrix} prog := f.body \\ mem := the(extend_{\text{stack}}(te, mc, vn, f, pl)) \end{bmatrix} \right\rfloor$$

Return

Let the program rest consist of a return statement with return expression re, i.e., $c.prog = Return(re)$. The execution of this statement can be divided into two steps.

1. Evaluate the return expression and store its value at the return destination specified by the current local memory frame.
2. Remove the current local memory frame from the stack.

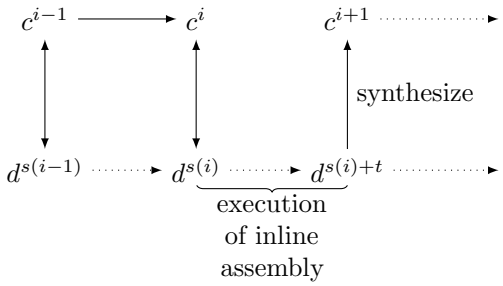


Figure 4.4: Modeling the Semantics of Inline Assembly Code

We require that the return expression re can be evaluated to some value v and is initialized, i.e., $\mathit{reval}(te, c.\mathit{mem}, re) = \lfloor v \rfloor$ and $\mathit{initialized}(te, c.\mathit{mem}, re)$, and that updating the return destination with the value of the return expression succeeds, i.e., $\mathit{memupd}(c.\mathit{mem}, \mathit{snd}(\mathit{hd}(c.\mathit{mem}.\mathit{lm})), v) = \lfloor mc' \rfloor$.

If one of the requirements is violated we set $\delta_{\mathrm{C0}}(te, ft, c) = \mathit{None}$. Otherwise, we define the new configuration by

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

Inline Assembly

Originally, we planned to introduce a separate computational model to define the semantics of $C0$ with inline assembly (cf. [GHLP05]). However, during the verification of Verisoft's academic system, it turned out to be more effective to verify chunks of $C0$ code and chunks of assembly code separately [ST08] and to combine the results later using the compiler correctness theorem from Chapter 8. This method heavily depends on the fact that the correctness theorem is formulated as a small-step simulation theorem which relates *every* state of a $C0$ computation to some consistent assembly state. The simulation theorem provides a 'double' computation consisting of a sequence of $C0$ states and a sequence of corresponding assembly states.

This allows to switch at the start of inline assembly code from the $C0$ layer in configuration c^i to some configuration $d^{s(i)}$ at the assembly layer. Starting from $d^{s(i)}$, we completely execute the inline assembly part (we do not support non-terminating inline assembly code). After its execution, we derive a new (consistent) $C0$ configuration c^{i+1} from that part of the final assembly state $d^{s(i)+t}$ which resembles the variables of the $C0$ program.[5]

This method of integrating inline assembly into the execution of $C0$ programs is illustrated in Figure 4.4. Observe that the dotted arrow in the assembly layer corresponds to several steps of the assembly machine.

[5] Observe that this method for the integration of assembly code into $C0$ computations is not a part of this thesis except for the development of the simulation theorem which it depends on. A. Tsyban has developed and successfully applied the method during the verification of low-level parts of Verisoft's academic system.

XCalls

As mentioned in Section 2.1.1, the meaning of each individual XCall is defined axiomatically. Confer [ASS08] for a detailed example where the authors use XCalls for the pervasive verification of a paging mechanism which is used in Verisoft's academic sub project.

4.4.4 Computations

In this section we combine the initial configuration of a $C0$ program and the $C0$ transition function in order to define computations of a $C0$ program. We start with the definition of a function which executes the $C0$ transition function several times.

Definition 4.36 (Multiple $C0$ transitions) We define the function $\delta^n_{\mathrm{C0}} :: \mathit{tenv} \times \mathit{functableT} \times \mathit{conf}_{\mathrm{C0}} \mapsto \mathit{conf}_{\mathrm{C0}\perp}$ which executes the $C0$ transition function n times starting from a given configuration. The definition is done by induction on n.

$$\delta^0_{\mathrm{C0}}(te, ft, c) = \lfloor c \rfloor$$
$$\delta^{i+1}_{\mathrm{C0}}(te, ft, c) = \begin{cases} \delta_{\mathrm{C0}}(te, ft, c') & \text{if } \delta^i_{\mathrm{C0}}(te, ft, c) = \lfloor c' \rfloor \\ \mathit{None} & \text{if } \delta^i_{\mathrm{C0}}(te, ft, c) = \mathit{None} \end{cases}$$

$C0$ programs consist of a type name environment te, a function table ft, and a symbol table gst for the global variables.

Definition 4.37 ($C0$ computation) We introduce the function $\mathit{nthstep}^n_{\mathrm{C0}} :: \mathit{tenv} \times \mathit{functableT} \times (\mathbb{S} \times \mathit{ty})\ \mathit{list} \mapsto \mathit{conf}_{\mathrm{C0}\perp}$ which computes the n-th configuration of a $C0$ program starting from its initial configuration.

$$\mathit{nthstep}^n_{\mathrm{C0}}(te, ft, gst) = \begin{cases} \delta^n_{\mathrm{C0}}(te, ft, c^0) & \text{if } \mathit{init}_{\mathrm{conf}}(ft, gst) = \lfloor c^0 \rfloor \\ \mathit{None} & \text{otherwise} \end{cases}$$

Chapter 5

Properties of the Small-Step Semantics

Contents

In this chapter, we will introduce several properties and invariants of the $C0$ small-step semantics which we will need in Part II for the compiler correctness proof.

We start with some basic properties of the $C0$ transition function in Section 5.1. Then, we introduce in Section 5.2 the set of *valid* $C0$ programs which requires expressions, statements, and other components of $C0$ programs to be consistent and well-formed. In Section 5.3, we introduce the notion of type correctness and show that the $C0$ expression evaluation *is* type correct. In Section 5.4, we define an invariant about the structure of program rests and prove that it is preserved by the $C0$ transition function. Finally, we define in Section 5.5 the set of valid $C0$ configurations. This definition includes among others static type correctness properties and well-formedness conditions on the program structure.

5.1 Basic Properties

The following lemma relates the execution of one step for a given program rest with the execution of one step when the program rest is replaced by its first non-compound statement, i.e., by $hd(s2l(c.prog))$. This lemma allow to prove properties about the next step of the $C0$ machine without doing induction on the structure of the program rest.

Lemma 5.1 (Execution of the first statement) *Let te be a type name environment, ft a function table, and c a $C0$ configuration and assume that the $C0$ transition function does not produce an error in the next step.*

Then, if we execute the $C0$ transition on a fictive configuration which equals c, except for the program rest which is replaced by the first non-compound

statement in the program rest of c, we get the same memory configuration as if we would have executed the transition function on the original configuration c.

$$\begin{aligned}&\delta_{C0}(te, ft, c) = \lfloor c' \rfloor \\ &\Longrightarrow c'.mem = the(\delta_{C0}(te, ft, c\,[prog := hd(s2l(c.prog))])).mem\end{aligned}$$

Proof This lemma is proved by a simple induction on the program rest. The different induction steps are trivial and omitted here.

5.2 The Set of Valid C0 Programs

Only a small fraction of the $C0$ configurations which could theoretically be build according to the definitions from Section 4.2 are reasonable. For example, we could build a configuration where one of the statements in a function is '5=true;'. This particular statement is invalid for two reasons:

1. '5' is a constant, i.e., it should not be used as a left expression, and
2. the types of left and right expression do not match.

In this section we define a predicate which separates the 'good' from the 'bad' $C0$ configurations. We start with the definition of valid expressions, extend this to valid statement, introduce valid type name environments, and define finally the set of valid function tables.

All these properties are *static* and the formal definitions are executable via Isabelle's code generation feature [BN02, Ber03]. Thus, the properties can be checked once and for all for a given $C0$ program which is in fact nothing else than a (huge) constant in Isabelle / HOL. For details how this approach can be used inside proofs see [Tzi07].

5.2.1 Valid Expressions

In this section we define the set of valid $C0$ expressions. We start with the definition of three predicates which specify which types are valid for the unary, binary, and lazy operators of $C0$. In particular, arithmetic operations for expressions of type $Char_{\mathrm{T}}$ are not supported.

Definition 5.1 (Valid unary operations) We define inductively the set of valid unary operations.

$$valid_{\mathrm{unop}} :: unop \times ty\ set$$

$$\frac{t = Int_{\mathrm{T}}}{(minus, t) \in valid_{\mathrm{unop}}}$$

$$\frac{t \in \{Int_{\mathrm{T}}, Unsigned_{\mathrm{T}}\}}{(neg, t) \in valid_{\mathrm{unop}}}$$

$$\frac{t = Bool_{\mathrm{T}}}{(not, t) \in valid_{\mathrm{unop}}}$$

$$\frac{\circ \in \{to_{int}, to_{unsgnd}, to_{char}\} \qquad t \in \{Int_{\mathrm{T}}, Unsigned_{\mathrm{T}}, Char_{\mathrm{T}}\}}{(\circ, t) \in valid_{\mathrm{unop}}}$$

Definition 5.2 (Valid binary operations) We define the set of valid binary operations.

$$valid_{\mathrm{binop}} :: binop \times ty \times ty\ set$$

For arithmetic and bitwise operators we require both operands to have the same type. Observe that division and modulo computation are only supported for unsigned operands.

$$\frac{\circ \in \{add, sub, mult, or_{\mathrm{b}}, and_{\mathrm{b}}, xor_{\mathrm{b}}\} \qquad t_1 = t_2 \qquad t_1 \in \{Int_{\mathrm{T}}, Unsigned_{\mathrm{T}}\}}{(\circ, t_1, t_2) \in valid_{\mathrm{binop}}}$$

$$\frac{\circ \in \{div, mod\} \qquad t_1 = t_2 \qquad t_1 = Unsigned_{\mathrm{T}}}{(\circ, t_1, t_2) \in valid_{\mathrm{binop}}}$$

The type of the shift distance can be an arbitrary numeric type.

$$\frac{\begin{array}{c}\circ \in \{shift_{\mathrm{l}}, shift_{\mathrm{r}}\} \\ t_1 \in \{Int_{\mathrm{T}}, Unsigned_{\mathrm{T}}\} \qquad t_2 \in \{Int_{\mathrm{T}}, Unsigned_{\mathrm{T}}, Char_{\mathrm{T}}\}\end{array}}{(\circ, t_1, t_2) \in valid_{\mathrm{binop}}}$$

$$\frac{\begin{array}{c}\circ \in \{comp_{\mathrm{less}}, comp_{\mathrm{le}}, comp_{\mathrm{greater}}, comp_{\mathrm{ge}}\} \\ t_1 = t_2 \qquad t_1 \in \{Int_{\mathrm{T}}, Unsigned_{\mathrm{T}}, Char_{\mathrm{T}}\}\end{array}}{(\circ, t_1, t_2) \in valid_{\mathrm{binop}}}$$

We allow equality tests if both types are equal and elementary. Additionally, we support one of them being a pointer and the other the special type $Null_{\mathrm{T}}$ which represents null pointer constants. This allows to compare arbitrary pointers with *Null*.

$$\frac{\begin{array}{c}\circ \in \{comp_{\mathrm{eq}}, comp_{\mathrm{neq}}\} \\ elem?_{\mathrm{t}}(t_1) \qquad t_1 = t_2 \vee (t_1 = Ptr_{\mathrm{T}} \wedge t_2 = Null_{\mathrm{T}}) \vee (t_1 = Null_{\mathrm{T}} \wedge t_2 = Ptr_{\mathrm{T}})\end{array}}{(binop, t_1, t_2) \in valid_{\mathrm{binop}}}$$

Definition 5.3 (Valid lazy operations) We define the set of valid lazy operations.

$$valid_{\mathrm{lazyop}} :: lazyop \times ty \times ty\ set$$

$$\frac{t_1 = Bool_{\mathrm{T}} \qquad t_2 = Bool_{\mathrm{T}}}{(and_{\mathrm{l}}, t_1, t_2) \in valid_{\mathrm{lazyop}}}$$

$$\frac{t_1 = Bool_{\mathrm{T}} \qquad t_2 = Bool_{\mathrm{T}}}{(or_{\mathrm{l}}, t_1, t_2) \in valid_{\mathrm{lazyop}}}$$

Next, we define the sets of valid primitive and aggregate literals. We restrict the range of values to come up with the bounded register size of computers.

Definition 5.4 (Valid literals) We define the set of valid literals.

$$valid_{\text{lit}} :: lit\ set$$

$$\frac{-2^{31} \leq i < 2^{31}}{Int(i) \in valid_{\text{lit}}}$$

$$\frac{0 \leq n < 2^{32}}{Unsigned(n) \in valid_{\text{lit}}}$$

$$\frac{-2^{7} \leq i < 2^{7}}{Char(i) \in valid_{\text{lit}}}$$

$$\frac{}{Bool(b) \in valid_{\text{lit}}}$$

$$\frac{}{Null \in valid_{\text{lit}}}$$

Definition 5.5 (Valid aggregate literals) We define the set of valid aggregate literals.

$$valid_{\text{alit}} :: lit_{\text{a}}\ set$$

$$\frac{}{ALPrim(p) \in valid_{\text{alit}}}$$

For array literals we require that all elements have the same type and are themselves valid aggregate literals.

$$\frac{vl \neq [\,] \qquad \forall v \in vl : v \in valid_{\text{alit}} \qquad \forall v \in vl : type_{\text{alit}}(v) = type_{\text{alit}}(hd(vl))}{ALArr(vl) \in valid_{\text{alit}}}$$

For structure literals we also require that the components are themselves valid aggregate literals and additionally that the component names are pairwise distinct.

$$\frac{cvl \neq [\,] \qquad distinct(map(fst, cvl)) \qquad \forall cv \in cvl : snd(cv) \in valid_{\text{alit}}}{ALStruct(cvl) \in valid_{\text{alit}}}$$

Isabelle Observe that for technical reasons the definition of valid aggregate literals in Isabelle / HOL has been done slightly different using three mutual recursive functions instead of an inductive set. Nevertheless, the semantics of both definitions are equivalent.

For the definition of valid expressions we need a predicate which checks whether a given expression refers to an object in the local memory. Of course

we could simply left evaluate the expression and check the memory name of the obtained g-variable. But this would require to have a complete memory configuration which conflicts with our intention to define a *static* property.

Definition 5.6 We define the predicate $lmexpr :: tenv \times (\mathbb{S} \times ty)\ list \times (\mathbb{S} \times ty)\ list \times expr \mapsto \mathbb{B}$ which tests whether a given expression belongs to a local memory. In $C0$, pointers to local variables are forbidden. Thus, in valid $C0$ programs, the address-of operator and pointer dereferencing cannot generate expressions which reference a local memory frame.

$$\begin{aligned} lmexpr(te, gst, lst, Var(vn)) &= vn \in map(fst, lst) \\ lmexpr(te, gst, lst, Arr(e_a, e_i)) &= lmexpr(te, gst, lst, e_a) \\ lmexpr(te, gst, lst, Str(e, cn)) &= lmexpr(te, gst, lst, e) \end{aligned}$$

For all other expressions e we set the predicate to *false*.

$$lmexpr(te, gst, lst, e) = false$$

Definition 5.7 (Valid expressions) We define the set of valid expressions. It depends on the type name environment *te* and on the symbol tables *lst* and *gst* for local and global variables.

$$valid_{\text{expr}} :: tenv \times (\mathbb{S} \times ty)\ list \times (\mathbb{S} \times ty)\ list \mapsto expr\ set$$

We require literals to be valid according to the definition on the facing page.

$$\frac{l \in valid_{\text{lit}}}{Lit(l) \in valid_{\text{expr}}(te, gst, lst)}$$

Variables must be defined in the local or in the global memory.

$$\frac{vn \in map(fst, lst) \vee vn \in map(fst, gst)}{Var(vn) \in valid_{\text{expr}}(te, gst, lst)}$$

An access to an array element is valid if all sub expressions are valid and have proper types. Observe that array indices have to be unsigned expressions.

$$\frac{\begin{array}{cc} type(te, gst, lst, e_a) = \lfloor Arr_{\text{T}} n, t \rfloor & type(te, gst, lst, e_i) = \lfloor Unsigned_{\text{T}} \rfloor \\ e_a \in valid_{\text{expr}}(te, gst, lst) & e_i \in valid_{\text{expr}}(te, gst, lst) \end{array}}{Arr(e_a, e_i) \in valid_{\text{expr}}(te, gst, lst)}$$

For access to structure components we require that the sub expression is valid and has a proper type and that the component name is present in the structure type.

$$\frac{\begin{array}{c} type(te, gst, lst, e) = \lfloor Str_{\text{T}}(scl) \rfloor \\ cn \in map(fst, scl) \qquad e \in valid_{\text{expr}}(te, gst, lst) \end{array}}{Str(e, cn) \in valid_{\text{expr}}(te, gst, lst)}$$

Operator application is a valid expression if all sub expressions are valid and their types go with the operator.

$$\frac{type(te, gst, lst, e) = \lfloor t \rfloor \qquad (\circ, t) \in valid_{\text{unop}} \qquad e \in valid_{\text{expr}}(te, gst, lst)}{UnOp(\circ, e) \in valid_{\text{expr}}(te, gst, lst)}$$

$$\frac{\begin{array}{c} e_1 \in valid_{\text{expr}}(te, gst, lst) \\ e_2 \in valid_{\text{expr}}(te, gst, lst) \qquad type(te, gst, lst, e_1) = \lfloor t_1 \rfloor \\ type(te, gst, lst, e_2) = \lfloor t_2 \rfloor \qquad (\circ, t_1, t_2) \in valid_{\text{binop}} \end{array}}{BinOp(\circ, e_1, e_2) \in valid_{\text{expr}}(te, gst, lst)}$$

$$\frac{\begin{array}{c} e_1 \in valid_{\text{expr}}(te, gst, lst) \\ e_2 \in valid_{\text{expr}}(te, gst, lst) \qquad type(te, gst, lst, e_1) = \lfloor t_1 \rfloor \\ type(te, gst, lst, e_2) = \lfloor t_2 \rfloor \qquad (\circ, t_1, t_2) \in valid_{\text{lazyop}} \end{array}}{LazyBinOp(\circ, e_1, e_2) \in valid_{\text{expr}}(te, gst, lst)}$$

Application of the address-of operator is a valid expression if the sub expression is valid, there exists a type name for its type, and it represents a memory object which does not refer to a local variable.

$$\frac{\begin{array}{c} e \in valid_{\text{expr}}(te, gst, lst) \qquad type(te, gst, lst, e) = \lfloor t \rfloor \\ t \in map(snd, te) \qquad mobj(te, gst, lst, e) \qquad \neg lmexpr(te, gst, lst, e) \end{array}}{AddrOf(e) \in valid_{\text{expr}}(te, gst, lst)}$$

Dereferencing of an expression e is valid if e is a valid expression and the type of e is a pointer to some type name which exists in the type name environment.

$$\frac{\begin{array}{c} e \in valid_{\text{expr}}(te, gst, lst) \\ type(te, gst, lst, e) = \lfloor Ptr_{\text{T}}(tn) \rfloor \qquad tn \in map(fst, te) \end{array}}{Deref(e) \in valid_{\text{expr}}(te, gst, lst)}$$

5.2.2 Valid Statements

In this section we define the set of valid $C0$ statements. We start with two predicates which test whether two types are suitable for assignments and whether the types of function call parameters go with the signature of the called function.

Definition 5.8 (Suitable types for assignments) We define the function $tmatch_{\text{ass}} :: ty \times ty \mapsto \mathbb{B}$ which tests for two types t_l and t_r if a value of type t_r can be assigned to a g-variable of type t_l. Such an assignment is possible if t_l and t_r are equal or differ only for sub variables where the right type has the special null pointer type, i.e., corresponds to a null pointer literal (cf. Definition 4.23 on page 53). Null pointer constants may be assigned to arbitrary pointer types,

even when they occur inside some aggregate literals.

$$\frac{}{tmatch_{\text{ass}}(t, t)}$$

$$\frac{}{tmatch_{\text{ass}}(Ptr_{\text{T}}(tn), Null_{\text{T}})}$$

$$\frac{|scl| = |scl'| \qquad \forall i < |scl| : tmatch_{\text{ass}}(scl!i, scl'!i)}{tmatch_{\text{ass}}(Str_{\text{T}}(scl), Str_{\text{T}}(scl'))}$$

$$\frac{tmatch_{\text{ass}}(t, t')}{tmatch_{\text{ass}}(Arr_{\text{T}}(n, t), Arr_{\text{T}}(n, t'))}$$

Definition 5.9 (Suitable parameter types) Based on $tmatch_{\text{ass}}$ we define the function $tmatch_{\text{para}} :: (ty_{\perp})\ list \times ty\ list \mapsto \mathbb{B}$ which tests if the types of a given list of function call parameters match the parameter types from the function's signature.

Let *stl* be the list of types from the function's signature. We define the function by induction on the list of (options of) parameter types. If one of the elements of this list is *None* we return *false*.

$$tmatch_{\text{para}}([\,], stl) = (stl = [\,])$$

$$tmatch_{\text{para}}(pt\#xs, stl) = \begin{cases} tmatch_{\text{ass}}(hd(stl), t) & \\ \wedge tmatch_{\text{para}}(xs, tl(stl)) & \text{if } pt = \lfloor t \rfloor \\ false & \text{otherwise} \end{cases}$$

Now, we can define the set of valid $C0$ statements.

Definition 5.10 (Valid statements) We define the set of valid statements. Let *te* be a type name environment, *ft* a function table, and *gst* and *lst* the symbol tables for the global and local variables.

$$valid_{\text{stmt}} :: tenv \times functableT \times (\mathbb{S} \times ty)\ list \times (\mathbb{S} \times ty)\ list \mapsto stmt\ set$$

Skip statements are always valid.

$$\frac{}{Skip \in valid_{\text{stmt}}(te, ft, gst, lst)}$$

For compound statements both sub statements have to be valid.

$$\frac{s_1 \in valid_{\text{stmt}}(te, ft, gst, lst) \qquad s_2 \in valid_{\text{stmt}}(te, ft, gst, lst)}{Comp(s_1, s_2) \in valid_{\text{stmt}}(te, ft, gst, lst)}$$

For normal assignments we require that both expressions are valid, that their types match, and that the left expression refers to a memory object.

$$\frac{\begin{array}{c} e_l \in valid_{\text{expr}}(te, gst, lst) \\ e_r \in valid_{\text{expr}}(te, gst, lst) \qquad type(te, gst, lst, e_l) = \lfloor t_l \rfloor \\ type(te, gst, lst, e_r) = \lfloor t_r \rfloor \qquad tmatch_{\text{ass}}(t_l, t_r) \qquad mobj(te, gst, lst, e_l) \end{array}}{Ass(e_l, e_r) \in valid_{\text{stmt}}(te, ft, gst, lst)}$$

The rule for assignments of aggregate literals is similar to the rule for normal assignments. It only differs in the requirements for the right expression.

$$\frac{\begin{array}{ccc} e_l \in valid_{\text{expr}}(te, gst, lst) & l_c \in valid_{\text{alit}} & type(te, gst, lst, e_l) = \lfloor t_l \rfloor \\ type_{\text{alit}}(l_c) = t_r & tmatch_{\text{ass}}(t_l, t_r) & mobj(te, gst, lst, e_l) \end{array}}{Ass_{\text{AL}}(e_l, l_c) \in valid_{\text{stmt}}(te, ft, gst, lst)}$$

For allocation of heap memory we require that the left expression is valid, of the correct pointer type, and represents a memory object.

$$\frac{\begin{array}{c} e \in valid_{\text{expr}}(te, gst, lst) \\ type(te, gst, lst, e) = \lfloor Ptr_{\text{T}}(tn) \rfloor \qquad mobj(te, gst, lst, e) \end{array}}{PAlloc(e, tn) \in valid_{\text{stmt}}(te, ft, gst, lst)}$$

For function calls we require that the function name exists, that the number and types of parameters match the definition of the function, that all parameters are valid expressions, and that the variable which is supposed to store the return value is defined and has a matching type.

$$\frac{\begin{array}{c} map\text{-}of(ft)(fn) = \lfloor f \rfloor \qquad |pl| = |f.params| \\ tmatch_{\text{para}}(map(\lambda\, e.\ type(te, gst, lst, e), pl), map(snd, f.params)) \\ \forall e \in pl : e \in valid_{\text{expr}}(te, gst, lst) \\ Var(vn) \in valid_{\text{expr}}(te, gst, lst) \qquad type(te, gst, lst, Var(vn)) = \lfloor f.rtype \rfloor \end{array}}{SCall(vn, fn, pl) \in valid_{\text{stmt}}(te, ft, gst, lst)}$$

For return statements we just require that the return expression is valid. Observe that here we do not require that the type of the return expression matches the return type of the function which the return statements belongs to. The reason is simply that we do not know which function that is. The requirement that the type of the return statement matches the return type of the function will be part of the definition of valid functions on page 78 where we have the necessary context information.

$$\frac{e \in valid_{\text{expr}}(te, gst, lst)}{Return(e) \in valid_{\text{stmt}}(te, ft, gst, lst)}$$

For conditional statements we require that the condition is a valid boolean expression and that both sub statements are valid and contain no return statements.

$$\frac{\begin{array}{c} e \in valid_{\text{expr}}(te, gst, lst) \\ s_1 \in valid_{\text{stmt}}(te, ft, gst, lst) \qquad s_2 \in valid_{\text{stmt}}(te, ft, gst, lst) \\ type(te, gst, lst, e) = \lfloor Bool_{\text{T}} \rfloor \qquad \#ret(s_1) = 0 \qquad \#ret(s_2) = 0 \end{array}}{Ifte(e, s_1, s_2) \in valid_{\text{stmt}}(te, ft, gst, lst)}$$

For loop statements we require that the condition is a valid boolean expression and that the loop body is a valid, non-empty statement without returns.

$$\frac{\begin{array}{c} e \in valid_{\text{expr}}(te, gst, lst) \qquad lb \in valid_{\text{stmt}}(te, ft, gst, lst) \\ type(te, gst, lst, e) = \lfloor Bool_{\text{T}} \rfloor \qquad \#ret(lb) = 0 \qquad s2l_{\text{ns}}(lb) \neq [\,] \end{array}}{Loop(e, lb) \in valid_{\text{stmt}}(te, ft, gst, lst)}$$

REMARK Observe that we do not give a definition of the $valid_{\text{stmt}}$ predicate for inline assembly or XCalls. For the latter, A. Tsyban has defined corresponding requirements. For the former, N. Schirmer and E. Alkassar have formulated the conditions.

5.2.3 Valid Type Name Environments

In this section we characterize valid type name environment. We start with a definition of valid types.

Definition 5.11 (Valid type) Let te be a type name environment. We call a type valid if it fulfills the following predicate:

$$valid_{\text{ty}} :: tenv \times ty \mapsto \mathbb{B}$$

Basic types are valid.

$$\frac{t \in \{Bool_{\text{T}}, Int_{\text{T}}, Char_{\text{T}}, Unsigned_{\text{T}}, Null_{\text{T}}\}}{valid_{\text{ty}}(te, t)}$$

Pointer types are valid if the type name is defined in the type name environment.

$$\frac{tn \in map(fst, te)}{valid_{\text{ty}}(te, Ptr_{\text{T}}(tn)}$$

Structure types are valid if the list of components is not empty and all component names are pairwise distinct and themselves valid types.

$$\frac{scl \neq [\,] \qquad distinct(map(fst, scl)) \qquad \forall sc \in scl : valid_{\text{ty}}(te, snd(sc))}{valid_{\text{ty}}(te, Str_{\text{T}}(scl)}$$

Array types are valid if they are not empty and the element type itself is a valid type.

$$\frac{0 < n \qquad valid_{\text{ty}}(te, t)}{valid_{\text{ty}}(te, Arr_{\text{T}}(n, t)}$$

Definition 5.12 (Valid type name environment) Based on the definition of valid types we define the set of valid type name environments. We say that a type name environment is valid if the type names are pairwise distinct and all types are valid.

$$valid_{\text{tenv}} :: tenv\ set$$

$$\frac{distinct(map(fst, te)) \qquad \forall (tn, t) \in te : valid_{\text{ty}}(te, t)}{te \in valid_{\text{tenv}}}$$

5.2.4 Valid Function Tables

In this section we characterize valid function tables. Roughly speaking, a function table is valid if all its functions are valid. This requires among others that the variable types in the function's symbol tables are valid and that the function body consists of valid statements. Additionally, we require that all statements in the function table have distinct identifiers. We start by defining the set of valid symbol tables.

Definition 5.13 (Valid symbol table) We call a symbol table valid if the variable names are pairwise distinct and the types of all variables are valid types.

$$valid_{\text{st}} :: tenv \mapsto ty\ set$$

$$\frac{distinct(map(fst, st)) \qquad \forall (vn, t) \in st : valid_{\text{ty}}(te, t)}{st \in valid_{\text{st}}(te)}$$

Definition 5.14 (Valid function) We define the set of valid functions. A function f is valid if its body is a valid statement, the last statement in the body of f (and only the last one) is a return statement, the type of the return expression matches the return type of the function, and the symbol table consisting of parameters and local variables of f is a valid symbol table.

$$valid_{\text{fun}} :: tenv \times functableT \times (\mathbb{S} \times ty)\ list \mapsto funcT\ set$$

$$\frac{\begin{array}{cc} f.body \in valid_{\text{stmt}}(te, ft, gst, st_{\text{fun}}(f)) & last(s2l(f.body)) = Return(re) \\ \forall s \in butlast(s2l(f.body)) : \neg is\_Return(s) & type(te, gst, st_{\text{fun}}(f), re) = \lfloor rt \rfloor \\ tmatch_{\text{ass}}(f.rtype, rt) & st_{\text{fun}}(f) \in valid_{\text{st}}(te) \end{array}}{f \in valid_{\text{fun}}(te, ft, gst)}$$

To be able to refer uniquely to statements of a $C0$ program we require all non-structural statements in valid function tables to be distinct. For statements which are otherwise identical, this can be achieved by adapting the statement identifier. Because this identifier is part of the constructor we can distinguish statements – except structural statements like *Skip* and *Comp* – by simple comparison which implicitly also compares the statement identifier.

Definition 5.15 (Distinct statements) We define the predicate $dstnct_{\text{s}} :: stmt \mapsto \mathbb{B}$ which tests whether all sub statements of a given statement – except for structural statements – are distinct.

$$dstnct_{\text{s}}(s) = \forall x, y \in sub_{\text{s}}(s) : \neg\,(stmt_{\text{structural}}(x) \vee stmt_{\text{structural}}(y)) \implies x \neq y$$

Isabelle In Isabelle, the definition of distinct statements has been formalized slightly different. However, the definition given above is more concise and has the same semantics as the original definition.

Definition 5.16 (Globally distinct statemens) We extend the predicate from the previous definition to complete function tables.

$$dstnct_{\mathrm{s}}^{ft} :: functableT \mapsto \mathbb{B}$$

$$\frac{}{dstnct_{\mathrm{s}}^{ft}([\,])}$$

$$\frac{\begin{array}{c} dstnct_{\mathrm{s}}(snd(h).body) \qquad dstnct_{\mathrm{s}}^{ft}(t) \\ \forall s \in sub_{\mathrm{s}}(snd(h).body) : \neg stmt_{\mathrm{structural}}(s) \longrightarrow \forall f \in t : s \notin sub_{\mathrm{s}}(snd(f).body) \end{array}}{dstnct_{\mathrm{s}}^{ft}(h\#t)}$$

Now, we can start with the definition of valid function tables.

Definition 5.17 (Valid function table) We define the set of valid function tables. We call a function table *ft* valid if all functions in *ft* are valid, the function names are pairwise distinct, all statements of all function bodies are distinct, and the function table is not empty.

$$valid_{ft} :: tenv \times (\mathbb{S} \times ty)\ list \mapsto functableT\ set$$

$$\frac{\begin{array}{c} \forall (fn, f) \in ft : f \in valid_{\mathrm{fun}}(te, ft, gst) \\ distinct(map(fst, ft)) \qquad dstnct_{\mathrm{s}}^{ft}(ft) \qquad ft \neq [\,] \end{array}}{ft \in valid_{ft}(te, gst)}$$

Observe that we do not require the presence of a 'main' function in valid function tables. The presence of a 'main' function is only needed when arguing about the initial state of $C0$ computations. In such cases we will state the requirement explicitly.

The non-emptiness requirement for the function table is – besides the fact the empty programs do not make much sense – only needed for rather technical arguments in the compiler correctness proof. However, we include the requirement here, to keep the list of assumptions in lemmas as concise as possible.

Remark Observe that, from now on, we often omit certain parameters of functions (e.g., the type name environment *te* or the symbol configuration $sc(mc)$) during proofs given that this does not introduce ambiguities.

Lemma 5.2 (Left and right evaluation are compatible) *Evaluating the right value of an expression via reval gives the same result as first evaluating the left value g of the expression via leval and then reading the corresponding part of the memory via* $value_g$.

Formally, let te be a type name environment, mc a memory configuration, and e an expression which represents a memory object. Then, the following formula is true.

$$\begin{array}{l} reval(te, mc, e) \neq None \\ \wedge\ mobj(te, gst(mc), lst_{top}(mc), e) \\ \wedge\ mc.lm \neq [\,] \\ \Longrightarrow value_g(mc, the(leval(te, mc, e))) = the(reval(te, mc, e)) \end{array}$$

PROOF This lemma is proved by structural induction on e. Because e represents a memory object, we only have to consider the following cases.

Case 1: $e = Var(vn)$
This case follows directly from the definitions of expression evaluation for variables and of the base address of g-variables from definition 4.18.

Case 2: $e = Arr(e_a, e_i)$
Let $type(e_a) = \lfloor Arr_T(n,t) \rfloor$ and $reval(e_i) = \lfloor m \rfloor$, where m is a numerical memory cell with value i. Then we have

$$\begin{aligned}
& value_g(the(leval(Arr(e_a, e_i)))) \\
=& value_g(gvar_{arr}(leval(e_a), i)) && \text{(definition of } leval\text{)} \\
=& value_g(leval(e_a))[i \cdot size_t(t), size_t(t)] && \text{(definition of } value_g\text{)} \\
=& the(reval(te, mc, e_a))[i \cdot size_t(t), size_t(t)] && \text{(induction hypothesis)} \\
=& reval(Arr(e_a, e_i)) && \text{(definition of } reval\text{)}
\end{aligned}$$

Case 3: $e = Str(e_s, cn)$
The proof for this case follows the same structure as the proof for array access.

Case 4: $e = Deref(e')$
This case follows directly from the definition of *reval*. q.e.d.

5.3 Type Correctness

In this section we introduce the notion of type correctness. In short, type correctness means that a value or a sequence of values matches a given type. This means both that the memory cell is of the right kind, i.e., the constructor matches the type, and that the value is in the range of the type, e.g., unsigned naturals should be smaller than 2^{32}.

We start in Section 5.3.1 with the formal definition of type correctness. We continue in Section 5.3.2 with basic lemmas and conclude in Sections 5.3.3 and 5.3.4 with the proof that expression evaluation produces type correct results.

5.3.1 Definitions

In this section we formally define type correctness. We start with defining *valid* memory cells. This includes the definition of valid g-variables and valid pointers.

Definition 5.18 (Valid memory cells) We define predicates for each basic type which specify if a given memory cell m is valid with respect to that type. The predicates can be *false* mainly for two reasons.

- If the memory cell has the wrong type (e.g., a memory cell *Int* for type unsigned int) or
- if the value of the memory cell is not in the domain of the type.

Formally we define

$$\frac{-2^{31} \leq i < 2^{31}}{mcell\surd_{Int}(Int(i))}$$

$$\frac{0 \leq i < 2^{32}}{mcell\surd_{Nat}(Nat(i))}$$

$$\frac{-2^{7} \leq i < 2^{7}}{mcell\surd_{Char}(Char(i))}$$

$$\frac{}{mcell\surd_{Bool}(Bool(b))}$$

Before we can give the definition of valid memory cells for pointers, we need to introduce some auxiliary definitions. We start with the definition of valid g-variables.

Definition 5.19 (Valid g-variables) We define the set $gvars\surd$ of valid g-variables which contains all g-variables with a well-formed structure for a given symbol configuration sc.

$$gvars\surd :: symbolconf \mapsto gvar\ set$$

For the base cases, the definition of valid g-variables is simple: we just require that the referenced variable exists.

$$\frac{vn \in map(fst, sc.gst)}{gvar_{\mathrm{gm}}(vn) \in gvars\surd(sc)}$$

$$\frac{vn \in map(fst, sc.lst!i) \qquad i < |sc.lst|}{gvar_{\mathrm{lm}}(i, vn) \in gvars\surd(sc)}$$

$$\frac{i < |sc.hst|}{gvar_{\mathrm{hm}}(i) \in gvars\surd(sc)}$$

For array g-variables we require that their parent g-variable is of array type, that the index is smaller than the number of array elements, and that the parent g-variable itself is a valid g-variable.

$$\frac{ty_{\mathrm{g}}(sc, g) = Arr_{\mathrm{T}}(n, t) \qquad i < n \qquad g \in gvars\surd(sc)}{gvar_{\mathrm{arr}}(g, i) \in gvars\surd(sc)}$$

For structure access g-variables we require that their parent g-variable is of structure type, that the component name is defined in that structure, and that the parent g-variable itself is a valid g-variable.

$$\frac{ty_{\mathrm{g}}(sc, g) = Str_{\mathrm{T}}(scl) \qquad cn \in map(fst, scl) \qquad g \in gvars\surd(sc)}{gvar_{\mathrm{str}}(g, cn) \in gvars\surd(sc)}$$

Based on this definition we define now the set of valid pointers.

Definition 5.20 (Valid pointers) We define the predicate $ptr\surd$ which specifies if a given pointer is valid. Null pointers are always valid pointers whereas pointers to g-variables g are valid if they are valid g-variables and refer to the global memory or to the heap.

$$ptr\surd :: symbolconf \times (gvar \cup \{\bot\}) \mapsto \mathbb{B}$$

$$\frac{}{ptr\surd(sc, \bot)}$$

$$\frac{mem_{\mathrm{g}}(g) = gm \vee mem_{\mathrm{g}}(g) = hm \qquad g \in gvars\surd(sc)}{ptr\surd(sc, g)}$$

Isabelle Observe that in Isabelle / HOL the preceding definition is split into several predicates. For example, the requirement that the pointer is a valid g-variable is formulated in an extra definition `valid_ptr`. Also the following definitions are split in Isabelle into two groups of predicates: one for non-pointer types and a second with additional requirements for pointers (see the theories `ptr_valid.thy` and `ptr_valid_lemmas.thy`). For simplicity, we merge these two groups of definitions here with the definitions regarding type correctness.

Definition 5.21 (Valid pointer memory cells) Similar to the definition of valid memory cells we define valid pointer memory cells. For a pointer memory cell which is not null we require that the g-variable which represents the pointer destination has the type specified by parameter tn.

$$mcell\surd_{Ptr} :: tenv \times symbolconf \times \mathbb{S} \times mcell_{\mathrm{C0}} \mapsto \mathbb{B}$$

$$\frac{ptr\surd(sc, p) \qquad p \neq \bot \Longrightarrow map\text{-}of(te, tn) = \lfloor ty_{\mathrm{g}}(sc, p) \rfloor}{mcell\surd_{Ptr}(te, sc, tn, Ptr(p))}$$

Now, we can formally define the notion of type correctness.

Definition 5.22 (Type correctness) We define the predicate $ty\surd :: tenv \times symbolconf \times (\mathbb{N} \mapsto mcell_{\mathrm{C0}}) \times ty \times \mathbb{N} \mapsto \mathbb{B}$ which check whether a given value (or a sequence of values) matches a given type. In addition to the obvious parameters (type name environment te, symbol configuration sc, value v, and type) the predicate has a parameter $b \in \mathbb{N}$ which specifies the base address inside the sequence of values where we start the comparison. For example, if $b = 0$ we start at the beginning of the sequence, if $b = 10$ we ignore the first ten

values. We define the predicate by induction on the type.

$$\begin{aligned}
ty\surd(te, sc, v, Bool_{\mathrm{T}}, b) &= mcell\surd_{Bool}(v[b]) \\
ty\surd(te, sc, v, Int_{\mathrm{T}}, b) &= mcell\surd_{Int}(v[b]) \\
ty\surd(te, sc, v, Char_{\mathrm{T}}, b) &= mcell\surd_{Char}(v[b]) \\
ty\surd(te, sc, v, Unsigned_{\mathrm{T}}, b) &= mcell\surd_{Nat}(v[b]) \\
ty\surd(te, sc, v, Ptr_{\mathrm{T}}(tn), b) &= mcell\surd_{Ptr}(te, sc, tn, v[b]) \\
ty\surd(te, sc, v, Null_{\mathrm{T}}, b) &= (v[b] = Ptr(\bot)) \\
ty\surd(te, sc, v, Str_{\mathrm{T}}([]), b) &= true \\
ty\surd(te, sc, v, Str_{\mathrm{T}}(h\#t), b) &= ty\surd(te, sc, v, snd(h), b) \\
&\quad \wedge ty\surd(te, sc, v, Str_{\mathrm{T}}(t), b + size_{\mathrm{t}}(snd(h))) \\
ty\surd(te, sc, v, Arr_{\mathrm{T}}(n, t), b) &= \forall i < n : ty\surd(te, sc, v, t, b + i * size_{\mathrm{t}}(t))
\end{aligned}$$

Isabelle In Isabelle / HOL the preceding definition does not require that pointer destinations are valid g-variables. However, to simplify presentation we include this requirement here into $ptr\surd$ and thus it is also part of $ty\surd$. In Isabelle the validity of pointer targets is formulated in the additional definition `content_valid_ptrs`.

Now we extend the definition of type correctness from values to complete memory frames. We call a memory type correct if all its *initialized* variables are type correct. Because *normal* memory frames with named variables and the *heap* memory frame differ slightly we introduce a separate definition for the heap memory.

Definition 5.23 (Type correct memory) For normal memory frames we introduce the predicate $ty\surd_{\mathrm{mem}} :: tenv \times symbolconf \times mframe \mapsto \mathbb{B}$ which checks whether a given memory is type correct.

$$\begin{aligned}
ty\surd_{\mathrm{mem}}(te, sc, m) = \forall vn \in (map(fst, m.st) \cap m.init) : \\
ty\surd(te, sc, m.ct, the(type_{\mathrm{v}}(m.st, vn)), the(ba_{\mathrm{v}}(m.st, vn)))
\end{aligned}$$

For the heap memory we need a slightly different definition of type correctness because the heap differs in two points from the other memory frames. The heap variables haves no names and they are automatically initialized, i.e., there is no set of initialized variables for the heap memory.

Definition 5.24 (Type correct heap) We define the predicate $ty\surd_{\mathrm{heap}} :: tenv \times symbolconf \times mframe \mapsto \mathbb{B}$ which checks whether a given memory frame is a type correct heap memory frame.

$$\begin{aligned}
ty\surd_{\mathrm{heap}}(te, sc, m) = \forall i < |m.st| : \\
ty\surd(te, sc, m.ct, snd(m.st!i), \sum_{j=0}^{i-1} size_{\mathrm{t}}(snd(m.st!j)))
\end{aligned}$$

5.3.2 Basic Lemmas

In this section we present some basic lemmas about type correctness. We only show the *non-trivial* basic lemmas; the most basic ones from the formal theories

in Isabelle / HOL are not given here because they are quite trivial and would distract the reader from the interesting facts.

Lemma 5.3 (Type correctness: base transformation) *In this lemma we prove that base parameter in the type correctness predicate can be removed if we appropriately trim the sequence of memory cells.*

$$ty\surd(te, sc, v, t, b) \Longrightarrow ty\surd(te, sc, v[b, size_t(t)], t, 0)$$

Proof The proof is done by induction on the type t. We omit the details here.

Lemma 5.4 *Type correctness is independent of the symbol tables of the local memory stack and of additional heap variables:*

$$\begin{aligned} &ty\surd(te, sc, v, t, b) \\ &\wedge\, sc'.gst = sc.gst \wedge |sc'.hst| \geq |sc.hst| \wedge sc'.hst[0, |sc.hst|] = sc.hst \\ &\Longrightarrow ty\surd(te, sc', v, t, b) \end{aligned}$$

Proof This lemma is proved by a simple induction on the type t. The correctness follows from the fact that the symbol configuration sc is only used to test the validity of pointers which can only point to the global or heap memory. Thus, changes in the local memory stack do not alter their validity. Additional heap variables in sc' also have no effect, because all relevant pointers in v must have been valid before and can only refer to variables which have already been present in the old symbol configuration. But those variables have not been changed. q.e.d.

Lemma 5.4 can easily be extended from $ty\surd$ to $ty\surd_{\text{mem}}$ and $ty\surd_{\text{heap}}$. The proofs are trivial and skipped here.

Lemma 5.5 *Type correctness of memory frames is independant of the symbol tables of the local memory stack and of additional heap variables:*

$$\begin{aligned} &ty\surd_{mem}(te, sc, m) \\ &\wedge\, sc'.gst = sc.gst \wedge |sc'.hst| \geq |sc.hst| \wedge sc'.hst[0, |sc.hst|] = sc.hst \\ &\Longrightarrow ty\surd_{mem}(te, sc', m) \end{aligned}$$

Lemma 5.6 *Type correctness of the heap memory is independant of the symbol tables of the local memory stack and of additional heap variables:*

$$\begin{aligned} &ty\surd_{heap}(te, sc, m) \\ &\wedge\, sc'.gst = sc.gst \wedge |sc'.hst| \geq |sc.hst| \wedge sc'.hst[0, |sc.hst|] = sc.hst \\ &\Longrightarrow ty\surd_{heap}(te, sc', m) \end{aligned}$$

5.3.3 Type Correctness of Initial Values

In this section we present lemmas which state that the initial values defined in Section 4.4.1 are type correct. The proofs of these lemmas are not interesting, so we omit them here.

Lemma 5.7 (Type correctness: initial values) *The inital value for a type t is type correct.*

$$ty\surd(te, sc, init_{val}(t), 0)$$

PROOF The proof of this lemma follows directly from the definitions. It can be done by a simple induction over t.

Lemma 5.8 (Type correctness: initial symbol table) *All variables of a memory frames which is initialized via $init_{st}$ are type correct.*

$$\forall vn \in map(fst, st) : ty\surd(te, sc, init_{st}(st), the(type_v(st, vn)), the(ba_v(st, vn)))$$

PROOF This lemma can be proved by a simple list induction over the symbol table st.

Corollary 5.9 (Type correctness: initial memories) *We conclude from the previous lemma that initialized memories are type correct.*

$$ty\surd_{mem}(te, sc, init_{vars}(m))$$

5.3.4 Type Correctness: Expression Evaluation

In this section we will sketch a proof of the type correctness of expression evaluation in the $C0$ small-step semantics. We will not give all details because expression evaluation is made up of too many cases. However, the main idea of the proof will be sketched.

Lemma 5.10 (Results of unary operators are type correct) *Let $\circ$ be a unary operator, t a type which is compatible with $\circ$, and m a memory cell. Moreover, let r be the result of applying $\circ$ to m. If m is type correct with respect to t then the result r will be type correct with respect to the type of the result.*

$$\begin{aligned} &ty\surd(te, sc, [m], t, 0) \wedge value_{\circ_1}(\circ, \lfloor m \rfloor) = \lfloor r \rfloor \\ &\Longrightarrow ty\surd(te, sc, [r], type_{\circ_1}(\circ, t), 0) \end{aligned}$$

PROOF The proof is done by a case distinction over all unary operators. The individual cases are trivial; thus, we skip the proofs here.

Observe that similar lemmas also hold for binary and lazy operators.

Now, we prove a lemma which states that a value which has been read from a type correct memory is itself type correct.

Lemma 5.11 (G-variables are type correct) *Let mc be a memory configuration where the global and all local memory frames are type correct memories, the heap memory frame is a type correct heap, and all symbol tables are valid. Further, let g be a valid and initialized g-variable. Then, the value of g is type correct.*

$$\begin{aligned} &ty\surd_{mem}(te, sc(mc), mc.gm) \wedge mc.gm.st \in valid_{st}(te) \\ &\wedge \forall lm \in mc.lm : ty\surd_{mem}(te, sc(mc), lm) \wedge lm.st \in valid_{st}(te) \\ &\wedge ty\surd_{heap}(te, sc(mc), mc.hm) \wedge \forall i < |mc.hm| : valid_{ty}(te, snd(mc.hm!i)) \\ &\wedge g \in gvars\surd(sc(mc)) \wedge initialized_g(mc, g) \\ &\Longrightarrow ty\surd(te, sc(mc), value_g(mc, g), ty_g(sc(mc), g), 0) \end{aligned}$$

PROOF The proof is done by induction on g:

Case 1: $g = gvar_{\mathrm{gm}}(x)$
Let $type_{\mathrm{v}}(mc.gm.st, x) = \lfloor t \rfloor$. From $gvar_{\mathrm{gm}}(x) \in gvars\surd(sc(mc))$ we conclude $x \in map(fst, sc(mc).gst)$. From this and $ty\surd_{\mathrm{mem}}(mc.gm)$ follows

$$\begin{aligned}
&ty\surd(mc.gm.ct, t, the(ba_{\mathrm{v}}(mc.gm.st, x))) \\
&= ty\surd(mc.gm.ct[ba_{\mathrm{g}}(g), size_{\mathrm{t}}(ty_{\mathrm{g}}(g))], t, 0) && \text{(Lemma 5.3)} \\
&= ty\surd(value_{\mathrm{g}}(mc, g), ty_{\mathrm{g}}(g), 0) && \text{(Defs. 4.15 and 4.22)}
\end{aligned}$$

which proves this case.

Case 2: $g = gvar_{\mathrm{lm}}(i, x)$
This case is proved similar to the previous case.

Case 3: $g = gvar_{\mathrm{hm}}(i)$
This case is proved similar to the first case.

Case 4: $g = gvar_{\mathrm{arr}}(g', i)$
Let $ty_{\mathrm{g}}(g') = Arr_{\mathrm{T}}(n, t)$. We conclude

$$\begin{aligned}
&ty\surd(value_{\mathrm{g}}(g'), Arr_{\mathrm{T}}(n, t), 0). && \text{(hypothesis)} \\
&= ty\surd(value_{\mathrm{g}}(g'), t, i * size_{\mathrm{t}}(t)) && \text{(Def. 5.22)} \\
&= ty\surd(value_{\mathrm{g}}(g), t, 0) && \text{(Def. 4.18 and Lemma 5.3)}
\end{aligned}$$

Case 5: $g = gvar_{\mathrm{str}}(g', cn)$
Let $ty_{\mathrm{g}}(g') = Str_{\mathrm{T}}(scl)$ and $type_{\mathrm{v}}(scl, x) = \lfloor t \rfloor$. We conclude

$$\begin{aligned}
&ty\surd(value_{\mathrm{g}}(g'), Str_{\mathrm{T}}(scl), 0). && \text{(hypothesis)} \\
&= ty\surd(value_{\mathrm{g}}(g'), t, the(ba_{\mathrm{v}}(scl, x))). && \text{(Def. 5.22)} \\
&= ty\surd(value_{\mathrm{g}}(g')[the(ba_{\mathrm{v}}(scl, x)), size_{\mathrm{t}}(t)], t, 0) && \text{(Lemma 5.3)} \\
&= ty\surd(value_{\mathrm{g}}(g), ty_{\mathrm{g}}(g), 0) && \text{(Def. 4.15 and 4.22)}
\end{aligned}$$

q.e.d.

Lemma 5.12 (Literals are type correct) *Let te be a type name environment and sc a symbol configuration. Then, the evaluation of a valid literal in the context of te and sc is type correct.*

$$l \in valid_{lit} \Longrightarrow ty\surd(te, sc, [reval_{lit}(l)], type_{lit}(l), 0)$$

Proof Trivial case distinction on the constructors for the literal data type.

Lemma 5.13 (Aggregate literals are type correct) *The evaluation of a valid aggregate literal c in this context of a symbol configuration sc and a type name environment te is type correct.*

$$c \in valid_{alit} \Longrightarrow ty\surd(te, sc, reval_{alit}(c), type_{alit}(c), 0)$$

Proof Trivial induction on the constructors of aggregate literals.

Theorem 5.14 (Expression evaluation is type correct)
Let te be a type name environment, mc a memory configuration, and e a valid expression. If the type name environment is valid, the symbol tables of the memory frames of mc are valid, the memory frames are type correct, and the evaluation of e is successful then the result of right evaluation is type correct and left evaluation produces a valid g-variable.

$$
\begin{aligned}
& e \in valid_{expr}(te, gst(mc), lst_{top}(mc)) \wedge reval(te, mc, e) = \lfloor v \rfloor \\
& \wedge\ initialized(te, mc, e) \wedge te \in valid_{tenv} \wedge mc.lm \neq [\,] \\
& \wedge\ gst(mc) \in valid_{st}(te) \wedge ty\surd_{mem}(te, sc(mc), mc.gm) \\
& \wedge\ \forall i < |mc.lm| : mc.lm!i.st \in valid_{st}(te) \wedge ty\surd_{mem}(te, sc(mc), mc.lm!i) \\
& \wedge\ \forall i < |mc.hm.st| : valid_{ty}(te, snd(mc.hm.st!i)) \wedge ty\surd_{heap}(te, sc(mc), mc.hm) \\
\implies\ & ty\surd(te, sc(mc), v, the(type(te, gst(mc), lst_{top}(mc), e)), 0) \\
& \wedge \left(\begin{array}{l} mobj(te, gst(mc), lst_{top}(mc), e) \\ \implies the(leval(te, mc, e)) \in gvars\surd(sc(mc))) \end{array} \right)
\end{aligned}
$$

Proof We prove this lemma by induction on expression e. We omit the boring details of unfolding the definitions of expression evaluation and type correctness and just present the main idea of the proof. Observe that we need to do the proof *simultaneously* for left and right evaluation because address-of and dereferencing expressions invert the usage of these functions. Thus, a statement only about one of the functions would not be inductive.

For expressions which represent memory objects, the type correctness proof is done using lemmas 5.2 and 5.11. That the left value is a valid g-variable follows by definition of *leval* from the fact that e is a valid expression and from the induction hypothesis.

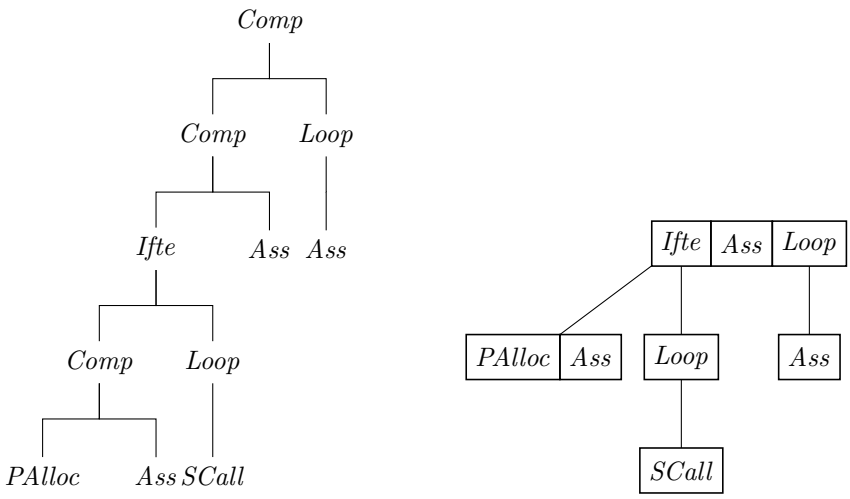
For literals the type correctness follows from Lemma 5.12 and for application of operators from Lemma 5.10 and the corresponding lemmas for binary and lazy operators. For these expressions we have $\neg mobj(te, gst(mc), lst_{\text{top}}(mc), e)$; thus, we do not have to argue about their left value.

For the address-of operator $e = AddrOf(e')$ we use the fact that the left value of e' is a valid g-variable (induction hypothesis) and that e is a valid expression to prove type correctness. q.e.d.

Isabelle In Isabelle, this theorem has a separate statement about the validity of pointers in the resulting data slice. However, in this thesis this requirement has been added to the definition of type correctness (cf. Definition 5.21).

5.4 Structure of the Program Rest

In this section we will first introduce some notation to talk about the structure of statement trees and the relationship of statements inside such a tree. Then we will prove a theorem about the structure of the program rest during the execution of a $C0$ program. This theorem will state that for every statement in the program rest, except for return statements, we know statically which statement follows them. This property will be useful in the compiler correctness proof in Chapters 8 and 10; for return statements we will need special machinery there.



(a) Syntax Tree (b) Flattened

Figure 5.1: Different Views to Statement Trees

5.4.1 Structure of Statement Trees

In this section we look in a special way at the syntax trees of $C0$ programs, which are spanned by the data type *stmt*. We interpret all statements of the same *level* simply as a list of statements (cf. function $s2l_{\text{ns}}$ on page 40) and organize them in a tree-like structure where the branches are built by statements which contain sub statements, i.e., *Ifte* and *Loop*. Observe that a statement *Comp* does not create branches in this structure because it is removed by $s2l_{\text{ns}}$. We illustrate this structure in Figure 5.1 where we face the conventional view to statement trees (cf. Figure 5.1(a)) and the flattened version which we explore in this section (cf. Figure 5.1(b)).

We start with some simple notation.

Definition 5.25 (Direct sub statement) Let s and t be statements. We call t a *direct sub statement* of s if s is a loop statement with body lb and $t \in s2l_{\text{ns}}(lb)$ or if s is a conditional statement with sub statements s_1 and s_2 and $t \in s2l_{\text{ns}}(s_1)$ or $t \in s2l_{\text{ns}}(s_2)$. Formally, this is written as $t \in sub_{\text{s}}^{\text{di}}(s)$.

Similary, we define the *parent statement* of a given statement s. In informal language, t is the parent statement of s if s is a direct sub statement of t. To make the definition of parent statements formal we need to put the cart before the horse: we start with a given 'top-level' statement x and search our way through the syntax tree until s is a direct sub statement of the current statement.

Definition 5.26 (Parent statements) Let s be a statement. We define the parent statement of s in the context of some top-level (or start) statement by the function $parent_{\text{s}} :: stmt \times stmt \mapsto stmt_{\perp}$. The definition is done by

induction on the structure of the top-level statement.

$$parent_{\mathrm{s}}(s, Comp(s_1, s_2)) = \begin{cases} parent_{\mathrm{s}}(s, s_1) & \text{if } s \in sub_{\mathrm{s}}(s_1) \\ parent_{\mathrm{s}}(s, s_2) & \text{otherwise} \end{cases}$$

$$parent_{\mathrm{s}}(s, Ifte(e, s_1, s_2)) = \begin{cases} \lfloor Ifte(e, s_1, s_2) \rfloor & \text{if } s \in sub_{\mathrm{s}}^{\mathrm{di}}(Ifte(e, s_1, s_2)) \\ parent_{\mathrm{s}}(s, s_1) & \text{if } s \in sub_{\mathrm{s}}(s_1) \\ parent_{\mathrm{s}}(s, s_2) & \text{otherwise} \end{cases}$$

$$parent_{\mathrm{s}}(s, Loop(e, lb)) = \begin{cases} \lfloor Loop(e, lb) \rfloor & \text{if } s \in sub_{\mathrm{s}}^{\mathrm{di}}(Loop(e, lb)) \\ parent_{\mathrm{s}}(s, lb) & \text{otherwise} \end{cases}$$

For all other top-level statements s' we set $parent_{\mathrm{s}}(s, s') = None$.

Definition 5.27 (*I*-th parent statement) We extend the previous definition in a natural way to the i-th father of a statement in the context of some top-level statement x.

$$parent_{\mathrm{s}}^{0}(s, x) = \lfloor s \rfloor$$

$$parent_{\mathrm{s}}^{i+1}(s, x) = \begin{cases} parent_{\mathrm{s}}(f, x) & \text{if } parent_{\mathrm{s}}^{i}(s, x) = \lfloor f \rfloor \\ None & \text{otherwise} \end{cases}$$

Definition 5.28 (Statement environment) We define the environment of a given statement s with respect to a given top-level statement. This environment consists of a list of statements which share the same parent statement f and, for the case that f is a conditional statement, are located in the same branch. For top-level statements, i.e., statements without parent statement, the environment is just the list of top-level statements. Formally, we define the function $env_{\mathrm{s}} :: stmt \times stmt \mapsto stmt\ list_{\perp}$ by induction on the top-level statement.

$$env_{\mathrm{s}}(s, Skip) = None$$

$$env_{\mathrm{s}}(s, Comp(s_1, s_2)) = \begin{cases} \lfloor s2l_{\mathrm{ns}}(Comp(s_1, s_2)) \rfloor & \text{if } s \in s2l_{\mathrm{ns}}(Comp(s_1, s_2)) \\ env_{\mathrm{s}}(s, s_1) & \text{if } s \in sub_{\mathrm{s}}(s_1) \\ env_{\mathrm{s}}(s, s_2) & \text{otherwise} \end{cases}$$

$$env_{\mathrm{s}}(s, Ifte(e, s_1, s_2)) = \begin{cases} \lfloor s \rfloor & \text{if } s = Ifte(e, s_1, s_2) \\ env_{\mathrm{s}}(s, s_1) & \text{if } s \in sub_{\mathrm{s}}(s_1) \\ env_{\mathrm{s}}(s, s_2) & \text{otherwise} \end{cases}$$

$$env_{\mathrm{s}}(s, Loop(e, lb)) = \begin{cases} \lfloor s \rfloor & \text{if } s = Loop(e, lb) \\ env_{\mathrm{s}}(s, lb) & \text{otherwise} \end{cases}$$

For all other statements s', i.e., the ones without sub statements, we define

$$env_{\mathrm{s}}(s, s') = \begin{cases} \lfloor s \rfloor & \text{if } s = s' \\ None & \text{otherwise} \end{cases}$$

Now, we can define the *direct successor* of a statement.

Definition 5.29 (Direct successor of a statement) Let s and x be statements. We define the direct successor of s in the context of x by the function $succ_{\text{direct}} :: stmt \times stmt \mapsto stmt_{\perp}$.

If s is not the last statement in its environment $env_{\text{s}}(s, x)$ then there exists a statement s' which follows s in $env_{\text{s}}(s, x)$. In this case we define $succ_{\text{direct}}(s, x) = \lfloor s' \rfloor$. Otherwise, we set $succ_{\text{direct}}(s, x) = None$.

Finally, the goal of this section is to prove a theorem which states that all statements in the program rest of a valid $C0$ configuration, except return statements, are followed by a particular unique statement. For statements which are not the last statement in their environment, this particular statement is just their direct successor. However, for statements which are the last statement in their environment we need some more definitions.

Definition 5.30 (Last statements) We start with a formalization of 'last statement in their environment'. Let s and x be statements. We define the predicate $is\_last_{\text{s}} :: stmt \times stmt \mapsto \mathbb{B}$ which checks whether s is the last statement in its environment.

$$is\_last_{\text{s}}(s, x) = \begin{cases} (s = last(env)) & \text{if } env_{\text{s}}(s, x) = \lfloor env \rfloor \\ false & \text{otherwise} \end{cases}$$

Additionally, we define two predicates which differentiate if s is a last statement in one of the branches of a conditional statement or if it is the last statement of a loop body. Formally, we define

$$\frac{is\_last_{\text{s}}(s, x) \qquad parent_{\text{s}}(s, x) \neq None \qquad is\_Loop(the(parent_{\text{s}}(s, x)))}{is\_last_{\text{s}}^{loop}(s, x) = true}$$

$$\frac{is\_last_{\text{s}}(s, x) \qquad parent_{\text{s}}(s, x) \neq None \qquad is\_Ifte(the(parent_{\text{s}}(s, x)))}{is\_last_{\text{s}}^{cond}(s, x) = true}$$

For conditionals we further distinguish whether a statement is the last statement in the if part ($is\_last_{\text{s}}^{if}$) or in the else part ($is\_last_{\text{s}}^{else}$).

If s is the last statement in the body of a while loop, i.e., $is\_last_{\text{s}}^{loop}(s, x)$, its successor statement will just be its parent statement, i.e., the loop statement (cf. the definition of the $C0$ transition function in Section 4.4.3). If $is\_last_{\text{s}}^{cond}(s, x)$ holds, the definition of successor statements involves induction because the successor statement of s equals the successor statement of the parent statement of s, i.e., the condition statement, which itself could again be a last statement.

Definition 5.31 (Condition nesting level) To compute how deep a given statement s is nested in such a stack of last statements in conditionals we introduce the function $cond_{\text{lvl}} :: stmt \times stmt \mapsto \mathbb{N}$.

$$cond_{\text{lvl}}(s, x) = \begin{cases} cond_{\text{lvl}}(the(parent_{\text{s}}(s, x)), x) + 1 & \text{if } is\_last_{\text{s}}^{cond}(s, x) \\ 0 & \text{otherwise} \end{cases}$$

Definition 5.32 (Condition senior) We use the definition of the condition nesting level ($cond_{\text{lvl}}$) to define the *condition senior* of a statement. This senior statement is the first statement in the sequence of parent statements of a given

statement which is not a last statement in a conditional statement. Formally, the condition senior is defined by the function $cond_{\text{sen}} :: stmt \times stmt \mapsto stmt_\perp$. We set $cond_{\text{sen}}(s,x) = parent_{\text{s}}^{cond_{\text{lvl}}(s,x)}(s,x)$.

Now, we can define the successor of a statement.

Definition 5.33 (Successor of a statement) We define the function $succ :: stmt \times stmt \mapsto stmt_\perp$.

Let s and x be statements. Then, $succ(s,x)$ computes the successor statement of s in the context of x. We define

$$succ(s,x) = \begin{cases} parent_{\text{s}}(s,x) & \text{if } is\_last_{\text{s}}^{loop}(s,x) \\ succ(the(cond_{\text{sen}}(s,x)),x) & \text{if } is\_last_{\text{s}}^{cond}(s,x) \\ succ_{\text{direct}}(s,x) & \text{otherwise} \end{cases}$$

Based on the definition of successor statements it is now simple to define what *valid successors* are.

Definition 5.34 (Valid successors) We define the predicate $successors\surd :: stmt\ list \times stmt \mapsto \mathbb{B}$ which checks whether in a given list sl of statements all statements are followed by their successor statement. There is one exception: for return statements we do not require this condition to hold because they are followed by that statement which has followed the corresponding function call statement when the function call has been executed. Functions may be called from several places in a statement tree; thus, return statements have no unique successor statement. Formally, we define the predicate by induction on the statement list.

$$\frac{}{successors\surd([],x)}$$

$$\frac{successors\surd(t,x) \qquad (t \neq [] \wedge \neg is\_Return\,(h)) \Longrightarrow succ(h,x) = \lfloor hd(t) \rfloor}{successors\surd(h\#t,x)}$$

5.4.2 Extending Statement Structure to Complete Programs

In the previous section we have defined several functions and predicates which describe the structure and relationship of statements with respect to a given top-level or starting statement. In this section we briefly describe how these definitions can be extended from the local context of a single statement to the context of a complete $C0$ program.

For this extension we introduce a new function $fos :: stmt \times functableT \mapsto (\mathbb{S} \times funcT)_\perp$ which computes for given function table and statement in which function of the function table the statement occurs. The formal definition of *fos* is straightforward:

$$fos(s,[]) = None$$
$$fos(s,h\#t) = \begin{cases} \lfloor h \rfloor & \text{if } s \in sub_{\text{s}}(snd(h).body) \\ fos(s,t) & \text{otherwise} \end{cases}$$

Using this function it is relatively easy to expand the 'local' definitions of statement structure to the context of complete programs. We demonstrate this

with the function $parent_s$ which computes the parent of a given statement. Let ft be a function table and s and x be statements. We extend the definition of $parent_s$ in the following way.

$$parent_s(s, ft) = \begin{cases} parent_s(s, f.body) & \text{if } fos(s, ft) = \lfloor f \rfloor \\ None & \text{otherwise} \end{cases}$$

In a similar way we can expand the remaining definitions from Section 5.4.1. We omit the details here.

Isabelle Observe that we overload the notation to minimize the number of symbols which the reader has to remember. In Isabelle / HOL there exists a completely new set of functions which work in the context of function tables instead of single statements.

5.4.3 Valid Program Rest

In this section we use the definitions from the previous section to define a predicate for the set of valid program rests. Then we will prove an important structural invariant about the program rest of $C0$ machines.

Definition 5.35 (Valid program rest) We define the predicate $progrest\surd :: stmt \times functableT \mapsto \mathbb{B}$ which checks whether a given statement s forms a valid program rest with respect to a given function table ft. We set $progrest\surd(s, ft) = true$ if all statements in the program rest are present in the function table and followed by their successor statement. Formally, we have

$$\frac{successors\surd(s2l_{ns}(s), ft)\ (\forall x \in s2l_{ns}(s) : fos(x, ft) \neq None)}{progrest\surd(s, ft)}$$

Lemma 5.15 *All statements in a function table fulfill (by definition) the predicate* $successors\surd$.

$$fos(s, ft) \neq None \Longrightarrow successors\surd(s2l_{ns}(s), ft)$$

Proof We prove this lemma by induction on s. The different cases follow directly from the definition of valid successors. We do not present the details here.

Theorem 5.16 (Transition function preserves valid program rests) *Let te be a type name environment, gst a symbol table for the global variables, ft be a valid function table, m a memory configuration, and c.prog a valid program rest in the context of ft. Then, the C0 transition function* δ_{C0} *preserves the predicate* $progrest\surd$. *This means in particular that in the new program rest all statements (except for return statements) are followed by their successor statement. Formally, this means*

$$progrest\surd(c.prog, ft) \wedge ft \in valid_{ft}(te, gst) \wedge \delta_{C0}(te, ft, c) = \lfloor c' \rfloor \\ \Longrightarrow progrest\surd(c'.prog, ft)$$

Proof We prove this theorem by induction on the program rest $c.prog$.

Case 1: $c.prog = Skip$
In this case, the program rest is unchanged by the transition function. Thus, the claim of the theorem is obviously true.

Case 2: $c.prog = Comp(s_1, s_2)$
If $s_1 \neq Skip$ the program rest of the new configuration has the structure $Comp(c''.prog, s_2)$ where $c'' = the(\delta_{\text{C0}}(c[prog := s_1]))$. We cannot prove $progrest\surd$ for $c''.prog$ and s_2 completely separately. This would be simple because $progrest\surd(c''.prog)$ follows from the induction hypothesis and $progrest\surd(s_2)$ from $progrest\surd(Comp(s_1, s_2))$.

Instead, we have to additionally prove that the first statement in $s2l_{\text{ns}}(s_2)$ is a valid successor of the last statement in $s2l_{\text{ns}}(c''.prog)$. The interesting case is when $s2l_{\text{ns}}(s_1)$ contains just a single statement, i.e., $s2l_{\text{ns}}(s_1) = [s]$, because otherwise the last statement of the first part of the composition statement would not change ($last(s2l_{\text{ns}}(c''.prog)) = last(s2l_{\text{ns}}(s_1))$) and thus the proof would become trivial. In the interesting case we have to make a case distinction on s.

If $is\_Ass(s)$, $is\_Ass_{\text{AL}}(s)$, $is\_PAlloc(s)$, $is\_Return(s)$, or $is\_Asm(s)$ the proof is trivial because $s2l_{\text{ns}}(c''.prog) = [\,]$ and thus, there is nothing to show.

If $is\_SCall(s)$ we know from the requirements on valid functions (cf. Definition 5.14 on page 78) that $is\_Return(last(s2l_{\text{ns}}(c''.prog))) = true$. Thus, we have nothing to show for $successors\surd$ because return statements are allowed to precede arbitrary statements.

If $is\_Loop(s)$ and the loop condition is $true$ we have $last(s2l_{\text{ns}}(c''.prog)) = s = last(s2l_{\text{ns}}(s_1))$, i.e., the last statement of the left part of the compound statement has not changed. If the loop condition is $false$ we have $s2l_{\text{ns}}(c''.prog) = [\,]$ and the proof is obvious.

If $is\_Ifte(s)$ the last statement of $s2l_{\text{ns}}(c''.prog)$ is the last statement of one of the branches of the conditional statement. In both cases we know by definition 5.33 that their successor statement is the successor statement of the conditional statement. Because $hd(s2l_{\text{ns}}(s_2))$ was a valid successor statement for the conditional statement we can conclude that it is also a valid successor of the new last statement in $s2l_{\text{ns}}(c''.prog)$.

This finishes the proof of the inductive case.

Case 3: $c.prog = Ass(e_l, e_2)$, $c.prog = Ass_{\text{AL}}(e_l, c)$, $c.prog = Asm(il)$, $c.prog = PAlloc(e_l, tn)$, or $c.prog = Return(e)$
In these cases we have by the definition of the transition function $c'.prog = Skip$; thus there is nothing to show.

Case 4: $c.prog = SCall(vn, fn, pl)$
In this case we have $c'.prog = f.body$ where $f = the(map\text{-}of(ft, fn))$ is the called function. From Lemma 5.15 we know that the function body fulfills the predicate $successors\surd$; this proves the first requirement of $progrest\surd$. Obviously, the function body is also present in the function table, which proves the second requirement.

Case 5: $c.prog = Ifte(e, s_1, s_2)$, $c.prog = Loop(e, lb)$
For these cases we know that the new program rest is either s_1 or s_2 (for

the conditional statement) or *lb* (for the loop). We know also that these statements are present in the function table because we can conclude from $progrest\surd(c.prog, ft)$ that their parent statements are in the function table. Using Lemma 5.15 we can easily show that the statements in the new program rest are followed by the correct successors. This proves the last case of the theorem. q.e.d.

5.5 Valid Configurations

As we have seen in Theorem 5.14 on page 87 we have sometimes quite big assumptions in lemmas where most of these assumptions are about validity of certain parts of the configuration. We combine this, the requirements on valid successors, and some additional properties in a predicate which defines the set of *valid configurations*. This will allow keeping the list of preconditions in lemmas as short as possible.

5.5.1 Definitions

Before defining the predicate for valid configurations we introduce two more auxiliary predicates. We start with the definition of valid stacks.

Definition 5.36 (Valid stack) Let *ft* be a function table and *lms* a list of memory frames (i.e., the stack of local memory frames of a *C*0 configuration). This stack of local memories is called valid if it goes with the program rest, i.e., if the sequence of statements in the program rest is in correspondance with the sequence of symbol tables in the local memory stack.

In more detail: we require for all statements in the program rest that the symbol table of the function which they belong to concerning their statement identifier (cf. the definition of *fos* on page 91) matches the symbol table which belongs to the statements position in the program rest. More formally, this requires for all statements between the i-th and the $i+1$-th return statement in the program rest that they origin from the function which belongs to stack frame $|lst| - i - 1$. For example, for the statements *before the first return* in the program rest the symbol table of their function has to match the top most local symbol table in the local memory stack (cf. Figure 5.2).

Formally, we define the predicate by induction on the list of statements. For the empty program rest, all stacks are valid.

$$stack\surd :: functableT \times mframe\ list \times stmt\ list \mapsto \mathbb{B}$$

$$\frac{}{stack\surd(ft, lst, [\,])}$$

For the inductive case, we do a case split on the first statement in the program rest.

$$\frac{\begin{array}{c} lst \neq [\,] \qquad st_{\text{fun}}(snd(the(fos(h, ft)))) = hd(lst).st \\ is\_Return(h) \longrightarrow stack\surd(ft, butlast(lst), t) \\ \neg is\_Return(h) \longrightarrow stack\surd(ft, lst, t) \end{array}}{stack\surd(ft, lst, h\#t)}$$

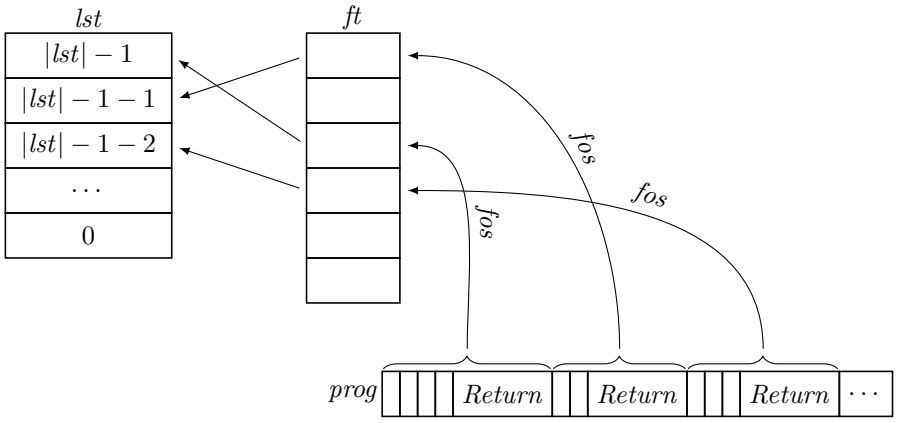


Figure 5.2: Correspondence Between Stack and Program Rest *prog*

As a last predicate before the definition of valid configurations we introduce the set of *valid return destinations*. This predicate argues about the return destinations (cf. Section 4.2.1 on page 44) which are stored in the stack of local memory frames and ensures that they are valid g-variables and that their type matches the return type of the corresponding return statement in the program rest.

Definition 5.37 (Valid return destinations) Let te be a type name environment and mc a memory configuration. We define the predicate $rdest\surd$, which ensures that the list of return destinations in mc matches a given list of statements. This definition is done by induction on the list of statements.

$$rdest\surd :: tenv \times memconf \times stmt\ list \mapsto \mathbb{B}$$

$$\frac{}{rdest\surd(te, mc, [\,])}$$

For the inductive case we do a case split on the head for the statement list. We first treat the case that the head of the statement list is a return statement.

$$\frac{\begin{array}{c} h = Return(re) \\ tmatch_{\text{ass}}(ty_{\text{g}}(sc(mc), snd(lm_{\text{top}}(mc))), type(te, gst(mc), lst_{\text{top}}(mc), re)) \\ snd(lm_{\text{top}}(mc)) \in gvars\surd(sc(mc)) \\ named_{\text{g}}(snd(lm_{\text{top}}(mc))) \qquad rdest\surd(te, mc\,[lm := butlast(mc.lm)]\,, t) \end{array}}{rdest\surd(te, mc, h\#t)}$$

If h is not a return statement the definition gets much simpler.

$$\frac{\neg is\_Return(h) \qquad rdest\surd(te, mc, t)}{rdest\surd(te, mc, h\#t)}$$

Finally, we can define the set of valid configurations.

Definition 5.38 (Valid configurations) Let te be a type name environment, ft a function table, and c a $C0$ configuration. We call c a valid configuration with respect to te and ft iff

1. the function table ft is valid with respect to the global symbol table,
2. the program rest of c is valid,
3. the number of return statements in the program rest is not greater than the recursion depth of c,
4. the stack of c is valid,
5. the type name environment te is valid,
6. the global symbol table of c is valid,
7. all local symbol tables of c belong to some function in ft,[1]
8. the types of all heap variables are valid types,
9. all memory frames of c are type correct, and
10. the return destinations of c are valid.

Formally, we define

$$conf\surd :: tenv \times functableT \mapsto conf_{\text{C0}}\ set$$

$$\frac{\begin{array}{c} ft \in valid_{ft}(te, gst(c.mem)) \\ progrest\surd(c.prog, ft) \qquad \#ret_{\text{top}}(s2l(c.prog)) < |c.mem.lm| \\ stack\surd(ft, map(fst, c.mem.lm), s2l_{\text{ns}}(c.prog)) \\ te \in valid_{\text{tenv}} \qquad gst(c.mem) \in valid_{\text{st}}(te) \\ \forall lm \in c.mem.lm : \exists f \in map(snd, ft) : st_{\text{fun}}(f) = fst(lm) \\ \forall i < |c.mem.hm.st| : valid_{\text{ty}}(te, snd(c.mem.hm.st!i)) \\ ty\surd_{\text{mem}}(te, sc(c.mem), c.mem.gm) \\ \forall lm \in c.mem.lm : ty\surd_{\text{mem}}(te, sc(c.mem), fst(lm)) \\ ty\surd_{\text{heap}}(te, sc(c.mem), c.mem.hm) \qquad rdest\surd(te, c.mem, s2l_{\text{ns}}(c.prog)) \end{array}}{c \in conf\surd(te, ft)}$$

5.5.2 Maintaining Valid Configurations

In this section, we present a theorem which states that the $C0$ transition function preserves the predicate $conf\surd$. In Isabelle, the proof of this theorem is quite complex and requires several auxiliary lemmas. However, we present here only the main theorem and omit its proof. The most complicated part of the proof has already been done in Theorem 5.16.

Theorem 5.17 (Transition function preserves valid configurations) *Let te be a type name environment, ft a function table, and c a valid $C0$ configuration with respect to te and ft. If executing the $C0$ transition function*

[1] This implies that they are valid symbol tables.

gives a next configuration c', i.e., $\delta_{C0}(te, ft, c) = \lfloor c' \rfloor$, then c' is also a valid $C0$ configuration. Formally,

$$c \in conf\surd(te, ft) \wedge \delta_{C0}(te, ft, c) = \lfloor c' \rfloor$$
$$\Longrightarrow c' \in conf\surd(te, ft)$$

Chapter 6

Target Language

Contents

In this chapter we describe the instruction set architecture and assembly language of the verified architecture microprocessor (VAMP) which is the target architecture of the verified $C0$ compiler. The VAMP [BJK+03, BJK+06, DHP05, Dal06] is a formally verified DLX-like processor. It is based on the designs and proofs from [MP00] and [Kro01].

The VAMP has first been implemented (on the gate-level) and verified using the PVS [OSR92] theorem prover. In the Verisoft project [Ver03], its implementation and correctness proof have been ported to Isabelle / HOL [NPW02]. Thereby, the proofs have been considerably simplified using automatic methods [Tve05]. The implementation of the VAMP has been synthesized on a Xilinx FPGA [BJK+06, Lei02].

The VAMP realizes the full DLX instruction set from [HP96], delayed branch with one delay slot to support pipelining of instruction fetches, a Tomasulo scheduler [Tom67, Kro01], maskable nested precise interrupts, and a memory management unit for virtual memory support. In addition, the PVS version of the VAMP supports separate instruction and data caches [Bey05] and a pipelined fully IEEE compatible dual precision floating point unit with variable latency. The correctness of a simple cache system for the VAMP processor has also been verified in Isabelle / HOL [Mül07] but has not yet been integrated into the main correctness proof. However, compared with the PVS work of [Bey05] the usage of automatic proof could be greatly increased. Currently, the VAMP is being extended with a translation look aside buffer (TLB). [HIP05] describes the integration of external devices into the VAMP ISA.

In this thesis we will refer to the Isabelle version of the VAMP without TLB, caches or devices.

In Section 6.1 we briefly introduce the VAMP instruction set architecture. We continue in Section 6.2 with a closer look at the semantics of the VAMP

assembly language. In Section 6.3 we list requirements which allow to prove that the VAMP ISA simulates the VAMP assembly language. We conclude in Section 6.4 with a verification methodology for VAMP assembly programs which is used for the low-level correctness proofs in Chapters 9 and 10.

Observe that the VAMP ISA and assembly semantics have not been developed as part of this thesis. They have been written by several other persons during the Verisoft project. The requirements from Section 6.3 have been developed in their original form by A. Tsyban. However, they have been slightly adapted and generalized for this thesis. The verification methodology in Section 6.4 has been developed completely in the scope of this thesis.

6.1 VAMP Instruction Set Architecture

In this section we give a brief overview on the VAMP instruction set architecture (ISA) as defined in the Isabelle formalization. We do not present any details; we introduce the ISA rather to argue why the introduction of a VAMP assembly machine in Section 6.2 is useful and what kind of special restrictions have to be met to be able to prove the equivalence between the assembly semantics and the ISA.

A configuration of the VAMP ISA consists of

- $GPR :: bv_5 \mapsto bv_{32}$, $SPR :: bv_5 \mapsto bv_{32}$
  Two register files for general purpose and special purpose registers. Each register file contains thirty two 32-bit registers; the registers are addressed by 5-bit addresses. Register $GPR(0)$ is always 0.

  The special purpose registers are

  - SR, ESR, ECA, EPC, EDPC, and EData for interrupt handling,
  - RM, IEEEf, and FCC for the floating point unit, and
  - PTO, PTL, EMODE, and MODE for virtual memory management.

- $DPC :: bv_{32}$, $PCP :: bv_{32}$
  Two program counters which implement the delayed branch mechanism. DPC is the current (delayed) program counter which stores the address where the current instruction is fetched from; PCP stores the address of the next instruction.

- $mem :: bv_{29} \mapsto bv_{64}$
  A double word addressed memory.

Transitions of the VAMP ISA are defined by its transition function $\delta_{\text{ISA}} :: conf_{\text{ISA}} \times bv_{19} \mapsto conf_{\text{ISA}}$. Let d be a VAMP ISA configuration and $intr_{\text{e}}$ a bit vector representing 19 flags for external interrupts. Then the next configuration d' is computed by the VAMP ISA transition function: $\delta_{\text{ISA}}(d, intr_{\text{e}}) = d'$. Instructions are stored as 32-bit bit vectors. If the MODE register is zero the VAMP executes in system mode (without address translation), otherwise it executes in user mode where address translation is activated. In user mode, a program is not allowed to access special purpose registers except for SR, RM, IEEEf, and FCC.

Due to the pipelined implementation of the VAMP self modifying code poses problems and the following sync criterion is required by the VAMP correctness

Table 6.1: Implemented Interrupts

index	name	maskable	type
0	reset	no	abort
1	illegal instruction	no	repeat
2	misalignment	no	repeat
3	page fault on fetch	no	repeat
4	page fault load store	no	repeat
5	trap	no	continue
6	arithmetic overflow	yes	continue
7 ... 12	FPU interrupts	yes	continue
13 ... 31	external interrupts	yes	continue

proof (cf. [BJK+06, Section 6.3]): between the modification of an instruction by a write access to the memory at address a and the fetch from the same address a the programmer (or compiler) is required to execute a special *sync* instruction. This *sync* instruction has the effect that all following instructions are stalled until the VAMP pipeline is empty and all pending memory accesses have been completed. Thus, after execution of the *sync* instruction we are guaranteed to fetch the updated instruction from address a.

Table 6.1 shows the interrupts supported by the VAMP. Observe that the floating point interrupts (index 7 to 12) are not covered by the ISA which is used in the Verisoft project. For a detailed explanation of the different interrupts see [Dal06, Chapter 3]. We highlight a few conditions which trigger interrupts:

- execution of illegal instructions, i.e., opcodes which are not defined by the instruction set
- misaligned memory accesses, i.e., instruction fetch from addresses which are not aligned to four and read or write instructions to addresses which are not aligned to the width (in bytes) of the access
- page faults, i.e., access to memory pages which are currently swapped out or (in user mode) which are not mapped with sufficient rights by the page table
- execution of the trap instruction
- arithmetic overflows when executing certain arithmetic instructions which have overflow signaling enabled

6.2 VAMP Assembly Language

Verification of programs at the ISA level is hard – for most kinds of programs unnecessary hard – for several reasons. We highlight the most important ones and show how the assembly semantics, which we will introduce in this section, solves these problems by abstraction from the VAMP ISA.

- The representation of instructions as bit vectors requires a quite complex instruction decoding scheme. This causes several non-trivial but nevertheless boring proof steps in order to extract the source and destination

registers as well as the instruction mnemonic from the instruction bit vector.

The VAMP assembly semantics represents instructions as an abstract data type. Each instruction is realized via its own constructor. Source and destination registers as well as immediate constants are represented by numerical parameters to this constructor.

- Usually, programs execute mostly numerical computations instead of bit vector operations. Thus, it costs quite some effort to convert bit vectors back and forth to numerical values for each machine step. The same holds for the program counters and addresses in general.

  In the VAMP assembly semantics, registers and memory content are represented as numerical values. Because – at least in the Verisoft project – programs make more frequently use of integers than of natural numbers, general purpose registers and the memory contents are modeled as integers. Only the program counters are modeled as natural numbers.

- At the ISA level there are many implicit conditions which have to be met in order not to cause unwanted interrupts.

  In the VAMP assembly machine interrupts are not modeled and thus one does not have to worry about them during verification. Obviously, this prevents verification of some programs at the assembly level; namely those which make explicit use of interrupts. Nevertheless, for most programs this is a convenient level of abstraction.

In order to transfer results from the assembly layer down to the ISA layer we need to justify the abstractions from above. In Section 6.3 we will explain in more details what additional properties need to be proved for an assembly program to ensure its correctness also on the ISA layer.

6.2.1 Instructions

Instructions of the VAMP assembly machine are modeled by the abstract data type *instr*. Each instruction is modeled by a separate constructor with parameters for register names and immediate constants. To distinguish register names from normal numbers we will prepend them with a lowercase r like in $r16$ which stands for register 16.

We list the constructors of *instr* and their meaning in Tables 6.2, 6.3, 6.4, 6.5, and 6.6; the detailed mathematical semantics for the instructions will be given in Section 6.2.5. For better readability of the constructors we introduce the type aliases $reg = \mathbb{N}_5$ for register names, $imm_{16} = \mathbb{Z}_{16}$ and $imm_{26} = \mathbb{Z}_{26}$ for the value of integer immediate constants, and $sa_5 = \mathbb{N}_5$ for immediate constants denoting shift distances.

Isabelle In Isabelle, the type for immediate constants is $\mathbb{Z}$, i.e., there is no range restriction. However, to ensure that immediate constants are not too small or too big, there exists a predicate `is_instr` with type $instr \mapsto \mathbb{B}$. We omit its lengthy definition here by using restricted types for the immediate constants and register names.

There is no special 'no operation' (nop) instruction for the VAMP assembly semantics. We use $ori(r1, r1, 0)$ for this purpose.[1] However, we will often simply write *nop* instead of $ori(r1, r1, 0)$.

For later use we introduce several simple predicates and functions on VAMP assembly instructions.

$$\begin{aligned} is\_store(i) &= i \in \{sb, sh, sw\} \\ is\_load(i) &= i \in \{lb, lh, lw, lbu, lhu\} \\ is\_loadstore(i) &= is\_store(i) \vee is\_load(i) \\ width_{\mathrm{a}}(i) &= \begin{cases} 4 & \text{if } i \in \{lw, sw\} \\ 2 & \text{if } i \in \{lh, lhu, sh\} \\ 1 & \text{otherwise} \end{cases} \end{aligned}$$

Additionally, we define functions $REG_i :: instr \mapsto \mathbb{N}$ and $IMM :: instr \mapsto \mathbb{N}$ which extract the i-th register parameter and the immediate constants of instructions.

6.2.2 Memory Model

Single memory cells of the VAMP assembly are modeled by an axiomatic type $mcell_{\mathrm{asm}}$. We introduce two uninterpreted functions $cell2int :: mcell_{\mathrm{asm}} \mapsto \mathbb{Z}$ and $int2cell :: \mathbb{Z} \mapsto mcell_{\mathrm{asm}}$. The expected behavior of these functions is defined by the following two axioms.

Axiom 6.1 $cell2int(int2cell(i)) = i$

Axiom 6.2 $int2cell(cell2int(m)) = m$

To handle storage of instructions in the memory we introduce two more functions $int2instr :: \mathbb{Z} \mapsto instr$ and $instr2int :: instr \mapsto \mathbb{Z}$. The formal definition of these two functions is based on the instruction encoding scheme of the VAMP ISA. The details are only of interest for the simulation proof between assembly and ISA layer (cf. Section 6.3) and will not be given in this thesis.

We combine the functions for conversion between instructions and integers and between memory cells and integers in the following way.

Definition 6.1 (Converting instructions and memory cells) We define two functions to convert between instructions given by the datatype *instr* and memory cells $mcell_{\mathrm{asm}}$.

$$\begin{aligned} cell2instr &:: mcell_{\mathrm{asm}} \mapsto instr \\ instr2cell &:: instr \mapsto mcell_{\mathrm{asm}} \end{aligned}$$

$$\begin{aligned} instr2cell(i) &= int2cell(instr2int(i)) \\ cell2instr(c) &= int2instr(cell2int(c)) \end{aligned}$$

[1] There are several other ways to encode a nop instruction for the VAMP processor. However, the *ori* instruction was the only one which allowed us to prove (formally) that it has no-op semantics without needing any preconditions (like source registers being in the correct range).

Using the definitions of *int2instr* and *instr2int* (which are not given in this thesis) we can prove the following lemma.

Lemma 6.3 $\mathit{cell2instr}(\mathit{instr2cell}(i)) = i$

Proof From the definitions and from Axiom 6.1 we conclude

$$\begin{aligned}\mathit{cell2instr}(\mathit{instr2cell}(i)) &= \mathit{int2instr}(\mathit{cell2int}(\mathit{int2cell}(\mathit{instr2int}(i))))\\ &= \mathit{int2instr}(\mathit{instr2int}(i))\\ &= i \qquad \text{(by definition)}\end{aligned}$$

q.e.d.

6.2.3 Integers vs. Natural Numbers

Bit vectors can represent numbers in two different ways (cf. Section 1.2.4). In normal binary representation for natural numbers or in two's complement representation for integers. Thus, the same bit vector has a completely different meaning depending on its interpretation as natural number or integer.

The VAMP assembly semantics stores register contents as numbers, more precisely as integers. So, if we want to interpret register contents as natural numbers rather than integers we have to use conversion functions. These conversion functions simulate numerically the convertion from integers to bit vectors and then back to naturals. The same holds (of course in the opposite order) if we try to store natural numbers in a register.

Definition 6.2 (Convertion: integers and natural numbers) We define the conversion from integers to natural numbers and vice versa by the two functions $\mathit{i2n} :: \mathbb{Z} \mapsto \mathbb{N}$ and $\mathit{n2i} :: \mathbb{N} \mapsto \mathbb{Z}$.

$$\mathit{i2n}(i) = \begin{cases} i + 2^{32} & \text{if } i < 0 \\ i & \text{otherwise} \end{cases}$$

$$\mathit{n2i}(n) = \begin{cases} n - 2^{32} & \text{if } 2^{31} \leq n < 2^{32} \\ n & \text{otherwise} \end{cases}$$

The semantics of the conversion functions is sketched in Figures 6.1 and 6.2. Observe that the conversion functions which we define here are only correct if the number to be converted is in the range of 32-bit numbers. If so, that half of the domain which is not present in the image of the function is just *moved* to that half of the image which is not present in the domain. For example, *i2n* moves the negative interval $[-2^{31}, \ldots, -1]$ of integers to the interval $[2^{31}, \ldots, 2^{32} - 1]$ of naturals.

One can choose to define assembly instruction semantics based on integers or based on naturals. Either decision would simplify reasoning about arithmetic on one type of numbers. Unfortunately, the authors of the VAMP assembly semantics have chosen a mixture of both concepts: register values are stored as integer values and semantics of arithmetic operators are defined based on natural numbers. This creates unnecessary (and sometimes tedious) proof obligations involving *i2n* and *n2i*. Fortunately, when using the VAMP assembly semantics for verification of concrete assembly code, most of these conversions occur pairwise and can be eliminated using the following lemmas.

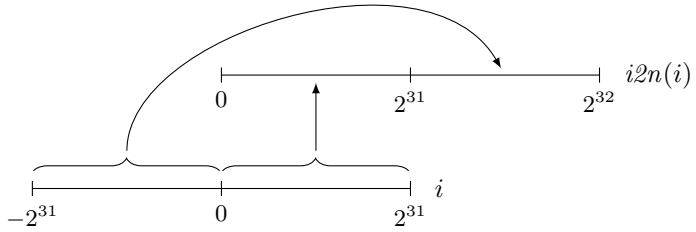


Figure 6.1: Conversion From Integers to Natural Numbers

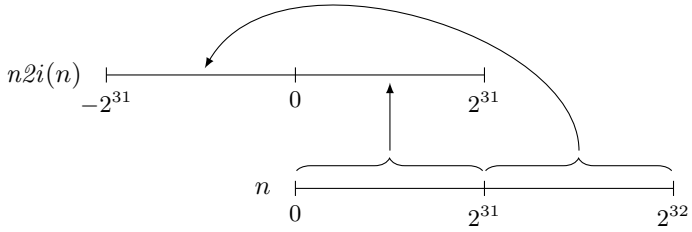


Figure 6.2: Conversion From Natural Numbers to Integers

Lemma 6.4 *For 32-bit integers conversion to natural numbers and then back to integers is transparent.*

$$-2^{31} \leq i < 2^{31} \Longrightarrow \mathit{n2i}(\mathit{i2n}(i)) = i$$

PROOF We omit the proof of this property since it follows directly from the definitions.

Lemma 6.5 *Conversion from natural number to integers and then back to natural numbers is transparent.*

$$\mathit{i2n}(\mathit{n2i}(n)) = n$$

PROOF We omit the proof of this property.

6.2.4 Configuration

Configurations $d :: \mathit{conf}_{\text{asm}}$ of the VAMP assembly semantics consist of five components:

- two program counters: $d.\mathit{dpc} :: \mathbb{N}_{32}$, $d.\mathit{pcp} :: \mathbb{N}_{32}$,
- two register files for general purpose and special purpose registers: $d.\mathit{gpr} :: \mathbb{Z}_{32}\ \mathit{list}$, $d.\mathit{spr} :: \mathbb{Z}_{32}\ \mathit{list}$, and
- a word addressed memory: $d.\mathit{mm} :: \mathbb{N} \mapsto \mathit{mcell}_{\text{asm}}$.

Observe that register $gpr!0$ is tied to zero. For a formalization of the VAMP assembly semantics, there are two ways to realize this: we can initialize the register with zero and henceforth ignore all writes to it or we modify the function which reads out register values and implement a special case for this register. The second solution has the advantage that we do not need to keep track of an invariant about the value of $gpr!0$ and has been implemented in the formalization in Isabelle / HOL.

Now, we introduce a technical predicate which defines the set of *valid* VAMP assembly configurations.

Definition 6.3 (Valid VAMP configuration) We define VAMP assembly configurations $d :: conf_{\text{asm}}$ to be valid if both register files have length 32 and all memory cells of the memory contain data in the domain of 32-bit integers. Formally, we introduce the predicate $conf_{\text{asm}}\surd :: conf_{\text{asm}} \mapsto \mathbb{B}$.

$$\frac{|d.gpr| = 32 \qquad |d.spr| = 32 \qquad \forall a : -2^{31} \leq cell2int(d.mm(a)) < 2^{31}}{conf_{\text{asm}}\surd(d)}$$

Isabelle In Isabelle the definition of valid VAMP assembly configurations is slightly more complicated because it additionally contains requirements about the value of registers and program counters. In this thesis we omit these restrictions and instead use types with bounded domain for the values.

We introduce some predicates and functions related to configurations of the VAMP assembly machine.

Definition 6.4 (A given memory region is unchanged) Let $m :: \mathbb{N} \mapsto mcell_{\text{asm}}$ and $m' :: \mathbb{N} \mapsto mcell_{\text{asm}}$ be memories of the VAMP assembly machine and $a :: \mathbb{N}$ and $b :: \mathbb{N}$ word addresses. We say that the memory between a and b has not been changed from memory m to memory m' if the following predicate holds.

$$\frac{\forall i : a \leq i < b \longrightarrow m'(i) = m(i)}{unchngd_{\text{mem}}(m, m', a, b)}$$

Definition 6.5 (Only given region is changed) Let $m :: \mathbb{N} \mapsto mcell_{\text{asm}}$ and $m' :: \mathbb{N} \mapsto mcell_{\text{asm}}$ be memories of the VAMP assembly machine and $a :: \mathbb{N}$ and $b :: \mathbb{N}$ word addresses. We say that only the memory between a and $a + l$ has been changed from memory m to memory m' if the following predicate holds.

$$\frac{\forall i : i < a \vee a + l \leq i \longrightarrow m'(i) = m(i)}{memchngd(m, m', a, l)}$$

6.2.5 Semantics

In this section we will define the semantics of the VAMP assembly language. We start with the semantics of single instructions and define then a transition function.

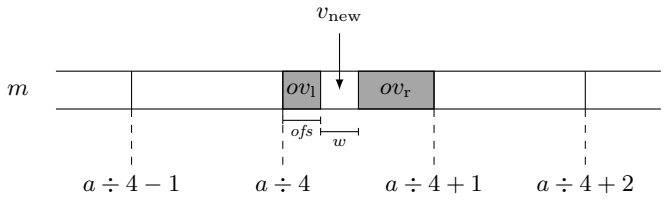


Figure 6.3: VAMP Assembly Machine: Storing

Instruction Semantics

To define the semantics of the VAMP assembly instructions we start with the introduction of several auxiliary functions which specify the behavior of load and store instructions.

We define the effective address of a memory instruction i in configuration d by the function $ea :: \mathit{conf}_{\text{asm}} \times \mathit{instr} \mapsto \mathbb{N}$. Formally, it is defined by

$$ea(d, i) = (i2n(d.gpr!REG_2(i)) + i2n(IMM(i))) \bmod_{\text{u}} 2^{32}.$$

Memory accesses have to be properly aligned. This means for a memory access with a width of w bytes that w divides the address a of the access. For properly aligned memory accesses we know that they are completely contained in a single 32-bit memory word; i.e., they do not cross word boundaries. Thus, we can define the access in two steps: we first load a complete 32-bit word from the memory and in a second step we extract the interesting part.

For the second step we introduce the notation $x_{[o,w]}$. Its meaning is that we extract w bytes from the word x starting at byte offset o.

$$x_{[o,w]} = (i2n(x) \div 2^{8 \cdot o}) \bmod_{\text{u}} 2^{8 \cdot w}$$

Definition 6.6 (Loading) We define the functions $load^{\text{u}}$ and $load^{\text{s}}$ to abbreviate the semantics of unsigned and signed load accesses to the memory. For signed load accesses we additionally convert the result to a 32 bit wide bit vector and then back to an integer.

$$load^{\text{u}} :: (\mathbb{N} \mapsto \mathit{mcell}_{\text{asm}}) \times \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$$
$$load^{\text{s}} :: (\mathbb{N} \mapsto \mathit{mcell}_{\text{asm}}) \times \mathbb{N} \times \mathbb{N} \mapsto \mathbb{Z}$$
$$load^{\text{u}}(m, a, w) = \mathit{cell2int}(m(a \div 4))_{[a \bmod_{\text{u}} 4, w]}$$
$$load^{\text{s}}(m, a, w) = [\mathit{bin}_{32}(8 \cdot w)\big(\mathit{cell2int}(m(a \div 4))_{[a \bmod_{\text{u}} 4, w]}\big)]$$

Store instructions are simpler than load instructions as we do not distinguish between signed and unsigned variants. Similar to loads, a store affects exactly one word in the memory. To support storage of parts of words, we split the original data word at the destination address in three parts. The medium part is replaced by the new value; the upper and lower parts (if they exist) remain unchanged.

Definition 6.7 (Storing) We define the function $\mathit{store} :: (\mathbb{N} \mapsto \mathit{mcell}_{\text{asm}}) \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z} \mapsto (\mathbb{N} \mapsto \mathit{mcell}_{\text{asm}})$ to abbreviate the semantics of store accesses

Table 6.2: Memory Instructions and Constant Loading

Instruction (i)	Meaning	Effect ($d' = exec(d, i)$)
$lb(reg, reg, imm_{16})$	load byte	$d'.gpr!r_1 := load^{s}(d.mm, ea(d, i), 1)$
$lh(reg, reg, imm_{16})$	load half word	$d'.gpr!r_1 := load^{s}(d.mm, ea(d, i), 2)$
$lw(reg, reg, imm_{16})$	load word	$d'.gpr!r_1 := load^{s}(d.mm, ea(d, i), 4)$
$lbu(reg, reg, imm_{16})$	load byte unsigned	$d'.gpr!r_1 := load^{u}(d.mm, ea(d, i), 1)$
$lhu(reg, reg, imm_{16})$	load half word unsigned	$d'.gpr!r_1 := load^{u}(d.mm, ea(d, i), 2)$
$sb(reg, reg, imm_{16})$	store byte	$d'.mm := store(d.mm, ea(d, i), 1, d.gpr!r_1)$
$sh(reg, reg, imm_{16})$	store half word	$d'.mm := store(d.mm, ea(d, i), 2, d.gpr!r_1)$
$sw(reg, reg, imm_{16})$	store word	$d'.mm := store(d.mm, ea(d, i), 4, d.gpr!r_1)$
$lhgi(reg, imm_{16})$	load high imm.	$d'.gpr!r_1 := imm \cdot 2^{16}$
$lhg(reg, reg)$	load high	$d'.gpr!r_1 := d.gpr!r_2 \cdot 2^{16}$

For all instructions holds $d'.dpc := d.pcp$ and $d'.pcp := d.pcp +_{32} 4$.

to the memory. Let m be a memory, a the byte address of the destination, w the width of the access in bytes, and v the value to be written. We use the abbreviation $ofs = a \bmod_u 4$ for the offset of address a inside the accessed word.

To store v we first have to read the original value $ov = i2n(cell2int(m(a \div 4)))$ from the memory and clip the two parts which should not be overwritten: the lower part $ov_l = ov \bmod_u 2^{8 \cdot ofs}$ and the higher part $ov_h = (ov \div 2^{8 \cdot (ofs+w)}) \cdot 2^{8 \cdot (ofs+w)}$. Then we shift and clip the new value accordingly: $v_{new} = (i2n(v) \bmod_u 2^{8 \cdot w}) \cdot 2^{8 \cdot ofs}$. Finally, we combine the three parts and update the memory

$$store(m, a, w, v) = m\left[(a \div 4) := n2i((ov_h + v_{new} + ov_l) \bmod_u 2^{32})\right].$$

This is illustrated in Figure 6.3 for a store at address a with width $w = 1$ and byte offset $ofs = a \bmod_u 4 = 1$.

Definition 6.8 (Instruction execution) The semantics of single VAMP assembly instructions are defined by the function $exec :: conf_{asm} \times instr \mapsto conf_{asm}$. We give its definition in Tables 6.2, 6.3, 6.4, 6.5, and 6.6.

In these tables we use r_i to refer to the i-th register parameter of the instruction's constructor, imm to refer to its immediate constant, and sa to refer to the shift amount. With d we denote the original configuration and with $d' = exec(d, i)$ the new configuration after executing instruction i. All parts x of the configuration which are not mentioned otherwise stay unchanged, i.e., $d'.x = d.x$.

Lemma 6.6 (Execution preserves $conf_{asm}\surd$) *Executing an instruction i in a valid VAMP assembly configuration d leads to a new configuration d' which is again valid.*

$$conf_{asm}\surd(d) \Longrightarrow conf_{asm}\surd(exec(d, i))$$

Table 6.3: Control and Special Instructions

Instruction (i)	Meaning	Effect ($d' = exec(d, i)$)
$beqz(reg, imm_{16})$	branch if equal zero	$d'.pcp := d.pcp +_{32} ((d.gpr!r_1 = 0)?i2n(imm) : 4)$
$bnez(reg, imm_{16})$	branch if not equal zero	$d'.pcp := d.pcp +_{32} ((d.gpr!r_1 \neq 0)?i2n(imm) : 4)$
$jal(imm_{26})$	jump and link	$d'.pcp := d.pcp +_{32} i2n(imm)$, $d'.gpr!31 := n2i(d.pcp +_{32} 4)$
$jalr(reg)$	jump and link register	$d'.pcp := i2n(d.gpr!r_1)$, $d'.gpr!31 := n2i(d.pcp +_{32} 4)$
$j(imm_{26})$	jump	$d'.pcp := d.pcp +_{32} i2n(imm)$
$jr(reg)$	jump register	$d'.pcp := i2n(d.gpr!r_1)$
$movs2i(reg, reg)$	move special to general purpose register	$\begin{cases} d'.gpr!r_1 := d.spr!r_2 & \text{if } spr\surd(d, r_2) \\ d' := d & \text{otherwise} \end{cases}$
$movi2s(reg, reg)$	move general to special purpose register	$\begin{cases} d'.spr!r_1 := d.gpr!r_2 & \text{if } spr\surd(d, r_1) \\ d' := d & \text{otherwise} \end{cases}$
$trap(imm_{26})$	trap	$d' := d$
rfe	return from exception	$d' := d$

For all instructions except the identity cases ($d' := d$) we have $d'.dpc := d.pcp$.
In the rows for *movi2s* and *movs2i* we use the following predicate as abbreviation:
$spr\surd(d, r) = (d.spr!MODE \neq 0 \Longrightarrow r \in \{SR, RM, IEEEf, FCC\})$.

Table 6.4: Test Instructions

Instruction (i)	Meaning	Effect ($d' = exec(d, i)$)
$clri(reg)$	clear imm.	$d'.gpr!r_1 := 0$
$sgri(reg, reg, imm_{16})$	set if greater imm.	$d'.gpr!r_1 := (d.gpr!r_2 > imm)?1 : 0$
$seqi(reg, reg, imm_{16})$	set if equal imm.	$d'.gpr!r_1 := (d.gpr!r_2 = imm)?1 : 0$
$sgei(reg, reg, imm_{16})$	set if greater equal imm.	$d'.gpr!r_1 := (d.gpr!r_2 \geq imm)?1 : 0$
$slsi(reg, reg, imm_{16})$	set if less imm.	$d'.gpr!r_1 := (d.gpr!r_2 < imm)?1 : 0$
$snei(reg, reg, imm_{16})$	set if not equal imm.	$d'.gpr!r_1 := (d.gpr!r_2 \neq imm)?1 : 0$
$slei(reg, reg, imm_{16})$	set if less equal imm.	$d'.gpr!r_1 := (d.gpr!r_2 \leq imm)?1 : 0$
$seti(reg)$	set imm.	$d'.gpr!r_1 := 1$
$clr(reg)$	clear	$d'.gpr!r_1 := 0$
$sgr(reg, reg, reg)$	set if greater	$d'.gpr!r_1 := (d.gpr!r_2 > d.gpr!r_3)?1 : 0$
$seq(reg, reg, reg)$	set if equal	$d'.gpr!r_1 := (d.gpr!r_2 = d.gpr!r_3)?1 : 0$
$sge(reg, reg, reg)$	set if greater equal	$d'.gpr!r_1 := (d.gpr!r_2 \geq d.gpr!r_3)?1 : 0$
$sls(reg, reg, reg)$	set if less	$d'.gpr!r_1 := (d.gpr!r_2 < d.gpr!r_3)?1 : 0$
$sne(reg, reg, reg)$	set if not equal	$d'.gpr!r_1 := (d.gpr!r_2 \neq d.gpr!r_3)?1 : 0$
$sle(reg, reg, reg)$	set if less equal	$d'.gpr!r_1 := (d.gpr!r_2 \leq d.gpr!r_3)?1 : 0$
$set(reg)$	set	$d'.gpr!r_1 := 1$

for all instructions: $d'.dpc := d.pcp$, $d'.pcp := d.pcp +_{32} 4$.

Table 6.5: Arithmetic and Bitwise Instructions

Instruction (i)	Meaning	Effect ($d' = exec(d, i)$)
$addi(reg, reg, imm_{16})$	add imm.	$d'.gpr!r_1 := n2i(i2n(d.gpr!r_2) +_{32} i2n(imm))$
$subi(reg, reg, imm_{16})$	subtract imm.	$d'.gpr!r_1 := n2i(i2n(d.gpr!r_2) -_{32} i2n(imm))$
$andi(reg, reg, imm_{16})$	bitwise and imm.	$d'.gpr!r_1 := [two_{32}(d.gpr!r_2) \wedge_{\mathrm{b}} two_{32}(imm)]$
$ori(reg, reg, imm_{16})$	bitwise or imm.	$d'.gpr!r_1 := [two_{32}(d.gpr!r_2) \vee_{\mathrm{b}} two_{32}(imm)]$
$xori(reg, reg, imm_{16})$	bitwise xor imm.	$d'.gpr!r_1 := [two_{32}(d.gpr!r_2) \otimes_{\mathrm{b}} two_{32}(imm)]$
$add(reg, reg, reg)$	add	$d'.gpr!r_1 := n2i(i2n(d.gpr!r_2) +_{32} i2n(d.gpr!r_3))$
$sub(reg, reg, reg)$	subtract	$d'.gpr!r_1 := n2i(i2n(d.gpr!r_2) -_{32} i2n(d.gpr!r_3))$
$and(reg, reg, reg)$	bitwise and	$d'.gpr!r_1 := [two_{32}(d.gpr!r_2) \wedge_{\mathrm{b}} two_{32}(d.gpr!r_3)]$
$or(reg, reg, reg)$	bitwise or	$d'.gpr!r_1 := [two_{32}(d.gpr!r_2) \vee_{\mathrm{b}} two_{32}(d.gpr!r_3)]$
$xor(reg, reg, reg)$	bitwise xor	$d'.gpr!r_1 := [two_{32}(d.gpr!r_2) \otimes_{\mathrm{b}} two_{32}(d.gpr!r_3)]$

for all instructions: $d'.dpc := d.pcp$, $d'.pcp := d.pcp +_{32} 4$.

Table 6.6: Shift Instructions

Instruction (i)	Meaning	Effect ($d' = exec(d, i)$)
$slli(reg, reg, sa_5)$	shift left logical imm.	$d'.gpr!r_1 := [two_{32}(d.gpr!r_2) \ll_{\mathrm{l}} i2n(sa)]$
$srli(reg, reg, sa_5)$	shift right logical imm.	$d'.gpr!r_1 := [two_{32}(d.gpr!r_2) \gg_{\mathrm{l}} i2n(sa)]$
$srai(reg, reg, sa_5)$	shift right arithmetic imm.	$d'.gpr!r_1 := [two_{32}(d.gpr!r_2) \gg_{\mathrm{a}} i2n(sa)]$
$sll(reg, reg, reg)$	shift left logical	$d'.gpr!r_1 := [two_{32}(d.gpr!r_2) \ll_{\mathrm{l}} (i2n(d.gpr!r_3) \bmod_{\mathrm{u}} 2^5)]$
$srl(reg, reg, reg)$	shift right logical	$d'.gpr!r_1 := [two_{32}(d.gpr!r_2) \gg_{\mathrm{l}} (i2n(d.gpr!r_3) \bmod_{\mathrm{u}} 2^5)]$
$sra(reg, reg, reg)$	shift right arithmetic	$d'.gpr!r_1 := [two_{32}(d.gpr!r_2) \gg_{\mathrm{a}} (i2n(d.gpr!r_3) \bmod_{\mathrm{u}} 2^5)]$

for all instructions: $d'.dpc := d.pcp$, $d'.pcp := d.pcp +_{32} 4$.

PROOF This lemma is proved by case distinction on i. We omit the details here.

ISABELLE Observe that in Isabelle the preceding lemma has an additional requirement (`is_instr`) about instruction i. This requirement claims that the register numbers and immediate constants of i are in the proper range. In this thesis we replace this predicate by using sub types for the immediate constants (cf. Section 6.2.1).

Transition Function

For the definition of the VAMP assembly transition function we need just one more auxiliary function which defines the *current instruction*.

Definition 6.9 (Current instruction) We define the current instruction of a configuration d of the VAMP assembly machine by the following function.

$$\begin{aligned} \mathit{currinstr} &:: \mathit{conf}_{\mathrm{asm}} \mapsto \mathit{instr} \\ \mathit{currinstr}(d) &= \mathit{cell2instr}(d.\mathit{mm}(d.\mathit{dpc} \div 4)) \end{aligned}$$

Definition 6.10 (VAMP transition function) We define the single step transition function $\delta_{\mathrm{asm}} :: \mathit{conf}_{\mathrm{asm}} \mapsto \mathit{conf}_{\mathrm{asm}}$ for the VAMP assembly semantics in the obvious way by applying the *exec* function to the current instruction.

$$\delta_{\mathrm{asm}}(d) = \mathit{exec}(d, \mathit{cell2instr}(((d \div 4)).\mathit{mm}, d.\mathit{dpc})$$

Similar to the $C0$ semantics we use the notation δ^i_{asm} to denote several steps of the VAMP assembly machine. This function is defined by induction on i.

$$\begin{aligned} \delta^0_{\mathrm{asm}}(d) &= d \\ \delta^{i+1}_{\mathrm{asm}}(d) &= \delta^i_{\mathrm{asm}}(\delta_{\mathrm{asm}}(d)) \end{aligned}$$

For later use we introduce some more functions and abbreviations regarding configurations of the VAMP assembly machine.

Definition 6.11 (Reading lists of memory cells) Let m be the memory of a VAMP assembly configuration, a a start address (in words), and l the number of words to be read. We define the function $\mathit{read}_{\mathrm{data}} :: \mathbb{N} \mapsto \mathit{mcell}_{\mathrm{asm}} \times \mathbb{N} \times \mathbb{N} \mapsto \mathit{mcell}_{\mathrm{asm}}\ \mathit{list}$ which reads the content of l consecutive memory cells from memory m starting at word address a. The function is defined by induction on l.

$$\begin{aligned} \mathit{read}_{\mathrm{data}}(m, a, 0) &= [\,] \\ \mathit{read}_{\mathrm{data}}(m, a, l+1) &= m(a) \# \mathit{read}_{\mathrm{data}}(m, a+1, l) \end{aligned}$$

Definition 6.12 (Reading lists of instructions) Let m be the memory of a VAMP assembly configuration, a a byte address, and l the number of instructions to be read. We define the function $\mathit{read}_{\mathrm{instr}} :: \mathbb{N} \mapsto \mathit{mcell}_{\mathrm{asm}} \times \mathbb{N} \times \mathbb{N} \mapsto \mathit{instr}\ \mathit{list}$ which reads l consecutive instructions from m starting at address a.

$$\mathit{read}_{\mathrm{instr}}(m, a, l) = \mathit{map}(\mathit{cell2instr}, \mathit{read}_{\mathrm{data}}(m, a \div 4, l))$$

6.3 Correctness of the VAMP Assembly Model

Obviously, it is a good idea to use an abstraction like the VAMP assembly semantics instead of arguing about compiler correctness directly at the ISA level. However, the Verisoft project aims at the *pervasive* verification of compuer systems which requires us to justify this abstraction formally. Such a justification has been done for user mode programs by A. Tsyban in form of a simulation theorem between the VAMP instruction set architecture and the VAMP assembly semantics.

In order to use this simulation theorem to transfer results from the assembly level to the ISA level we have to meet several additional requirements. Observe that we do not formally define the requirements in this section but instead we introduce a verification methodology for VAMP assembly programs in the following section which incorporates sufficient conditions for the justification of the abstraction. We highlight the most important requirements.

No Self Modifying Code. As mentioned in Section 6.1 on page 101, the VAMP correctness proof requires a special sync instruction between the modification of code and its execution. The *C*0 compiler does not generate self modifying code. Thus, we can fulfill this requirement by just proving that we *never* execute code from memory locations where the code has previously written to.

To distinguish between memory locations from which we will fetch instructions at some point in time and other locations we introduce the concept of a *code range*. The code range $range_c :: \mathbb{N} \times \mathbb{N}$ of a given assembly program is a pair of start address and end address. Roughly speaking, all memory locations inside $range_c$ are considered to contain instructions and all other to contain data.

Alignment of Data and Instructions. The VAMP ISA requires memory accesses to be properly aligned. The VAMP assembly machine does not contain explicit requirements about alignment of accesses; nevertheless, the definitions of memory accesses in the assembly machine are only correct (and meaningful) if proper alignment is given.

No Access Outside the Address Space. The VAMP ISA generates page fault interrupts mainly for two reasons.

1. A memory access is in principle permissible but the affected page is currently swapped out. In this case, the page fault handler of the operating system will swap in the accessed page transparently and the assembly program will not observe any problems. For the compiler correctness proof, we do not have to care about this kind of page fault interrupts.

2. A program tries to access a memory location which it is not allowed to. In this case, there is in general no way for the operating system to fulfill the program's request and it will probably terminate the program. We have to prevent this case by ensuring that the program does not access memory outside its own address space neither by load or store instructions nor by instruction fetches.

Similar to $range_{\text{c}}$ we introduce the notion of an *address range*. With $range_{\text{a}} :: \mathbb{N} \times \mathbb{N}$ we represent the address space of a program. The program is not supposed to do any memory access to an address which is not inside $range_{\text{a}}$.

ISABELLE Observe that in Isabelle the address range is a pair of start address and length in contrast to the code range which is a pair of start and end address. This difference is just for historical reasons and thus we do not reflect it here. In this thesis, all ranges are modeled as pair of start and end address.

No Execution of Non-Modeled Instructions. The VAMP assembly semantics does not model all instructions of the VAMP ISA. The instructions *trap* and *rfe* which (in the VAMP ISA) generate trap interrupts and return from exceptions are just modeled by the identity function and thus must not be executed in a program at the assembly level. For user mode, this also holds for instructions *movs2i* and *movi2s* if they access certain special purpose registers.

6.3.1 Formalizing the Requirements

We define two predicates which formally sum up the requirements from above.

Definition 6.13 (Advancing instructions) We call an instruction *advancing* if its semantics is not the identity mapping. Based on the definition of instruction semantics from Section 6.2.5 we know that only few instructions can lead to an identity step. Thus, we define the predicate $advancing :: instr \mapsto \mathbb{B}$ formally as $advancing(i) = i \notin \{rfe, trap\}$.

Definition 6.14 (Interrupt free configuration) Let d be a configuration of the VAMP assembly semantics, $range_{\text{c}}$ the code range, and $range_{\text{a}}$ the address range. Additionally, let $i = currinstr(d)$ be the current instruction of configuration d. The predicate $interruptfree :: conf_{\text{asm}} \times (\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}) \mapsto \mathbb{B}$ states that the next step of the VAMP assembly machine would not generate interrupts on the ISA layer. This allows to prove for that step the equivalence between the VAMP assembly machine and the ISA layer.

If the current instruction is neither a load or store nor a *movi2s* the rules are quite simple. We just require that the delayed program counter is a multiple of four and inside the code range.

$$\frac{\neg is\_loadstore(i) \qquad i \neq movi2s \qquad 4 \mid d.dpc \qquad d.dpc \in range_{\text{c}}}{interruptfree(d, range_{\text{c}}, range_{\text{a}})}$$

We forbid moves to or from special purpose registers other than floating point registers because this would generate an *illegal* interrupt [Dal06, Chapter 3].

$$\frac{\begin{array}{cc} 4 \mid d.dpc & d.dpc \in range_{\text{c}} \\ i \in \{movi2s, movs2i\} & REG_1(i) \in \{IEEEf, RM, FCC\} \end{array}}{interruptfree(d, range_{\text{c}}, range_{\text{a}})}$$

For load and store instructions we additionally require that the effective addresses of the memory access is inside the address range and that the access is properly aligned.

$$\frac{\begin{array}{c} is\_load(i) \qquad 4 \mid d.dpc \qquad width_{\mathrm{a}}(i) \mid REG_2(i) \qquad fst(range_{\mathrm{a}}) \leq ea(d,i) \\ ea(d,i) + width_{\mathrm{a}}(i) \leq snd(range_{\mathrm{a}}) \qquad d.dpc \in range_{\mathrm{c}} \end{array}}{interruptfree(d, range_{\mathrm{c}}, range_{\mathrm{a}})}$$

Furthermore, for stores the effective address must not be inside the code range.

$$\frac{\begin{array}{c} is\_store(i) \qquad 4 \mid d.dpc \qquad width_{\mathrm{a}}(i) \mid REG_2(i) \\ fst(range_{\mathrm{a}}) \leq ea(d,i) \qquad ea(d,i) + width_{\mathrm{a}}(i) \leq snd(range_{\mathrm{a}}) \\ ea(d,i) + width_{\mathrm{a}}(i) \leq fst(range_{\mathrm{c}}) \vee snd(range_{\mathrm{c}}) < ea(d,i) \qquad d.dpc \in range_{\mathrm{c}} \end{array}}{interruptfree(d, range_{\mathrm{c}}, range_{\mathrm{a}})}$$

6.4 Assembly Code Verification Methodology

The formal verification of VAMP assembly programs directly at the level of the transition function δ_{asm} is non-practical for several reasons. First, repeated unfolding of definitions is time consuming and makes proofs long and hard to maintain. Second, the need to keep track of side conditions ($conf_{\mathrm{asm}}\checkmark$, etc.) and a lot of bookkeeping regarding requirements for absence of interrupts and the simulation proof from the previous section complicates pre and post conditions of lemmas. Additionally, it is hard to carry the preconditions from one instruction to the next. Third and last, the bulk of proof goals is similar for each instruction and need to be done over and over again. Thus, they should be moved into 'general' lemmas which abstract from the concrete situation and can be applied with less effort.

In this section we introduce a methodology which allows us to verify VAMP assembly programs in a straightforward fashion. First, we define a predicate which states that a certain block of assembly instructions executes without generating interrupts. Then we introduce lemmas about the effect of single assembly instructions which are formulated based on this predicate. These lemmas generate only those verification conditions which are really necessary for the given instruction and cannot be concluded in a general way. Finally, we prove lemmas about the successive execution of several assembly instructions or even of several blocks of assembly instructions. This allows us to combine the lemmas about single instructions to lemmas about the behavior of larger pieces of assembly code.

6.4.1 Predicates for Execution of Assembly Code

We need to formalize what we mean exactly when talking about the execution of a certain piece of assembly code starting in a configuration d and ending in another configuration d'.

Definition 6.15 (Execution of assembly code) We use the notation

$$d \xrightarrow[range_{\mathrm{c}}, range_{\mathrm{a}}]{t, dest} d'$$

to state that the VAMP assembly machine started in configuration d reaches in t steps a configuration d' with $d'.dpc = dest$ and $d'.pcp = d'.dpc + 4$ using $range_{\mathrm{c}}$ and $range_{\mathrm{a}}$ as code range and address range, respectively.

The predicate requires that the final configuration is reached by executing t steps of the assembly machine, that its delayed program counter is equal to the destination, that the second program counter *pcp* points to the next instruction, that the code stays unchanged, that no interrupt generating events occur, and that all instructions which are being executed are advancing.

Formally, this is defined by

$$\frac{\begin{array}{c} d'.dpc = dest \qquad d'.pcp = d'.dpc + 4 \qquad \delta^t_{\mathrm{asm}}(d) = d' \qquad conf_{\mathrm{asm}}\surd(d') \\ \forall t' \leq t : unchngd_{\mathrm{mem}}(d.mm, d'.mm, fst(range_{\mathrm{c}}) \div 4, snd(range_{\mathrm{c}}) \div 4) \\ \forall t' < t : interruptfree(\delta^{t'}_{\mathrm{asm}}(d), range_{\mathrm{c}}, range_{\mathrm{a}}) \\ \forall t' < t : advancing(currinstr(\delta^{t'}_{\mathrm{asm}}(d))) \end{array}}{d \xrightarrow[range_{\mathrm{c}}, range_{\mathrm{a}}]{t, dest} d'}$$

To keep formulas in the rest of this thesis concise we combine a set of preconditions which are common for lemmas about the execution of assembly code.

Definition 6.16 (Preconditions for assembly execution) Let $range_{\mathrm{c}}$ be the code range, d a configuration of the VAMP assembly machine, and il a list of assembly instructions. We define the predicate $asm_{\mathrm{pre}} :: conf_{\mathrm{asm}} \times (\mathbb{N} \times \mathbb{N}) \times instr\ list \mapsto \mathbb{B}$ which combine several preconditions for the successful execution of VAMP assembly code.

The predicate requires that d is a valid VAMP assembly configuration, that the delayed program counter is a multiple of four, that the second program counter *pcp* points to the next instruction, that the instruction list il is stored in the memory of d starting at address $d.dpc$, that il is completely located within the code range, and that the code range fits into 32-bit addresses.

Formally, we define

$$\frac{\begin{array}{ccc} conf_{\mathrm{asm}}\surd(d) & 4 \mid d.dpc & \\ d.pcp = d.dpc + 4 & read_{\mathrm{instr}}(d.mm, d.dpc, |il|) = il & fst(range_{\mathrm{c}}) \leq d.dpc \\ d.dpc + 4 * |il| < snd(range_{\mathrm{c}}) & snd(range_{\mathrm{c}}) + 4 < 2^{32} & \end{array}}{asm_{\mathrm{pre}}(d, range_{\mathrm{c}}, il)}$$

Isabelle In Isabelle, the predicate asm_{pre} has an additional conjunction `is_instr` with type $instr \mapsto \mathbb{B}$. It requires that all instructions in il are valid VAMP assembly instructions, i.e., that register names and immediate constants are in the correct domain. However, in this thesis the property follows directly from the fact that we use bounded types for register names and immediate constants (cf. remark on page 102). Thus, there is no need to explicitly state it.

6.4.2 Executing Single Instructions

We have proved dozens of lemmas in Isabelle which state, given a small set of preconditions, the meaning of certain assembly instructions. These lemmas have been very useful in the low-level correctness proofs for the assembly code which is being generated by the $C0$ compiler. But in this thesis we will not give detailed

correctness proofs about the execution of small building blocks of assembly code; instead we will concentrate on the interesting parts of the compiler correctness proof, where $C0$ layer and assembly layer are linked together. Thus, we will show only a few lemmas of the aforementioned kind to give the reader an idea of the kind of low-level proofs which have been done.

We start with a simple lemma about the execution of the *addi* instruction.

Lemma 6.7 (Correctness of addition) *If the preconditions from asm_{pre} hold for a configuration d where addi is the current instruction then the next step of the VAMP assembly transition function has the expected behavior.*

$$asm_{pre}(d, range_c, [addi(r_d, r_s, i)])$$
$$\Longrightarrow d \xrightarrow[range_c, range_a]{1, d.dpc+4} d \left[\begin{array}{l} dpc := d.dpc + 4 \\ pcp := d.pcp + 4 \\ gpr!r_d := n2i(i2n(d.gpr!r_s) +_{32} i2n(i)) \end{array} \right]$$

Proof We do not give a detailed proof of this lemma here. In Isabelle the proof is done mainly by unfolding the corresponding definitions of the VAMP assembly semantics and by some simple arguments about modulo arithmetic.

The previous lemma is not very impressive and its formal proof is quite short. Thus, the question may arise what this kind of lemmas is good for. But, even if the lemma and its proofs are quite short its repeated use while proving larger pieces of assembly code saves quite some time. However, the main advantage is that most pre and post conditions of these lemmas are encapsulated in special predicates; this gives a *standard* way of encoding this conditions. Thereby, this kind of lemmas can be easily combined to lemmas about larger pieces of assembly code as demonstrated in the following section.

Usually, a programmer does not make use of the fact the the program counter is increased using modulo arithmetic. Thus, these lemmas abstract the program counter computation to ordinary addition as long as the preconditions from asm_{pre} are fulfilled.

The second example is a lemma about the execution of a branch-equal-zero instruction (*beqz*) in case that the condition is fulfilled. In contrast to the first lemma, we cannot use the predicate for the execution of assembly code from Definition 6.15 on page 114 directly for lemmas about branch and jump instructions. The reason is that in a configuration d' after executing a jump or branch instruction (given that the branch has been taken) it is *not* true that $d'.pcp = d'.dpc + 4$. Nevertheless, this condition is very helpful as it allows to look at the execution of a given code block in isolation without knowing too much details about the past of the assembly machine. Thus, we formulate lemmas about the execution of branch and jump instructions always as lemmas about two instructions: the branch and the instruction in the delay slot. After executing the delay slot instruction (which must not be a control flow instruction, cf. [BJK+06]) the condition $d'.pcp = d'.dpc + 4$ holds again. In the code generated by the $C0$ compiler there are not many different instructions in the delay slots of branches. This keeps the number of different combinations of branch and delay slot instructions – and the number of lemmas – small.

We present here exemplarily the combination of a *beqz* instruction with an *addi* in the delay slot. Observe that the jump distance can be negative.

Lemma 6.8 (Correctness of a conditional branch) *Let d be a configuration where the preconditions from asm_{pre} hold and beqz is the current instruction followed by addi. If the source register is zero and the destination address of the branch is reasonable then the branch is taken and the next step of the VAMP assembly transition function has the expected behavior.*

$$
\begin{aligned}
& asm_{pre}(d, range_c, [beqz(r_s, dist), addi(r_d, r_s, i)]) \\
& \wedge\, d.gpr!r_s = 0 \\
& \wedge\, 0 \leq d.dpc + 4 + dist + 4 < 2^{32} \wedge -2^{31} \leq dist < 2^{31} \\
& \Longrightarrow d \xrightarrow[range_c, range_a]{2, d.pcp + dist} d \left[\begin{array}{l} dpc := d.pcp + dist \\ pcp := d.pcp + dist + 4 \\ gpr!r_d := n2i(i2n(d.gpr!r_s) +_{32} i2n(i)) \end{array} \right]
\end{aligned}
$$

Proof We omit the proof.

For this kind of lemmas there are basically two ways to specify the new configuration d'. We can either define the new configuration explicitly like in the lemmas above or we can specify its interesting properties implicitly. For the compiler correctness proofs we have preferred the latter way in lemmas which argue about more than one or two instructions (cf. Section 6.4.3). The main advantage of this kind of formalization is that we can work on a slightly higher level of abstraction. Additionally, Isabelle's simplifier is not overstressed by unsuccessfully trying to exploit the explicit representation of the new configuration and the user is not overwhelmed by the exploding size of formulas.

6.4.3 Combining Code Pieces

In this thesis we developed a verification methodology for VAMP assembly code which allows to combine lemmas about the execution of single assembly instructions (cf. the previous section) or about small code pieces to correctness proofs about larger code pieces. This verification is done in a *forward* style.

1. Have asm_{pre} for the whole list il of assembly instructions in a configuration d .
2. From step 1 deduce asm_{pre} for a certain prefix a of $il = a \circ b$ for which we already have a lemma.
3. Apply the lemma for a. This gives a new configuration d' with certain properties.
4. From the execution of a, infer asm_{pre} for b in configuration d'.
5. Now, repeat steps 2 to 4 starting in configuration d' for a prefix a' of the remaining assembly instructions. This leads to a new configuration d'' after execution of this prefix.
6. Combine the execution of a and a' to the execution of $a \circ a'$ from configurations d to d'' and start again with step 5 until the original assembly program il is completely processed.

In formal proofs, small differences in the way of proving lemmas can make a big difference in the required effort. A big advantage of the described verification methodology is that it allows to keep the formal proof context in Isabelle concise. The same holds for the proof scripts themselves. With the right kind of lemmas (cf. below) and a good way of applying them we can process the assembly instructions step-by-step without poisoning the proof state with obsolete facts and without having to state complicated facts by copy-and-paste.

Now, we present some lemmas which implement the verification methodology from above. We start with a lemma which allows to combine results about the execution of two separate code blocks a and b to the execution of $a@b$ (cf. step 6 of the methodology).

Lemma 6.9 (Combined execution of code blocks) *Given that a list a of assembly instructions is executed from configuration d to configuration d' in t_a steps and a code piece b is executed from d' to d'' in t_b steps we can conclude that the composed list $a \circ b$ is executed from d to d'' in $t_a + t_b$ steps.*

$$d \xrightarrow[range_c, range_a]{t_a, dest_a} d' \wedge d' \xrightarrow[range_c, range_a]{t_b, dest_b} d'' \Longrightarrow d \xrightarrow[range_c, range_a]{t_a + t_b, dest_b} d''$$

Proof The proof of this lemma is quite simple; even in Isabelle it does only require 15 proof steps. We have to verify seven conditions for this lemma (cf. Definition 6.15 on page 114):

1. $d''.dpc = dest_{\mathrm{b}}$
2. $d''.pcp = d''.dpc + 4$
3. $\delta_{\mathrm{asm}}^{t_{\mathrm{a}}+t_{\mathrm{b}}}(d) = d''$
4. $conf_{\mathrm{asm}}\surd(d'')$
5. $\forall t' \leq t_{\mathrm{a}}+t_{\mathrm{b}} : unchngd_{\mathrm{mem}}(d.mm, d''.mm, fst(range_{\mathrm{c}}) \div 4, snd(range_{\mathrm{c}}) \div 4)$
6. $\forall t' < t_{\mathrm{a}} + t_{\mathrm{b}} : interruptfree(\delta_{\mathrm{asm}}^{t'}(d), range_{\mathrm{c}}, range_{\mathrm{a}})$
7. $\forall t' < t_{\mathrm{a}} + t_{\mathrm{b}} : advancing(currinstr(\delta_{\mathrm{asm}}^{t'}(d)))$

Conditions 1, 2, and 4 follow directly from the correct execution of b. Condition 3 follows by a simple argument about the VAMP assembly transition function. Finally, conditions 5, 6, and 7 follow for $t' < t_{\mathrm{a}}$ ($t' \leq t_{\mathrm{a}}$ for condition 5) from the proper execution of a and for the other cases from the execution of b. q.e.d.

We continue with statements about concluding asm_{pre} for prefixes of a list of assembly instructions (cf. step 2 of the methodology)

Lemma 6.10 *Given that we have proper preconditions for the execution of a list il of assembly instructions in configuration d we also have the preconditions for any prefix of il.*

$$asm_{pre}(d, range_c, il) \wedge n \leq |il| \Longrightarrow asm_{pre}(d, range_c, il[0, n])$$

Proof The proof of this lemma follows directly from the definition of asm_{pre}.

To further simplify formal reasoning in Isabelle we also use the following corollaries of the previous lemma. These corollaries are trivial with paper-and-pencil and their proof is trivial also in Isabelle / HOL. However, using them instead of the original lemma often saves several additional proof steps which would be required to bring the current proof state in correspondence with the lemma on the facing page.

Corollary 6.11 *Given that we have proper preconditions for the execution of a list of assembly instructions in configuration d we also have the preconditions for the execution of the first instruction in that list.*

$$asm_{pre}(d, range_c, a\#b) \Longrightarrow asm_{pre}(d, range_c, [a])$$

Corollary 6.12 *Given that we have proper preconditions for the execution of a list of assembly instructions which is composed of two parts the precondition holds in the same configuration also for the first part of the list.*

$$asm_{pre}(d, range_c, a \circ b) \Longrightarrow asm_{pre}(d, range_c, a)$$

Finally, we present a lemma which formalizes step 4 of the verification methodology on page 117.

Lemma 6.13 *Let the preconditions for the execution of assembly code hold for a list $a \circ b$ of assembly instructions in configuration d. Furthermore, assume that the instructions a are executed from configuration d to configuration d' such that the program counter of d' points to the start of instruction list b. Then, we can conclude that the preconditions also hold for the execution of b in configuration d'.*

$$asm_{pre}(d, range_c, a \circ b) \wedge d \xrightarrow[range_c, range_a]{t, d.dpc+4\cdot|a|} d' \Longrightarrow asm_{pre}(d', range_c, b)$$

Proof We omit the simple proof of this lemma.

We also present the following corollary of the previous lemma which simplifies formal reasoning.

Corollary 6.14 *If the preconditions for the execution of assembly code hold in configuration d for a list of assembly instructions, the first statement of that list is executed from configuration d to d', and the program counter of d' points to the second instructions of the list we can conclude that the preconditions holds for the tail of the instruction list in d'.*

$$asm_{pre}(d, range_c, a\#b) \wedge d \xrightarrow[range_c, range_a]{t, d.dpc+4} d' \Longrightarrow asm_{pre}(d', range_c, b)$$

6.4.4 Automation

The assembly level correctness proofs which have been done in this thesis using the assembly verification methodology from Section 6.4, can probably be automated to a large extent. In [Eme05], the author presents a method for an automatic proof that no interrupts are generated by a given piece of assembly code. When more and larger assembly programs have to be verified, it could pay off to develop a semi-automated assembly verification environment as has been done for $C0$ programs in [Sch06].

Part II

Compiler

Chapter 7

Code Generation Algorithm

Contents

In this chapter we describe the code generation algorithm of the $C0$ compiler.

A $C0$ program consists of a type name environment, a function table, and a symbol table for the global variables (cf. Section 4.1). The code generation algorithm of the $C0$ compiler is quite simple and follows directly the structure of the input program. The $C0$ compiler starts by iterating over all functions in the function table and generates code for the function bodies (which are formed by a single statement). The code generation for statements and expressions – in the context of a certain function – is done by a simple recursive algorithm which follows the structure of the corresponding data types.

The code generation algorithm does not use any optimizations. Registers are not used to store (intermediate) results or memory content between different expressions or statements. Thus, there is no need for a special register allocation phase. Register allocation is done together with instruction selection in one step.

In this chapter we introduce the code generation algorithm bottom up. We start in Section 7.1 with some basic definitions and introduce the memory layout of the compiled programs in Section 7.2. We continue with the code generation for expressions in Section 7.3, with the code generation for statements in Section 7.4, and conclude in Section 7.5 with the code generation algorithm for complete $C0$ programs. In Section 7.6 we investigate restrictions on input programs which are caused by restrictions of the target machine. We conclude this chapter in Section 7.7 with a discussion on how the Isabelle specification of the code generation algorithm can be executed. This is a possible solution of the bootstrap problem (cf. [GH98]).

7.1 Introduction to the Code Generation

7.1.1 Code Size

On several occasions we need to know the size of the generated code. To name just a few: we need to compute jump distances (e.g., to miss out the body of a loop if the condition is *false*), to compute the absolute start address of functions (for function calls), and to compute the base address of variables, which depends – at least for the $C0$ compiler – on the size of the program (cf. Figure 7.2 on page 129).

For the first and for the last example we do not necessarily need to know the code size in advance, i.e., before generating the corresponding code.

As we will see later, code is generated by recursion, thus we know the code for the loop body already when we generate the code which reacts to the condition. Thus, we can figure out the size of the loop body by just looking at its code.

For the last example, we anticipate the meaning of special registers (cf. Table 7.2 on page 129) which store the base address of the global memory frame and the base address of the current stack frame. We compute the address of variables relative to one of these registers and can keep the code which accesses variables independent of the real values for the base address. Obviously, these two registers need to be initialized.

This is done by some initialization code which is located at the beginning of the compiled program (cf. Figure 7.12 on page 160). Thus, the initialization code need to know the code size of the program, in order to compute the initial register values. However, the initialization code – which is of fixed size – could be generated at the end, after generating the code for the remaining program, when we already know the size of the remaining code.

Nevertheless, the solutions from above do not work for function calls. $C0$ supports mutual recursion, thus there is – at least for some $C0$ programs – no way to generate code without having to jump to functions which have not yet been compiled. To compute the jump distance to or the absolute address of these functions we would need to know the size of the current function, which we do not.

There are two solutions to this problem.

- We could compile a program in two passes: in a first pass we would miss out jump distances from function calls – taking notes of where we need to insert the jump distance to which function – and in a second pass, when we know the size of all functions, we would insert the jump distances.

- We can introduce simple functions which compute, given a necessary set of parameters, the size of the generated code, without having to know all details like jump distances.

In terms of correctness proofs, we would have to prove that the selected solution generates correct jump distances for function calls. For the two-pass compilation this would require us to keep a list of positions where we have to fill in jump distances and to prove that inserting the distances works correctly. Such a proof is be quite complicated to do in a formal verification system. The second approach is much simpler as it just requires to prove for all code

generation functions that the corresponding *code size functions* compute the correct result. Thus, we have chosen the second approach to specify the code generation algorithm in Isabelle. For the compiler implementation in $C0$ [Pet07], Elena Petrova has chosen to implement the two-pass approach. This was mainly done for efficiency and required a quite lengthy equivalence proof between both solutions.

Even if not mandatory, the code size functions also simplify code generation for the first and third problem from above because they decouple computation of addresses and actual code generation.

We will not give a detailed (and boring) definition of the code size functions, here. Instead we present the signatures of the most important code size function and leave their concrete definitions and some auxiliary functions to the Isabelle theories. One can easily calculate the code size by hand from the code generation algorithm in Sections 7.3 and 7.4. Observe that we count the size of code in words, which corresponds directly to the number of VAMP instructions. To have the code size in bytes we have to multiply by four.

Definition 7.1 (Code size of expressions) Let e be an expression, te a type name environment, gst and lst global and local symbol tables, and rf a flag which specifies if e should be evaluated as a left or as a right expression. Then, the function $csize_{\mathrm{e}}(te, gst, lst, rf, e)$ computes the size of the code generated for e.

$$csize_{\mathrm{e}} :: tenv \times (\mathbb{S} \times ty)\ list \times (\mathbb{S} \times ty)\ list \times \mathbb{B} \times expr \mapsto \mathbb{N}$$

Definition 7.2 (Code size of statements) Let s be a statement, te a type name environment, and let gst and lst be global and local symbol tables, respectively. Then, the function $csize_{\mathrm{s}}(te, gst, lst, s)$ computes the size of the code generated for statement s.

$$csize_{\mathrm{s}} :: tenv \times (\mathbb{S} \times ty)\ list \times (\mathbb{S} \times ty)\ list \times stmt \mapsto \mathbb{N}$$

As an abbreviation we introduce a second variant of the above function which computes the neccessary local symbol table depending on the function to which the statement belongs. The result of this function is of option type to cope with statements which are not present in the program.

$$csize_{\mathrm{s}} :: tenv \times (\mathbb{S} \times ty)\ list \times functableT\, stmt \mapsto \mathbb{N}_{\perp}$$

$$csize_{\mathrm{s}}(te, gst, ft, s) = \begin{cases} \lfloor csize_{\mathrm{s}}(te, gst, st_{\mathrm{fun}}(f), s) \rfloor & \text{if } fos(s, ft) = \lfloor f \rfloor \\ None & \text{otherwise} \end{cases}$$

Definition 7.3 (Code size of function lists) Let te a type name environment, gst the global symbol table, and ft a function table. Then we introduce the function $csize_{\mathrm{fl}} :: tenv \times (\mathbb{S} \times ty)\ list \times functableT \mapsto \mathbb{N}$ which computes the code size of a given list of functions. We define by induction on the list of functions

$$\begin{aligned} csize_{\mathrm{fl}}(te, gst, ft, [\,]) &= 0 \\ csize_{\mathrm{fl}}(te, gst, ft, (fn, f)\#fs) &= csize_{\mathrm{fl}}(te, gst, ft, fs) \\ &\quad + csize_{\mathrm{s}}(te, gst, st_{\mathrm{fun}}(f), f.body) \end{aligned}$$

Definition 7.4 (Size of the init code) We introduce the following constant which denotes the size of the init code.

$$csize_{\text{ini}} :: \mathbb{N}$$

The code size of a $C0$ program is composed of the code size of the init code and of all functions of the program.

Definition 7.5 (Code size of programs) Let te a type name environment, gst the global symbol table, and ft a function table. We define the code size of a $C0$ program by

$$csize_{\text{prog}}(te, gst, ft) = csize_{\text{ini}} + csize_{\text{fl}}(te, gst, ft, ft)$$

7.1.2 Base Addresses of Code

Now, we define several functions which compute the (byte) base addresses of code. We start with a function which computes the start address of the code which has been generated for a given function. Then, we define for a statement s and a sub statement s' of s the *code displacement* of s' in s. The code displacement tells us the displacement of the code generated for s' with respect to the start of the code which has been generated for s. Finally, we will define the code base of statements using the start address of the statement's function and the code displacement of the statement with respect to the start of the code for that function.

Definition 7.6 (Displacement of functions) We define the displacement of a function with name fn with respect to the start of the code generated for a given list of functions. We do this definition by induction on the function list. As we are not interested in the case that fn is not present in the function list we can omit the induction start.

$$displ_{\text{f}} :: tenv \times (\mathbb{S} \times ty)\ list \times \mathbb{S} \times functableT \times functableT \mapsto \mathbb{N}$$

$$displ_{\text{f}}(te, gst, fn, ft, (fn', f)\#fl) = \begin{cases} 0 & \text{if } fn' = fn \\ 4 \cdot csize_{\text{s}}(te, gst, st_{\text{fun}}(f), f.body) & \\ \quad + displ_{\text{f}}(te, gst, fn, ft, fl) & \text{otherwise} \end{cases}$$

Based on the displacement of a function it is now simple to define its absolute start address.

Definition 7.7 (Start address of functions) We define the start address of a function f by adding its displacement to the program base and the size of the initialization code. Formally, we define

$$cbase_{\text{f}} :: tenv \times (\mathbb{S} \times ty)\ list \times \mathbb{S} \times functableT \mapsto \mathbb{N}$$
$$cbase_{\text{f}}(te, gst, f, ft) = progbase + 4 \cdot csize_{\text{ini}} + displ_{\text{f}}(te, gst, f, ft, ft)$$

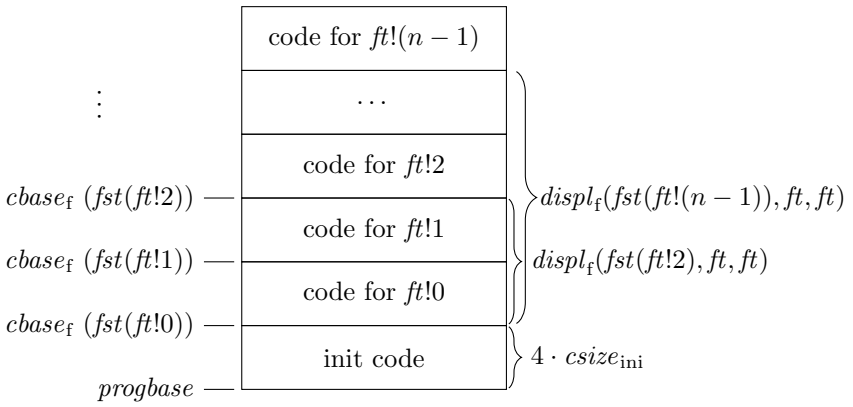


Figure 7.1: Start Address of Functions for Function Table ft

Let te be a type name environment, ft a function table, and gst and lst the global and local symbol tables. Furthermore, let s be a statement and s' a non-structural sub statement of s. We introduce the code displacement (in bytes) of statement s' with respect to statement s.

$$displ_{\mathrm{s}} :: tenv \times functableT \times (\mathbb{S} \times ty)\ list \times (\mathbb{S} \times ty)\ list \times stmt \times stmt \mapsto \mathbb{N}_{\perp}$$

For easier understanding of the formulas, we will not give the concrete definition of $displ_{\mathrm{s}}$ until Section 7.4.9 after we have defined the code generation for statements.

Based on these definitions we can finally define the base address of arbitrary statements.

Definition 7.8 (Base address of statements) Let te a type name environment, ft a function table, gst the symbol table for the global memory, and s a statement. We define the function $cbase_{\mathrm{s}} :: tenv \times (\mathbb{S} \times ty)\ list \times functableT \times stmt \mapsto \mathbb{N}_{\perp}$ which computes the base address of s. To do that it adds the base address of the function f to which s belongs and the displacement of s inside the function body of f.

$$cbase_{\mathrm{s}}(te, gst, ft, s) = \begin{cases} \lfloor\ cbase_{\mathrm{f}}(te, gst, fn, ft) + \\ \quad the(displ_{\mathrm{s}}(te, ft, gst, st_{\mathrm{fun}}(f), s, f.body))\ \rfloor & \text{if } fos(s, ft) = \lfloor (fn, f) \rfloor \\ None & \text{otherwise} \end{cases}$$

We prove the following simple lemma about the base address of sub statements.

Lemma 7.1 (Base address of sub statements) *Let statements s and f be non-structural. Furthermore, let s be a sub statement of f and f a sub statement of f'. Then, the code displacement of s with respect to f' equals the displacement*

of s with respect to f plus the displacement of f with respect to a.

$$
\begin{aligned}
& s \in sub_s(f) \wedge \neg stmt_{structural}(s) \\
& \wedge f \in sub_s(f') \wedge \neg stmt_{structural}(f) \\
& \wedge dstnct_s(f') \\
& \Longrightarrow the(displ_s(s, f')) = the(displ_s(s, f)) + the(displ_s(f, f'))
\end{aligned}
$$

Proof We prove this lemma by induction on f'. Most induction cases follow directly from the definition of $displ_s$.

The condition that all sub statements of f' are distinct is not strictly necessary for the lemma. However, it simplifies the proof for the inductive cases, i.e., for the statement with several sub statements: $Comp(s_1, s_2)$ and $Ifte(e, s_1, s_2)$. In these cases, we have to argue that s and f are in the same sub tree, e.g., that the case $s \in sub_s(s_1) \wedge f \notin sub_s(s_1) \wedge f \in sub_s(s_2)$ cannot occur. This is trivial, if we have $dstnct_s(f')$ because $f \in sub_s(s_2)$ implies $s \in sub_s(s_2)$ which is a contradiction to the distinctness requirement.

Without distinctness, the goal would also be provable but would require more auxiliary lemmas which argue that f is in some sense *smaller* than f' (because $f \in sub_s(s_2)$) and so it has to be a sub statement of s_1 because s is a sub statement of s_1. Then, we would not care whether s or f are sub statements of s_2 because in the definition of $displ_s$ the left sub statement has higher priority.

We omit other details of the proof. q.e.d.

7.2 Memory Layout

The data memory layout of the $C0$ compiler consists of three main areas: the global memory frame, a stack of local memory frames, and the heap. A rough overview on the layout of these areas in the memory of a VAMP assembly machine is depicted in Figure 7.2.

The memory layout starts with the assembly code for the compiled program. The code starts at address *progbase* which is a parameter to the $C0$ compiler and thus treated as a constant here. Behind an unused area of size $bubble_{\text{code}}$ – this is also a parameter to the compiler and can be zero – follows the global memory frame. The global memory is followed by an unused area of size $bubble_{\text{gm}}$.

The global memory frame is followed by the stack of local memories. Each local memory frame starts with a *stack frame header* which occupies three words and stores the *return address*, the *return destination*, and the *previous stack pointer*. The offsets and meaning of these fields are explained in Table 7.1. New stack frames are placed at the upper end of the stack, i.e., the $C0$ compiler lets the stack grow to the top.

The last part of the memory is the heap. The heap starts at address $abase_{\text{heap}}$, which is also a constant parameter to the compiler, and grows to the top. The size of the heap memory is bounded by the constant $asize_{\text{heap}}^{\max}$.

The compiler stores the base address of the global memory, the first unused address on the heap, and the base address of the current stack frame in registers (cf. Table 7.2).

Table 7.1: Frame Header Layout

Offset	Name	Meaning
0	return address	where to jump after completion of the corresponding function
4	return destination	where to store the result of the function
8	previous stack pointer	start address of the previous stack frame

Table 7.2: Special Registers Used in the $C0$ Compiler

Register	Mnemonic	Meaning
$r28$	r_{sbase}	base address of the global memory frame
$r29$	r_{htop}	first unused address on the heap
$r30$	r_{lframe}	base address of the current (top) stack frame
$r31$	r_{jal}	used by the VAMP for jump-and-link instructions

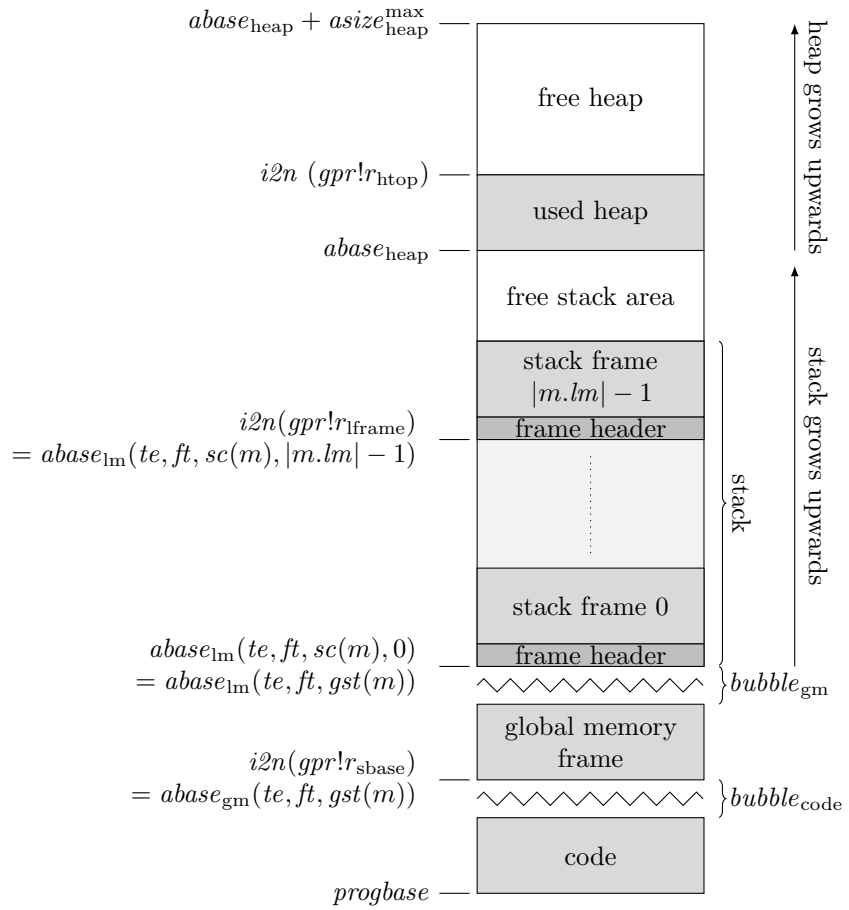


Figure 7.2: Memory Layout of the $C0$ Compiler

7.2.1 Alignment and Allocated Size

We define inductively the alignment of types and the number of bytes needed to store a value of certain type in the target machine. We call the latter the *allocated size* of a type.

Remark The following definitions of alignment and allocated size of types and the displacement of structure components are unnecessary complex for the current version of the $C0$ compiler which reserves a complete word for all basic types. For such a compiler, all structure components are automatically aligned to four bytes and there is no need to insert holes in the memory layout for proper alignment.

However, having these complex definitions allows us to switch without too much effort to a memory layout where some basic types (e.g., boolean and character) occupy only a single byte. The generated code – with the exception of the code for big assignments – works also for this case. The proofs, however, would requirement some adjustments.

Definition 7.9 (Alignment of types) We define the function $algn :: ty \mapsto \mathbb{N}$ which computes the neccessary alignment for a given type.

$$\begin{aligned} algn(Bool_{\mathrm{T}}) &= 4 \\ algn(Int_{\mathrm{T}}) &= 4 \\ algn(Char_{\mathrm{T}}) &= 4 \\ algn(Unsigned_{\mathrm{T}}) &= 4 \\ algn(Ptr_{\mathrm{T}}(tn)) &= 4 \\ algn(Null_{\mathrm{T}}) &= 4 \end{aligned}$$

The alignment of arrays is the alignment of the element type.

$$algn(Arr_{\mathrm{T}}(n, t)) = algn(t)$$

The alignment of a structure is the maximum of the alignments of its components.

$$algn(Str_{\mathrm{T}}(scl)) = \max(map(algn, scl))$$

Definition 7.10 (Allocated size of types and displacement) We define simultaneously the function $asize_{\mathrm{t}} :: ty \mapsto \mathbb{N}$ which computes the allocated size for a given type and the function $displ_{\mathrm{v}} :: \mathbb{N} \times (\mathbb{S} \times ty)\ list \times \mathbb{S} \mapsto \mathbb{N}$ which computes the displacement of structure components inside a structure.

$$\begin{aligned} asize_{\mathrm{t}}(Bool_{\mathrm{T}}) &= 4 \\ asize_{\mathrm{t}}(Int_{\mathrm{T}}) &= 4 \\ asize_{\mathrm{t}}(Char_{\mathrm{T}}) &= 4 \\ asize_{\mathrm{t}}(Unsigned_{\mathrm{T}}) &= 4 \\ asize_{\mathrm{t}}(Ptr_{\mathrm{T}}(tn)) &= 4 \\ asize_{\mathrm{t}}(Null_{\mathrm{T}}) &= 4 \end{aligned}$$

All array elements have to be properly aligned. However, the allocated size of array elements may not correspond to their alignment: e.g., consider the

case of an array of structures which contain a pointer followed by a boolean and imagine that booleans would have an allocated size of just one byte. In this case, the allocated size of the array elements would be five, whereas the required alignment would be four; accordingly, the compiler would have to insert three unused bytes behind every array element (this example is illustrated in Figure 7.3). Thus, we have to round the element size up to a multiple of its alignment.

$$asize_{\mathrm{t}}(Arr_{\mathrm{T}}(n,t)) = n \cdot \lceil asize_{\mathrm{t}}(t) \rceil_{algn(t)}$$

The allocated size of a structure equals the displacement of the last component plus its allocated size.

$$asize_{\mathrm{t}}(Str_{\mathrm{T}}(scl)) = displ_{\mathrm{v}}(0, scl, fst(last(scl))) + asize_{\mathrm{t}}(snd(last(scl)))$$

The displacement of a component x inside a structure is defined using an accumulator parameter i. The intended meaning of i is that it represents the displacement of the previous component plus the allocated size of that component. Then, as soon as we reach component x, we just have to round i up to a multiple of the alignment of the type of component x.

Observe that we apply $asize_{\mathrm{t}}$ only to the type of single structure components not to the type of the whole structure. This allows us to use $displ_{\mathrm{v}}$ to define the allocated size of structures while still having well defined functions.

$$displ_{\mathrm{v}}(i, [\,], x) = i$$
$$displ_{\mathrm{v}}(i, (cn,t)\#xs, x) = \begin{cases} \lceil i \rceil_{algn(t)} & \text{if } x = cn \\ displ_{\mathrm{v}}(\lceil i \rceil_{algn(t)} + asize_{\mathrm{t}}(t), xs, x) & \text{otherwise} \end{cases}$$

Isabelle To avoid the need for a simultaneous definition of $asize_{\mathrm{t}}$ and $displ_{\mathrm{v}}$, the definition of $asize_{\mathrm{t}}$ for structures has been done differently in Isabelle. It uses an auxiliary accumulator parameter and is far from being as intuitive as the definition above. However, we have proved a lemma[1] which shows the equivalence between the Isabelle definition and the intuitive definition above.

Lemma 7.2 (Allocated size is a multiple of four) *Let te be a type name environment and t a valid C0 type. Then, the allocated size of t is a multiple of four.*

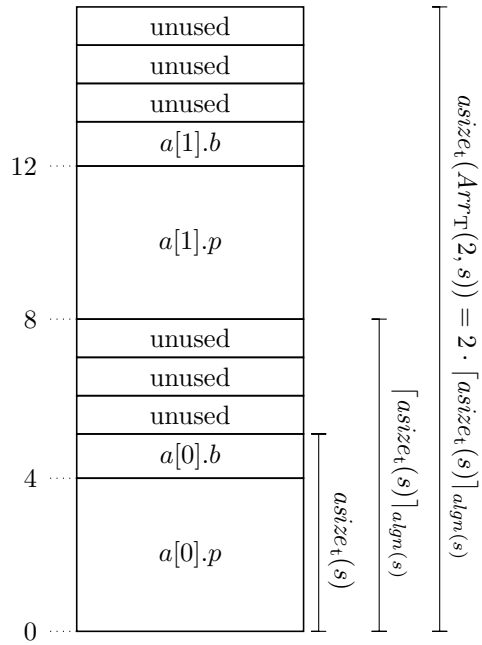
$$valid_{ty}(te,t) \Longrightarrow 4 \mid asize_t(t)$$

Proof This lemma is proved by induction on the structure of t. For the basic types, the statement follows directly from the definition of $asize_{\mathrm{t}}$. The induction steps for arrays and structures are done using some basic properties of the ceiling function $\lceil x \rceil_y$. q.e.d.

As an abbreviation we introduce the function $asize_{\mathrm{algn}}$ which aligns the allocated size of a type by the alignment of the type. We set

$$asize_{\mathrm{algn}} :: ty \mapsto \mathbb{N}$$
$$asize_{\mathrm{algn}}(t) = \lceil asize_{\mathrm{t}}(t) \rceil_{algn(t)}.$$

[1] `asize_struct_components_as_displ_var_of_last`



In this example consider an array a with two elements where the array's element type is a structure type s with components $p : Ptr_{\mathrm{T}}$ and $b : Bool_{\mathrm{T}}$.

Figure 7.3: Example for the Allocated Size of Arrays

We extend the definition of allocated size from types to symbol tables and memories.

Definition 7.11 (Allocated size of symbol tables) Compared to a global memory frame, the stack frames of the local memory store additional information in the frame header. Thus, the function $asize_{\mathrm{st}} :: (\mathbb{S} \times ty)\ list \times \mathbb{N} \mapsto \mathbb{N}$, which computes the allocated size of a given symbol table st has a second parameter which represents the size of the frame header. For the global memory frame, this parameter is zero, for the stack frame it is twelve. We define

$$asize_{\mathrm{st}}(st, hs) = \begin{cases} hs & \text{if } st = [\,] \\ displ_{\mathrm{v}}(hs, st, fst(last(st))) + asize_{\mathrm{t}}(snd(last(st))) & \text{otherwise} \end{cases}$$

which means that the size of an empty symbol table equals the header size and for a non-empty symbol table equals the displacement of the last variable plus the allocated size of the variable.

Based on this definition we define the allocated size of a memory by

$$asize_{\mathrm{mem}} :: mframe \times \mathbb{N} \mapsto \mathbb{N}$$
$$asize_{\mathrm{mem}}(m, hs) = asize_{\mathrm{st}}(m.st, hs)$$

Variables in the heap have no names so we have to define the allocated size of the heap differently.

Definition 7.12 (Allocated size of the heap) We define the allocated size of the heap by induction. Observe that all heap objects are aligned to word boundaries.

$$asize_{\text{heap}} :: (\mathbb{S} \times ty)\ list \mapsto \mathbb{N}$$

$$\begin{aligned} asize_{\text{heap}}([\,]) &= 0 \\ asize_{\text{heap}}((x,t)\#xs) &= \lceil asize_{\text{t}}(t) \rceil_4 + asize_{\text{heap}}(xs) \end{aligned}$$

7.2.2 Displacement of Variables and G-Variables

The displacement of a variable in a memory frame is equal to the displacement of a structure component inside a structure. Thus, we reuse the function $displ_{\text{v}}$ from Definition 7.10 on page 130 to compute displacement of variables. Observe that for variables in stack frames we have to set the headersize to twelve.

Definition 7.13 (Displacement of g-variables) We introduce a function $displ_{\text{g}} :: symbolconf \times gvar \mapsto \mathbb{N}$ which computes the displacement of a g-variable g with respect to its root g-variable. The function is defined by induction on g.

$$\begin{aligned} displ_{\text{g}}(sc, gvar_{\text{gm}}(x)) &= 0 \\ displ_{\text{g}}(sc, gvar_{\text{lm}}(i,x)) &= 0 \\ displ_{\text{g}}(sc, gvar_{\text{hm}}(i)) &= 0 \\ displ_{\text{g}}(sc, gvar_{\text{arr}}(g,i)) &= displ_{\text{g}}(sc,g) \\ &\quad + i \cdot asize_{\text{algn}}(ty_{\text{g}}(sc, gvar_{\text{arr}}(g,i))) \end{aligned}$$

For structure access let $ty_{\text{g}}(sc, gvar_{\text{arr}}(g,i)) = Str_{\text{T}}(scl)$. Then we set

$$\begin{aligned} displ_{\text{g}}(sc, gvar_{\text{str}}(g,cn)) &= displ_{\text{g}}(sc,g) \\ &\quad + displ_{\text{v}}(0, scl, cn) \end{aligned}$$

Definition 7.14 (Relative displacement) We define the relative displacement $displ_{\text{rel}}$ of a g-variable g with respect to another g-variable h by

$$\begin{aligned} displ_{\text{rel}} &:: symbolconf \times \times gvar \times gvar \mapsto \mathbb{N} \\ displ_{\text{rel}}(sc,g,h) &= displ_{\text{g}}(sc,h) - displ_{\text{g}}(sc,g) \end{aligned}$$

7.2.3 Base Addresses

After defining the size of the generated code in the previous section, we can define the absolute base addresses of variables and g-variables. We start by defining the absolute base address of the global memory and of the local memory stack.

Definition 7.15 (Base address of the global memory) Let te be a type name environment, gst the global symbol table, and ft a function table. To compute the base address of the global memory we have to add the size of the program (in bytes) and the size of $bubble_{\text{code}}$ to the base address of the code (cf. Figure 7.2). Formally, this is defined as

$$\begin{aligned} abase_{\text{gm}} &:: tenv \times functableT \times (\mathbb{S} \times ty)\ list \mapsto \mathbb{N} \\ abase_{\text{gm}}(te, ft, gst) &= progbase + 4 \cdot csize_{\text{prog}}(te, gst, ft) + 4 \cdot bubble_{\text{code}} \end{aligned}$$

Definition 7.16 (Base address of stack frames) Let te a type name environment, ft a function table, and sc the current symbol configuration. We define the base address of the i-th local memory frame. The base address of the lowest stack frame can be easily computed from the base address of the global memory by adding the size of the global memory frame and $bubble_{\mathrm{gm}}$. For the other stack frame we define the base by induction on i.

$$abase_{\mathrm{lm}} :: tenv \times functableT \times symbolconf \times \mathbb{N} \mapsto \mathbb{N}$$

$$\begin{aligned} abase_{\mathrm{lm}}(te, ft, sc, 0) &= abase_{\mathrm{gm}}(te, ft, sc.gst) + \lceil asize_{\mathrm{st}}(sc.gst, 0) \rceil_4 \\ &\quad + 4 \cdot bubble_{\mathrm{gm}} \\ abase_{\mathrm{lm}}(te, ft, sc, i+1) &= abase_{\mathrm{lm}}(te, ft, sc, i) + \lceil asize_{\mathrm{st}}(sc.lst!i, 12) \rceil_4 \end{aligned}$$

For simplicity, we will sometimes use the following abbreviation for the base address of the first stack frame: $abase_{\mathrm{lm}}(te, ft, gst) = abase_{\mathrm{lm}}(te, ft, sc, 0)$, where $gst = sc.gst$ is the global symbol table of the given symbol configuration.

Using the definition from above, we define the allocated base address of g-variables. Observe that this definition only works for g-variables in the global or in one the local stack frames. For g-variables in the heap we cannot statically define the allocated base address because they may be moved or even deleted by the garbage collector.

Definition 7.17 (Base address of g-variables) Let te a type name environment, ft a function table, sc the current symbol configuration, and g a local or global g-variable. We define the allocated base address of a g-variable by case distinction on the location of the root g-variable of g.

$$abase_{\mathrm{g}} :: tenv \times functableT \times symbolconf \times gvar \mapsto \mathbb{N}$$

$$\begin{aligned} abase_{\mathrm{g}}(te, ft, sc, gvar_{\mathrm{gm}}(x)) &= abase_{\mathrm{gm}}(te, ft, sc.gst) + displ_{\mathrm{v}}(0, sc.gst, x) \\ abase_{\mathrm{g}}(te, ft, sc, gvar_{\mathrm{lm}}(i, x)) &= abase_{\mathrm{lm}}(te, ft, sc, i) + displ_{\mathrm{v}}(12, sc.lst!i, x) \\ abase_{\mathrm{g}}(te, ft, sc, gvar_{\mathrm{arr}}(g, i)) &= abase_{\mathrm{g}}(te, ft, sc, g) \\ &\quad + i \cdot asize_{\mathrm{algn}}(ty_{\mathrm{g}}(sc, gvar_{\mathrm{arr}}(g, i))) \end{aligned}$$

For structure access let $ty_{\mathrm{g}}(sc, gvar_{\mathrm{arr}}(g, i)) = Str_{\mathrm{T}}(scl)$.

$$abase_{\mathrm{g}}(te, ft, sc, gvar_{\mathrm{str}}(g, cn)) = abase_{\mathrm{g}}(te, ft, sc, g) + displ_{\mathrm{v}}(0, scl, cn)$$

Lemma 7.3 *The allocated base address of g-variables and the displacement of g-variables are consistent:*

$$\begin{aligned} &mem_g(g) \neq hm \\ &\Longrightarrow abase_g(te, ft, sc, g) = abase_g(te, ft, sc, root_g(g)) + displ_g(sc, g) \end{aligned}$$

Proof We omit the simple induction proof of this lemma.

Isabelle In Isabelle, this lemma is split into two lemmas: one for global g-variables and one for local ones.

7.3 Code Generation for Expressions

In this section we describe the code generation algorithm for $C0$ expressions. For this purpose we introduce the following function which computes the code for a given expression:

$$\begin{aligned} \mathit{code}_{\text{expr}} ::& \mathit{tenv} \times (\mathbb{S} \times \mathit{ty})\ \mathit{list} \times (\mathbb{S} \times \mathit{ty})\ \mathit{list} \times \mathbb{B} \times \mathit{reg} \times \mathit{reg}\ \mathit{list} \times \mathit{expr} \\ & \mapsto \mathit{instr}\ \mathit{list} \end{aligned}$$

The function has the following parameters: a type name environment, a symbol table for the global variables, a symbol table for the current local variables, a flag which specifies whether the code should compute the left or the right value of the expression, the name of the destination register, and list of registers which may be used during evaluation of sub expressions.

In the following sections we will define $\mathit{code}_{\text{expr}}$ by induction on the expression structure. Before we actually start with the definition of $\mathit{code}_{\text{expr}}$ for literals, we describe in Section 7.3.1 some general properties of the code generation algorithm.

For the remainder of this section, let te be a type name environment, gst and lst two symbol tables (for global and local variables), rf a flag which defines whether evaluation should return the value of the expression ($rf = true$) or its address ($rf = false$), r the name of the destination register, $fregs$ a list of available (free) registers, and finally e the expression to be translated.

7.3.1 Introduction

In this section we discuss some basic properties of the code generation algorithm for expressions before we continue with the formal definitions.

Basic Evaluation Strategy

The $C0$ compiler uses a quite simple and direct expression evaluation strategy by induction on the datastructure for an expression. Intermediate results are stored in the processor's registers. We do not use intermediate results from one part of an expression in another part (this is sometimes called elemination of common sub expressions); instead, we recompute the value of sub expressions for each occurence. We also do not remember the result of expression evaluation between $C0$ statements.

Left and Right Evaluation

We distinguish between expressions for which we want to compute the *value* and expressions for which we want to compute the *address*. We call the former case evaluation as a *right* expression and the latter evaluation as a left expression. The naming (left and right evaluation) arises from the situation in assignments where we compute the address of the left expression and the value of the right expression.

To differentiate between left and right evaluation we use the boolean flag rf of the $\mathit{code}_{\text{expr}}$ function. We set this flag to *true* for right evaluation and to *false* for left evaluation. During the inductive descent we can determine this flag for the sub expression from the context (cf. below). For the top-level call

of $code_{\text{expr}}$ during the code generation for statements, the flag depends on the position of the expression in the statement (cf. Section 7.4).

There is one exception to the intuitive meaning of left and right evaluation. If the value of an expression does not fit into a processor register we generate code for left evaluation (i.e., to compute the address) even if the expression is located on the right side of an assignment. In this case, the assignment code determines and assigns the *value* of the expression step-by-step in chunks that fit into a register.

Observe that the previous exception implies the invariant that *rf* is only *true* for expressions which fit into the processor registers.

Register Allocation and Order of Sub Expressions

Before we start with the actual definition of the code generation function for expressions, we give a short overview on the (simple) register allocation strategy and about the order in which we generate code for sub expressions.

The $C0$ compiler uses registers $r1$ to $r3$ as scratchpad registers and registers $r28$ to $r31$ to store special values (cf. Table 7.2 on page 129). Register $r0$ is tied to zero. Thus, registers $r4$ to $r27$ remain available for expression evaluation. We call this the set of *initially free registers* or short $fregs_{\text{ini}}$.

We introduce the function $size_{\text{e}} :: expr \mapsto \mathbb{N}$ which computes the (abstract) *size* of an expression. We define the size of an expression e as the number of data type constructors in e.

The code generation function for expressions gets a destination register r_d and a list of currently available registers *fregs* as input. If there are no sub expressions we just compute the result and store it in the destination register r_d. If there is a single sub expression we use $code_{\text{expr}}$ recursively and set $r'_d = hd(fregs)$ and $fregs' = r_d \# tl(fregs)$ for the sub expression. Observe that we use the destination register of an expression as free register for the computation of its sub expressions.

If the expression has two sub expressions (binary operators, array access, structure access) and is not a lazy expression ('logical and' or 'logical or') we use $r'_d = hd(fregs)$ and $fregs' = r_d \# tl(fregs)$ to evaluate the first sub expression and $r''_d = hd(tl(fregs))$ and $fregs' = r_d \# tl(tl(fregs))$ to evaluate the second. In order to have more registers available we evaluate the larger sub expression first.

For lazy operators, we cannot choose the order of sub expression evaluation arbitrarily because the $C0$ semantics force use not to evaluate the right sub expression if the left sub expression suffices to compute the result of the lazy operator. This makes a difference if the evaluation of the right operand would generate a runtime error like division-by-zero. Thus, for lazy operators we evaluate the sub expressions strictly from left to right.

Figure 7.4 demonstrates the register allocation strategy for expressions. We state between the operands of binary operators how their sizes relate to each other and print the number of available free registers to the right of every node.

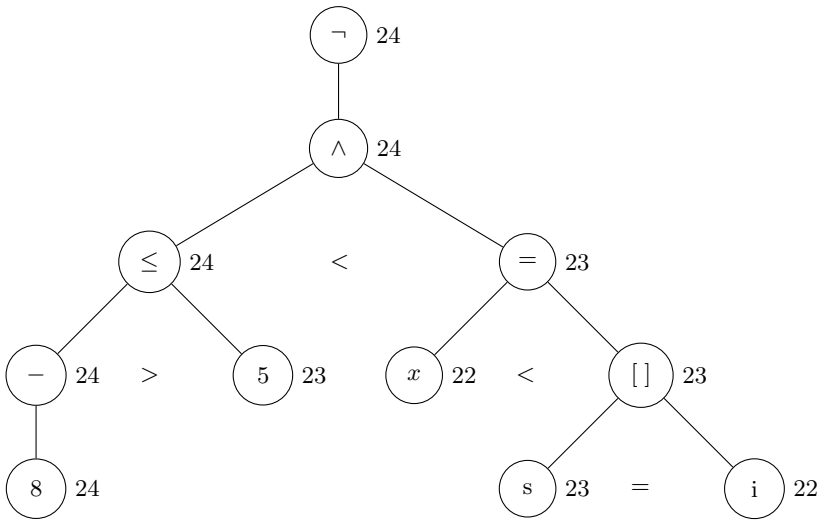


Figure 7.4: Register Allocation Strategy

7.3.2 Literals

For the VAMP processor, code generation for literals is not completely trivial because there is no support 32-bit immediate constants. Thus, 32-bit literals have to be loaded in two steps.

Definition 7.18 (Code generation for literals) We define by case distinction the function $code_{\text{lit}} :: reg \times lit \mapsto instr\ list$ which generates code for literals.

The definition for null pointers, boolean, and character literals is simple.

$$\begin{aligned} code_{\text{lit}}(r_d, Null) &= addi(r_d, r0, 0) \\ code_{\text{lit}}(r_d, Bool(b)) &= \begin{cases} addi(r_d, r0, 1) & \text{if } b \\ addi(r_d, r0, 0) & \text{if } \neg b \end{cases} \\ code_{\text{lit}}(r_d, Char(c)) &= ori(r_d, r0, c) \end{aligned}$$

Loading of 32-bit signed and unsigned literals is done in two steps. First, we load the upper half word to the upper half of the destination register using the *lhgi* instruction. After that, we combine it with the lower half word using exclusive or. Because the *xori* instruction sign extends the immediate constant we have to invert the upper half word in the case that the most significant bit of the lower half word is set.

$$\begin{aligned} code_{\text{lit}}(r_d, Unsigned(u)) &= \left[\begin{array}{l} lhgi(r_d, [bin_{32}(u)[31:16] \otimes_{\text{b}} bin_{32}(u)[15]^{16}]), \\ xori(r_d, r_d, [bin_{32}(u)[15:0]]) \end{array} \right] \\ code_{\text{lit}}(r_d, Int(i)) &= \left[\begin{array}{l} lhgi(r_d, [two_{32}(i)[31:16] \otimes_{\text{b}} two_{32}(i)[15]^{16}]), \\ xori(r_d, r_d, [bin_{32}(i)[15:0]]) \end{array} \right] \end{aligned}$$

Using $code_{\text{lit}}$ we define the code generation for literals by

$$code_{\text{expr}}(te, gst, lst, true, r_d, fregs, Lit(l)) = code_{\text{lit}}(r_d, l).$$

REMARK Observe that we will omit the first three parameters of $code_{\text{expr}}$ in the following because they are passed-trough without change. This keeps formulas shorter and focuses the readers attention to the interesting parts.

7.3.3 Variable Access

Accesses to variables can be used both as left and as right expressions. If they are used as left expressions we just compute the address of the variable. For right expressions we have to additionally load the corresponding value from the memory.

Definition 7.19 (Code generation for memory accesses) We define the function $code_{\text{load}} :: \mathbb{N} \times reg \mapsto instr\ list$ which generates code to load values of a given type from the memory. Let n be the allocated size of the type (in bytes) and r_d a register name. This register r_d specififfies the address where we want to read from. After execution of the code, r_d is also supposed to contain the value which has been read. We define

$$code_{\text{load}}(n, r_d) = \begin{cases} lbu(r_d, r_d, 0) & \text{if } n = 1 \\ lhu(r_d, r_d, 0) & \text{if } n = 2 \\ lw(r_d, r_d, 0) & \text{otherwise} \end{cases}$$

Obviously, this code can only be used for values which fit into a 32-bit register.

Now, we can define the code for variable access for the case of a left expression. We distinguish two cases: the variable is either local or global. The base address of the memory frame is stored in register r_{lframe} and r_{sbase}, respectively (cf. Figure 7.2 and Table 7.2 on page 129).

- If x is a local variable, i.e., $lst \in map(fst, x)$, we define

$$code_{\text{expr}}(false, r_d, fregs, Var(x)) = \left[\begin{array}{l} code_{\text{lit}}(r1, displ_{\text{v}}(12, lst, x)), \\ add(r_d, r_{\text{lframe}}, r1) \end{array} \right]$$

- Otherwise, x is a global variable (if it is defined at all) and we define

$$code_{\text{expr}}(false, r_d, fregs, Var(x)) = \left[\begin{array}{l} code_{\text{lit}}(r1, displ_{\text{v}}(0, gst, x)), \\ add(r_d, r_{\text{sbase}}, r1) \end{array} \right]$$

For right expressions we add code to load the value from the memory. Let t be the type of the variable, i.e., $t = the(type_{\text{v}}(lst, x))$ for local variables and $t = the(type_{\text{v}}(gst, x))$ for global ones.

$$code_{\text{expr}}(true, r_d, fregs, Var(x)) = \left[\begin{array}{l} code_{\text{expr}}(false, r_d, fregs, Var(x)), \\ code_{\text{load}}(asize_{\text{t}}(t), r_d) \end{array} \right]$$

7.3.4 Unary Operators

Unary operators are the first example of an expression which contains sub expressions. Thus, we need to apply the code generation for expressions recursively. The code for unary operators starts with the code for the sub expression which computes the result in r_d. Then we append the real code for the unary operator.

Definition 7.20 (Code for unary operators) We introduce the function $code_{\circ_1} :: reg \times unop \mapsto instr\ list$ which generates code for unary operators. Let r_d be both the source and the destination register. We define the function by case distinction

$$\begin{aligned} code_{\circ_1}(r_d, minus) &= [sub(r_d, r0, r_d)] \\ code_{\circ_1}(r_d, neg) &= [xori(r_d, r_d, -1)] \\ code_{\circ_1}(r_d, not) &= [xori(r_d, r_d, 1)] \end{aligned}$$

For conversion to integers or to unsigned integers we do not need to do anything.

$$\begin{aligned} code_{\circ_1}(r_d, to_{int}) &= [] \\ code_{\circ_1}(r_d, to_{unsgnd}) &= [] \end{aligned}$$

For conversion to 8 bit signed numbers we need to ensure that the result is in the correct domain. The necessary modulo computation is done by first shifting the value 24 bits to the left and afterwards shifting it back to the right.

$$code_{\circ_1}(r_d, to_{char}) = \left[\begin{array}{l} slli(r_d, r_d, 24), \\ srai(r_d, r_d, 24) \end{array} \right]$$

Finally, code generation for unary operators is defined by

$$code_{\text{expr}}(true, r_d, fregs, UnOp(\circ, e)) = \left[\begin{array}{l} code_{\text{expr}}(true, r_d, fregs, e), \\ code_{\circ_1}(r_d, \circ) \end{array} \right]$$

7.3.5 Binary Operators

Code generation for binary operators works similar to code generation for unary operators. We first generate code for the sub expressions. For the first and second sub expression, we use $hd(fregs)$ and $hd(tl(fregs))$ as destination registers and $r_d \# tl(fregs)$ and $r_d \# tl(tl(fregs))$ as free registers, respectively. Then we generate code for the operator itself which computes the result of the operation and stores it in the destination register. The structure of the generated code is illustrated in Figure 7.5.

Larger expressions may need more free registers to store intermediate results than smaller expressions. For the first we have one additional free register for the evaluation of the first sub expression. Thus, we evaluate the larger sub expression of a binary operator first [AU73]. Figure 7.4 clarifies the register allocation strategy.

Definition 7.21 (Code for binary operators) We introduce the function $code_{\circ_2} :: reg \times reg \times reg \times ty \times ty \times binop \mapsto instr\ list$. Application of $code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, \circ)$ generates code for a given binary operator $\circ$, assuming that the left and right source operands of type t_l and t_r, respectively, are stored in registers r_l and r_r. Below, we will define $code_{\circ_2}$ for the various operators.

Figure 7.5: Code Generation for Expressions With Binary Operators

Based on $code_{\circ_2}$ we easily define the code generation for binary operators. Let $t_l = the(type(te, gst, lst, e_l))$ and $t_r = the(type(te, gst, lst, e_r))$ be the types of the sub expressions. To keep formulas short, we define abbreviations for the bigger and the smaller sub expression. If $size_{\text{e}}(e_l) < size_{\text{e}}(e_r)$ we set $e_b = e_r$ and $e_s = e_l$. Otherwise, we set $e_b = e_l$ and $e_s = e_r$.

For the evaluation of the bigger sub expression we use $r_b = hd(fregs)$ as destination register; for the smaller sub expression we use $r_s = hd(tl(fregs))$. Depending on the size of the expressions, we also have to use the right order of source registers for the application of the binary operator. For this purpose we define $r_l = r_s$ and $r_r = r_b$ if $size_{\text{e}}(e_l) < size_{\text{e}}(e_r)$. Otherwise, we set $r_l = r_b$ and $r_r = r_s$. Finally, we define

$$code_{\text{expr}}(true, r_d, fregs, BinOp(\circ, e_l, e_r)) = \left[\begin{array}{l} code_{\text{expr}}(true, r_b, r_d\#tl(fregs), e_b), \\ code_{\text{expr}}(true, r_s, r_d\#tl(tl(fregs)), e_s), \\ code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, \circ) \end{array}\right]$$

Simple Binary Operators

In this section we define the function $code_{\circ_2}$ for simple binary operators whose generated code is very short. We do the definition by case distinction

$$\begin{aligned} code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, add) &= [add(r_d, r_l, r_r)] \\ code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, sub) &= [sub(r_d, r_l, r_r)] \\ code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, shift_{\text{l}}) &= [sll(r_d, r_l, r_r)] \\ code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, shift_{\text{r}}) &= [srl(r_d, r_l, r_r)] \\ code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, or_{\text{b}}) &= [or(r_d, r_l, r_r)] \\ code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, and_{\text{b}}) &= [and(r_d, r_l, r_r)] \\ code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, xor_{\text{b}}) &= [xor(r_d, r_l, r_r)] \\ code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, comp_{\text{eq}}) &= [seq(r_d, r_l, r_r)] \\ code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, comp_{\text{neq}}) &= [sne(r_d, r_l, r_r)] \end{aligned}$$

Comparison

Comparison of unsigned values is not directly supported by the VAMP processor, except for equality and inequality tests where signed and unsigned operand are equivalent. In fact, using one of the test instructions from Table 6.4 on page 109 for unsigned values $v_a = i2n(d.gpr!a)$ and $v_b = i2n(d.gpr!b)$ gives a wrong result in case that one source register is negative the other one is not. Observe that this is equivalent to the case that one of the unsigned values v_a and v_b is greater than 2^{31} and the other one is not (cf. definition of *i2n* on page 104).

Assume, without loss of generality, that $v_a > 2^{31}$ and $v_b \leq 2^{31}$. In this case $d.gpr!a$ would be negative and $d.gpr!b$ positive or zero. Execution of

a $sls(r1, a, b)$ instruction would set $r1$ to one although $v_a > v_b$. To fix this behavior we do code generation for unsigned operands in two steps: first, we generate the normal test code for signed operands and in a second step we add code to correct the result in the special case that one register is negative and the other one is not.

Definition 7.22 (Code for signed comparison) Code for signed comparison is generated by the function $code^{\mathrm{s}}_{\mathrm{cmp}} :: reg \times reg \times reg \times bool \mapsto instr\ list$. Its fourth parameter is used as an equality flag and specifies whether equal operands make the test successful or not. We define

$$code^{\mathrm{s}}_{\mathrm{cmp}}(r_d, r_l, r_r, eq) = \begin{cases} [sle(r_d, r_l, r_r)] & \text{if } eq \\ [sls(r_d, r_l, r_r)] & \text{if } \neg eq \end{cases}$$

Code generated by $code^{\mathrm{s}}_{\mathrm{cmp}}(r_d, r_l, r_r, false)$ tests if $r_l < r_r$ and sets the destination register appropriately. Similarly, $code^{\mathrm{s}}_{\mathrm{cmp}}(r_d, r_l, r_r, true)$ generates code which tests for $r_l \leq r_r$.

Definition 7.23 (Code for unsigned comparison) For comparison of unsigned operands we define the following code generation function

$$code^{\mathrm{u}}_{\mathrm{cmp}}(r_d, r_l, r_r, eq) = \begin{bmatrix} code^{\mathrm{s}}_{\mathrm{cmp}}(r1, r_l, r_r, eq), \\ xor(r_d, r_l, r_r), \\ sls(r_d, r_d, r0), \\ xor(r_d, r1, r_d) \end{bmatrix}$$

The correctness of this code pattern is shown in Section 9.3 on page 189.

Finally, for unsigned values (i.e., $t_l = Unsigned_{\mathrm{T}}$) we define the code generation for comparison operators by

$$\begin{aligned} code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, comp_{\mathrm{less}}) &= code^{\mathrm{u}}_{\mathrm{cmp}}(r_d, r_l, r_r, false) \\ code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, comp_{\mathrm{le}}) &= code^{\mathrm{u}}_{\mathrm{cmp}}(r_d, r_l, r_r, true) \\ code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, comp_{\mathrm{greater}}) &= code^{\mathrm{u}}_{\mathrm{cmp}}(r_d, r_r, r_l, false) \\ code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, comp_{\mathrm{ge}}) &= code^{\mathrm{u}}_{\mathrm{cmp}}(r_d, r_r, r_l, true) \end{aligned}$$

We define accordingly for signed values

$$\begin{aligned} code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, comp_{\mathrm{less}}) &= code^{\mathrm{s}}_{\mathrm{cmp}}(r_d, r_l, r_r, false) \\ code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, comp_{\mathrm{le}}) &= code^{\mathrm{s}}_{\mathrm{cmp}}(r_d, r_l, r_r, true) \\ code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, comp_{\mathrm{greater}}) &= code^{\mathrm{s}}_{\mathrm{cmp}}(r_d, r_r, r_l, false) \\ code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, comp_{\mathrm{ge}}) &= code^{\mathrm{s}}_{\mathrm{cmp}}(r_d, r_r, r_l, true) \end{aligned}$$

Multiplication

The VAMP processor does not provide hardware support for multiplication or division. Thus, we have to generate code which computes the result of multiplication in software.

The code which we generate for multiplication (cf. Listing 7.1) executes the following algorithm which computes unsigned multiplication: $i2n(r_d) = i2n(r_l) \cdot i2n(r_r) \bmod_{\mathrm{u}} 2^{32}$. However, the algorithm is also correct for signed multiplication (cf. remarks in the proof of Theorem 9.2). Observe that arithmetic in this algorithm corresponds to VAMP arithmetic, i.e., is modulo 2^{32}.

```
0   xor(r_d, r_d, r_d)
4     andi(r2, r_r, 1)
8     beqz(r2, 8)
12    srli(r_r, r_r, 1)
16      add(r_d, r_d, r_l)
20    bnez(r_r, −20)
24    slli(r_l, r_l, 1)
```

Listing 7.1: Code for Software Multiplication: $code_{\text{mult}}(r_d, r_l, r_r)$

$r_d \leftarrow 0$
repeat
  $r_r \leftarrow r_r \gg 1$
  if lowest bit of r_r is set **then**
    $r_d \leftarrow r_d + r_l$
  end if
  $r_l \leftarrow r_l \ll 1$
until $r_r = 0$

The algorithm uses the fact that multiplication of a bit vector a with a single bit b can be trivially computed by a case distinction on the bit's value. If $b = 1$ the result is a, otherwise the result is 0.

In a first step we set the destination register to zero. Then we loop over the bits of r_r and add, depending on the value of the currently lowest bit, a properly shifted version of r_l. i.e., for the i-th bit of r_r we add (if the bit is set) $r_l \ll i = r_l \cdot 2^i$ to the result. In the end this sums up to the expected result.

The formal correctness proof for the multiplication code from Listing 7.1 has been done by Hristo Pentchev in his Master's Thesis [Pen07]. There, you find a more detailed description of the multiplication algorithm and of its correctness proof.

Division and Modulo

In this section we present the code which is generated for division and modulo computation. The algorithm which has been chosen computes both values at the same time. Thus, the code generated for division and modulo computation is the same except for an additional instruction in the modulo case which moves the result to the destination register.

The implementation of the division algorithm described in this section has been developed by Mark A. Hillebrand and has been verified by Hristo Pentchev within the scope of his Master's Thesis [Pen07]. Observe that $C0$ currently only supports division and modulo for unsigned integers (cf. Section 5.2.1), thus we do not have to care about negative values of dividend or divisor.

Basic Idea of the Algorithm. Let r_l and r_r be the registers which hold dividend and divisor, respectively. The basic idea of the algorithm is to divide by continuously subtracting the divisor from the dividend until the dividend becomes smaller than the divisor.

$i \leftarrow 0$
while $r_l \geq r_r$ **do**
  $r_l \leftarrow r_l - r_r$
  $i \leftarrow i + 1$
end while

After execution of the algorithm, the number i of successful subtractions equals $a \div b$ and the final value of the dividend is $a \bmod_u b$.

However, this simple algorithm has one major drawback: if the divisor is small it requires many iterations. In fact, the runtime of the algorithm is linear in the dividend which means for our case that it could require up to 2^{32} iterations.

To improve the performance of the division algorithm we proceed in two phases:

1. Shifting the divisor. As long as the divisor is smaller than the dividend, we shift it step-by-step to the left. Assume, that we shift i times.

2. Subtracting. In a second step we act similar to the trivial division algorithm from above. We subtract the shifted divisor from the dividend and add 2^i to the result. Then, we shift the divisor to the right and subtract again, this time adding 2^{i-1}. We finish after i subtraction steps.

Below, we describe the two steps in more details.

Shift Loop. In the shift loop, we iteratively shift the divisor i times to the left, where i is the smallest natural number such that $2^i \cdot i2n(r_r) \geq i2n(r_l)$. We have to be careful not to shift ones out of r_r; thus, we also stop shifting if the most significant bit of r_r is 1. We store the number i of iterations in register $r3$.

Instead of using unsigned comparison, which needs three instructions, we implement the algorithm using signed comparison and treat negative values of r_l in a special test.

while $r_r \geq 0 \wedge (r_l < 0 \vee r_r < r_l)$ **do**
  $r_r \leftarrow r_r \ll 1$
  $r3 \leftarrow r3 + 1$
end while

The assembly code for the shift loop is given in Listing 7.2. Observe that the condition of the while loop has been implemented in assembly in a quite efficient but hard to understand way in instructions zero to three.

Subtract Loop. The subtract loop implements the following algorithm:

```
1 repeat
2     r_d ← r_d ≪ 1
3     if r_l ≥ r_r then
4         r_l ← r_l − r_r
5         r_d ← r_d + 1
6     end if
7     r_r ← r_r ≫ 1
8     r3 ← r3 − 1
9 until r3 = 0
```

$$
\begin{array}{rl}
0 & sgei(r2, r_r, 0) \\
4 & sls(r1, r_r, r_l) \\
8 & or(r1, r1, r_d) \\
12 & and(r2, r2, r1) \\
16 & add(r3, r3, r2) \\
20 & bnez(r2, -24) \\
24 & sll(r_r, r_r, r2)
\end{array}
$$

Listing 7.2: Code for Software Division: Shift Loop

$$
\begin{array}{rl}
0 & slli(r_d, r_d, 1) \\
4 & code^{\mathrm{u}}_{\mathrm{cmp}}(r2, r_l, r_r, false) \\
 & \quad \ldots \\
20 & addi(r2, r2, -1) \\
24 & sub(r_d, r_d, r2) \\
28 & and(r2, r2, r_r) \\
32 & srli(r_r, r_r, 1) \\
36 & subi(r3, r3, 1) \\
40 & bnez(r3, -44) \\
44 & sub(r_l, r_l, r2)
\end{array}
$$

Listing 7.3: Code for Software Division: Subtract Loop

The algorithm uses $r3$ as loop counter which was initialized by the shift loop. In each iteration it shift the *current* quotient in the r_d register one bit to the left and, if $r_l \geq r_r$, adds one to r_d and subtracts r_r from the current dividend in r_l. After that it shift the current divisor one bit to the right and proceeds to the next loop iteration. Finally, the quotient is stored in register r_d and the remainder in register r_l.

The main difference between the algorithm and the basic idea from above is that instead of adding 2^{i-1} we just add one and shift the quotient $i-1$ bits to the left in the remaining loop iterations. This approach has the same effect but is more efficient.

The code for the subtract loop is given in Listing 7.3. As it is by no means obvious that this code implements the algorithm we explain its critical part in more details. The critical part is the implementation of the two conditionally executed statements $r_l = r_l - r_r$ and $r_d = r_d + 1$. To improve performance, this part has been realized without branch or jump instructions with the seven instructions starting at offset 4 and with the last instruction, which is in the delay slot of the branch instruction.

The first four out of the seven instructions mentioned above do an unsigned comparison of r_l and r_r, as done in line 3 in the algorithm. The next instruction subtracts one from register $r2$. It sets $r2$ to zero if r_l was smaller than r_r and to -1 otherwise. This implies that the following *sub* instruction adds one to r_d if $r_l \geq r_r$ and otherwise leaves it unchanged; this resembles the behavior of the

conditional execution of the statement in line 5 of the algorithm. The following instruction ($and(r2, r2, r_r)$) set register $r2$ to r_r if $r2 = -1$, i.e., if $r_l \geq r_r$, and to zero otherwise. This together with the last instruction of the code block for the subtract loop realizes the statement in line 4 of the algorithm.

Combining the Pieces. Assembling the code for division and modulo computation by combining shift and subtract loop is not hard.

Definition 7.24 (Code for division and modulo) Let r_d be the destination register and r_l and r_r the registers which contain the dividend and the divisor, respectively. We define the code module for division and modulo by

$$code_{\text{div}}(r_d, r_l, r_r) = \begin{bmatrix} slsi(r_d, r_l, 0) \\ addi(r3, r0, 1) \\ code_{\text{div}}^{\text{shift}}(r_d, r_l, r_r) \\ addi(r_d, r0, 0) \\ code_{\text{div}}^{\text{subtract}}(r_d, r_l, r_r) \end{bmatrix}$$

First, the code initializes the destination register with the most significant bit of the dividend and sets r_l, which is used as the shift counter in the shift loop, to one. This prepares the work of the shift loop. Then we have the code for the shift loop itself and afterwards we initialize the destination register to zero to prepare the execution of the subtract loop which computes the result.

Finally, we define code generation for divide and modulo expressions by

$$code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, div) = [code_{\text{div}}(r_d, r_l, r_r)]$$
$$code_{\circ_2}(r_d, r_l, r_r, t_l, t_r, mod) = \begin{bmatrix} code_{\text{div}}(r_d, r_l, r_r), \\ addi(r_d, r_l, 0) \end{bmatrix}$$

Isabelle Observe that in Isabelle the final *addi* instruction for the division case is placed inside the $code_{\text{div}}$ function. Thus, this function has an additional parameter which specifies whether code should be generated for division or modulo. For clarify of definition, we changed this slightly in this text.

7.3.6 Lazy Binary Operators

Code generation for lazy binary operators (and_{l} and or_{l}) differs from code generation for other binary operators in two ways.

1. For lazy binary operators we have no choice in the execution order of sub expressions. We have to compute the result of the left sub expression first because this may forbid to evaluate the second sub expression.

2. We have to add code in between the code for the two sub expressions in order to check whether the second one needs to be evaluated or not. We will call this the *central code* for the lazy operator.

Figure 7.6 depicts the code generation for expressions with lazy binary operators.

In the following, we introduce two functions which generate the code for the lazy binary operator and the corresponding central code. The code for the lazy operators is quite trivial.

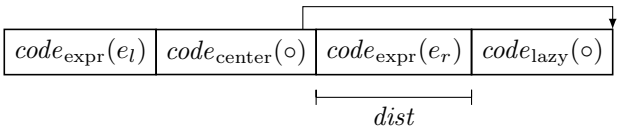


Figure 7.6: Code Generation for Expressions With Lazy Binary Operators

Definition 7.25 (Code for lazy operators) The function $code_{\text{lazy}} :: reg \times reg \times reg \times lazyop \mapsto instr\ list$ generates code for lazy operators.

$$code_{\text{lazy}}(r_d, r_l, r_r, and_{\text{l}}) = [and(r_d, r_l, r_r)]$$
$$code_{\text{lazy}}(r_d, r_l, r_r, or_{\text{l}}) = [or(r_d, r_l, r_r)]$$

The central code also depends on the operator but is more complicated. In case that the result of the left sub expression suffices to compute the result of the whole expression, we jump over the code for the right sub expression and for the lazy operator. Instead we set the destination register directly to the correct value and jump behind the code for the complete lazy expression.

Definition 7.26 (Lazy operators: central code) We introduce the function $code_{\text{center}} :: reg \times reg \times \mathbb{Z} \times lazyop \mapsto instr\ list$ which generates the central code for lazy operators. Let r_l be the register which contains the result of the left sub expression, r_d the destination register, and $dist$ the size of the code which is generated for the right sub expression and for the lazy operator. Then we define the central code as follows.

$$code_{\text{center}}(r_d, r_l, dist, and_{\text{l}}) = \begin{bmatrix} bnez(r_l, 12), \\ nop, \\ j(dist + 4), \\ addi(r_d, r0, 0) \end{bmatrix}$$

$$code_{\text{center}}(r_d, r_l, dist, or_{\text{l}}) = \begin{bmatrix} beqz(r_l, 12), \\ nop, \\ j(dist + 4), \\ addi(r_d, r_l, 0) \end{bmatrix}$$

Finally, the code generation for lazy binary expressions is defined by

$$code_{\text{expr}}(true, r_d, fregs, LazyBinOp(\circ, e_l, e_r)) = \begin{bmatrix} code_{\text{expr}}(true, hd(fregs), r_d \# tl(fregs), e_l), \\ code_{\text{center}}(r_d, hd(fregs), 4 \cdot csize_{\text{e}}(true, e_r) + 4, \circ), \\ code_{\text{expr}}(true, hd(tl(fregs)), r_d \# tl(tl(fregs)), e_r), \\ code_{\text{lazy}}(r_d, hd(fregs), hd(tl(fregs)), \circ) \end{bmatrix}$$

7.3.7 Structure and Array Access

For access to structure components and to array elements we proceed in a similar way. We first generate code for the base expression (i.e., the structure or the array) and in a second step we generate code which computes the offset of the component or the element in the base expression. For structures the second step is similar to variable access (cf. Section 7.3.3) whereas for access to array elements we need to reuse the multiplication code from Section 7.3.5.

Access to Structure Components

After having the address of the structure in a register, say r_l, the code for structure access is nearly the same as the code for variable access.

Let $t = the(type(te, gst, lst, e)) = Str_{\mathrm{T}}(scl)$ be the type of the structure and $t_c = the(type_{\mathrm{v}}(scl, cn))$ the type of component cn. Then, we define for left expressions

$$code_{\mathrm{expr}}(false, r_d, fregs, Str(e, cn)) = \left[\begin{array}{l} code_{\mathrm{expr}}(false, hd(fregs), r_d \# tl(regs), e), \\ addi(r_d, hd(fregs), displ_{\mathrm{v}}(0, scl, cn)) \end{array}\right]$$

For right expressions we define

$$code_{\mathrm{expr}}(true, r_d, fregs, Str(e, cn)) = \left[\begin{array}{l} code_{\mathrm{expr}}(false, r_d, fregs, Str(e, cn)), \\ code_{\mathrm{load}}(asize_{\mathrm{t}}(t_c), r_d) \end{array}\right]$$

Access to Array Elements

Code generation for arrays is similar to code generation for structure access: generate code for the sub expression, compute an offset, in case of a right expressions load the value from memory. Only the computation of the offset inside the array is done differently. We first have to generate code to evaluate the index expression and then we have to multiply the result with the element size of the array.

As for other binary expressions we evaluate the larger sub expression first. Thus, let e_b and e_s be the bigger and smaller sub expression, respectively. Also, let r_i be the register into which we evaluate the index expression and r_a the register containing the address of the array expression. Thus, if the index expression is bigger ($e_b = e_i$) we have $r_i = hd(fregs)$ and $r_a = hd(tl(fregs))$. In the other case we have $r_i = hd(tl(fregs))$ and $r_a = hd(fregs)$.

Let $t = the(type(te, gst, lst, e_a)) = Arr_{\mathrm{T}}(n, t_e)$ be the type of the array which has n elements of type t_e. Then, we define for left expressions

$$code_{\mathrm{expr}}(false, r_d, fregs, Arr(e_a, e_i)) = \left[\begin{array}{l} code_{\mathrm{expr}}(true, hd(fregs), r_d \# tl(regs), e_b), \\ code_{\mathrm{expr}}(false, hd(tl(fregs)), r_d \# tl(tl(regs)), e_s), \\ addi(r1, r0, asize_{\mathrm{algn}}(t_e)), \\ code_{\mathrm{mult}}(r_d, r1, r_i), \\ add(r_d, r_d, r_a) \end{array}\right]$$

The code behind the evaluation of the sub expression just initializes $r1$ with the element size of the array. Then we multiply $r1$ with the index which has been stored in r_i. This computes the element's offset inside the array and stores it in r_d. Finally, we add the offset to the base address of the array which has been stored in r_a.

For right expressions we simply append code to load the value of the array

element from the memory.

$$code_{\text{expr}}(true, r_d, fregs, Arr(e_a, e_i)) = \left[\begin{array}{l} code_{\text{expr}}(false, r_d, fregs, Arr(e_a, e_i)) = \\ code_{\text{load}}(asize_{\text{t}}(t_e), r_d) \end{array}\right]$$

7.3.8 Address-of and Pointer Dereferencing

Code generation for dereferencing pointers and for the address-of operator is quite simple.

The address-of operator can only be used as a right expression in valid $C0$ programs. Code generation for it works by simply generating code for its sub expression as a left expression. This code computes the address of the sub expression and thus fulfills the semantics of the address-of operator.

$$code_{\text{expr}}(true, r_d, fregs, AddrOf(e)) = code_{\text{expr}}(false, r_d, fregs, e)$$

Dereferencing a pointer, $Deref(e_p)$, can occur both as left and as right expression. In both cases we first generate code which evaluates the sub expression as a right expression. If $Deref(e_p)$ is used as a left expression, we are done. In this case the address of the expression, which is just the *value* of e_p, has already been computed by the code for e_p. Otherwise, if it is used as a right expression, we have to append code which loads the value from the memory. Let $t = the(type(te, gst, lst, e_p)) = Ptr_{\text{T}}(tn)$ be the type of expression e_p and let $t_p = the(map\text{-}of(te, tn))$ be the type to which the pointer of type t refers.

$$code_{\text{expr}}(false, r_d, fregs, Deref(e_p)) = code_{\text{expr}}(true, r_d, fregs, e_p)$$
$$code_{\text{expr}}(true, r_d, fregs, Deref(e_p)) = \left[\begin{array}{l} code_{\text{expr}}(true, r_d, fregs, e_p), \\ code_{\text{load}}(asize_{\text{t}}(t_p), r_d) \end{array}\right]$$

7.4 Code Generation for Statements

In this section we describe the code generation for statements. We introduce the function $code_{\text{stmt}}$ which computes the code for a given statement.

$$code_{\text{stmt}} :: tenv \times functableT \times (\mathbb{S} \times ty)\ list \times (\mathbb{S} \times ty)\ list \times stmt \mapsto instr\ list$$

As an abbreviation we introduce a second variant of the above function which computes the necessary local symbol table depending on the function to which the statement belongs. If the statement is not present in the program the function is undefined.

$$code_{\text{stmt}} :: tenv \times (\mathbb{S} \times ty)\ list \times functableT \times stmt \mapsto instr\ list$$
$$code_{\text{stmt}}(te, gst, ft, s) = \begin{cases} \lfloor code_{\text{stmt}}(te, ft, gst, st_{\text{fun}}(f), s) \rfloor & \text{if } fos(s, ft) = \lfloor f \rfloor \\ undef & \text{otherwise} \end{cases}$$

Remark The code generation algorithm presented in this thesis does not cover XCalls or inline assembly code. Support for the latter has been added recently by A. Tsyban. Support for XCalls is not required for the compiler because they are replaced by an appropriate implementation (using inline assembly code) before invocation of the $C0$ compiler (cf. [ASS08]).

In the remainder of this section we define $code_{\text{stmt}}$ by induction on the statement structure. As in the previous section for the definition of $code_{\text{expr}}$ we omit the parameters for type name environment, function table, and symbol tables to keep formulas simple. Thus, let *te* be the current type name environment, *ft* the function table, and *gst* and *lst* the global and current local symbol table, respectively.

7.4.1 Skip and Compound

We start the definition of $code_{\text{stmt}}$ with the simplest cases. For *Skip* we need no code at all and for the compound statement we recursively generate code for the sub statements.

$$code_{\text{stmt}}(gst, lst, Skip) = [\,]$$
$$code_{\text{stmt}}(gst, lst, Comp(s_1, s_2)) = \begin{bmatrix} code_{\text{stmt}}(gst, lst, s_1), \\ code_{\text{stmt}}(gst, lst, s_2) \end{bmatrix}$$

7.4.2 Assignments

For assignments $Ass(e_l, e_r)$ we first generate code for the expressions e_l and e_r. We evaluate e_l as a left expression to get the allocated address of the expression. For the evaluation of e_r we distinguish two cases. If the type of e_r is elementary we evaluate e_r as a right expression. Otherwise, e_r is an array or a structure and we evaluate it as a left expression. Finally, we add code which copies the value of the right expression to the address of the left expression.

For the first case this code is quite simple.

Definition 7.27 (Code for storing basic values) We define the function $code_{\text{store}} :: \mathbb{N} \times reg \times reg \times \mathbb{N} \mapsto instr\ list$ which generates code to copy a value of a given type from a register to a given destination address in the memory. Let n be the allocated size of the type (in bytes), r_d and r_s register names, and *ofs* an (byte) offset. The register r_d and the offset define the destination address of the access; the value to be copied is stored in register r_s.

We define depending on the given offset. If the offset is zero we set

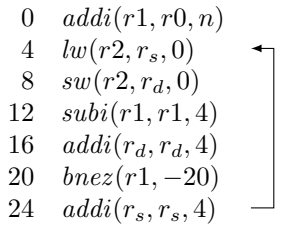
$$code_{\text{store}}(n, r_d, r_s, 0) = \left[\begin{cases} sb(r_s, r1, 0) & \text{if } n = 1 \\ sh(r_s, r1, 0) & \text{if } n = 2 \\ sw(r_s, r1, 0) & \text{otherwise} \end{cases} \right]$$

If the offset is greater than zero we define

$$code_{\text{store}}(n, r_d, r_s, ofs) = \begin{bmatrix} code_{\text{lit}}(r1, ofs), \\ add(r1, r1, r_d), \\ code_{\text{store}}(n, r1, r_s, 0) \end{bmatrix}$$

Obviously, this code can only be used for values which fit into a single 32-bit register.

For the second case, i.e., assignments of arrays and structures, we use a loop which copies a memory region word-by-word. The code of this loop (cf. Listing 7.4) is generated by the function $code^{\text{loop}}_{\text{cpy}} :: reg \times reg \times \mathbb{N} \mapsto instr\ list$.



Listing 7.4: Code for Copying Memory Regions: $code^{\text{loop}}_{\text{cpy}}(r_d, r_s, n)$

Let n be the size of the memory region in words and let registers r_s and r_d contain the source and destination address, respectively. Then, the code generated by $code^{\text{loop}}_{\text{cpy}}(r_d, r_s, n)$ implements the following algorithm:

$r1 \leftarrow n$
repeat
  $mm[r_d \div 4] \leftarrow mm[r_s \div 4]$
  $r1 \leftarrow r1 - 4$
  $r_d \leftarrow r_d + 4$
  $r_s \leftarrow r_s + 4$
until $r1 = 0$

Definition 7.28 (Code for copying memory regions) Similar to assignments of elementary values we combine this code template with code to add an offset to the destination address.

If the offset is zero, we set $code_{\text{cpy}}(r_d, r_s, n, 0) = code^{\text{loop}}_{\text{cpy}}(r_d, r_s, n)$. Otherwise, we define

$$code_{\text{cpy}}(r_d, r_s, n, ofs) = \left[\begin{array}{l} code_{\text{lit}}(r3, ofs), \\ add(r3, r_d, r3), \\ code^{\text{loop}}_{\text{cpy}}(n, r3, r_s, 0) \end{array}\right]$$

Definition 7.29 We combine the code generation for assignments of basic and of big values.

Let *big* be a flag which specifies whether we assign a basic value ($big = false$) or a big value, r_d and r_s the destination and source register, n the number of bytes to be copied, and *ofs* an offset to the destination address. Then, the function $code_{\text{ass}} :: \mathbb{B} \times reg \times reg \times \mathbb{N} \times \mathbb{N} \mapsto instr\ list$ copies the value specified by the source register – either stored *in* the register (if $\neg big$) or at the address which is specified by the register (if *big*) – to the memory at the address specified by the destination register.

$$code_{\text{ass}}(big, r_d, r_s, n, ofs) = \begin{cases} code_{\text{cpy}}(r_d, r_s, n, ofs) & \text{if } big \\ code_{\text{store}}(n, r_d, r_s, ofs) & \text{if } \neg big \end{cases}$$

Now, we define the code generation for an assignment statement $Ass(e_l, e_r)$. Let $t = the(type(te, gst, lst, e_l))$ be the type of the left expression and $rf = elem?_{\text{t}}(t)$ be the flag which specifies whether we have to evaluate e_r as a right

or as a left expression. Then, we set

$$code_{\text{stmt}}(Ass(e_l, e_r)) = \left[\begin{array}{l} code_{\text{expr}}(false, hd(fregs_{\text{ini}}), tl(fregs_{\text{ini}}), e_l), \\ code_{\text{expr}}(rf, hd(tl(fregs_{\text{ini}})), tl(tl(fregs_{\text{ini}})), e_r), \\ code_{\text{ass}}(\neg rf, hd(fregs_{\text{ini}}), hd(tl(fregs_{\text{ini}})), asize_{\text{t}}(t), 0) \end{array}\right]$$

7.4.3 Assignments of Aggregate Literals

In the previous case for 'normal' assignments we had a value which either fit into a register a the target machine or was stored in the memory. Assignments of aggregate literals need special treatment; they are not stored in the memory and they do not fit into a register.

Code generation for assignments of aggregate literals, which are composed of basic literals, works by appending code for assigning these basic literals at the destination address with a proper offset.

Definition 7.30 (Assignments of aggregate literals) We define the function $code_{\text{alit}}$ which generates code to store an aggregate literal at a given destination address. For the handling of structure literals we introduce the auxiliary function $code^{\text{struct}}_{\text{alit}}$. Let r_d be the register which contains the destination address of the assignment and *ofs* an offset to this destination address.

$$code_{\text{alit}} :: reg \times \mathbb{N} \times lit_{\text{a}} \mapsto instr\ list$$
$$code^{\text{struct}}_{\text{alit}} :: reg \times \mathbb{N} \times \mathbb{N} \times (\mathbb{S} \times lit_{\text{a}})\ list \mapsto instr\ list$$

Simple literals are evaluated into an auxiliary register and then stored in the memory.

$$code_{\text{alit}}(r_d, ofs, ALPrim(p)) = \left[\begin{array}{l} code_{\text{lit}}(r2, p), \\ code_{\text{store}}(asize_{\text{t}}(type_{\text{lit}}(p)), r_d, r2, ofs) \end{array}\right]$$

For arrays we just generate code for the single elements with corresponding offsets.

$$code_{\text{alit}}(r_d, ofs, ALArr([\,])) = [\,]$$
$$code_{\text{alit}}(r_d, ofs, ALArr(h\#t)) = \left[\begin{array}{l} code_{\text{alit}}(r_d, ofs, h), \\ code_{\text{alit}}(r_d, ofs + asize_{\text{algn}}(type_{\text{alit}}(h)), ALArr(t)) \end{array}\right]$$

For structures it is a little bit harder to compute the right offset. We introduce a new accumulator parameter which keeps track of the offset *within* the structure; this local offset is initially zero.

$$code_{\text{alit}}(r_d, ofs, ALStruct(scl)) = code^{\text{struct}}_{\text{alit}}(r_d, ofs, 0, scl)$$

Let in the following $i' = \lceil i \rceil_{align(type_{\text{alit}}(snd(h)))}$ be the aligned local offset, i.e., the local offset where we store the next component of the structure.

$$code^{\text{struct}}_{\text{alit}}(r_d, ofs, i, [\,]) = [\,]$$
$$code^{\text{struct}}_{\text{alit}}(r_d, ofs, i, h\#t) = \left[\begin{array}{l} code_{\text{alit}}(r_d, ofs + i', snd(h)), \\ code^{\text{struct}}_{\text{alit}}(r_d, ofs, i' + asize_{\text{t}}(type_{\text{alit}}(snd(h))), t) \end{array}\right]$$

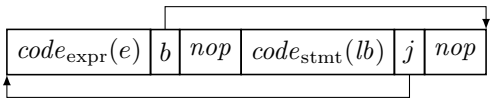


Figure 7.7: Code Generation Template for Loops

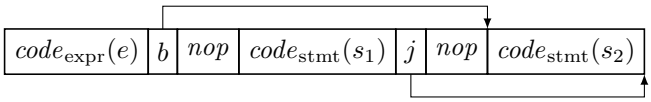


Figure 7.8: Code Generation Template for Conditional Statements

Finally, we define the code generation for aggregate literals. Observe that the actual copying of the literals is hidden in the definition of $code_{\text{alit}}$ for the case of primitive literals.

$$code_{\text{stmt}}(Ass_{\text{AL}}(e_l, Lit(l))) = \left[\begin{array}{l} code_{\text{expr}}(false, hd(fregs_{\text{ini}}), tl(fregs_{\text{ini}}), e_l), \\ code_{\text{alit}}(hd(fregs_{\text{ini}}), 0, l) \end{array} \right]$$

7.4.4 Loops

For while loops $Loop(c, lb)$ we generate code in four steps. First, we generate code for the condition c. Then, we generate code to conditionally jump behind the code of the loop body if the condition is *false*. This code is followed by the code for the body of the loop. Finally, we generate code to jump back to the evaluation of c to start another loop iteration. This code generation template is depicted in Figure 7.7.

Formally, we define

$$code_{\text{stmt}}(Loop(c, lb)) = \left[\begin{array}{l} code_{\text{expr}}(true, hd(fregs_{\text{ini}}), tl(fregs_{\text{ini}}), c), \\ beqz(hd(fregs_{\text{ini}}), 4 \cdot (csize_{\text{s}}(lb) + 3)), \\ nop, \\ code_{\text{stmt}}(lb), \\ j(-4 \cdot (csize_{\text{e}}(true, c) + csize_{\text{s}}(lb) + 3)), \\ nop \end{array} \right]$$

7.4.5 Conditional Statements

For conditional statements $Ifte(c, s_1, s_2)$ we generate code in five steps. We start with code generation for the condition, which is followed by a branch which jumps to the code for the second sub statements if the condition is *false*. This branch is followed by the code for the first sub statement. To skip the second sub statement if the condition was *true*, we append a jump behind the last instruction of the code for the conditional statement. Finally, we generate code for the second sub statement. This code generation template is depicted in Figure 7.8.

$$\begin{array}{ll} 0 & sw(r0, r_z, 0) \\ 4 & addi(r_c, r_c, -4) \\ 8 & bnez(r_c, -12) \\ 12 & addi(r_z, r_z, 4) \end{array}$$

Listing 7.5: Code for Zero-Filling a Memory Region: $code_{\text{zerofill}}(r_z, r_c)$

Formally, we define

$$code_{\text{stmt}}(\mathit{Ifte}(c, s_1, s_2)) = \left[\begin{array}{l} code_{\text{expr}}(\mathit{true}, hd(\mathit{fregs}_{\text{ini}}), tl(\mathit{fregs}_{\text{ini}}), c), \\ beqz(hd(\mathit{fregs}_{\text{ini}}), 4 \cdot (csize_{\text{s}}(s_1) + 3)), \\ nop, \\ code_{\text{stmt}}(s_1), \\ j(4 \cdot (csize_{\text{s}}(s_2) + 1)), \\ nop, \\ code_{\text{stmt}}(s_2) \end{array}\right]$$

7.4.6 Allocation of Heap Variables

The code which we generate for $\mathit{PAlloc}(e, t_n)$, i.e., for the allocation of new heap variables, is composed of two parts. The first part tests whether there is enough heap memory available and stores a pointer to the new heap variable (or a null pointer in case there was not enough memory) in one of the registers. The second part assigns this pointer to the allocated address of e and – given that the allocation was successful – initializes the new memory region with zeroes.

We start with the function $code^{\text{test}}_{\text{alloc}} :: \mathbb{N} \times \mathit{reg} \mapsto \mathit{instr\ list}$ which specifies the first part. The code $code^{\text{test}}_{\text{alloc}}(n, r)$ tests whether there are at least n more bytes available in the heap and correspondingly stores a null pointer or a pointer to the newly allocated memory in register r. We define

$$code^{\text{test}}_{\text{alloc}}(n, r) = \left[\begin{array}{l} code_{\text{lit}}(r, \mathit{Unsigned}(abase_{\text{heap}} + asize^{\text{max}}_{\text{heap}} - \lceil n \rceil_4)), \\ code^{\text{u}}_{\text{cmp}}(r, r, r_{\text{htop}}, \mathit{true}), \\ subi(r, r, 1), \\ and(r, r, r_{\text{htop}}) \end{array}\right]$$

The compiler will ensure that the value of r_{htop} is always aligned to multiples of four. This guarantees that newly allocated heap variables are always properly aligned. Thus, we need to round n up to a multiple of four before the test.

Additionally, the subtraction and bitwise-and instructions allow to conditionally set r without a branch instruction which is usually (at least for modern processors with branch prediction) faster.

For the second part, we introduce the function $code_{\text{zerofill}} :: \mathit{reg} \times \mathit{reg} \mapsto \mathit{instr\ list}$ which generates code to fill a certain region of the memory with zeroes. Let r_z be the register which contains the start address of the region and r_c the register which contains the length of the region. Then, the code in Listing 7.5 is generated by $code_{\text{zerofill}}(r_z, r_c)$.

Now, we define the second code part for allocation using the function $code^{\text{zero}}_{\text{alloc}} :: \mathbb{N} \times \mathit{reg} \times \mathit{reg} \mapsto \mathit{instr\ list}$. Let r be the register which contains either

$$
\begin{array}{ll}
0 & beqz(r, 28) \\
4 & sw(r, r_d, 0) \\
8 & code_{\text{lit}}(r3, Unsigned(\lceil n \rceil_4)) \\
16 & code_{\text{zerofill}}(r_{\text{htop}}, r3) \\
32 & \cdots
\end{array}
$$

Listing 7.6: Second Part of the Code for Allocation: $code^{\text{zero}}_{\text{alloc}}(n, r_d, r)$

a zero (null pointer) or the address of the newly allocated heap variable. Further, let n be the number of bytes to allocate and r_d the address where the content of register r should be copied two, i.e., the allocated address of the left side expression of the allocation statement. Then, $code^{\text{zero}}_{\text{alloc}}(n, r_d, r)$ generates the code depicted in Listing 7.6. Observe that the store word instruction is executed in any case because it is located in the delay slot of the branch instruction.

Finally, we combine the two parts. Let $Ptr(t_n) = the(type(te, gst, lst, e))$ be the type of the expression and $t = the(map\text{-}of\,(te, t_n))$ the type of the newly allocated heap variable. Then we define

$$
code_{\text{stmt}}(PAlloc(e, t_n)) = \left[\begin{array}{l}
code_{\text{expr}}(false, hd(fregs_{\text{ini}}), tl(fregs_{\text{ini}}), e), \\
code^{\text{test}}_{\text{alloc}}(asize_{\text{t}}(t), r2), \\
code^{\text{zero}}_{\text{alloc}}(asize_{\text{t}}(t), hd(fregs_{\text{ini}}), r2)
\end{array} \right]
$$

7.4.7 Function Calls

The code generated for function calls $SCall(vn, fn, [p_1, \ldots, p_n])$ works in four phases.

1. It computes the address of the return destination.
2. It evaluates the parameters p_1 to p_n and copies their values to the corresponding position in the new stack frame.
3. It initializes the frame header of the new stack frame and updates the stack pointer in register r_{lframe}.
4. It jumps to the called function and stores the return address, i.e., the current value of PCP, in the frame header of the new stack frame.

Observe that we perform the evaluation of the parameters still in the old memory frame (before updating the stack pointer). This will allow us to smoothly reuse the correctness proofs for expression evaluation for the evaluation of the parameters.

In the remainder of this section we first introduce code generation for parameter passing (phase 2). Then we define the code generation for the initialization of the new frame header and the jump to the function. Finally, we put the code pieces together in the definition of $code_{\text{stmt}}$ for function calls.

Parameter Passing

Passing a single parameter is similar to an assignment. We evaluate the parameter as a right or left expression depending on its type. Structures and

arrays are evaluated as left expressions; all other parameter are evaluated as right expressions. Using expression evaluation for the target of the parameter passing would generate some problems, because the destination would have to be evaluated in the context of the new frame although the stack is still unchanged. However, the destination of the parameter passing is always a local variable of the function. Thus, we can easily compute the destination address via specialized code.

To copy a single parameter we use the same code as for assignments. The code for passing several parameters is generated by induction on the parameter list.

Definition 7.31 (Code for parameter passing) We introduce a function $code_{\text{passing}} :: tenv \times funcT \times (\mathbb{S} \times ty)\ list \times (\mathbb{S} \times ty)\ list \times reg\ list \times \mathbb{N} \times expr\ list \times \mathbb{S}\ list \mapsto instr\ list$ which generates code for passsing a list of parameters. Let te be a type name environment, f the called function, gst and lst global and current local symbol table, $fregs$ a list of free register names, fs the allocated size of the current stack frame, and pnl a list which contains the names of the local variables which have to be initialized with the values of the parameters or short the parameter names.

Then, we define $code_{\text{passing}}(te, f, gst, lst, fregs, fs, pl, pnl)$ by induction on the list pl of parameter expressions. Observe that we omit te, f, gst, lst, and fs in the formulas below to keep them short and readable.

For the induction start, the definition is trivial.

$$code_{\text{passing}}(fregs, [\,], [\,]) = [\,]$$

For the non-empty case let $t = the(type(te, gst, lst, p_h))$ be the type of the first parameter, $rf = elem?_{\text{t}}(t)$ a flag which specifies if we evaluate the parameter as right or left expression, and $ofs = fs + displ_{\text{v}}(12, st_{\text{fun}}(f), pn_h)$ the offset of the first destination variable with respect to the base of the current frame.

$$code_{\text{passing}}(fregs, p_h \# p_t, pn_h \# pn_t) = \left[\begin{array}{l} code_{\text{expr}}(rf, hd(fregs_{\text{ini}}), tl(fregs_{\text{ini}}), p_h), \\ code_{\text{ass}}(\neg rf, r_{\text{lframe}}, hd(fregs_{\text{ini}}), asize_{\text{t}}(t), ofs), \\ code_{\text{passing}}(fregs, p_t, pn_t) \end{array}\right]$$

Initialization of the New Frame Header and Jump to the Function

Now, we define the code which sets up the headers of the new stack frame, adapts the stack pointer, and jumps to the called function. We start with the computation of the distance to the called function (cf. Figure 7.9).

Definition 7.32 (Call distance) We introduce the function $dist_{\text{scall}} :: tenv \times functableT \times (\mathbb{S} \times ty)\ list \times (\mathbb{S} \times ty)\ list \times stmt \mapsto \mathbb{Z}$ which computes the distance from the *jal* instruction of a function call to the start of the called function. The function subtracts the address of the jump-and-link instruction from the start address of the called function. Let n_{pp} be the size of the parameter passing code from above and observe that $code_{\text{lit}}$ for unsigned numbers has size 2. Observe that we additionally subtract one from the call distance to compensate the fact that the immediate constant for jumps is added to pcp which equals $dpc + 4$

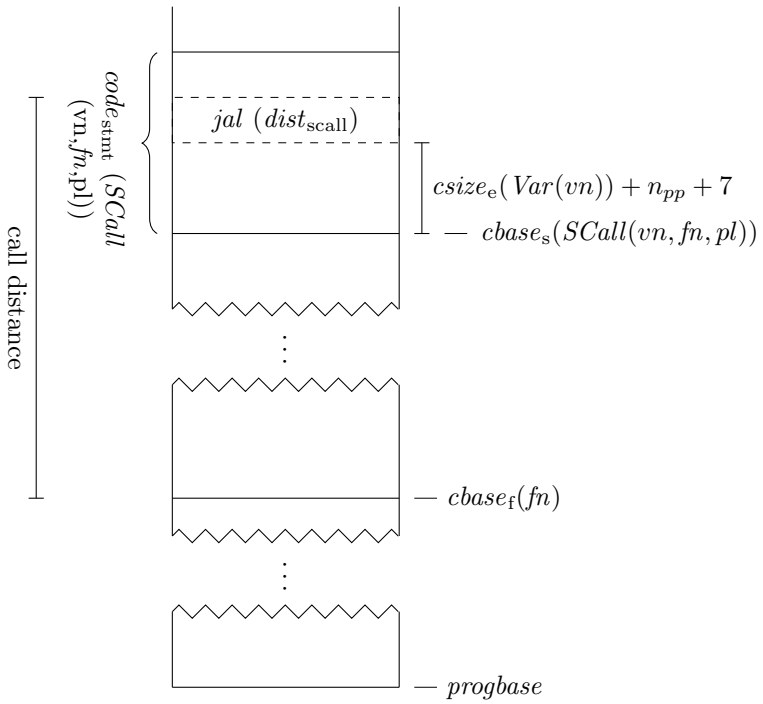


Figure 7.9: Call Distance for Function Calls

(cf. Table 6.3 on page 109). We define

$$\begin{aligned} dist_{\text{scall}}(te, ft, gst, lst, SCall(vn, fn, [p_1, \ldots, p_n])) = \\ & cbase_{\text{f}}(te, gst, fn, ft) \\ & - cbase_{\text{s}}(te, gst, ft, SCall(vn, fn, [p_1, \ldots, p_n])) \\ & - 4 \cdot (csize_{\text{e}}(false, Var(vn)) + n_{pp} + 2 + 4 + 1) \end{aligned}$$

Using this definition, we can easily define the code which sets up the new frame and jumps to the called function. The effect of this code fragment is illustrated in Figure 7.10 (d is the assembly configuration *before* and d' *after* the execution of $code_{\text{newfr}}$).

Definition 7.33 (Code for setting up the new frame) Let fs_{cur} be the size of the current stack frame, fs_{new} the size of the new one, and let $s = SCall(vn, fn, [p_1, \ldots, p_n])$ be the function call statement. We define the function $code_{\text{newfr}} :: tenv \times functableT \times (\mathbb{S} \times ty)\ list \times (\mathbb{S} \times ty)\ list \times \mathbb{N} \times \mathbb{N} \times stmt \mapsto instr\ list$. Observe that we omit some parameters for readability.

$$code_{\text{newfr}}(fs_{\text{cur}}, fs_{\text{new}}, s) = \left[\begin{array}{ll} (1) & code_{\text{lit}}(r1, fs_{\text{cur}}), \\ (2) & add(r2, r0, r_{\text{lframe}}), \\ (3) & add(r_{\text{lframe}}, r_{\text{lframe}}, r1), \\ (4) & sw(hd(fregs_{\text{ini}}), r_{\text{lframe}}, 4), \\ (5) & sw(r2, r_{\text{lframe}}, 8), \\ (6) & jal(dist_{\text{scall}}(te, ft, gst, lst, s)), \\ (7) & sw(r_{\text{jal}}, r_{\text{lframe}}, 0) \end{array}\right]$$

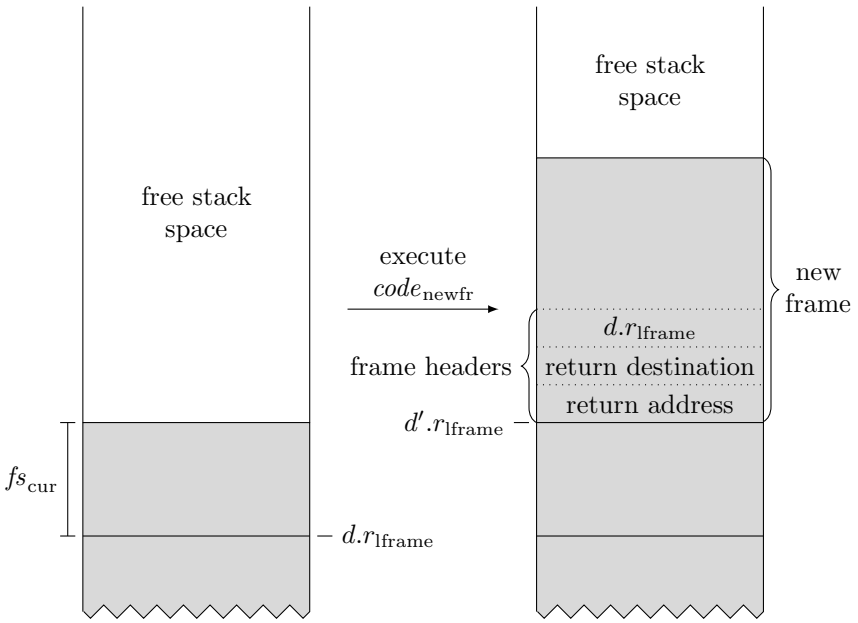


Figure 7.10: Construction of the New Stack Frame

In the first line of the code from $code_{\text{newfr}}$ we store the size of the current frame in $r1$ for later use. In the second line we save the current stack pointer in $r2$ and update in the third line the stack pointer to the base of the new stack frame. In the fourth and fifth line we store the return destination, i.e., the address where the result of the function has to be copied to, and the old stack pointer to the frame header of the new stack frame. In line six we jump to the new function. Finally, in line seven, i.e., in the delay slot of the jump, we store the return address, which has been stored by the jump-and-link instruction in register r_{jal}, in the new frame header.

Putting it All Together

Building on the definitions from above, we define the code generation for function calls. Let $f = \mathit{the}(\mathit{map\text{-}of}(\mathit{ft}, \mathit{fn}))$ be the function which is called. Furthermore, let $fs_{\text{cur}} = \lceil \mathit{asize}_{\text{st}}(\mathit{lst}, 12) \rceil_4$ and $fs_{\text{new}} = \lceil \mathit{asize}_{\text{st}}(\mathit{st}_{\text{fun}}(f), 12) \rceil_4$ be the size of the current and of the new stack frame, respectively.

We define

$$
\begin{aligned}
&\mathit{code}_{\text{stmt}}(\mathit{SCall}(\mathit{vn}, \mathit{fn}, \mathit{pl})) = \\
&\quad \left[\begin{array}{l}
\mathit{code}_{\text{expr}}(\mathit{false}, \mathit{hd}(\mathit{fregs}_{\text{ini}}), \mathit{tl}(\mathit{fregs}_{\text{ini}}), \mathit{Var}(\mathit{vn})), \\
\mathit{code}_{\text{passing}}(\mathit{te}, f, \mathit{gst}, \mathit{lst}, \mathit{tl}(\mathit{fregs}_{\text{ini}}), fs_{\text{cur}}, \mathit{pl}, \mathit{map}(\mathit{fst}, f.\mathit{params})), \\
\mathit{code}_{\text{newfr}}(\mathit{te}, \mathit{ft}, \mathit{gst}, \mathit{lst}, fs_{\text{cur}}, fs_{\text{new}}, \mathit{SCall}(\mathit{vn}, \mathit{fn}, \mathit{pl}))
\end{array} \right]
\end{aligned}
$$

7.4.8 Return

Compared to function calls, code generation for return statements $\mathit{Return}(\mathit{re})$ is quite simple. First we evaluate the return expression, then we copy the result to

the destination specified in the header of the current frame. Finally we restore the stack pointer with the base address of the previous frame and jump back to the return address, which was also stored in the frame header.

Let $t = the(type(te, gst, lst, re))$ be the type of the return expression and let $rf = elem?_{\mathrm{t}}(t)$ be a flag which specifies if we evaluate the return expression as right or as left expression. We define

$$code_{\mathrm{stmt}}(Return(re)) = \left[\begin{array}{ll} (1) & code_{\mathrm{expr}}(rf, hd(fregs_{\mathrm{ini}}), tl(fregs_{\mathrm{ini}}), re), \\ (2) & lw(r3, r_{\mathrm{lframe}}, 4), \\ (3) & code_{\mathrm{ass}}(\neg rf, asize_{\mathrm{t}}(t), r3, hd(fregs_{\mathrm{ini}}), 0), \\ (4) & lw(r3, r_{\mathrm{lframe}}, 0), \\ (5) & lw(r_{\mathrm{lframe}}, r_{\mathrm{lframe}}, 8), \\ (6) & jr(r3), \\ (7) & nop \end{array}\right]$$

In the first line the code evaluates the return expression. In the second line it loads the return destination from the frame header and in line three copy the result. In line four and five it loads the return address and the base address of the previous stack frame from the frame header and stores them in registers $r3$ and r_{lframe}, respectively. Finally, in line six the code jumps back behind the function call statement which initiated the execution of the current function.

7.4.9 Code Displacement of Statements

After having given the code generation for statements, we now give a concrete definition of $displ_{\mathrm{s}}$ as announced in Section 7.1.2. We illustrate the displacement exemplarily in Figure 7.11.

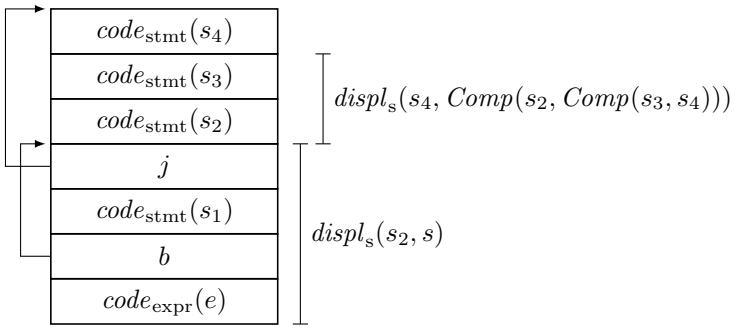
Definition 7.34 (Displacement of statements) Let te be a type name environment, ft a function table, and gst and lst the global and local symbol tables. Furthermore, let s and s' be statements. We define the code displacement of statement s' with respect to statement s by induction on s. This definition requires s' to be non-structural and to be a sub statement of s.

$$displ_{\mathrm{s}} :: tenv \times functableT \times (\mathbb{S} \times ty)\ list \times (\mathbb{S} \times ty)\ list \times stmt \times stmt \mapsto \mathbb{N}_{\perp}$$

For clarity of formulas, we omit most parameters in the following formulas.

$$displ_{\mathrm{s}}(s', Skip) = None$$

$$displ_{\mathrm{s}}(s', Comp(s_1, s_2)) = \begin{cases} \lfloor displ_{\mathrm{s}}(s', s_1) \rfloor & \text{if } s' \in sub_{\mathrm{s}}(s_1) \\ \lfloor 4 \cdot csize_{\mathrm{s}}(s_1) + displ_{\mathrm{s}}(s', s_2) \rfloor & \text{if } s' \in sub_{\mathrm{s}}(s_2) \\ None & \text{otherwise} \end{cases}$$



Example: $s = Ifte(e, s_1, Comp(s_2, Comp(s_3, s_4)))$

Figure 7.11: Code Displacement of Statements

For conditional statements $Ifte(e, s_1, s_2)$ let l_c be the number of instructions which have been generated for the evaluation of the condition e and the corresponding branch to either s_1 or s_2. Additionally, let l_j be the size of the code which has been inserted behind the code for s_1 to jump over the code for s_2 (cf. Figure 7.8).

$$displ_{\text{s}}(s', Ifte(e, s_1, s_2)) = \begin{cases} \lfloor 4 \cdot l_c + displ_{\text{s}}(s', s_1) \rfloor & \text{if } s' \in sub_{\text{s}}(s_1) \\ \left\lfloor \begin{array}{l} 4 \cdot (l_c + csize_{\text{s}}(s_1) + l_j) \\ + displ_{\text{s}}(s', s_2) \end{array} \right\rfloor & \text{if } s' \in sub_{\text{s}}(s_2) \\ None & \text{otherwise} \end{cases}$$

For a loop statement $Loop(e, lb)$ let l_c be the number of instructions which have been generated for the evaluation of the condition e and the corresponding conditional branch behind the code of the loop (cf. Figure 7.7).

$$displ_{\text{s}}(s', Loop(e, lb)) = \begin{cases} 4 \cdot l_c + displ_{\text{s}}(s', lb) & \text{if } s' \in sub_{\text{s}}(lb) \\ None & \text{otherwise} \end{cases}$$

For all other (non-structural) statements s we set

$$displ_{\text{s}}(s', s) = \begin{cases} \lfloor 0 \rfloor & \text{if } s' = s \\ None & \text{otherwise} \end{cases}$$

Observe that the code displacement is measured in bytes.

7.5 Code Generation for Complete C0 Programs

Having code generation functions for expressions and statements is an important part of the *C0* compiler specification but two things are still missing. We need a function which generates the initialization code of a program and we need functions which extend the code generation for single functions – more precisely for the body of single functions – to code generation of lists of function (i.e., function tables) and prepend the initialization code (cf. Figure 7.12).

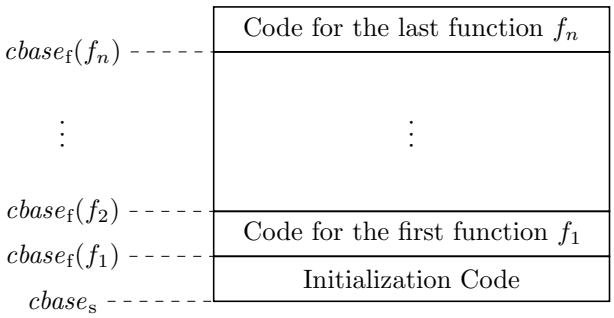


Figure 7.12: Overall Structure of the Generated Code

7.5.1 Init Code

The initialization code of a $C0$ program sets up the special registers which store the allocated base address of the global memory, the top most used heap address, and the base address of the current stack frame (cf. Table 7.2 on page 129). Let te be a type name environment, gst a symbol table for the global variables, and ft a function table. We introduce the function $code_{\mathrm{ini}} :: tenv \times (\mathbb{S} \times ty)\ list \times functableT \mapsto instr\ list$ which defines the code generation of the init code for user mode programs.

$$code_{\mathrm{ini}}(te, gst, ft) = \left[\begin{array}{l} code_{\mathrm{lit}}(r_{\mathrm{sbase}}, abase_{\mathrm{gm}}(te, ft, gst)), \\ code_{\mathrm{lit}}(r_{\mathrm{htop}}, abase_{\mathrm{heap}}), \\ code_{\mathrm{lit}}(r_{\mathrm{lframe}}, abase_{\mathrm{lm}}(te, ft, gst)) \end{array}\right]$$

Observe that $abase_{\mathrm{heap}}$ in the second line is a constant whereas $abase_{\mathrm{gm}}$ and $abase_{\mathrm{lm}}$ are computed using te, gst, and ft.

7.5.2 Code Generation for Programs

The code generation for complete $C0$ programs is done by appending code for all functions of the function table and the init code (cf. Figure 7.12).

Definition 7.35 (Code generation for lists of functions) The code generation for a list fl of functions is done by the function $code_{\mathrm{flist}} :: tenv \times functableT \times (\mathbb{S} \times ty)\ list \times funcT\ list$. This function is defined by induction on the list of functions.

$$code_{\mathrm{flist}}(te, ft, gst, [\,]) = [\,]$$
$$code_{\mathrm{flist}}(te, ft, gst, f\#t) = \left[\begin{array}{l} code_{\mathrm{stmt}}(te, ft, gst, st_{\mathrm{fun}}(f), f.body), \\ code_{\mathrm{flist}}(te, ft, gst, t) \end{array}\right]$$

Definition 7.36 (Code generation for programs) Let te be a type name environment, ft a function table, and gst a symbol table for the global variables. We introduce the function $code_{\mathrm{prog}} :: tenv \times functableT \times (\mathbb{S} \times ty)\ list \mapsto instr\ list$ which generates code for a complete $C0$ program.

$$code_{\mathrm{prog}}(te, ft, gst) = \left[\begin{array}{l} code_{\mathrm{ini}}(te, gst, ft), \\ code_{\mathrm{flist}}(te, ft, gst, map(snd, ft)) \end{array}\right]$$

We will later sometimes just write *code* for different versions of the code generation functions when it is obvious from the context which version is actually used. For example, we will write $code(s)$ or $code(e)$ instead of $code_{\text{stmt}}(s)$ or $code_{\text{expr}}(e)$, respectively.

7.6 Translatable Programs

The code generation from the previous sections does not work for arbitrary $C0$ programs. For example, if a program contains an expression which uses too many registers, or if the jump distance for a function call is too large, the generated code will not work.

In this section we introduce predicates which restrict expressions, statements, and programs in such a way

1. that the generated code fulfills the necessary requirements regarding the size of immediate operands and
2. that there are enough free registers to evaluate the expressions.

We call expressions, statements, and programs which have this property *translatable*.

Isabelle In Isabelle / HOL we have proved that the code which is generated for translatable expressions, statements, and programs indeed fulfills the requirements regarding the size of immediate operands and the number of registers. However, these proofs are quite lengthy and not very interesting; we omit them here. The proofs can be found in the theory `is_instr_lemmas.thy`.

We start by introducing a predicate which tests if there are enough free registers for a given expression.

Definition 7.37 (Enough free registers) Let te be a type name environment, gst and lst symbol tables for the global and (current) local variables, rf a flag which specifies if the expression has to be evaluated as right or left expression, and n the number of available free registers. We introduce the predicate $enough_{\text{regs}} :: tenv \times (\mathbb{S} \times ty)\ list \times (\mathbb{S} \times ty)\ list \times \mathbb{B} \times \mathbb{N} \times expr \mapsto \mathbb{B}$ which is defined by induction on the expression.

$$\frac{}{enough_{\text{regs}}(te, gst, lst, rf, n, Lit(l))}$$

$$\frac{}{enough_{\text{regs}}(te, gst, lst, rf, n, Var(v))}$$

$$\frac{0 < n \qquad enough_{\text{regs}}(te, gst, lst, false, n, e)}{enough_{\text{regs}}(te, gst, lst, rf, n, Str(e, cn))}$$

$$\frac{\begin{array}{c}2 \leq n \\ enough_{\text{regs}}(te, gst, lst, true, n, e_l) \qquad enough_{\text{regs}}(te, gst, lst, true, n-1, e_r)\end{array}}{enough_{\text{regs}}(te, gst, lst, rf, n, LazyBinOp(e_l, e_r))}$$

$$\frac{enough_{\text{regs}}(te, gst, lst, true, e)}{enough_{\text{regs}}(te, gst, lst, rf, n, UnOp(e))}$$

$$\frac{enough_{\text{regs}}(te, gst, lst, false, n, e)}{enough_{\text{regs}}(te, gst, lst, rf, n, AddrOf(e))}$$

$$\frac{enough_{\text{regs}}(te, gst, lst, true, n, e)}{enough_{\text{regs}}(te, gst, lst, rf, n, Deref(e))}$$

We define the predicate for array access and binary operators using a case distinction. We have one case if the left expression is bigger and another one if it is not.

$$\frac{size_{\text{e}}(e_a) < size_{\text{e}}(e_i) \qquad 2 \leq n \qquad enough_{\text{regs}}(te, gst, lst, true, n, e_i) \qquad enough_{\text{regs}}(te, gst, lst, false, n-1, e_a)}{enough_{\text{regs}}(te, gst, lst, rf, n, Arr(e_a, e_i)}$$

$$\frac{size_{\text{e}}(e_a) \geq size_{\text{e}}(e_i) \qquad 2 \leq n \qquad enough_{\text{regs}}(te, gst, lst, false, n, e_a) \qquad enough_{\text{regs}}(te, gst, lst, true, n-1, e_i)}{enough_{\text{regs}}(te, gst, lst, rf, n, Arr(e_a, e_i)}$$

$$\frac{size_{\text{e}}(e_l) < size_{\text{e}}(e_r) \qquad 2 \leq n \qquad enough_{\text{regs}}(te, gst, lst, true, n, e_r) \qquad enough_{\text{regs}}(te, gst, lst, true, n-1, e_l)}{enough_{\text{regs}}(te, gst, lst, rf, n, BinOp(e_l, e_r))}$$

$$\frac{size_{\text{e}}(e_l) \geq size_{\text{e}}(e_r) \qquad 2 \leq n \qquad enough_{\text{regs}}(te, gst, lst, true, n, e_l) \qquad enough_{\text{regs}}(te, gst, lst, true, n-1, e_r)}{enough_{\text{regs}}(te, gst, lst, rf, n, BinOp(e_l, e_r))}$$

We have a few additional requirements for translatable expressions.

Definition 7.38 (Translatable expressions) Let te be a type name environment, ft a function table, and gst and lst symbol tables for the global and local variables. We introduce the predicate $xltbl_{\text{expr}}$ which checks for translatable expressions. This predicate is mainly motivated by the restricted size of immediate constants in the generated code.

$$xltbl_{\text{expr}} :: tenv \times functableT \times (\mathbb{S} \times ty)\ list \times (\mathbb{S} \times ty)\ list \mapsto expr\ set$$

$$\frac{}{Lit(ALPrim(p)) \in xltbl_{\text{expr}}(te, ft, gst, lst)}$$

$$\frac{}{Var(v) \in xltbl_{\text{expr}}(te, ft, gst, lst)}$$

$$\frac{e_l \in xltbl_{\text{expr}}(te, ft, gst, lst) \qquad e_r \in xltbl_{\text{expr}}(te, ft, gst, lst)}{BinOp(e_l, e_r) \in xltbl_{\text{expr}}(te, ft, gst, lst)}$$

For lazy binary operators we require that the jump distance from the center code to the end of the expression fits into a 26 bit immediate constant.

$$\frac{e_l \in \mathit{xltbl}_{\mathrm{expr}}(te, ft, gst, lst) \quad e_r \in \mathit{xltbl}_{\mathrm{expr}}(te, ft, gst, lst) \quad 4 \cdot (1 + \mathit{csize}_{\mathrm{e}}(te, gst, lst, \mathit{true}, e_r) + 1) < 2^{25}}{\mathit{LazyBinOp}(e_l, e_r) \in \mathit{xltbl}_{\mathrm{expr}}(te, ft, gst, lst)}$$

$$\frac{e \in \mathit{xltbl}_{\mathrm{expr}}(te, ft, gst, lst)}{\mathit{UnOp}(e) \in \mathit{xltbl}_{\mathrm{expr}}(te, ft, gst, lst)}$$

For structures we require that the allocated size of the structure's type fits into a 16 bit immediate constant.

$$\frac{e \in \mathit{xltbl}_{\mathrm{expr}}(te, ft, gst, lst) \quad \mathit{asize}_{\mathrm{t}}(\mathit{the}(\mathit{type}(te, gst, lst, e))) < 2^{15}}{\mathit{Str}(e, cn) \in \mathit{xltbl}_{\mathrm{expr}}(te, ft, gst, lst)}$$

For arrays we have the same requirement for the size of an individual array element. Let $t = \mathit{the}(\mathit{type}(te, gst, lst, e_a)) = \mathit{Arr}_{\mathrm{T}}(n, t')$ be the type of the array.

$$\frac{e_a \in \mathit{xltbl}_{\mathrm{expr}}(te, ft, gst, lst) \quad e_i \in \mathit{xltbl}_{\mathrm{expr}}(te, ft, gst, lst) \quad \mathit{asize}_{\mathrm{t}}(t') < 2^{15}}{\mathit{Arr}(e_a, e_i) \in \mathit{xltbl}_{\mathrm{expr}}(te, ft, gst, lst)}$$

$$\frac{e \in \mathit{xltbl}_{\mathrm{expr}}(te, ft, gst, lst)}{\mathit{AddrOf}(e) \in \mathit{xltbl}_{\mathrm{expr}}(te, ft, gst, lst)}$$

$$\frac{e \in \mathit{xltbl}_{\mathrm{expr}}(te, ft, gst, lst)}{\mathit{Deref}(e) \in \mathit{xltbl}_{\mathrm{expr}}(te, ft, gst, lst)}$$

Definition 7.39 (Translatable statements) Let *te* be a type name environment, *ft* a function table, and *gst* and *lst* symbol tables for the global and local variables. We introduce the predicate $\mathit{xltbl}_{\mathrm{stmt}}$ which checks if a given statement is translatable. This predicate requires that we have sufficiently many registers for expression evaluation, that the expressions are translatable, and contains some additional requirements which ensure that we do not create immediate constants which are too large.

$$\mathit{xltbl}_{\mathrm{stmt}} :: \mathit{tenv} \times \mathit{functableT} \times (\mathbb{S} \times \mathit{ty})\ \mathit{list} \times (\mathbb{S} \times \mathit{ty})\ \mathit{list} \mapsto \mathit{stmt}\ \mathit{set}$$

$$\frac{}{\mathit{Skip} \in \mathit{xltbl}_{\mathrm{stmt}}(te, ft, gst, lst)}$$

$$\frac{s_1 \in \mathit{xltbl}_{\mathrm{stmt}}(te, ft, gst, lst) \quad s_2 \in \mathit{xltbl}_{\mathrm{stmt}}(te, ft, gst, lst)}{\mathit{Comp}(s_1, s_2) \in \mathit{xltbl}_{\mathrm{stmt}}(te, ft, gst, lst)}$$

For assignments we require that the left and right expressions are translatable and that we have enough register to do this. Additionally, we require that the size of the assigned expressions is less than 2^{15} bytes, s.t. it fits into a 16 bit immediate constant. Let $t = the(type(te, gst, lst, e_l))$ be the type of the left expression.

$$\frac{\begin{array}{c} e_l \in xltbl_{\text{expr}}(te, ft, gst, lst) \qquad e_r \in xltbl_{\text{expr}}(te, ft, gst, lst) \\ asize_{\text{t}}(t) < 2^{15} \qquad enough_{\text{regs}}(te, gst, lst, false, |fregs_{\text{ini}}| - 1, e_l) \\ enough_{\text{regs}}(te, gst, lst, elem?_{\text{t}}(t), |fregs_{\text{ini}}| - 2, e_r) \end{array}}{Ass(e_l, e_r) \in xltbl_{\text{stmt}}(te, ft, gst, lst)}$$

$$\frac{\begin{array}{c} e_l \in xltbl_{\text{expr}}(te, ft, gst, lst) \\ asize_{\text{t}}(t) < 2^{15} \qquad enough_{\text{regs}}(te, gst, lst, false, |fregs_{\text{ini}}| - 1, e_l) \end{array}}{Ass_{\text{AL}}(e_l, l_c) \in xltbl_{\text{stmt}}(te, ft, gst, lst)}$$

For memory allocation, let $t = the(map\text{-}of(te, tn))$ be the type for which we allocate the memory. We require that the allocated size of t is less than the maximum heap address.

$$\frac{\begin{array}{c} e \in xltbl_{\text{expr}}(te, ft, gst, lst) \qquad \lceil asize_{\text{t}}(t) \rceil_4 < abase_{\text{heap}} + asize^{\max}_{\text{heap}} \\ enough_{\text{regs}}(te, gst, lst, false, |fregs_{\text{ini}}| - 1, e) \end{array}}{PAlloc(e, tn) \in xltbl_{\text{stmt}}(te, ft, gst, lst)}$$

For function calls, we require that all parameters are translatable expressions, that we have enough registers to evaluate them, and that the allocated size of each parameter is not too big.

$$\frac{\begin{array}{c} \forall e \in pl : e \in xltbl_{\text{expr}}(te, ft, gst, lst) \\ \forall e \in pl : enough_{\text{regs}}(te, gst, lst, elem?_{\text{t}}(the(type(te, gst, lst, e))), |fregs_{\text{ini}}| - 2, e) \\ \forall e \in pl : asize_{\text{t}}(the(type(te, gst, lst, e))) < 2^{15} \end{array}}{SCall(vn, fn, pl) \in xltbl_{\text{stmt}}(te, ft, gst, lst)}$$

$$\frac{\begin{array}{c} re \in xltbl_{\text{expr}}(te, ft, gst, lst) \\ enough_{\text{regs}}(te, gst, lst, elem?_{\text{t}}(the(type(te, gst, lst, re))), |fregs_{\text{ini}}| - 1, re) \\ asize_{\text{t}}(the(type(te, gst, lst, re))) < 2^{15} \end{array}}{Return(re) \in xltbl_{\text{stmt}}(te, ft, gst, lst)}$$

The most interesting requirement for conditional statements and while loops is that the sub statements are not too big, s.t. the jump distances fit into the immediate constants (cf. Figures 7.7 on page 152 and 7.8 on page 152).

$$\frac{\begin{array}{c} c \in xltbl_{\text{expr}}(te, ft, gst, lst) \qquad enough_{\text{regs}}(te, gst, lst, true, |fregs_{\text{ini}}| - 1, c) \\ s_1 \in xltbl_{\text{stmt}}(te, ft, gst, lst) \qquad s_2 \in xltbl_{\text{stmt}}(te, ft, gst, lst) \\ 4 \cdot (csize_{\text{s}}(te, gst, lst, s_1) + 3) < 2^{15} \qquad 4 \cdot (csize_{\text{s}}(te, gst, lst, s_2) + 1) < 2^{25} \end{array}}{Ifte(c, s_1, s_2) \in xltbl_{\text{stmt}}(te, ft, gst, lst)}$$

$$\frac{\begin{array}{c} c \in xltbl_{\text{expr}}(te, ft, gst, lst) \qquad enough_{\text{regs}}(te, gst, lst, true, |fregs_{\text{ini}}| - 1, c) \\ lb \in xltbl_{\text{stmt}}(te, ft, gst, lst) \qquad 4 \cdot (csize_{\text{s}}(te, gst, lst, lb) + 3) < 2^{15} \\ -4 \cdot (csize_{\text{e}}(te, gst, lst, true, c) + csize_{\text{s}}(te, gst, lst, lb) + 3) < 2^{25} \end{array}}{Loop(c, lb) \in xltbl_{\text{stmt}}(te, ft, gst, lst)}$$

Definition 7.40 (Translatable functions) Let te be a type name environment, ft a function table, gst the symbol tables for the global variables, and f a $C0$ function. We introduce the predicate $xltbl_{\text{func}}$ which checks whether f is a translatable function, i.e., whether the body of f is a translatable statement and whether the size of the function symbol table is small enough.

$$xltbl_{\text{func}} :: tenv \times functableT \times (\mathbb{S} \times ty)\ list \mapsto funcT\ set$$

$$\frac{f.body \in xltbl_{\text{stmt}}(te, ft, gst, st_{\text{fun}}(f)) \qquad 2 \cdot \lceil asize_{\text{st}}(st_{\text{fun}}(f), 12) \rceil_4 < 2^{32}}{f \in xltbl_{\text{func}}(te, ft, gst)}$$

Remark When copying function call parameters, their offset with respect to the current value of register r_{lframe} is the sum of the old frame's size and (at most) the size of the new frame. The factor two in the above predicate ensures that this offset not exceeds $2^{32} - 1$. Of course, this definition could be strengthened. For two functions f and f' where f contains a function call to f', it would suffice to require that the size of the frame for f and the offset of the last parameter in the frame of f' is smaller than 2^{32}. However, the above definition simplifies proofs and will probably not be an actual restriction.

Definition 7.41 (Translatable program) Let te be a type name environment, ft a function table, and gst the symbol tables for the global variables. We introduce the predicate $xltbl_{\text{prog}}$ which checks whether these form a translatable $C0$ program or not.

$$xltbl_{\text{prog}} :: (tenv \times functableT \times (\mathbb{S} \times ty)\ list)\ \ set$$

$$\frac{\begin{array}{c} progbase + 4 \cdot csize_{\text{prog}}(te, gst, ft) < 2^{25} \\ \forall (fn, f) \in ft : f \in xltbl_{\text{func}}(te, ft, gst) \qquad 2 \cdot \lceil asize_{\text{st}}(gst, 0) \rceil_4 < 2^{32} \\ abase_{\text{lm}}(te, ft, gst) < 2^{32} \qquad abase_{\text{heap}} + asize_{\text{heap}}^{\text{max}} < 2^{32} \end{array}}{(te, ft, gst) \in xltbl_{\text{prog}}}$$

7.7 Execution of the Compiler Specification

To solve the bootstrap problem [GH98], i.e., to obtain a trustworthy binary of the $C0$ compiler, it is not sufficient to verify the code generation algorithm or the $C0$ implementation of the compiler [Pet07]. We additionally need to translate the compiler implementation to VAMP assembly and machine language.

Of course, we could simply translate the compiler implementation using some unverified existing compiler. But the probability that this translation induces errors in the generated compiler binary is quite high. In Verisoft, we follow two different ways how to reduce this probability.

First, B. Finkbeiner's group applies translation validation techniques to show that a binary compiler which has been generated by some untrusted bootstrap compiler is a correct translation of the compiler implementation from [Pet07].

Second, we have used Isabelle's built-in ML code generation feature [Ber03, BN02]. The (verified) compiling specification from Chapter 7 is executable using this ML code generation. In particular, it can be used to compile the compiler's $C0$ implementation from [Pet07] to obtain a trustworthy compiler binary. This approach is presented in more details in [Tzi07].

Of course, both approaches extend the trusted code base: the first by the translation validation tool, the second by Isabelle's code generation module. However, we can drastically reduce the likelihood of errors by applying all three methods and comparing the resulting binaries.[2] The probability that all methods produce the same error is negligible.

[2] For the first method we would have to proceed in two steps. First, we would compile the compiler implementation with an unverified existing compiler. The resulting binary would not be comparable to the binary generated by the other two methods because the unverified compiler is likely to use a different code generation algorithm. After that, we would use this binary (which uses *our* code generation algorithm) to compile the implementation again.

Chapter 8

Simulation Theorem

Contents

We present in this chapter the simulation theorem which states the correctness of the *compiling specification* of the $C0$ compiler. Additionally, E. Petrova has proved the correctness of the compiler *implementation* [Pet07] using the $C0$ verification environment from [Sch06]. Her results are briefly sketched in Chapter 11. Recently, she has formally combined her result with the simulation theorem from this chapter.

Observe that we first present the main theorem and the structure of its induction proof. This gives the reader some idea of the overall structure of the compiler correctness proof. The correctness of the individual induction cases will be proved later in Chapters 9 and 10.

We start this chapter in Section 8.1 with a short overview. We continue in Section 8.2 with the formal definition of the simulation relation (this includes the definition of reachability and allocation functions) and in Section 8.3 with some requirements about available memory in the target machine. We conclude the chapter in Section 8.4 with the formalization of the main correctness theorem for the compiling specification and a sketch of its correctness proof.

8.1 Overview

We prove the correctness of the $C0$ compiler using a *small-step simulation theorem* between the execution of a $C0$ program $p = (te, ft, gst)$ on the abstract $C0$ machine from Chapter 4 and the execution of the compiled code $code_{\text{prog}}(te, ft, gst)$ of the program on the VAMP assembly machine from Section 6.2.

The important words in the previous sentence are ‘small-step’ and ‘simulation theorem’.

- 'Simulation theorem' means that we define a simulation relation *consis* between states of the $C0$ small-step semantics and states of the VAMP assembly machine. Then, we prove that the execution of the compiled program preserves this simulation relation.

- 'Small-step' means that we assure the simulation relation after *every single step* of the $C0$ small-step semantics. In contrast, a big-step simulation theorem only states simulation between the final states of program execution.

We call a state c of the $C0$ machine and a state d of the VAMP assembly machine *consistent* if they fulfill the simulation relation.

We illustrate the correctness statement in Figure 8.2 on page 183. Let $c^0 = init_{\text{conf}}(ft, gst)$ be the initial state of a given $C0$ program p and d^0 a (arbitrary) state of the VAMP assembly machine where the compiled code $code_{\text{prog}}(p)$ is stored at address *progbase*. The correctness statement assures that for every state $c^i = \delta^i_{\text{C0}}(te, ft, c^0)$ in the computation of a $C0$ program there exists a number of assembly steps $s(i)$ such that $\delta^{s(i)}_{\text{asm}}(d^0)$ and c^i are consistent.

8.2 Simulation Relation

In this section we define the simulation relation of the compiler correctness proof. We start in Section 8.2.1 with some thoughts about why we believe that the simulation relation is correct, sufficiently expressible, and consistent. In Section 8.2.2 we define the set of reachable g-variables. Then, in Section 8.2.3 we introduce the concept of allocation functions which specify how g-variables are located in the assembly machine. We define in Section 8.2.4 when a $C0$ value and an assembly value are *matching*, i.e., when they are considered equivalent. Finally, in Section 8.2.5 we formally define the compiler simulation relation.

8.2.1 Correctness of the Simulation Relation

The compiler correctness proof in the following sections ensures that the compiled code preserves some simulation relation. But how can we be sure that the simulation relation itself is correct, i.e., that it correctly reflects the intuitive meaning of *correctness*? How can we be sure that a property of a certain $C0$ program also holds for the compiled assembly code?

We could for instance use $consis(c, d) = true$ as definition for the simulation relation. This would make the compiler correctness proof quite simple but would not guarantee any meaningful property of the compiled code.

There are three main reasons why we have confidence in the simulation relation which we specify in this thesis.

1. The definition of the simulation relation *consis* is quite simple. This makes it easy to manually inspect it and look for errors. Of course, such a manual inspection can never clear out all skepticism; after all, we are doing formal verification here, i.e., we try to rely on manual checks as little as possible.

2. The Verisoft approach, i.e., *pervasive verification*, can give us ultimate confidence in the correctness of the simulation relation; at least for concrete properties of individual programs.

   Let P_{C0} be a property on the $C0$ layer and P_{asm} be the corresponding property on the assembly layer. We first prove that P_{C0} holds for a given $C0$ program. Then we use the compiler simulation relation and the simulation theorem to show that we can conclude the validity of P_{asm} for the compiled program. Then we know that the property P_{asm} holds for the concrete program. Observe that this does not completely guarantee the compiler correctness or the correctness of the simulation relation for *arbitrary* programs. Nevertheless, it shows that the compiler and the simulation relation are good enough for *one* concrete property of *one* concrete program.

   Exemplary, this has been done for the page fault handler used in Verisoft's academic sub project [ASS08].

3. If we would only prove the induction step correct, i.e., that from simulation consistency in step i we can conclude simulation consistency in step $i+1$, we still could have contradictions in the simulation relation. For example, for the simulation relation $sim(c,d) = \mathit{false}$ it would be quite simple to conclude $sim(c^i, d^i) \longrightarrow sim(c^{i+1}, d^{i+1})$. Instead, we additionally prove the induction start correct which would be impossible for a contradictory simulation relation because from a contradiction in $sim(c,d)$ we could prove $sim(c,d) = \mathit{false}$ and would then have to show $sim(c^0, d^0)$ which could not be done. Thus, we can be sure that the simulation relation is consistent.

8.2.2 Reachability

In the following sections we require only for *reachable* g-variables that their value from the $C0$ configuration is properly stored in the assembly memory. Having this condition only for reachable variables makes the compiler simulation proof a little bit more complicated because we have to show that g-variables which are reachable in the new configuration

- have either been freshly allocated and are properly stored in the assembly machine by the correctness of the allocation code, or
- have also been reachable in the previous $C0$ configuration which allows to apply the induction hypothesis.

On the other hand, having this condition only for reachable variables will simplify the integration of a garbage collector which can remove unreachable g-variables from the heap memory. The current version of the $C0$ compiler does not yet include a garbage collector. However, the implementation of a copying garbage collector for the $C0$ compiler has already been verified using the $C0$ verification environment [Sch06]. Only the integration of the code into the code generation of the $C0$ compiler and the necessary adaptations of the compiler simulation proof have to be done.

In this section we will formally define the set of reachable g-variables.

Definition 8.1 (Pointer g-variable) Let te be a type name environment, sc a symbol configuration, and g a g-variable. We say that g is a pointer g-variable if it is of type Ptr_{T} or $Null_{\mathrm{T}}$. Formally,

$$is\_gvar_{\mathrm{p}}(sc, g) = is\_Ptr_{\mathrm{T}}(ty_{\mathrm{g}}(sc, g)) \vee ty_{\mathrm{g}}(sc, g) = Null_{\mathrm{T}}$$

Definition 8.2 (Reachable nameless g-variables) Let mc a memory configuration of the $C0$ machine. We introduce the set $reachable_{\mathrm{g}}^{\mathrm{nameless}}$ which defines the set of reachable g-variables in the heap memory.

We do this definition by induction: we start with the g-variables in the global and local memories and – doing pointer chasing – we add step by step all g-variables which are reachable from these starting points to the set.

$$reachable_{\mathrm{g}}^{\mathrm{nameless}} :: memconf \mapsto gvar\ set$$

For the induction start let x be an initialized pointer g-variable in the global or in one of the local memories. If the value of x points to some nameless g-variable g we add g to the set of reachable nameless g-variables.

$$\frac{\begin{array}{c} x \in gvars\surd(sc(mc)) \\ named_{\mathrm{g}}(x) \qquad is\_gvar_{\mathrm{p}}(sc(mc), x) \qquad initialized_{\mathrm{g}}(mc, x) \\ g \in gvars\surd(sc(mc)) \qquad \neg named_{\mathrm{g}}(g) \qquad value_{\mathrm{g}}(mc, x) = Ptr(g) \end{array}}{g \in reachable_{\mathrm{g}}^{\mathrm{nameless}}(mc)} \tag{8.1}$$

Similarly to the induction start, we call a nameless g-variable g reachable if a second nameless g-variable h – which is already in the set of reachable g-variables – points to g.

$$\frac{\begin{array}{c} h \in reachable_{\mathrm{g}}^{\mathrm{nameless}}(mc) \qquad initialized_{\mathrm{g}}(mc, h) \qquad is\_gvar_{\mathrm{p}}(sc(mc), h) \\ g \in gvars\surd(sc(mc)) \qquad \neg named_{\mathrm{g}}(g) \qquad value_{\mathrm{g}}(mc, h) = Ptr(g) \end{array}}{g \in reachable_{\mathrm{g}}^{\mathrm{nameless}}(mc)} \tag{8.2}$$

Finally, sub g-variables of reachable nameless g-variables are also reachable.

$$\frac{h \in reachable_{\mathrm{g}}^{\mathrm{nameless}}(mc) \qquad g \in sub_{\mathrm{g}}(h) \qquad g \in gvars\surd(sc(mc))}{g \in reachable_{\mathrm{g}}^{\mathrm{nameless}}(mc)} \tag{8.3}$$

Definition 8.3 (Reachable g-variables) Let mc a memory configuration of the $C0$ machine. We define the set $reachable_{\mathrm{g}} :: memconf \mapsto gvar\ set$ which contains all reachable g-variables of memory configuration mc.

$$\frac{g \in reachable_{\mathrm{g}}^{\mathrm{nameless}}(mc)}{g \in reachable_{\mathrm{g}}(mc)}$$

$$\frac{g \in gvars\surd(sc(mc)) \qquad named_{\mathrm{g}}(g)}{g \in reachable_{\mathrm{g}}(mc)}$$

Definition 8.4 (Reachable pointers) Now, we extend the previous definition from g-variables to pointers by defining

$$memconf \mapsto (gvar \cup \{\bot\})\ set$$

$$\frac{}{\bot \in reachable_{\text{ptr}}(mc)}$$

$$\frac{g \in reachable_{\text{g}}(mc)}{g \in reachable_{\text{ptr}}(mc)}$$

Definition 8.5 (Reachable pointer memory cells) We define a predicate for memory cells which contain reachable pointers.

$$reachable_{\text{mcell}} :: memconf \times mcell_{\text{C0}} \mapsto \mathbb{B}$$

For pointer memory cells we require that the contained pointer is reachable.

$$\frac{p \in reachable_{\text{ptr}}(mc)}{reachable_{\text{mcell}}(mc, Ptr(p))}$$

For other memory cells there are no requirements.

$$\frac{c \in \{Int(i), Nat(u), Char(c), Bool(b)\}}{reachable_{\text{mcell}}(mc, c)}$$

The following theorem shows that the values and addresses which we obtain by expression evaluation are reachable.[1] For *AddrOf* and *Deref* operators we switch between left value and right value during evaluation. Thus, we have to state the theorem simultaneously for both left and right values.

Theorem 8.1 (Evaluation gives reachable g-variables)
Let c be a valid C0 configuration and e an expression which can be evaluated both as a left and as a right expression. Then, given that e represents a memory object, the left value of e is a reachable g-variable and, given that e is initialized, all pointer memory cells in the right value of e are reachable.

$$\begin{aligned} & c \in conf\surd(te, ft) \\ & \wedge\ leval(te, c.mem, e) = \lfloor g \rfloor \wedge reval(te, c.mem, e) = \lfloor v \rfloor \\ \Longrightarrow\quad & (mobj(e) \longrightarrow g \in reachable_g(c.mem)) \\ & \wedge\ (initialized(e) \longrightarrow \forall i < size_t(type(e)) : reachable_{mcell}(c.mem, v(i))) \end{aligned}$$

Proof We prove this lemma by induction on e.

For variable accesses the first goal follows directly from the definition of reachable g-variables because g is a named g-variable. The second goal holds because reachability is transitive: if g is reachable then the content of g in the memory is also reachable (via g).

[1] This is the expected meaning of the word *reachability*; thus, we should not be surprised that the theorem holds.

Literals do not represent memory objects. Thus, we only have to prove the second goals which follows from the fact that literals may only contain null pointers.

For array and structure access, the first goal follows from the fact that sub g-variables of reachable g-variables are also reachable (cf. Definition 8.2). The second goal holds because the content of the sub g-variables is contained in the content of g.

Application of operators is trivial: they give results which are not memory objects and which contain no pointers at all.

The address-of operator $AddrOf(e)$ also does not give a memory object; thus, we only have to prove the second goal which follows from the fact that the value of $AddrOf(e)$ is just a pointer to the address of e which – by the induction hypothesis – is a reachable g-variable.

For dereferencing $Deref(e)$ the first goal is true because – by the induction hypothesis – the value of e, which is the left value of $Deref(e)$, is a reachable pointer. For the second goal we use the same arguments as for the second goal for variable access. q.e.d.

Theorem 8.2 (No new reachable g-variables by memory updates) *Let c be a valid configuration and let m' be the C0 memory configuration after a successful update of the valid g-variable g_l with some value v. Furthermore, assume that all pointers in v point to reachable g-variables. Then, all heap g-variables which are reachable in the new memory configuration m' have already been reachable in the old memory configuration $c.mem$.*

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge g \in reachable_g^{nameless}(m') \\
& \wedge\, g_l \in gvars\surd(sc(c.mem)) \wedge memupd(c.mem, g_l, v) = \lfloor m' \rfloor \\
& \wedge\, tmatch_{ass}(ty_g(sc(c.mem), g_l), t) \wedge ty\surd(te, sc(c.mem), v, t, 0) \\
& \wedge\, \forall i < size_t(t) : reachable_{mcell}(c.mem, v(i)) \\
& \Longrightarrow g \in reachable_g^{nameless}(c.mem)
\end{aligned}
$$

Proof We prove this theorem by induction according to the definition of $reachable_{\mathrm{g}}^{\mathrm{nameless}}$. There are three ways how g can be reachable in m'.

Case 1: Induction start (Equation (8.1))
In this case, g is reachable in m' via some named g-variable x (i.e., the value of x in m' is $Ptr(g)$). We consider two cases.

1. $x \in sub_{\mathrm{g}}(g_{\mathrm{l}})$: In this case we use the last precondition of the theorem. This precondition ensures that every pointer memory cell inside v points to a g-variable which was reachable in the old configuration. During the memory update x is updated with one of these reachable pointers (observe that we need the preconditions $ty\surd$ and $tmatch_{\mathrm{ass}}$ to know that the embedded pointers in both types are at corresponding positions). So, the new value of x – which is $Ptr(g)$ – has also been reachable in the old configuration and thus, g itself has been reachable.
2. $x \notin sub_{\mathrm{g}}(g_{\mathrm{l}})$: In this case the value of x has not been changed by the memory update and it must have pointed to g also in the old configuration. Thus, g was reachable.

Case 2: Induction step: indirectly reachable (Equation (8.2))
In this case, g is reachable in m' via some *reachable* nameless g-variable x. From the induction hypothesis we know that x has also been reachable in the old configuration. The rest of this proof case follows the proof of case 1.

Case 3: Induction step: sub g-variable (Equation (8.3))
This case is easy. From Equation (8.3) we know that g is a sub g-variable of some other reachable nameless g-variable x. From the induction hypothesis we conclude that x was also reachable in the old configuration. However, the fact of being a sub g-variable is a structural property of the g-variables and does not depend on any memory configuration. Thus, g was also a sub g-variable of x in the old configuration and reachable by Equation (8.3). q.e.d.

8.2.3 Allocation Function

The simulation relation *consis* which we define in Section 8.2.5 is parameterized with the allocation function $alloc :: gvar \mapsto (\mathbb{N} \times \mathbb{N})$. The allocation function maps a g-variable g to a pair $alloc(g) = (b, s)$ of an *allocated base address* b for g and the *allocated size* s of g's type.

Later we will instantiate *alloc* for global and local variables with values computed by $abase_{\mathrm{g}}$ and $asize_{\mathrm{t}}$. For heap variables we have to keep track of their actual position in the heap. For the current version of the $C0$ compiler this address stays constant as we have not yet integrated a garbage collector. After the integration of a garbage collector, the allocated address of heap variables may change with every execution of an allocation statement.

8.2.4 Matching Values

In this section, we introduce three predicates which test for given values of the $C0$ and the assembly semantics whether the $C0$ value is properly represented by the assembly value. This gives us some kind of equality measure between the two kinds of values.

We start with the definition of a predicate for non-pointer values.

Definition 8.6 (Matching values) Let $v_{\mathrm{C0}} :: mcell_{\mathrm{C0}}$ be a $C0$ value and $v_{\mathrm{asm}} :: \mathbb{Z}$ an assemble value. We introduce the predicate *vmatch* which checks whether v_{C0} and v_{asm} match, i.e., whether they contain equivalent values. We define *vmatch* by case distinction on the $C0$ value. Observe that pointer values never fulfill the predicate.

$$vmatch :: mcell_{\mathrm{C0}} \times \mathbb{Z} \mapsto \mathbb{B}$$

$$\frac{v_{\mathrm{asm}} = i}{vmatch(Int(i), v_{\mathrm{asm}})}$$

$$\frac{v_{\mathrm{asm}} = i}{vmatch(Char(i), v_{\mathrm{asm}})}$$

For unsigned values we have to interpret the assembly value as a natural number.

$$\frac{i2n(v_{\text{asm}}) = n}{vmatch(Nat(n), v_{\text{asm}})}$$

We represent *false* by 0 and *true* by 1.

$$\frac{v_{\text{C0}} = Bool(v_{\text{asm}} \neq 0) \qquad v_{\text{asm}} \in \{0, 1\}}{vmatch(Bool(b), v_{\text{asm}})}$$

For pointers the equality test slightly differs from the above definition.

Definition 8.7 (Matching pointer values) Let *alloc* be an allocation function, $p :: gvar \cup \{\bot\}$ a $C0$ pointer, and $v_{\text{asm}} :: \mathbb{Z}$ a value of the assembly machine. We introduce the predicate $vmatch_{\text{ptr}}$ which checks whether p and v_{asm} correspond to each other. Formally, we define

$$vmatch_{\text{ptr}} :: (gvar \mapsto (\mathbb{N} \times \mathbb{N})) \times gvar \cup \{\bot\} \times \mathbb{Z} \mapsto \mathbb{B}$$

For null pointers the definition is trivial.

$$\frac{v_{\text{asm}} = 0}{vmatch_{\text{ptr}}(alloc, \bot, v_{\text{asm}})}$$

For pointers to g-variables g we require that the assembly value – interpreted as a natural number – equals the allocated base address of g as defined by the allocation function.

$$\frac{i2n(v_{\text{asm}}) = fst(alloc(g))}{vmatch_{\text{ptr}}(alloc, g, v_{\text{asm}})}$$

Finally, we combine the definitions of matching values for pointers and normal values in the following definition.

Definition 8.8 (Matching right values) Let *alloc* be an allocation function, $v_{\text{C0}} :: mcell_{\text{C0}}$ a $C0$ value, and $v_{\text{asm}} :: \mathbb{Z}$ a value of the assembly machine. We introduce the predicate $vmatch_{\text{rval}} :: (gvar \mapsto (\mathbb{N} \times \mathbb{N})) \times mcell_{\text{C0}} \times \mathbb{Z} \mapsto \mathbb{B}$ which checks whether v_{C0} and v_{asm} correspond to each other.

$$vmatch_{\text{rval}}(alloc, v_{\text{C0}}, v_{\text{asm}}) = \begin{cases} vmatch_{\text{ptr}}(alloc, p, v_{\text{asm}}) & \text{if } v_{\text{C0}} = Ptr(p) \\ vmatch(v_{\text{C0}}, v_{\text{asm}}) & \text{otherwise} \end{cases}$$

Finally, we extend the definition of matching values from single values to complete g-variables.

Definition 8.9 (Content consistent g-variables) Let *sc* a symbol configuration and *alloc* an allocation function. Further, let $v_g :: \mathbb{N} \mapsto mcell_{\text{C0}}$ be the content of a given g-variable g and b the allocated base address of g in assembly configuration d. Then we define the predicate $vmatch_{\text{gvar}} :: symbolconf \times (\mathbb{N} \mapsto mcell_{\text{C0}}) \times conf_{\text{asm}} \times (gvar \mapsto (\mathbb{N} \times \mathbb{N})) \times gvar \times \mathbb{N} \mapsto \mathbb{B}$ which checks for *all* elementary sub g-variables of g whether their value at the proper offset in v_g

matches the value stored in d. If all sub g-variables match we say that v_g is *content consistent* with g in assembly configuration d. Formally, we define

$$\begin{aligned} &\mathit{vmatch}_{\mathrm{gvar}}(sc, v_g, d, \mathit{alloc}, g, b) = \\ &\qquad \forall g' \in \mathit{sub}_{\mathrm{g}}(g) : \big(\mathit{elem?}_{\mathrm{t}}(\mathit{ty}_{\mathrm{g}}(sc, g')) \wedge g' \in \mathit{gvars}\surd(sc)\big) \\ &\qquad\quad \Longrightarrow \mathit{vmatch}_{\mathrm{rval}}(\mathit{alloc}, v_g(\mathit{ba}_{\mathrm{g}}(sc, g') - \mathit{ba}_{\mathrm{g}}(sc, g)), \\ &\qquad\qquad\qquad \mathit{cell2int}(d.\mathit{mm}(b + \mathit{displ}_{\mathrm{rel}}(sc, g, g') \div 4))) \end{aligned}$$

8.2.5 Definition of the Simulation Relation

The simulation relation $\mathit{consis} :: \mathit{tenv} \times \mathit{functableT} \times \mathit{conf}_{\mathrm{C0}} \times (\mathit{gvar} \mapsto (\mathbb{N} \times \mathbb{N})) \times \mathit{conf}_{\mathrm{asm}} \mapsto \mathbb{B}$ defines whether a $C0$ configuration and an assembly configuration are equivalent with respect to a given allocation function.

Basically, the simulation relation is supposed to state that all (reachable) g-variables of the $C0$ configuration are properly stored at their allocated address in the assembly configuration. This part of *consis* will be used to translate and transfer correctness statements about $C0$ programs to the assembly layer (cf. Section 8.2.1).

Additionally, we need many more properties in the simulation relation to make the compiler correctness statement inductive, i.e., to be able to infer consistency between $\delta_{\mathrm{C0}}^{i+1}(te, ft, c^0)$ and $\delta_{\mathrm{asm}}^{s(i+1)}(d^0)$ given that $\delta_{\mathrm{C0}}^{i}(te, ft, c^0)$ and $\delta_{\mathrm{asm}}^{s(i)}(d^0)$ are consistent. Among other things, these additional properties argue

- that the compiled code has not been changed,
- that the program counters of the assembly machine point to the code of the current statement of the $C0$ machine, and
- that all stack frames are properly reprented.

We will define the simulation relation in several steps. But before we start with the formal definition of consistency we introduce some auxiliary functions.

Definition 8.10 (Reading frame headers) We define three functions with signature $\mathit{tenv} \times \mathit{functableT} \times \mathit{symbolconf} \times (\mathbb{N} \mapsto \mathit{mcell}_{\mathrm{asm}}) \times \mathbb{N} \mapsto \mathbb{N}$ which read the return address, the return destination, and the previous stack pointer from the frame headers in the stack. Let te be a type name environment, ft a function table, sc a symbol configuration, and mm the memory of a VAMP assembly configuration. Then, the following functions read frame header information from the i-th stack frame

$$\begin{aligned} \mathit{fh}_{\mathrm{ra}}(te, ft, sc, mm, i) &= \mathit{i2n}(\mathit{cell2int}(\mathit{mm}((\mathit{abase}_{\mathrm{lm}}(te, ft, sc, i)) \div 4))) \\ \mathit{fh}_{\mathrm{rd}}(te, ft, sc, mm, i) &= \mathit{i2n}(\mathit{cell2int}(\mathit{mm}((\mathit{abase}_{\mathrm{lm}}(te, ft, sc, i) + 4) \div 4))) \\ \mathit{fh}_{\mathrm{psp}}(te, ft, sc, mm, i) &= \mathit{i2n}(\mathit{cell2int}(\mathit{mm}((\mathit{abase}_{\mathrm{lm}}(te, ft, sc, i) + 8) \div 4))) \end{aligned}$$

We give the top-level definition of the simulation relation *consis* before we formally define its parts in the following sections.

Definition 8.11 (Consistency) Let te be a type name environment, ft a function table, c a $C0$ configuration, d a configuration of the VAMP assembly

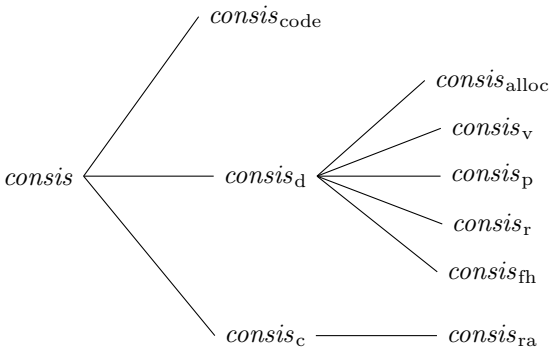


Figure 8.1: Overall Structure of the Compiler Simulation Relation

machine, and *alloc* an allocation function. We say that c and d are consistent if they fulfill the predicate *consis* which requires code consistency, control consistency, and data consistency.

$$consis :: tenv \times functableT \times conf_{\text{C0}} \times (gvar \mapsto (\mathbb{N} \times \mathbb{N})) \times conf_{\text{asm}} \mapsto \mathbb{B}$$

$$\frac{consis_{\text{code}}(te, ft, c, d) \qquad consis_{\text{c}}(te, ft, c, d) \qquad consis_{\text{d}}(te, ft, c, alloc, d)}{consis(te, ft, c, alloc, d)}$$

For a better overview during the following definitions we illustrate the structure of the consistency relation in Figure 8.1.

Code Consistency

Code consistency requires that the compiled code is stored at address *progbase* in the assembly configuration.

Definition 8.12 (Code consistency) Let te be a type name environment, ft a function table, c a $C0$ configuration, and d a configuration of the VAMP assembly machine. We define the predicate $consis_{\text{code}}$ which checks whether c and d are code consistent.

$$consis_{\text{code}} :: tenv \times functableT \times conf_{\text{C0}} \times conf_{\text{asm}} \mapsto \mathbb{B}$$

$$\frac{\begin{array}{c} read_{\text{instr}}(d.mm, progbase, csize_{\text{prog}}(te, gst(c.mem), ft)) \\ = code_{\text{prog}}(te, ft, gst(c.mem)) \end{array}}{consis_{\text{code}}(te, ft, c, d)}$$

Control Consistency

Control consistency requires that the program counters of the assembly machine point to the start address of the code which has been generated for the head of the current program rest and that return addresses of all stack frames are correct. The latter requirement is formally defined via the predicate $consis_{\text{ra}}$.

Definition 8.13 (Return address consistency) Let te be a type name environment, ft a function table, c a $C0$ configuration, and d a configuration of the VAMP assembly machine. We define the predicate $consis_{\mathrm{ra}}$. It checks whether for all $i < |c.mem.lm| - 1$ there exists a function call statement s in the function table such that

1. the successor statement of s follows the i-th return statement in the program rest and
2. the return address in stack frame $|c.mem.lm| - (i + 1)$ points directly behind the code of s, i.e., s has been the function call which generated that stack frame.

Formally, we define

$$
\begin{aligned}
&consis_{\mathrm{ra}}(te, ft, c, d) = \\
&\quad \forall i < |c.mem.lm| - 1 : \exists s : \\
&\qquad is\_SCall(s) \\
&\qquad \wedge\, fos(s, ft) \neq None \\
&\qquad \wedge\, succ_{\mathrm{ithret}}(s2l_{\mathrm{ns}}(c.prog), i) = \lfloor x \rfloor \implies succ(s, ft) = \lfloor x \rfloor \\
&\qquad \wedge\, fh_{\mathrm{ra}}(te, ft, sc(c.mem), d.mm, |c.mem.lm| - (i + 1)) \\
&\qquad\quad = cbase_{\mathrm{s}}(te, gst(c.mem), ft, s) + 4 \cdot csize_{\mathrm{s}}(te, gst(c.mem), ft, s)
\end{aligned}
$$

Consistency of the return addresses together with the correctness of the program counters gives control consistency.

Definition 8.14 (Control consistency) Let ft be a function table, te a type name environment, c a $C0$ configuration, and d a configuration of the VAMP assembly machine. If the program rest is not empty, we say that c and d are control consistent if $consis_{\mathrm{ra}}(te, ft, c, d)$ holds and the program counters of d point to the code of the head of the current program rest of c. However, if the program rest is empty we have no requirements for control consistency.

$$consis_{\mathrm{c}} :: tenv \times functableT \times conf_{\mathrm{C0}} \times conf_{\mathrm{asm}} \mapsto \mathbb{B}$$

$$\frac{d.dpc = the(cbase_{\mathrm{s}}(te, gst(c.mem), ft, hd(s2l_{\mathrm{ns}}(c.prog)))) \quad d.pcp = d.dpc + 4 \qquad consis_{\mathrm{ra}}(te, ft, c, d) \qquad s2l_{\mathrm{ns}}(c.prog) \neq [\,]}{consis_{\mathrm{c}}(te, ft, c, d)}$$

$$\frac{s2l_{\mathrm{ns}}(c.prog) = [\,]}{consis_{\mathrm{c}}(te, ft, c, d)}$$

Data Consistency

Data consistency states that reachable g-variables are correctly stored in the assembly machine, that some auxiliary information about stack and heap are stored correctly, and that the allocation function conforms with the allocated base address of global and local g-variables as computed by $abase_{\mathrm{g}}$. We define data consistency using several auxiliary definitions.

Allocation Consistency. First, we require that the allocation function returns meaningful values. We require that the second component of the allocation function returns for all g-variables their allocated size. For named g-variables we require additionally that the first component returns the same values as $abase_{\mathrm{g}}$. We cannot argue about the absolute position of nameless g-variables in the assembly memory because they might be moved by a garbage collector. Instead, we require that sub g-variables are properly placed relatively to their root g-variables and that nameless g-variables with different roots do not overlap in the assembly memory.

Definition 8.15 (Named allocation consistency) We define the predicate $consis_{\mathrm{alloc}}^{\mathrm{named}} :: tenv \times functableT \times conf_{\mathrm{C0}} \times (gvar \mapsto (\mathbb{N} \times \mathbb{N})) \mapsto \mathbb{B}$ which tests whether a given allocation function contains the correct data for named g-variables.

$$
\begin{aligned}
&consis_{\mathrm{alloc}}^{\mathrm{named}}(te, ft, c, alloc) = \\
&\quad \forall g \in gvars\surd(sc(c.mem)) : \\
&\qquad named_{\mathrm{g}}(g) \implies \begin{array}{l} fst(alloc(g)) = abase_{\mathrm{g}}(te, ft, sc(c.mem), g) \\ \wedge snd(alloc(g)) = asize_{\mathrm{t}}(ty_{\mathrm{g}}(sc(c.mem), g)) \end{array}
\end{aligned}
$$

Definition 8.16 (Heap allocation consistency) The introduce the predicate $consis_{\mathrm{alloc}}^{\mathrm{heap}} :: tenv \times functableT \times conf_{\mathrm{C0}} \times (gvar \mapsto (\mathbb{N} \times \mathbb{N})) \mapsto \mathbb{B}$ which tests whether a given allocation function returns reasonable data for reachable nameless g-variables. We have three conditions for heap allocation consistency.

1. We require that the end of the top most local frame $|c.mem.lm| - 1$ (which coincides with the base address of the non-existing stack frame $|c.mem.lm|$, cf. Definition 7.16 on page 134) is below the heap base.
2. We require for all valid named g-variables g that the second component of $alloc(g)$ equals the allocated size of the g-variable g.
3. We require for all reachable nameless g-variables g' (i) that the allocated base address $fst(alloc(g'))$ of g' corresponds to the allocated base address of its root g-variable $root_{\mathrm{g}}(g')$ plus the relative displacement of g', (ii) that the root g-variable of g' is properly aligned, (iii) that no g-variable h with a different root g-variable does overlap with g', and (iv) that the allocated base address of g' is above the heap base.

Formally, this is defined as

$$
\begin{aligned}
&consis_{\mathrm{alloc}}^{\mathrm{heap}}(te, ft, c, alloc) = \\
&\quad abase_{\mathrm{lm}}(te, ft, sc(c.mem), |c.mem.lm|) \leq abase_{\mathrm{heap}} \\
&\wedge \forall g \in gvars\surd(sc(c.mem)) : \\
&\qquad \neg named_{\mathrm{g}}(g) \implies snd(alloc(g)) = asize_{\mathrm{t}}(ty_{\mathrm{g}}(sc(c.mem), g)) \\
&\wedge \forall g' \in reachable_{\mathrm{g}}^{\mathrm{nameless}}(c.mem) : \\
&\qquad fst(alloc(g')) = fst(alloc(root_{\mathrm{g}}(g'))) + displ_{\mathrm{g}}(sc(c.mem), g') \\
&\quad \wedge algn(ty_{\mathrm{g}}(sc(c.mem), root_{\mathrm{g}}(g'))) \,|\, fst(alloc(root_{\mathrm{g}}(g'))) \\
&\quad \wedge abase_{\mathrm{heap}} \leq fst(alloc(g')) \\
&\quad \wedge \forall h \in reachable_{\mathrm{g}}^{\mathrm{nameless}}(c.mem) : \\
&\qquad root_{\mathrm{g}}(g') \neq root_{\mathrm{g}}(h) \implies alloc(g') \asymp alloc(h)
\end{aligned}
$$

Definition 8.17 (Allocation consistency) We combine allocation consistency for nameless and named g-variables.

$$consis_{\text{alloc}} :: tenv \times functableT \times conf_{\text{C0}} \times (gvar \mapsto (\mathbb{N} \times \mathbb{N})) \mapsto \mathbb{B}$$

$$\frac{consis^{\text{named}}_{\text{alloc}}(te, ft, c, alloc) \qquad consis^{\text{heap}}_{\text{alloc}}(te, ft, c, alloc)}{consis_{\text{alloc}}(te, ft, c, alloc)}$$

Value Consistency. Value consistency requires that the values of reachable non-pointer g-variables are properly represented in the assembly configuration. We define value consistency in two steps. We define first a predicate for a *single* g-variable. Then we extend this predicate for *all* g-variables.

Definition 8.18 (Value consistent g-variable) Let mc a memory configuration, $alloc$ an allocation function, d a VAMP assembly configuration, and g a g-variable. We define the predicate $consis^{\text{g}}_{\text{v}} :: memconf \times (gvar \mapsto (\mathbb{N} \times \mathbb{N})) \times conf_{\text{asm}} \times gvar \mapsto \mathbb{B}$ which tests whether the value of g is properly stored in the memory of assembly configuration d.

Observe that we define the predicate for *arbitrary* g-variables although it is only meaningful for reachable non-pointer g-variables of elementary type. For other g-variables the predicate is *true* by default.

Formally, we define

$$consis^{\text{g}}_{\text{v}}(mc, alloc, d, g) = \begin{cases} vmatch(value_{\text{g}}(mc, g)(0), & \\ cell2int(d.mm(fst(alloc(g)) \div 4))) & \text{if } g \in reachable_{\text{g}}(mc) \\ & \quad \wedge elem?_{\text{t}}(ty_{\text{g}}(sc(mc), g)) \\ & \quad \wedge \neg \left(is\_gvar_{\text{p}}(sc(mc), g)\right) \\ & \quad \wedge initialized_{\text{g}}(mc, g) \\ true & \text{otherwise} \end{cases}$$

Definition 8.19 (Value consistency) We say that a $C0$ configuration c is value consistent with assembly configuration d in the context of alloction function $alloc$ if all g-variables are value consistent. Formally, we define

$$consis_{\text{v}}(c, alloc, d) = \forall g.consis^{\text{g}}_{\text{v}}(c.mem, alloc, d, g).$$

Pointer Consistency. Pointer consistency states that the values of reachable pointer g-variables are properly represented in the assembly configuration. In principle, its definition is similar to value consistency except that we have a different notion of *equality* between values in the assembly and the $C0$ machine.

As in the definition of value consistency, we first define when a single g-variable is called pointer consistent.

Definition 8.20 (Pointer consistent g-variable) Let mc a memory configuration, $alloc$ an allocation function, d a VAMP assembly configuration, and g a g-variable. Furthermore, let the value of g be a pointer value, i.e., $value_{\text{g}}(mc, g)(0) = Ptr(p)$. We define the predicate $consis^{\text{g}}_{\text{p}} :: memconf \times (gvar \mapsto (\mathbb{N} \times \mathbb{N})) \times conf_{\text{asm}} \times gvar \mapsto \mathbb{B}$ which tests whether the pointer value

of g is properly stored in the memory of assembly configuration d. Similar to $consis^{\mathrm{g}}_{\mathrm{v}}$, the predicate only tests the value of reachable and initialized g-variables of pointer type; for other g-variables it is *true* by default.

$$consis^{\mathrm{g}}_{\mathrm{p}}(mc, alloc, d, g) = \begin{cases} vmatch_{\mathrm{ptr}}(p, & \\ cell2int(d.mm(fst(alloc(g)) \div 4))) & \text{if } g \in reachable_{\mathrm{g}}(mc) \\ & \quad \wedge \left(is\_gvar_{\mathrm{p}}(sc(mc), g)\right) \\ & \quad \wedge initialized_{\mathrm{g}}(mc, g) \\ true & \text{otherwise} \end{cases}$$

Definition 8.21 (Pointer consistency) We say that $C0$ configuration c is pointer consistent with assembly configuration d in the context of allocation function *alloc* if all g-variables are pointer consistent. Formally, we define

$$consis_{\mathrm{p}}(c, alloc, d) = \forall g.consis^{\mathrm{g}}_{\mathrm{p}}(c.mem, alloc, d, g).$$

Register Consistency. Register consistency argues about the content of some special registers (cf. Table 7.2 on page 129).

Definition 8.22 (Register consistency) We introduce a predicate $consis_{\mathrm{r}}$ which checks whether $C0$ configuration c and assembly configuration d are register consistent. For this we require

1. that the contents of registers r_{sbase}, r_{lframe}, and r_{htop} have values according to their intended meaning from Table 7.2,
2. that no reachable nameless g-variables are allocated above the top most heap address from r_{htop}, and
3. that the value of r_{htop} is properly aligned to the maximum alignment.

Formally, this is defined as

$$consis_{\mathrm{r}} :: tenv \times functableT \times conf_{\mathrm{C0}} \times (gvar \mapsto (\mathbb{N} \times \mathbb{N})) \times conf_{\mathrm{asm}} \mapsto \mathbb{B}$$

$$\frac{\begin{array}{c} i2n(d.gpr!r_{\mathrm{sbase}}) = abase_{\mathrm{gm}}(te, ft, gst(c.mem)) \\ i2n(d.gpr!r_{\mathrm{lframe}}) = abase_{\mathrm{lm}}(te, ft, sc(c.mem), |c.mem.lm| - 1) \\ i2n(d.gpr!r_{\mathrm{htop}}) = abase_{\mathrm{heap}} + asize_{\mathrm{heap}}(hst(c.mem)) \\ \forall g \in reachable^{\mathrm{nameless}}_{\mathrm{g}}(c.mem) : fst(alloc(g)) + snd(alloc(g)) \leq i2n(d.gpr!r_{\mathrm{htop}}) \\ 4 \mid i2n(d.gpr!r_{\mathrm{htop}}) \end{array}}{consis_{\mathrm{r}}(te, ft, c, alloc, d)}$$

Frame Header Consistency. Frame header consistency requires the values in the frame headers of the local stack frames to be consistent with their intended meaning (cf. Table 7.1 on page 129). Observe that we do not argue about the return addresses here; they are already part of control consistency.

Definition 8.23 (Frame header consistency) We introduce the predicate $consis_{\mathrm{fh}}$ which requires the previous stack pointers and the return destinations in all stack frames – except for stack frame 0 – to be consistent with their intended

meaning. This means that the previous stack pointer of frame i contains the allocated base address of frame $i-1$ and that the return destination of frame i contains the allocated base address of the i-th return destination of the $C0$ configuration.

Formally, we define

$$\begin{aligned} &\mathit{consis}_{\mathrm{fh}}(te, ft, c, \mathit{alloc}, d) = \forall 0 < i < |c.\mathit{mem}.\mathit{lm}| : \\ &\qquad \mathit{fh}_{\mathrm{psp}}(te, ft, \mathit{sc}(c.\mathit{mem}), d.\mathit{mm}, i) = \mathit{abase}_{\mathrm{lm}}(te, ft, \mathit{sc}(c.\mathit{mem}), i-1) \\ &\qquad \wedge \mathit{fh}_{\mathrm{rd}}(te, ft, \mathit{sc}(c.\mathit{mem}), d.\mathit{mm}, i) = \mathit{fst}(\mathit{alloc}(\mathit{snd}(c.\mathit{mem}.\mathit{lm}!i))) \end{aligned}$$

Data Consistency. We combine the above definitions to data cosistency.

Definition 8.24 (Data consistency) Let te be a type name environment, ft a function table, c a $C0$ configuration, *alloc* an allocation function, and d a configuration of the VAMP assembly machine. We define the predicate $\mathit{consis}_{\mathrm{d}}(te, ft, c, \mathit{alloc}, d)$ which checks whether c and d are data consistent via the allocation function *alloc*.

$$\mathit{consis}_{\mathrm{d}} :: \mathit{tenv} \times \mathit{functableT} \times \mathit{conf}_{\mathrm{C0}} \times (\mathit{gvar} \mapsto (\mathbb{N} \times \mathbb{N})) \times \mathit{conf}_{\mathrm{asm}} \mapsto \mathbb{B}$$

$$\frac{\begin{array}{ccc} & \mathit{consis}_{\mathrm{alloc}}(te, ft, c, \mathit{alloc}) & \mathit{consis}_{\mathrm{v}}(c, \mathit{alloc}, d) \\ \mathit{consis}_{\mathrm{p}}(c, \mathit{alloc}, d) & \mathit{consis}_{\mathrm{r}}(te, ft, c, \mathit{alloc}, d) & \mathit{consis}_{\mathrm{fh}}(te, c, \mathit{alloc}, d) \end{array}}{\mathit{consis}_{\mathrm{d}}(te, ft, c, \mathit{alloc}, d)}$$

8.3 Resource Restrictions

Besides the requirement that the compiled code fits into memory (cf. Section 7.6), an important proof goal of the compiler correctness proof will be that during execution the compiled program does not use more memory than is available on the target machine.

There are basically two ways in which a program can use too much memory. First, if the program allocates to much data on the heap, it produces a *heap overflow*. Second, too many nested function calls may lead to a *stack overflow*.

The definition of memory allocation in the $C0$ semantics (cf. Section 4.4.3 on page 63) allows to discharge the proof goals regarding heap overflow at the $C0$ layer if we instantiate the $\mathit{avail}_{\mathrm{heap}}$ parameter of the $C0$ transition function correctly. For the current version of the $C0$ compiler, i.e., without garbage collection, we instantiate

$$\mathit{avail}_{\mathrm{heap}}(mc, t) = \left(\mathit{asize}_{\mathrm{heap}}(\mathit{hst}(mc)) + \lceil \mathit{asize}_{\mathrm{t}}(t) \rceil_4 < \mathit{asize}_{\mathrm{heap}}^{\max}\right).$$

This instantiation ensures that memory allocation has in every situation the same result – a null pointer or a pointer to the newly allocated memory – both for the original $C0$ program and for the compiled code (cf. the correctness proof for memory allocation in Section 10.7).

The $C0$ compiler does not annotate the function call code with a stack watch nor does the $C0$ semantics check for stack size when executing function calls. To allow the $C0$ program to react on a stack overflow would require major changes in the $C0$ language; e.g., adding support for exception handling or similar features to $C0$.

Because of the missing stack watch, we cannot guarantee equivalent execution of function calls in case of a stack overflow. However, we can formulate a predicate which checks at the $C0$ level whether we have enough stack or not. This predicate allows to lift argumentation about the required stack size from the assembly to the $C0$ layer.

Definition 8.25 (Enough stack available) Let te be a type name environment, ft a function table, and c a $C0$ configuration. We define the predicate $avail_{\text{stack}}$ which tests whether the top most stack frame ends below the heap base, i.e., whether we have a stack overflow or not.

$$avail_{\text{stack}} :: tenv \times functableT \times conf_{\text{C0}} \mapsto \mathbb{B}$$

$$\frac{abase_{\text{lm}}(te, ft, sc(c.mem), |c.mem.lm|) \leq abase_{\text{heap}}}{avail_{\text{stack}}(te, ft, c)}$$

We combine the predicate $avail_{\text{stack}}$ with a statement about a sufficient amount of heap memory.

Definition 8.26 (Sufficient memory available) Let te be a type name environment, ft a function table, and c a $C0$ configuration. Furthermore, let $addr_{\text{max}}$ denote the maximum address in the program's address space. We define the predicate $avail_{\text{mem}}$ which combines the previously defined predicate $avail_{\text{stack}}$ with requirements regarding heap consumption.

$$avail_{\text{mem}} :: \mathbb{N} \times tenv \times functableT \times conf_{\text{C0}} \mapsto \mathbb{B}$$

$$\frac{avail_{\text{stack}}(te, ft, c) \qquad abase_{\text{heap}} + asize_{\text{heap}}(hst(c.mem)) < addr_{\text{max}}}{avail_{\text{mem}}(addr_{\text{max}}, te, ft, c)}$$

8.3.1 Lifting Resource Restrictions to More Abstract Layers

Above, we have defined predicates which allow to ensure at the $C0$ small-step layer that a compiled program does not violate resource restrictions of the target machine. However, formally proving at the small-step layer that these predicates hold for all steps of the program is quite costly because a lot of work from the program's functional verification in the $C0$ Hoare logic would have to be repeated. It seems worthwhile to lift the requirements to the Hoare logic layer and to discharge them as part of the program's functional correctness proof.

Without a garbage collector, tracking heap usage in the $C0$ verification environment is treated in [Sch06] via a ghost variable which counts the number of free heap memory cells (similar to the work in [BPS05]). However, in the presence of a garbage collector, this simple method only gives an upper bound of heap memory usage because it does not keep track of deallocation of unreachable memory objects (neither manually nor via garbage collection). A garbage collector may automatically deallocate all memory objects which are not reachable anymore.

Because of the big-step nature of the $C0$ verification environment only the current stack frame is directly accessible. To keep track of the maximum

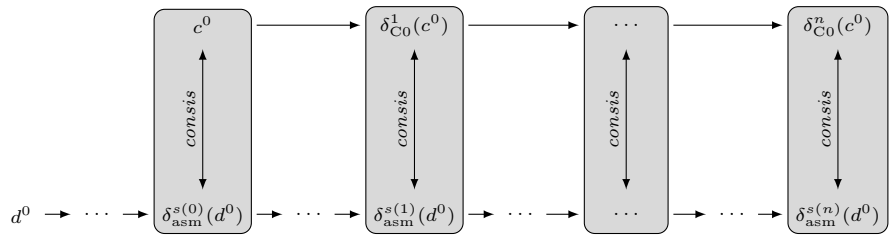


Figure 8.2: Small-Step Compiler Simulation Theorem

recursion depth and of all root pointers for a reachability analysis it would be necessary to have (at least partly) access to all stack frames. This requires to add additional ghost state. To examine the best way of formulating such ghost state and to prove the necessary meta theorems remains as future work.

8.4 Simulation Theorem

In this section, we present the top-level correctness theorem of the $C0$ compiler specification and sketch the structure of its proof which is done by induction on the steps of the $C0$ machine. For each step we do the proof by case distinction on the next statement in the program rest. The correctness proofs for the different statements are given later in Chapter 10.

As already stated in Section 8.1 we prove the correctness of the $C0$ compiler by a small-step induction proof over the execution steps of the $C0$ program (cf. Figure 8.2). This means that we prove for *every* step of a $C0$ program that the assembly machine executing the compiled code will reach an equivalent state; of course this theorem has several additional requirements both to the $C0$ program and to the initial state of the assembly machine.

Theorem 8.3 (Compiler correctness)
Let te be a type name environment, ft a function table, and gst the symbol table for the global variables. Furthermore, let $p = (te, ft, gst)$ *be the corresponding* $C0$ *program. Let d be the initial state of the VAMP assembly machine.*

Then for all steps i of the $C0$ *machine executing program p there exists a step number t of the assembly machine such that the* $C0$ *machine after i steps is consistent with the assembly machine after t steps. However, this is only true if the following requirements about p and d are fulfilled.*

- *The* $C0$ *program p is a translatable program with a non-empty function table which starts with the main function.*
- *The type name environment te is valid, ft is a valid function table, and gst is a valid symbol table.*
- *The* $C0$ *computation does not produce a 'None' configuration in step i.*
- *The constants* $addr_{max}$*, which denotes the maximum address in the program's address space,* $abase_{heap}$*, and* $asize^{max}_{heap}$ *are meaningful.*
- *The assembly configuration d is valid and its program counters point to the start of the compiled program.*

- *The global memory is zero initialized.*
- *For all steps of the C0 machine up to step i we had not have a stack overflow.*
- *The C0 program p does not contain inline assembly.*[2]

Formally, this theorem is stated as follows

$$
\begin{aligned}
&(te, ft, gst) \in xltbl_{prog} \wedge ft \neq [\,] \wedge fst(hd(ft)) = \text{"}main\text{"} \\
&\wedge\ te \in valid_{tenv} \wedge gst \in valid_{st}(te) \wedge \exists c' : nthstep^{i}_{C0}(te, ft, gst) = \lfloor c' \rfloor \\
&\wedge\ conf_{asm}\surd(d) \wedge d.dpc = progbase \wedge d.pcp = d.dpc + 4 \\
&\wedge\ addr_{max} \leq 2^{32} \wedge abase_{heap} + asize^{max}_{heap} < addr_{max} \wedge 0 < asize^{max}_{heap} \\
&\wedge\ consis_{code}(te, ft, the(init_{conf}(ft, gst)), d) \\
&\wedge\ \forall j < \lceil asize_{st}(gst, 0) \rceil_4 : cell2int(d.mm((abase_{gm}(te, ft, gst) + j) \div 4)) = 0 \\
&\wedge\ \forall i' \leq i : avail_{stack}(te, ft, the(nthstep^{i'}_{C0}(te, ft, gst))) \\
&\wedge\ \forall i' < i : \neg is\_Asm(hd(s2l((the(nthstep^{i'}_{C0}(te, ft, gst))).prog))) \\
&\Longrightarrow \exists t, alloc, d' : \quad d \xrightarrow[(progbase, progbase+4\cdot csize_{prog}(te,gst,ft)),(progbase, addr_{max})]{t, d'.dpc} d' \\
&\qquad\qquad \wedge consis(te, ft, c', alloc, d') \\
&\qquad\qquad \wedge d'.spr = d.spr
\end{aligned}
$$

Proof We prove this theorem by induction on the step number i of the $C0$ machine.

For the induction start we have to show that the initialization code $code_{\text{ini}}$ from Section 7.5.1 correctly sets up an assembly configuration which is consistent with the initial $C0$ configuration. We omit details of this (simple) proof here.

For the induction step we have to conclude from step i to $i + 1$. We know that there exists an intermediate assembly configuration d'' such that $c'' = the(nthstep^{i}_{\text{C0}}(te, ft, gst))$ and d'' are consistent with respect to some allocation function $alloc''$. We distinguish two cases.

Case 1: $s2l_{\text{ns}}(c''.prog) = [\,]$
In this case, the program has terminated (or only contains *Skip* statements) and the $C0$ transition function has reached a fix point (i.e., $c' = c''$ modulo *Skip* statements in the program rest). Thus, it is easy to prove the conclusion of the theorem by just instantiating the existence quantifier with $t = 0$, $d' = d''$, and $alloc = alloc''$.

Case 2: $s2l_{\text{ns}}(c''.prog) = h\#t$
We prove this case by a case distinction on h. The proofs for the respective cases are given in Chapter 10. q.e.d.

[2] The compiler correctness proof has been extended by A. Tsyban with support for inline assembly statements. However, this extension is not part of this thesis.

Chapter 9

Correct Compilation of Expressions

Contents

In this section we sketch the proof that the code generation algorithm for expressions from Section 7.3 produces correct code. This means that – given that some preconditions hold – the generated code computes a result in the destination register which is equivalent to the result computed by the evaluation functions *leval* and *reval* in the *C*0 machine.

Observe that we not present details of this proof. We present the overall structure of the proof – which is done by induction on the structure of the expressions – but we do not look deeper into the various cases. A thorough discussion of the correctness proof for several types of expressions can be found in the theses [Pen07] (multiplication, division, and modulo), [Pen06] (literals, bitwise operators, and comparison), and [Kre06] (addition and subtraction). These theses were supervised by the author. The remaining expressions have been verified as part of this thesis.

We start in Section 9.1 with some notation and continue in Section 9.2 with a more formal definition of the correctness statement for expression evaluation. Finally, we sketch the proof of correct compilation of expressions in Section 9.3.

9.1 Execution of Assembly Code for Expressions

Because the verification of expression evaluation was started and finished before the correctness proof for statements, the formulations of 'execution of assembly code' slightly differs between both proofs. The execution predicate for assembly code from Section 6.4.1 on page 114 has been improved after finishing the expression proofs. Nevertheless, in order to keep the expression proofs as stable as possible we have not adapted the execution predicate there but instead verified that we can use the expression code execution predicate inside the statement correctness proofs.

Definition 9.1 (Execution of expression code) Let d and d' be two assembly configurations, il a list of assembly instructions, $range_{\mathrm{c}}$ and $range_{\mathrm{a}}$ the code and address range, and t a step number. We define the predicate $exec_{\mathrm{asm}}$ which states – for the context of the expression correctness proofs – that the list of assembly instructions il is being executed from d to d'.

$$exec_{\mathrm{asm}} :: (conf_{\mathrm{asm}} \times instr\ list \times (\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}) \times \mathbb{N} \times conf_{\mathrm{asm}})\ set$$

$$\frac{\begin{array}{c} read_{\mathrm{instr}}(d.mm, d.dpc, |il|) = il \qquad d.pcp = d.dpc + 4 \\ d'.dpc = d.dpc + 4 \cdot |il| \qquad d'.pcp = d'.dpc + 4 \qquad d' = \delta^{t}_{\mathrm{asm}} d \\ \forall t' < t : interruptfree(\delta^{t'}_{\mathrm{asm}} d, range_{\mathrm{c}}, range_{\mathrm{a}}) \wedge advancing(currinstr(\delta^{t'}_{\mathrm{asm}}(d))) \\ \forall t' \leq t : unchngd_{\mathrm{mem}}(d.mm, \delta^{t'}_{\mathrm{asm}} d.mm, fst(range_{\mathrm{c}}) \div 4, snd(range_{\mathrm{c}}) \div 4) \end{array}}{(d, il, range_{\mathrm{c}}, range_{\mathrm{a}}, t, d') \in exec_{\mathrm{asm}}}$$

9.2 Correctness Statement for Expressions

In this section, we define the correctness statement for the assembly code which handles expression compilation. To give the reader a better intuition we start top-down by first defining the correctness predicate using uninterpreted predicates for pre and post conditions. Only then we give concrete definitions for them.

Definition 9.2 (Correct compilation of expressions) Let ft be a function table, te a type name environment, c a $C0$ configuration, $addr_{\max}$ the maximum address in the address space of the program, e an expression, and rf the flag which specifies whether we evaluate the expression as right or left value.

Furthermore, let $code_{\mathrm{e}}$ be the code which is generated for expression e, destination register r_d, and the list $fregs$ of free registers. That is, $code_{\mathrm{e}} = code_{\mathrm{expr}}(te, gst(c.mem), lst_{\mathrm{top}}(c.mem), rf, r_d, fregs, e)$.

We define the predicate $code_{\mathrm{expr}}\surd$ which checks whether e has been correctly compiled. Correct compilation means here that for all assembly configurations, allocation functions, destination registers, and set of free registers the precondition $expr_{\mathrm{pre}}$ implies that there exists a step number t and a new configuration such that the code for the expression is being executed in t steps and the postcondition $expr_{\mathrm{pos}}$ holds.

Formally, we define $code_{\mathrm{expr}}\surd(te, ft, c, range_{\mathrm{c}}, addr_{\max}, e, rf)$ to be true iff the following implication is true for all assembly configurations d, allocation functions $alloc$, destination registers r_d, and lists of free registers $fregs$.

$$\begin{array}{l} expr_{\mathrm{pre}}\,(d, te, ft, c, alloc, e, code_{\mathrm{e}}, rf, r_d, fregs, range_{\mathrm{c}}, addr_{\max}) \\ \quad \Longrightarrow \exists t, d' : (d, code_{\mathrm{e}}, range_{\mathrm{c}}, (progbase, a), t, d') \in exec_{\mathrm{asm}} \\ \qquad\qquad \wedge expr_{\mathrm{pos}}\,(d, d', te, c, alloc, e, code_{\mathrm{e}}, range_{\mathrm{c}}, addr_{\max}, t, rf, r_d, fregs) \end{array}$$

In the following we give a formal definition of the precondition and postcondition which are used above. Observe that the definition of the precondition contains a lot of technical details. Most of these details can later – when we use the expression correctness in the correctness proofs for statements – be directly derived from the predicates for valid configurations and the compiler consistency relation.

To guide the reader to the important parts of the conditions we define them in three steps. We start with an informal introduction of the most important conditions, followed by an informal list of more technical requirements, and finally we present the formal definition.

9.2.1 Precondition for Correct Execution of Expressions

We define the predicate $expr_{\text{pre}}$ which holds if all preconditions for the correct execution of code for a given expression hold. The main requirements of the predicate are (i) data consistency between c and d (this is only required if the expression is not a literal), (ii) that c and d are valid configurations, (iii) that e is a valid and translatable expression, (iv) that the evaluation of e does return some value, (v) that the expression is a memory object if $rf = false$, and (vi) that the expression is of elementary type and initialized if $rf = true$.

For literals we cannot assume data consistency because we use the code generation for literals in intermediate steps of the execution of assignments for aggregate literals (cf. Section 7.4.3 for the code generation and Section 10.6 for the correctness proof). Initialization is mainly needed for the correctness of division and modulo expressions where we need to know that the divisor is not zero.

Additionally, we have the following technical requirements: that there are enough free registers, that the global and the top most local symbol tables are translatable, that there is sufficient memory, i.e., that all memory objects are allocated below 2^{32}, that the maximum address $addr_{\max}$ is a valid 32-bit number, that program counters and code range fulfill some basic requirements, that the code for expression e is located at the address referenced by the program counters, and that the destination register and the list of free registers are ok.

Definition 9.3 (Expression code: preconditions) Let $gst = gst(c.mem)$ and $lst = lst_{\text{top}}(c.mem)$ be the global and local symbol tables. Then, the predicate for correct expression evaluation is formally defined as

$$\frac{\begin{array}{c}\neg is\_lit(e) \longrightarrow consis_{\text{d}}(te, ft, c, alloc, d) \qquad conf_{\text{asm}}\surd(d) \\ c \in conf\surd(te, ft) \qquad e \in valid_{\text{expr}}(te, gst, lst) \qquad e \in xltbl_{\text{expr}}(te, ft, gst, lst) \\ rf \longrightarrow reval(te, c.mem, e) \neq None \qquad \neg rf \longrightarrow leval(te, c.mem, e) \neq None \\ enough_{\text{regs}}(te, gst, lst, rf, |fregs|, e) \qquad 2 \cdot \lceil asize_{\text{st}}(gst, 0) \rceil_4 < 2^{32} \\ 2 \cdot \lceil asize_{\text{st}}(lst, 12) \rceil_4 < 2^{32} \qquad avail_{\text{mem}}(addr_{\max}, te, ft, c) \\ addr_{\max} \leq 2^{32} \qquad \neg rf \longrightarrow mobj(te, gst, lst, e) \\ rf \longrightarrow elem?_{\text{t}}(the(type(te, gst, lst, e))) \qquad rf \longrightarrow initialized(te, c.mem, e) \\ 4 \mid d.dpc \qquad d.pcp = d.dpc + 4 \qquad d.dpc + 4 \cdot (|code_{\text{e}}| + 1) < 2^{32} \\ fst(range_{\text{c}}) \leq d.dpc \qquad d.dpc + 4 \cdot |code_{\text{e}}| \leq snd(range_{\text{c}}) \\ snd(range_{\text{c}}) + 4 < 2^{32} \qquad read_{\text{instr}}(d.mm, d.dpc, |code_{\text{e}}|) = code_{\text{e}} \\ r_d \in fregs_{\text{ini}} \qquad distinct(fregs) \wedge \forall r \in fregs : fregs \in fregs_{\text{ini}} \qquad r_d \notin fregs \end{array}}{expr_{\text{pre}}(d, te, ft, c, alloc, e, code_{\text{e}}, rf, r_d, fregs, range_{\text{c}}, addr_{\max}))}$$

9.2.2 Postcondition for Correct Execution of Expressions

The definition of the postcondition for correct expression execution is much simpler than the definition of the precondition. The predicate $expr_{\text{pos}}$ states that (i) the result of the expression has been correctly computed, (ii) only free registers have been changed, (iii) the memory has not been changed, (iv) the

special purpose registers have not been changed, and, finally, (v) that the new configuration is a valid assembly configuration.

Before we give a formal definition of $expr_{\text{pos}}$ we have to define what it means that the result of expression evaluation is correct.

Definition 9.4 (Correct expression result) Let te be a type name environment, c a $C0$ configuration, and $alloc$ an allocation function. Let furthermore e be an expression and $v_{\text{asm}} :: \mathbb{Z}$ a register content.

We define the predicate $expr_{\text{result}}\surd$ which tests whether the value of expression e in configuration c and the value v_{asm} in the assembly machine match.

If e cannot be evaluated, we define the predicate to be $true$; this will later slightly simplify our formulas and lemmas. Otherwise, let $reval(te, c.mem, e) = \lfloor v_{\text{C0}} \rfloor$ be the $C0$ value of the expression and $leval(te, c.mem, e) = \lfloor g \rfloor$ the g-variable which represents its address.

$$expr_{\text{result}}\surd :: tenv \times conf_{\text{C0}} \times (gvar \mapsto (\mathbb{N} \times \mathbb{N})) \times expr \times \mathbb{B} \times \mathbb{Z} \mapsto \mathbb{B}$$

For right expressions we require the assembly value to represent the value of the expression; if the expression is not yet initialized, every assembly value is fine.

$$\frac{initialized(te, c.mem, e) \longrightarrow vmatch_{\text{rval}}(alloc, v_{\text{C0}}(0), v_{\text{asm}})}{expr_{\text{result}}\surd(te, c, alloc, e, true, v_{\text{asm}})}$$

For left expressions we require the assembly value to contain the allocated address of the g-variable which represents the address of the expression.

$$\frac{i2n(v_{\text{asm}}) = fst(alloc(g))}{expr_{\text{result}}\surd(te, c, alloc, e, false, v_{\text{asm}})}$$

Definition 9.5 (Expression code: postconditions) Now, we can formally define the predicate $expr_{\text{pos}}$ which formalizes the five requirements on the previous page.

$$\frac{\begin{array}{c} expr_{\text{result}}\surd(te, c, alloc, e, rf, d'.gpr!r_d) \\ \forall r \notin (\{r_d, r1, r2, r3\} \cup fregs) : d.gpr!r = d'.gpr!r \\ d'.mm = d.mm \qquad d'.spr = d.spr \qquad conf_{\text{asm}}\surd(d') \end{array}}{expr_{\text{pos}}(d, d', te, c, alloc, e, code_{\text{e}}, range_{\text{c}}, a, t, rf, r_d, fregs)}$$

9.2.3 Compatibility

We present one of the lemmas which guarantee compatibility between the predicates about the proper execution of expression code and the predicates which are used for the statement correctness proofs in Chapter 10 (cf. the start of Section 9.1).

Lemma 9.1 *Let e be a valid, translatable, and correctly compiled expression, let c be a valid $C0$ configuration, and (te, ft, gst) a translatable $C0$ program. Additionally, let $gst = gst(c.mem)$ and $lst = lst_{top}(c.mem)$ be abbreviations for the global and local symbol tables. This implies, given that a lot of additional technical requirements (which resemble essentially the predicates $expr_{pre}$ and*

asm_{pre}) are fulfilled (cf. below), that the expression code can be executed according to the definitions in Section 6.4.1 with a result that is correct according to Definition 9.4.

$$
\begin{aligned}
& code_{expr}\surd(te, ft, c, range_c, addr_{max}, e, rf) \\
& \wedge\, e \in valid_{expr}(te, gst, lst) \wedge e \in xltbl_{expr}(te, ft, gst, lst) \\
& \wedge\, c \in conf\surd(te, ft) \wedge (te, ft, gst) \in xltbl_{prog} \\
& \wedge\, addr_{max} \leq 2^{32} \wedge avail_{mem}(addr_{max}, te, ft, c) \\
& \wedge\, \neg is\_lit(e) \longrightarrow consis_d(te, ft, c, alloc, d) \\
& \wedge\, enough_{regs}(te, gst, lst, rf, |fregs|, e) \\
& \wedge\, \neg rf \longrightarrow mobj(te, gst, lst, e) \\
& \wedge\, rf \longrightarrow (elem?_t(the(type(te, gst, lst, e))) \wedge initialized(te, c.mem, e)) \\
& \wedge\, fregs \neq [\,] \wedge distinct(fregs) \wedge \forall r \in fregs : fregs \in fregs_{ini} \\
& \wedge\, rf \longrightarrow reval(te, c.mem, e) \neq None \\
& \wedge\, \neg rf \longrightarrow leval(te, c.mem, e) \neq None \\
& \wedge\, asm_{pre}(d, range_c, code_{expr}(te, gst, lst, rf, hd(fregs), tl(fregs), e)) \\
\Longrightarrow\ & \exists t, d' : \quad d \xrightarrow[range_c, range_a]{t, d.dpc+4\cdot csize_e(e)} d' \\
& \qquad \wedge d'.mm = d.mm \\
& \qquad \wedge d'.spr = d.spr \\
& \qquad \wedge \forall r \notin fregs \circ [r1, r2, r3] : d'.gpr!r = d.gpr!r \\
& \qquad \wedge expr_{result}\surd(te, c, alloc, e, rf, d.gpr!(hd(fregs)))
\end{aligned}
$$

Proof We omit the lengthy proof of this lemma as it mostly just discharges all the technical preconditions of $expr_{\text{pre}}$.

9.3 Correctness Theorem

Using the definitions from Section 9.2 we can state the correctness theorem for expression evaluation in a compact way. Note however, that a lot of additional conditions are hidden in the predicates $expr_{\text{pre}}$ and $expr_{\text{pos}}$ from the previous section.

Theorem 9.2 (Correct compilation of expressions)
Let te be a type name environment, ft a function table, $range_c$ a code range, $addr_{max}$ the maximum address in the address space, c a C0 configuration, and e a C0 expression.

Then, e is correctly compiled by the C0 compiler for arbitrary right flags rf. Formally the theorem states

$$\forall rf : code_{expr}\surd(te, ft, c, range_c, addr_{max}, e, rf)$$

Proof The correctness of the code generation algorithm for expressions has been proved in Isabelle / HOL by induction on the expression's structure.

In this thesis we will not give detailed proofs for the different induction cases; instead, we highlight very briefly some interesting parts of the proof. Detailed proof transcripts for some of the cases can be found in several bachelor and master theses [Kre06, Pen06, Pen07]. q.e.d.

We sketch some proof details which are not covered by the above theses:

- Hristo Pentchev presents in [Pen07] correctness proofs for signed multiplication, division, and modulo. His proofs for signed multiplication relies on the fact that (for $C0$) an assembly multiplication algorithm for unsigned numbers can be reused for signed multiplication without change. To allow this, the following lemma has been proved as part of this thesis.

  Lemma 9.3 (Signed multiplication) *Let a and b be natural numbers. Then the signed multiplication of a and b interpreted as integers gives the same result as first doing unsigned multiplication of a and b and then interpreting the result as an integer.*

  $$0 \leq a < 2^{32} \wedge 0 \leq b < 2^{32}$$
  $$\Longrightarrow n2i(a) \cdot n2i(b) \bmod_s 2^{31} = n2i(a \cdot b \bmod_u 2^{32})$$

- The correctness of the equality test for pointers requires some special argument.

  In $C0$, equality or inequality of pointers is tested by comparing the embedded g-variables. However, the code generation for equality tests uses directly the corresponding instructions *seq* and *sne* of the VAMP processor (cf. Section 7.3.5). Thus, we have to show that equality of the register values (i.e., the allocated base addresses of the pointers) implies equality of the pointers on the $C0$ layer (i.e., both pointers are null pointers or refer to the same g-variable). This sounds quite trivial, but there are two hidden problems.

  1. We have to show that no g-variable is allocated at address zero because this would coincide with the register representation of null pointers. However, this follows directly from the fact that g-variables are allocated behind the code and that the program's code is not empty.
  2. There exist indeed multiple valid g-variables which are allocated at the same address on the target architecture. For example, let g be a g-variable of type array. Then, g and $gvar_{\text{arr}}(g, 0)$ are both allocated at the same address. However, only g-variables of different types may be allocated at the same address and, for type correct equality tests in $C0$, we know that both sub expressions have to be of the same type.

  We have proved the following lemma to discharge these proof obligations.

  Lemma 9.4 *Let c be a $C0$ configuration, alloc an allocation function, $p_1 :: gvar \cup \{\bot\}$ and $p_2 : gvar \cup \{\bot\}$ be $C0$ pointers, and r_1 and r_2 register values. If c is a valid configuration and consistent with alloc, both p_1 and p_2 are reachable and of the same type, and p_1 and p_2 are proper representations of r_1 and r_2, respectively, then we can show that equality*

of r_1 and r_2 is equivalent to equality of p_1 and p_2.

$$
\begin{aligned}
& c \in \mathit{conf}\surd(\mathit{te}, \mathit{ft}) \\
& \wedge\ \mathit{consis}_{\mathit{alloc}}(\mathit{te}, \mathit{ft}, c, \mathit{alloc}) \\
& \wedge\ p_1 \in \mathit{reachable}_{\mathit{ptr}}(c.\mathit{mem}) \wedge p_2 \in \mathit{reachable}_{\mathit{ptr}}(c.\mathit{mem}) \\
& \wedge\ \mathit{vmatch}_{\mathit{ptr}}(\mathit{alloc}, p_1, r_1) \wedge \mathit{vmatch}_{\mathit{ptr}}(\mathit{alloc}, p_2, r_2) \\
& \wedge\ (p_1 \neq \bot \wedge p_2 \neq \bot \Longrightarrow \mathit{ty}_g(\mathit{sc}(c.\mathit{mem}), p_1) = \mathit{ty}_g(\mathit{sc}(c.\mathit{mem}), p_2)) \\
& \wedge\ -2^{31} \leq r_1 < 2^{31} \wedge -2^{31} \leq r_2 < 2^{31} \\
& \Longrightarrow (r_1 = r_2) \equiv (p_1 = p_2)
\end{aligned}
$$

- The correctness of the code for unsigned comparison (cf. Definition 7.23 on page 141) is not obvious. A correctness proof is given in [Pen06].

Chapter 10

Correct Compilation of Statements

Contents

In this Chapter we prove the correct compilation of statements. We start in Section 10.1 with a theorem which allows us to pull out a large part of the control consistency proof from the correctness proofs for individual statements. In the remaining sections of this chapter, we prove theorems about the code generation for the different $C0$ statements. These theorem are used in the induction step of the main compiler correctness theorem (cf. the proof of Theorem 8.3 on page 183). Observe that we do not argue about compilation of *Skip* and *Comp* here because they are treated in a special way in the proof of Theorem 8.3.

10.1 Control Consistency

The code for control statements (conditionals and loops) is fragmented, i.e., it is not generated in a single piece but interrupted by sub statements. Figure 10.1 demonstrates this for conditional statements. The code which is generated for the conditional starts with code for the test expression followed by a branch. Then follows the code for the first sub statement s, a jump instruction, and finally the code for the second sub statement s'.

During the execution of such fragmented code we run into situations where we have to execute code (the jump instruction in the example) but the $C0$ statements for which this code has been generated has already been removed from the program rest.

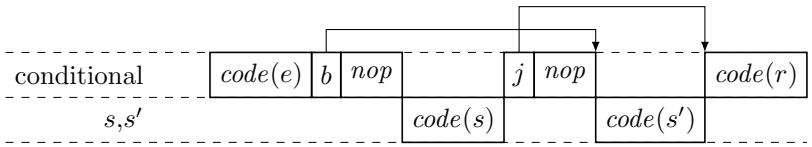
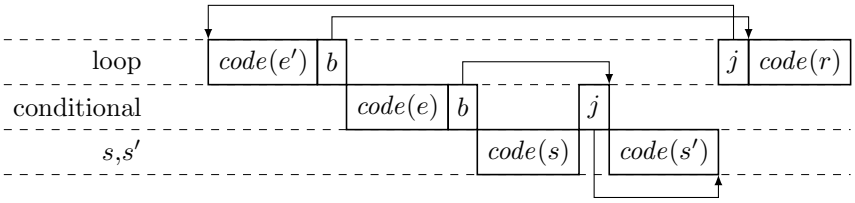


Figure 10.1: Fragmented Generation for Control Statements



Nop instructions are omitted in this figure

Figure 10.2: Fragmented Generation for Control Statements Cont.

We demonstrate this with the execution of the example code from Figure 10.1; we assume that the condition evaluates to *true*. Initially, the program rest is $c^0.prog = \mathit{Ifte}(e, s, s'); r$ and the program counter points to the start of $code(e)$. After one step of the $C0$ machine, the conditional statement has been executed and the program rest is $c^1.prog = s; r$; now, the program counter of the assembly machine points to the start of $code(s)$. Given that the execution of statement s terminates, we eventually reach a $C0$ configuration where s has just been completely executed and the program rest is $c^i.prog = r$. When we just have executed the last instruction of $code(s)$ the program counter points to the address of the jump instruction which has been generated for the conditional statement.

This means that after the execution of the code of a certain statement, e.g., the last statement in the if-part, we may have to execute code pieces from other statements (in this example: the conditional) before we reach a control consistent state; i.e., before the program counter reaches the start of $code(r)$.

Figure 10.2 shows a slightly more involved example, where two control statements are nested. In this example we would have to execute two *old* jump instructions to reach a control consistent state after the execution of $code(s)$. In the following, we will call such separated jump instructions *control code*. In general, such structures can be arbitrarily deeply nested.

The remainder of this section is divided into two parts.

1. In Section 10.1.1, we prove the correctness of the control code, i.e., that if the program counter points directly behind the last instruction of $code(s)$ for some statement s we will eventually reach an assembly configuration in which the program counter points to the first instruction of the successor statement of s and the memory has not been altered.
2. In Section 10.1.2, we use Theorem 5.16 on page 92 to show that the first statement in $s2l_{\mathrm{ns}}(r)$ is the successor statement of the last statement in $s2l_{\mathrm{ns}}(s)$. This allows us, together with the lemmas from 1, to show that we eventually reach a control consistent state.

Observe that this method does not work for *Return* statements because they do not have a unique successor statement but are followed by the successor statement of the corresponding function call statement. Thus, we need the special consistency invariant about return addresses (cf. Definition 8.13 on page 177).

10.1.1 Correctness of the Control Code

We split the correctness proof of the control code into several cases. The first and simplest case is the control code behind the last statement in a while loop.

Lemma 10.1 (Correct control code: loops) *Let (te, ft, gst) be a translatable C0 program, c a valid C0 configuration, d a valid assembly configuration, $range_c = (progbase, progbase + 4 \cdot csize_{prog}(te, gst, ft))$ the code range of the program, and $range_a = (progbase, addr_{max})$ the address range. Furthermore, let s be a statement which is the last statement in some loop body and let f be the father statement of s. Finally, let the program counter of assembly configuration d point behind the last instruction of $code(s)$.*

Then we will eventually reach an assembly configuration d' in which the program counter points to the code base of f and neither the memory nor the special registers have changed.

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge (te, ft, gst) \in xltbl_{prog} \wedge conf_{asm}\surd(d) \\
& \wedge is\_last_s^{loop}(ft, s) \wedge parent_s(s, ft) = \lfloor f \rfloor \wedge consis_{code}(te, ft, c, d) \\
& \wedge d.dpc = the(cbase_s(te, gst, ft, s)) + 4 \cdot the(csize_s(te, gst, ft, s)) \\
& \wedge d.pcp = d.dpc + 4 \\
\Longrightarrow \exists t, d' : \quad & d \xrightarrow[range_c, range_a]{t, the(cbase_s(te, gst, ft, f))} d' \\
& \wedge d'.mm = d.mm \\
& \wedge d'.spr = d.spr \\
& \wedge \forall r \in \{r_{sbase}, r_{htop}, r_{lframe}, r_{jal}\} : d'.gpr!r = d.gpr!r
\end{aligned}
$$

Proof We conclude from Definition 5.26 on page 88 that the parent statement (and thus by Definition 5.33 also the successor statement) of s is a loop statement; thus, let $f = Loop(e, lb)$. From the code generation algorithm for statements we can conclude that the code of statement s is followed by the jump instruction which is generated at the end of $code(f)$ (cf. Figure 7.7 on page 152). Thus, we just have to show that the jump distance is correct. If we instantiate the existential quantifiers with $t = 2$ and $d' = \delta^2_{\text{asm}}(d)$ we have

$$
\begin{aligned}
\delta^2_{\text{asm}}(d).dpc &= d.pcp - 4 \cdot (csize_{\text{e}}(true, e) + csize_{\text{s}}(lb) + 3) \\
&= the(cbase_{\text{s}}(s)) + 4 \cdot the(csize_{\text{s}}(s)) + 4 \\
&\quad - 4 \cdot (csize_{\text{e}}(true, e) + csize_{\text{s}}(lb) + 3) && \text{(precondition)} \\
&= the(cbase_{\text{s}}(f)) && \text{(Lemma 7.1 and correctness of } displ_{\text{s}}\text{)}
\end{aligned}
$$

q.e.d.

Lemma 10.2 (Correct control code: conditionals) *Let c be a valid C0 configuration, (te, ft, gst) a translatable C0 program, d a valid assembly configuration, $range_c = (progbase, progbase + 4 \cdot csize_{prog}(te, gst, ft))$ the code range of*

the program, and $range_a = (progbase, addr_{max})$ *the address range. Furthermore, let* s *be a statement which is the last statement either in the if or in the else part of a conditional statement, let* $i \leq cond_{lvl}(s, ft)$ *be natural number which is not greater than the condition nesting level of* s*, and let* f *be the* i*-th father of* s*. Finally, let the program counter of assembly configuration* d *point behind the last instruction of* $code(s)$*.*

Then we will eventually reach an assembly configuration d' *in which the program counter points behind the last instruction of* $code(f)$ *and neither the memory nor the special registers have changed.*

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge (te, ft, gst) \in xltbl_{prog} \wedge conf_{asm}\surd(d) \\
& \wedge is\_last_s^{cond}(ft, s) \wedge i \leq cond_{lvl}(s, ft) \wedge parent_s^i(s, ft) = \lfloor f \rfloor \\
& \wedge consis_{code}(te, ft, c, d) \wedge d.pcp = d.dpc + 4 \\
& \wedge d.dpc = the(cbase_s(te, gst, ft, s)) + 4 \cdot the(csize_s(te, gst, ft, s)) \\
\Longrightarrow \exists t, d' : \quad & d \xrightarrow[range_c, range_a]{t, the(cbase_s(f)) + 4 \cdot the(csize_s(f))} d' \\
& \wedge d'.mm = d.mm \\
& \wedge d'.spr = d.spr \\
& \wedge \forall r \in \{r_{sbase}, r_{htop}, r_{lframe}, r_{jal}\} : d'.gpr!r = d.gpr!r
\end{aligned}
$$

Proof We prove this lemma by induction on i.

For the induction start there is nothing to prove because we have $s = f$ by the Definition 5.27 and simply instantiate $t = 0$ and $d' = d$.

For the induction step – from i to $i + 1$ – we know that there exists a t' and some intermediate configuration d'' such that the program counter after t' steps points behind the code of the i-th father $f^i = the(parent_s^i(s, ft))$. So it just remains to show that the control code behind the i-th father is correct and that the program counter points behind the code of the $i + 1$-th father f^{i+1}. We distinguish two cases.

Case 1: f^i is the last statement in an else part: $is\_last_s^{else}(ft, f^i)$
This is the easy case: behind the last statement in the else part there is no additional control code (at least not from the current conditional statement). Thus, we can simply instantiate $t = t'$ and $d' = d''$.

Case 2: f^i is the last statement in an if part: $is\_last_s^{if}(ft, f^i)$
In this case we have to prove – similar to the proof of Lemma 10.1 that the jump instruction between the if and the else part is correct (cf. Figure 7.8 on page 152). If we instantiate the existential quantifiers with $t = 2$ and $d' = \delta^2_{\text{asm}}(d'')$ and assuming that the else part of the conditional statement is s_2 we have

$$
\begin{aligned}
\delta^2_{\text{asm}}(d'').dpc &= d''.pcp + 4 \cdot (the(csize_s(s_2)) + 1) \\
&= the(cbase_s(f^i)) + 4 \cdot the(csize_s(f^i)) + 4 \\
&\quad + 4 \cdot (the(csize_s(s_2)) + 1) \qquad \text{(induction hypothesis)} \\
&= the(cbase_s(f^{i+1})) + 4 \cdot the(csize_s(f^{i+1})) \\
& \qquad\qquad \text{(from code structure)}
\end{aligned}
$$

q.e.d.

Corollary 10.3 (Correct control code: conditionals) *The correctness of the control code for conditionals can be extended to the condition senior of s by using $i = cond_{lvl}(s, ft)$.*

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge (te, ft, gst) \in xltbl_{prog} \wedge conf_{asm}\surd(d) \\
& \wedge is\_last_s^{cond}(ft, s) \wedge cond_{sen}(s, ft) = \lfloor sen \rfloor \wedge consis_{code}(te, ft, c, d) \\
& \wedge d.dpc = the(cbase_s(te, gst, ft, s)) + 4 \cdot the(csize_s(te, gst, ft, s)) \\
& \wedge d.pcp = d.dpc + 4 \\
\Longrightarrow \exists t, d' : \quad & d \xrightarrow[range_c, range_a]{t, the(cbase_s(sen)) + 4 \cdot the(csize_s(s))} d' \\
& \wedge d'.mm = d.mm \\
& \wedge d'.spr = d.spr \\
& \wedge \forall r \in \{r_{sbase}, r_{htop}, r_{lframe}, r_{jal}\} : d'.gpr!r = d.gpr!r
\end{aligned}
$$

We combine Lemma 10.1 and Corollary 10.3 into the following theorem.

Theorem 10.4 (Correct control code)
Let (te, ft, gst) be a translatable C0 program, c a valid C0 configuration, and d a valid assembly configuration. Further denote by $range_c = (progbase, progbase + 4 \cdot csize_{prog}(te, gst, ft))$ the code range of the program and its address range by $range_a = (progbase, addr_{max})$. Furthermore, let s be a non-structural statement; we assume that s is not a Return statement. Finally, let the program counter of assembly configuration d point behind the last instruction of $code(s)$.

Then, after a certain number t of assembly steps, we reach a configuration where the program counter points to the first instruction of the successor statement of s and neither the memory nor important registers have been altered.

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge (te, ft, gst) \in xltbl_{prog} \wedge conf_{asm}\surd(d) \\
& \wedge \neg stmt_{structural}(s) \wedge \neg is\_Return(s) \wedge succ(s, ft) = \lfloor s' \rfloor \\
& \wedge consis_{code}(te, ft, c, d) \wedge d.pcp = d.dpc + 4 \\
& \wedge d.dpc = the(cbase_s(te, gst, ft, s)) + 4 \cdot the(csize_s(te, gst, ft, s)) \\
\Longrightarrow \exists t, d' : \quad & d \xrightarrow[range_c, range_a]{t, the(cbase_s(s'))} d' \\
& \wedge d'.mm = d.mm \\
& \wedge d'.spr = d.spr \\
& \wedge \forall r \in \{r_{sbase}, r_{htop}, r_{lframe}, r_{jal}\} : d'.gpr!r = d.gpr!r
\end{aligned}
$$

Proof We prove this theorem by case distinction.

Case 1: s is not a last statement: $\neg is\_last_s(ft, s)$
In this case the code for the successor statement of s is directly placed behind the code of s. Thus, we simply instantiate $t = 0$ and $d' = d$ and the proof is done.

Case 2: s is the last statement in a while body: $is\_last_s^{loop}(ft, s)$
In this case we use Lemma 10.1 to obtain t and d' for the instantiation. From Definition 5.33 on page 91 we conclude that the successor of the last statement in a loop is the loop statement itself. Thus, our instantiation is correct and we are done with this case.

Case 3: s is the last statement in a conditional: $is\_last_s^{cond}(ft, s)$
This case is the most complicated one. Firstly, we have to use Corollary 10.3 which argues about the correct ascent up to to the condition senior sen of s. I.e., we know that there exists t such that the program counter after t assembly steps points directly behind $code(sen)$.

However, sen itself can be the last statement in a loop. In this case we have to argue as in the previous case that we correctly reach the beginning of the loop statement.

If sen is not a last statement in a loop we know by Definition 5.33 the successor statement of s is the successor statement of the condition senior. And because sen is not a last statement, the code of its successor is placed directly behind $code(sen)$ in the code. Thus, the program counter already points to the code base of the successor statement s'. q.e.d.

10.1.2 Proof of Control Consistency

We combine Theorems 5.16 and 10.4 to the following theorem which allows to conclude that after reaching a data consistent state directly behind the code of a statement we will eventually reach a state which is completely consistent. This lemma will be used in the following sections to conclude from the *local* correctness of certain statements (i.e., that they correctly manipulate memory and registers) to the global correctness including the correct execution of potentially present control code.

Theorem 10.5 (Control consistency)
Let (te, ft, gst) be a translatable C0 program, c a valid C0 configuration, and d a valid assembly configuration. Further, denote by $range_c = (progbase, progbase + 4 \cdot csize_{prog}(te, gst, ft))$ the code range of the program and its address range by $range_a = (progbase, addr_{max})$. Furthermore, let c' be the next configuration after one step of the C0 machine. Finally, let the first statement in the program rest of c not be a Return statement which will be executed completely in a single C0 step (i.e., not a loop with true condition, not a conditional with non-empty branch, and not a function call statement) and let the program counter of assembly configuration d point behind the code of that statement.

Then, – given that we have data and return address consistency between d and c' – after a certain number t of assembly steps we reach a configuration which is completely consistent with c'.

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge (te, ft, gst) \in xltbl_{prog} \wedge conf_{asm}\surd(d) \\
& \wedge\, consis_{code}(te, ft, c', d) \wedge consis_d(te, ft, c', alloc, d) \wedge consis_{ra}(te, ft, c', d) \\
& \wedge\, s2l_{ns}(c.prog) \neq [\,] \wedge \delta_{C0}(te, ft, c) = \lfloor c' \rfloor \wedge s2l_{ns}(c'.prog) = tl(s2l_{ns}(c.prog)) \\
& \wedge\, \neg is\_Return(hd(s2l_{ns}(c.prog))) \wedge d.pcp = d.dpc + 4 \\
& \wedge\, d.dpc = the(cbase_s(hd(s2l_{ns}(c.prog)))) + 4 \cdot the(csize_s(hd(s2l_{ns}(c.prog))))
\end{aligned}
$$

$$
\begin{aligned}
\Longrightarrow \exists t, d' : \quad & d \xrightarrow[range_c, range_a]{t, d'.dpc} d' \\
& \wedge consis(te, ft, c', alloc, d') \\
& \wedge d'.spr = d.spr \\
& \wedge \forall r \in \{r_{sbase}, r_{htop}, r_{lframe}, r_{jal}\} : d'.gpr!r = d.gpr!r
\end{aligned}
$$

PROOF We prove this theorem by case distinction on the length of the old program rest.

Case 1: $|s2l_{\text{ns}}(c.prog)| = 0$
In this case the precondition $s2l_{\text{ns}}(c.prog) \neq [\,]$ is violated.

Case 2: $|s2l_{\text{ns}}(c.prog)| = 1$
In this case – which happens exactly when the $C0$ program has terminated – the proof of control consistency is simple because we have $s2l_{\text{ns}}(c'.prog) = [\,]$ and thus there is nothing else to show (cf. Definition 8.14 on page 177). We instantiate $t = 0$ and $d' = d$ and are done.

Case 3: $|s2l_{\text{ns}}(c.prog)| \geq 2$
Here we use Theorem 10.4 with instantiation $s' = hd(s2l_{\text{ns}}(c'.prog))$ to show that we eventually reach the base of the code of the successor statement. In order to discharge the last precondition of the lemma we exploit that $c.prog$ forms a valid program rest according to Definition 5.35 on page 92 and thus $hd(s2l_{\text{ns}}(c'.prog))$ is the valid successor statement.

Now, we just have to show that data consistency and return address consistency are not destroyed by the control code. This holds because the memory stays unchanged (data consistency) and because we do not change the structure of *Return* statements in the program rest (return address consistency). q.e.d.

10.2 Auxiliary Lemmas for Data Consistency

In this section we present several lemmas which will be used later in the correctness proofs for the different $C0$ statements.

10.2.1 Basic Lemmas About Allocation Functions

We start with some basic properties of allocation functions. The proofs for most of these lemmas are straightforward and will not be presented here.

Lemma 10.6 (Base address is multiple of four) *Let c be a valid $C0$ configuration which is allocation consistent with allocation function alloc. Then, the first component of $alloc(g)$ for any reachable g-variable g is a multiple of four.*

$$c \in conf\surd(te, ft) \wedge consis_{alloc}(te, ft, c, alloc) \wedge g \in reachable_g(c.mem)$$
$$\Longrightarrow 4 \,|\, fst(alloc(g))$$

Lemma 10.7 (Base address is behind the code region) *Let c be a valid $C0$ configuration which is allocation consistent with the allocation function alloc. Then, the allocated base address of any reachable g-variable is behind the code region.*

$$c \in conf\surd(te, ft) \wedge consis_{alloc}(te, ft, c, alloc) \wedge g \in reachable_g(c.mem)$$
$$\Longrightarrow fst(alloc(g)) \geq snd((progbase, progbase + 4 \cdot csize_{prog}(te, gst(c.mem), ft)))$$

Lemma 10.8 (Allocation below maximum address) *Let c be a valid $C0$ configuration which is data consistent with an assembly configuration d via allocation function alloc. Further, assume that we have sufficient memory. Then, any reachable g-variable is allocated completely below the maximum address.*

$$\begin{aligned} & c \in conf\surd(te, ft) \wedge consis_d(te, ft, c, alloc, d) \\ & \wedge avail_{mem}(addr_{max}, te, ft, c) \wedge g \in reachable_g(c.mem) \\ & \Longrightarrow fst(alloc(g)) + snd(alloc(g)) < addr_{max} \end{aligned}$$

The two previous lemmas can be easily combined in the following way.

Lemma 10.9 (Allocation inside the address range) *Let c be a valid $C0$ configuration which is data consistent with an assembly configuration d via allocation function alloc. Further, assume that we have sufficient memory. Then, any reachable g-variable is allocated completely in the address range.*

$$\begin{aligned} & c \in conf\surd(te, ft) \wedge consis_d(te, ft, c, alloc, d) \\ & \wedge avail_{mem}(addr_{max}, te, ft, c) \wedge g \in reachable_g(c.mem) \\ & \Longrightarrow alloc(g) \subseteq (progbase, addr_{max}) \end{aligned}$$

Lemma 10.10 (Allocation: no overlap with frame headers) *Let c be a valid and allocation consistent $C0$ configuration and let g be a reachable g-variable. Then, the allocated memory region of g does not overlap with any frame header.*

$$\begin{aligned} & c \in conf\surd(te, ft) \wedge consis_{alloc}(te, ft, c, alloc) \\ & \wedge g \in reachable_g(c.mem) \wedge i < |c.mem.lm| \\ & \Longrightarrow alloc(g) \asymp (abase_{lm}(te, ft, sc(c.mem), i), 12) \end{aligned}$$

Lemma 10.11 (Elementary sub g-variables do not overlap) *Let c be a valid $C0$ configuration which is allocation consistent with the allocation function alloc. Further, let g and g' be two reachable g-variables and assume that g is an elementary g-variable which is not a sub g-variable of g'. Then, the allocated range of g and g' does not overlap.*

$$\begin{aligned} & c \in conf\surd(te, ft) \wedge consis_{alloc}(te, ft, c, alloc) \\ & \wedge g \in reachable_g(c.mem) \wedge g' \in reachable_g(c.mem) \\ & \wedge elem?_t(ty_g(sc(c.mem), g)) \wedge g \notin sub_g(g') \\ & \Longrightarrow alloc(g) \asymp alloc(g') \end{aligned}$$

Proof First, we conclude from g being an elementary g-variable that g' is not a sub g-variable of g. In principle, they could be *equal*, but this would contradict with $g \notin sub_g(g')$. The only interesting remaining case is that g and g' have a common ancestor; all other cases are trivial.

If g and g' have a common ancestor let w.l.o.g. g be the 'simpler' one (i.e., the one with fewer recursive steps from its root g-variable). Then, we prove the goal by induction on the structure of g. Eventually, we will reach the junction where the ancestry of g and g' split up. Here, it is easy to show that the allocated ranges of the split sub trees do not overlap. q.e.d.

Lemma 10.12 (G-variables do not overlap) *Let c be a valid $C0$ configuration which is allocation consistent with the allocation function alloc. Further, let g and g' be two reachable g-variables which have the same allocated size. Then, the allocated range of g and g' either does not overlap at all or is identical.*

$$\begin{aligned}
&c \in conf\surd(te, ft) \wedge consis_{alloc}(te, ft, c, alloc) \\
&\wedge g \in reachable_g(c.mem) \wedge g' \in reachable_g(c.mem) \\
&\wedge asize_t(ty_g(sc(c.mem), g)) = asize_t(ty_g(sc(c.mem), g')) \\
&\Longrightarrow alloc(g) \asymp alloc(g') \vee alloc(g) = alloc(g')
\end{aligned}$$

Proof If $g \in sub_{\mathrm{g}}(h)$ or $h \in sub_{\mathrm{g}}(g)$, i.e., one is a sub g-variable of the other, a simple induction proof following the structure of the sub g-variable shows that both g-variables have the same displacement. Observe that they do not need to be equal: imagine the case that $g \in sub_{\mathrm{g}}(h)$ where h is a one-elementary array and g is this one element. Obviously, g and h would be different but still would have the same displacement.

If g and h are not sub g-variables the proof is trivial. q.e.d.

10.2.2 Consistency After Updating Memory

The statement correctness proofs in the following sections will often argue that updating a g-variable g in the $C0$ machine and simultaneously updating the memory of an assembly machine at the allocated base address of g with the corresponding assembly value preserves data consistency. In this section we will prove several lemmas which state this formally.

First, we show that value and pointer consistency are preserved by updates of g-variables. Then, we extend this to return address, frame header, and allocation consistency. Finally, we combine those results to the preservation of data consistency.

Value and Pointer Consistency

We start this section with proofs about value and pointer consistency when updating a given g-variable g.

First, we present a lemma which shows that value and pointer consistency are preserved for sub g-variables of the updated g-variable g.

Lemma 10.13 (Updated g-variables: consistency inside) *Let c be some valid $C0$ configuration which is allocation consistent with the allocation function alloc, g_l a reachable g-variable, and g a sub g-variable of g_l. Further, let value v match the value stored at the allocated address of g_l in an assembly configuration d' and be type correct with a given type t which matches the type of g_l. Finally, assume that updating g_l with v is successful and gives a new $C0$ memory configuration m'.*

Then, we know that – after the memory update – g is value and pointer consistent with respect to the $C0$ memory m', the allocation function alloc, and

the assembly configuration d'.

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge consis_{alloc}(te, ft, c, alloc) \\
& \wedge\, g_l \in reachable_g(c.mem) \wedge g \in sub_g(g_l) \\
& \wedge\, memupd(c.mem, g_l, v) = \lfloor m' \rfloor \\
& \wedge\, ty\surd(te, sc(c.mem), v, t, 0) \wedge tmatch_{ass}(ty_g(sc(c.mem), g_l), t) \\
& \wedge\, vmatch_{gvar}(sc(c.mem), v, d', alloc, g_l, fst(alloc(g_l))) \\
& \Longrightarrow consis_v^g(m', alloc, d', g) \wedge consis_p^g(m', alloc, d', g)
\end{aligned}
$$

PROOF We know from the definition of $consis_{\mathrm{v}}^{\mathrm{g}}$ and $consis_{\mathrm{p}}^{\mathrm{g}}$ that g is reachable – and thus also valid – in memory m' and of an elementary type; otherwise the lemma is trivially true.

Finally, we have to show that the value $v'_{\mathrm{C0}} = value_{\mathrm{g}}(m', g)(0)$ of g in m' matches the value $v'_{\mathrm{asm}} = cell2int(d'.mm(fst(alloc(g)) \div 4))$ which is stored at the allocated base address of g in assembly machine d'. We can conclude from the definition of $vmatch_{\mathrm{gvar}}$ that the value $v_{\mathrm{C0}} = v[ba_{\mathrm{g}}(sc(c.mem, g_{\mathrm{l}}) - ba_{\mathrm{g}}(sc(c.mem, g)]$ at the relative base address of g inside g_{l} in the old memory $c.mem$ matches the value $cell2int(d'.mm(fst(alloc(g_{\mathrm{l}})) + displ_{\mathrm{rel}}(sc, g_{\mathrm{l}}, g) \div 4))$ at the allocated base address of g_{l} plus the relative displacement of g inside g_{l} in the assembly configuration d'. Using allocation consistency we can show that this address corresponds to the allocated base address of g. Thus, we know that v_{C0} matches v'_{asm}. Additionally, we can show from the semantics of memory updates that v_{C0} equals the value v'_{C0} which we read from m' at the address of g. q.e.d.

ISABELLE Observe that the preconditions $tmatch_{\mathrm{ass}}$ and $ty\surd$ in the previous and some of the following lemmas are merely needed for technical reasons: The definition of $vmatch_{\mathrm{rval}}$ (e.g., used inside $vmatch_{\mathrm{gvar}}$) makes a case distinction based on the value of a memory cell (whether or not $is\_Ptr(v_{\mathrm{C0}})$). In contrast, the definition of $consis_{\mathrm{v}}^{\mathrm{g}}$ and $consis_{\mathrm{p}}^{\mathrm{g}}$ require a certain *type* of the g-variable. Thus, to relate the value and types of the g-variables we have to know that type correctness holds. However, we can easily discharge these additional preconditions using, e.g., Theorem 5.14 on page 87.

Similar, to the previous lemma we now prove value and pointer consistency for g-variables outside the updated memory region.

Lemma 10.14 (Updated g-variables: value consistency outside) *Let c be an allocation consistent valid $C0$ configuration, g_l a reachable g-variable, v a value which, again, is type correct with a matching type t, and m' the updated memory configuration. Now, in contrast to the previous lemma, g is not a sub g-variable of g_l and we require that g is value consistent with respect to c, alloc, and assembly configuration d. Additionally, we require that the reachability of g is not changed during the update and that between assembly configurations d and d' only that part of the memory has changed which is allocated to g_l.*

Then, we can show that g is value consistent with respect to $C0$ memory m',

allocation function alloc, and assembly configuration d′.

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge consis_{alloc}(te, ft, c, alloc) \\
& \wedge g_l \in reachable_g(c.mem) \wedge g \notin sub_g(g_l) \\
& \wedge consis_v^g(c.mem, alloc, d, g) \\
& \wedge memupd(c.mem, g_l, v) = \lfloor m' \rfloor \\
& \wedge (g \in reachable_g(c.mem) = g \in reachable_g(m')) \\
& \wedge tmatch_{ass}(ty_g(sc(c.mem), g_l), t) \\
& \wedge memchngd(d.mm, d'.mm, fst(alloc(g_l)) \div 4, snd(alloc(g_l)) \div 4) \\
& \Longrightarrow consis_v^g(m', alloc, d', g)
\end{aligned}
$$

Proof This lemma is quite simple to prove. Basically, we have to show that neither in the $C0$ machine nor in the assembly machine the value of g is changed. Additionally, we need to show that g was already initialized in the old configuration c; otherwise, we could not use the precondition $consis_{\mathrm{v}}^{\mathrm{g}}(c.mem, alloc, d, g)$.

The second goal follows from the fact that memory updates of a g-variable g_{l} in the $C0$ machine do not affect whether $g \notin sub_{\mathrm{g}}(g_{\mathrm{l}})$ is initialized or not. The first goal has two parts. The $C0$ value of g is constant because memory updates of g_{l} do only affect values of g-variables which overlap with g_{l}. Because g is an elementary g-variable it would only overlap with g_{l} if $g \in sub_{\mathrm{g}}(g_{\mathrm{l}})$ which does not hold.[1] The allocated memory regions of g and g_{l} are distinct by Lemma 10.11. Thus, the assembly value of g is constant because the last preconditions ensures that only memory which is allocated to g_{l} has been altered between d and d'. q.e.d.

Corollary 10.15 *The preceding lemma holds also for pointer consistency.*

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge consis_{alloc}(te, ft, c, alloc) \\
& \wedge g_l \in reachable_g(c.mem) \wedge g \notin sub_g(g_l) \\
& \wedge consis_p^g(c.mem, alloc, d, g) \\
& \wedge memupd(c.mem, g_l, v) = \lfloor m' \rfloor \\
& \wedge (g \in reachable_g(c.mem) = g \in reachable_g(m')) \\
& \wedge tmatch_{ass}(ty_g(sc(c.mem), g_l), t) \\
& \wedge memchngd(d.mm, d'.mm, fst(alloc(g_l)) \div 4, snd(alloc(g_l)) \div 4) \\
& \Longrightarrow consis_p^g(m', alloc, d', g)
\end{aligned}
$$

Isabelle In Isabelle, we have proved several corollaries from the preceding lemmas which shorten some of the formal correctness proofs about code generation for statements. However, we will not present them here because they are quite trivial combinations and / or extensions and can be explained inline in the informal proofs.

Now, we know that updating a g-variable with content consistent values preserves value and pointer consistency. The next step will be to show that the value of a g-variable g is content consistent with g if the corresponding

[1] Observe that a non-elementary g-variable g' may overlap with g_{l} even if $g' \notin sub_{\mathrm{g}}(g_{\mathrm{l}})$, namely if $g_{\mathrm{l}} \in sub_{\mathrm{g}}(g')$.

$C0$ and assembly configurations are value, pointer, and allocation consistent (cf. Lemma 10.16). Then, we will show in Corollary 10.17 that updating a g-variable g_l with the value of some other g-variable g both in the $C0$ and in the assembly configuration preserves value and pointer consistency.

Lemma 10.16 (*value*$_\mathrm{g}$ provides matching values) *Let c be an allocation consistent valid $C0$ configuration which is value and pointer consistent with an assembly configuration d. Additionally, let g be a reachable and initialized g-variable. Then, the value of g in configuration c is content consistent with g in configuration d, i.e., the value which is stored in d at the allocated address of g matches the value of g in the $C0$ configuration.*

$$
\begin{aligned}
&c \in conf\surd(te, ft)\\
&\wedge consis_v(c, alloc, d) \wedge consis_p(c, alloc, d) \wedge consis_{alloc}(te, ft, c, alloc)\\
&\wedge g \in reachable_g(c.mem) \wedge initialized_g(c.mem, g)\\
&\Longrightarrow vmatch_{gvar}(sc(c.mem), value_g(c.mem, g), d, alloc, g, fst(alloc(g)))
\end{aligned}
$$

PROOF To prove content consistency we need to show for all elementary sub g-variables g' of g that their value at the proper offset in $value_\mathrm{g}(c.mem, g)$ matches the value stored in d at address

$$a = fst(alloc(g)) + displ_\mathrm{rel}(sc(c.mem), g, g').$$

In the remainder of this proof we will concentrate on non-pointer values; the proof for pointers is done likewise.

Let in the following $ofs = ba_\mathrm{g}(sc(c.mem), g') - ba_\mathrm{g}(sc(c.mem), g)$ be the offset of g' inside of g. We instantiate the all quantifier in the definition of value consistency with g' and get

$$
\begin{aligned}
&vmatch(value_\mathrm{g}(c.mem, g')(0), cell2int(d.mm(fst(alloc(g')) \div 4)))\\
=&vmatch(value_\mathrm{g}(c.mem, g')(0), cell2int(d.mm(a \div 4)))\\
&\hfill \text{(by correctness of } displ_\mathrm{rel})\\
=&vmatch(value_\mathrm{g}(c.mem, g)(ofs), cell2int(d.mm(a \div 4)))\\
&\hfill \text{(by definition of } ofs)\\
=&vmatch_\mathrm{rval}(alloc, value_\mathrm{g}(c.mem, g)(ofs), cell2int(d.mm(a \div 4)))\\
&\hfill \text{(by Def. 8.8)}
\end{aligned}
$$

q.e.d.

Corollary 10.17 (Memory updates: value consistency) *Let c be an allocation consistent valid $C0$ configuration which is value and pointer consistent with an assembly configuration d. Further, let g_l and g_r be two reachable g-variables whose types match and let m' be the $C0$ memory configuration after a successful update of g_l with the value of g_r. Finally, let $v_{asm} = read_{data}(d.mm, fst(alloc(g_r)) \div 4, snd(alloc(g_r)) \div 4)$ be the value of g_r in the old assembly configuration and let d' be the new assembly configuration after updating the allocated memory region of g_l with v_{asm}. Then, the $C0$ configuration $c' = c[mem := m']$ after the update and the assembly configuration*

d' are value and pointer consistent.

$$
\begin{aligned}
& c \in conf\surd(te, ft) \\
& \wedge\ consis_v(c, alloc, d) \wedge consis_p(c, alloc, d) \wedge consis_{alloc}(te, ft, c, alloc) \\
& \wedge\ g_l \in reachable_g(c.mem) \wedge g_r \in reachable_g(c.mem) \\
& \wedge\ memupd(c.mem, g_l, value_g(c.mem, g_r)) = \lfloor m' \rfloor \\
& \wedge\ tmatch_{ass}(ty_g(sc(c.mem), g_l), ty_g(sc(c.mem), g_r)) \\
& \wedge\ d'.mm = d.mm\,([fst(alloc(g_l)) \div 4, snd(alloc(g_r)) \div 4] := v_{asm}) \\
& \Longrightarrow consis_v(c', alloc, d') \wedge consis_p(c', alloc, d')
\end{aligned}
$$

Proof We prove this fact by a combination of the previous lemmas and corollaries.

We have to show for all elementary g-variables g' in c' that they are value and pointer consistent. From Lemma 10.16 we know that $value_{\mathrm{g}}(c.mem, g_{\mathrm{r}})$ is content consistent with g_{r} in the old $C0$ configuration. If g' is a sub g-variable of g_{l} this fact allows us to use Lemma 10.13. Otherwise, if g' is not a sub g-variable of g_{l} we use Lemma 10.14 and Corollary 10.15 to show value and pointer consistency. q.e.d.

Return Address and Frame Header Consistency

In this paragraph we will present two lemmas which state that neither return address nor frame header consistency are destroyed by updates of g-variables in an assembly machine.

Lemma 10.18 (Memory updates: frame header consistency) *Let c be an allocation consistent valid $C0$ configuration which is frame header consistent with assembly configuration d. Updating the allocated memory region of a reachable g-variable g will not destroy frame header consistency.*

$$
\begin{aligned}
& c \in conf\surd(te, ft) \\
& \wedge\ consis_{fh}(te, ft, c, alloc, d) \wedge consis_{alloc}(te, ft, c, alloc) \\
& \wedge\ g \in reachable_g(c.mem) \wedge 4 \cdot |vl| = snd(alloc(g)) \\
& \wedge\ d'.mm = d.mm\,([fst(alloc(g) \div 4, |vl|] := vl) \\
& \Longrightarrow consis_{fh}(te, ft, c, alloc, d')
\end{aligned}
$$

Proof The main idea of this proof is that – if allocation consistency is given – the allocated memory regions of reachable g-variables and frame headers do not overlap (cf. Lemma 10.10). Thus, updating the allocated memory regions of reachable g-variables does no harm and it is easy to show that frame header consistency is preserved. q.e.d.

Lemma 10.19 (Memory update: return address consistency) *Let c be an allocation consistent valid $C0$ configuration which is return address consistent with assembly configuration d. Updating the allocated memory region of a*

reachable g-variable g will not destroy return address consistency.

$$
\begin{aligned}
& c \in \mathit{conf}\surd(te, ft) \\
& \wedge\ \mathit{consis}_{ra}(te, ft, c, d) \wedge \mathit{consis}_{alloc}(te, ft, c, \mathit{alloc}) \\
& \wedge\ g \in \mathit{reachable}_{g}(c.\mathit{mem}) \wedge 4 \cdot |vl| = \mathit{snd}(\mathit{alloc}(g)) \\
& \wedge\ d'.\mathit{mm} = d.\mathit{mm}\,([\mathit{fst}(\mathit{alloc}(g)) \div 4, |vl|] := vl) \\
& \Longrightarrow \mathit{consis}_{ra}(te, ft, c, d')
\end{aligned}
$$

Proof As long as the program rest and the return addresses of the $C0$ machine are not changed return address consistency only depends on the return addressees in the frame headers of the assembly machine. Thus, the proof of this lemma follows the same arguments as the proof of Lemma 10.18. q.e.d.

Allocation Consistency

In this paragraph we will show that allocation consistency is preserved while updating g-variables. These lemmas are simpler than those in the previous paragraphs since allocation consistency does not argue about assembly configurations but only about the relationship between a $C0$ configuration and an allocation function.

Lemma 10.20 (Memory update: heap allocation consistency) *Let c be a valid $C0$ configuration which is heap allocation consistent with an allocation function alloc. Furthermore, let g_l be a valid g-variable and let m' be the $C0$ memory configuration after a successful update of g_l with value v which is type correct with respect to a type t which matches the type of g_l. Finally, we require that all pointers in v point to reachable g-variables. Then, we can show that the new configuration $c[\mathit{mem} := m']$ is also heap allocation consistent with alloc.*

$$
\begin{aligned}
& c \in \mathit{conf}\surd(te, ft) \wedge \mathit{consis}^{\mathit{heap}}_{\mathit{alloc}}(te, ft, c, \mathit{alloc}) \\
& \wedge\ g_l \in \mathit{gvars}\surd(\mathit{sc}(c.\mathit{mem})) \wedge \mathit{memupd}(c.\mathit{mem}, g_l, v) = \lfloor m' \rfloor \\
& \wedge\ \mathit{tmatch}_{ass}(\mathit{ty}_{g}(\mathit{sc}(c.\mathit{mem}), g_l), t) \wedge \mathit{ty}\surd(te, \mathit{sc}(c.\mathit{mem}), v, t, 0) \\
& \wedge\ \forall i < \mathit{size}_{t}(t) : \mathit{reachable}_{\mathit{mcell}}(c.\mathit{mem}, v(i)) \\
& \Longrightarrow \mathit{consis}^{\mathit{heap}}_{\mathit{alloc}}(te, ft, c[\mathit{mem} := m'], \mathit{alloc})
\end{aligned}
$$

Proof The first conjunct in the definition of $\mathit{consis}^{\mathrm{heap}}_{\mathrm{alloc}}$ does not depend on the whole memory configuration but only on symbol configuration and recursion depth which are not affected by memory updates.

The same holds for the second conjunct because the set of valid heap g-variables is also not affected by memory updates.

Finally, for the third conjunct only the last part is of interest. There, we have to show that the allocated memory region of two reachable heap g-variables which do not have the same root g-variable does not overlap. Using Theorem 8.2 on page 172 we can show that both g-variables must have been reachable in the old configuration as well. Thus, this goal follows from the precondition $\mathit{consis}^{\mathrm{heap}}_{\mathrm{alloc}}(te, ft, c, \mathit{alloc})$. q.e.d.

Lemma 10.21 (Memory update: named allocation consistency) *Let c be a $C0$ configuration which is named allocation consistent with the allocation*

function alloc. Updating the memory of c preserves named allocation consistency.

$$consis^{named}_{alloc}(te, ft, c, alloc) \wedge memupd(c.mem, g_l, v) = \lfloor m' \rfloor$$
$$\Longrightarrow consis^{named}_{alloc}(te, ft, c[mem := m'], alloc)$$

Proof This lemma follows directly from Definition 8.15 on page 178: named allocation consistency does only depend on the symbol configuration which is not altered by memory updates. q.e.d.

Data Consistency

In this paragraph we will combine the lemmas from the previous sections into theorems which ensure that data consistency is preserved by memory updates.

Theorem 10.22 (Updating memory preserves data consistency) *Let c be a valid C0 configuration which is data consistent with assembly configuration d via allocation function alloc. Furthermore, let g_l be a reachable g-variable and let m' be the C0 memory configuration after a successful update of g_l with value v which is type correct with respect to a type t which matches the type of g_l. Finally, let v be content consistent with g_l in assembly configuration d' which differs from d only in the allocated memory region of g_l and in non-special registers.*

Then, the updated C0 configuration is data consistent with d' via alloc.

$$\begin{aligned}
&c \in conf\surd(te, ft) \wedge consis_d(te, ft, c, alloc, d)\\
&\wedge g_l \in reachable_g(c.mem) \wedge memupd(c.mem, g_l, v) = \lfloor m' \rfloor\\
&\wedge ty\surd(te, sc(c.mem), v, t, 0) \wedge tmatch_{ass}(ty_g(sc(c.mem), g_l), t)\\
&\wedge \forall i < size_t(t) : reachable_{mcell}(c.mem, v(i))\\
&\wedge \forall r \in \{r_{sbase}, r_{htop}, r_{lframe}, r_{jal}\} : d'.gpr!r = d.gpr!r\\
&\wedge memchngd(d.mm, d'.mm, fst(alloc(g_l)) \div 4, snd(alloc(g_l)) \div 4)\\
&\wedge vmatch_{gvar}(sc(c.mem), v, d', alloc, g_l, fst(alloc(g_l)))\\
&\Longrightarrow consis_d(te, ft, c[mem := m'], alloc, d')
\end{aligned}$$

Proof To prove this theorem we first unpack the definition of data consistency and then apply the previous Lemmas 10.13, 10.14, 10.15, 10.18, 10.20, and 10.21.

For some of these lemmas we need to know that a g-variable g which is reachable in $c[mem := m']$ has also been reachable in c; we prove this using Theorem 8.2.

The only remaining sub goal is to show that register consistency is preserved. This is easy to show because (except for the reachability aspect from the previous paragraph) register consistency only uses the symbol configuration which has not been changed by the memory update. q.e.d.

Corollary 10.23 *Let c be a valid C0 configuration which is data consistent with assembly configuration d via allocation function alloc. Furthermore, let g_l and g_r be reachable g-variables with matching types and let m' be the C0 memory configuration after a successful update of g_l with the value of g_r in the old C0 configuration. Finally, let d' be a new assembly configuration which is updated in the memory region allocated to g_l with the value $v_{asm} =$*

$read_{data}(d.mm, fst(alloc(g_r)) \div 4, snd(alloc(g_r)) \div 4)$ of g_r in the old assembly configuration.

Then, the new C0 configuration $c[mem := m']$ is data consistent with d' via alloc.

$$\begin{aligned}
&c \in conf\surd(te, ft) \wedge consis_d(te, ft, c, alloc, d)\\
&\wedge g_l \in reachable_g(c.mem) \wedge g_r \in reachable_g(c.mem)\\
&\wedge memupd(c.mem, g_l, value_g(c.mem, g_r)) = \lfloor m' \rfloor\\
&\wedge tmatch_{ass}(ty_g(sc(c.mem), g_l), ty_g(sc(c.mem), g_r))\\
&\wedge d'.mm = d.mm\,([fst(alloc(g_l)) \div 4, snd(alloc(g_r))] := v_{asm})\\
&\Longrightarrow consis_d(te, ft, c[mem := m'], alloc, d')
\end{aligned}$$

Proof We prove this corollary using Theorem 10.22. The only difficulty is that we have to discharge the last precondition

$$vmatch_{\mathrm{gvar}}(sc(c.mem), value_{\mathrm{g}}(c.mem, g_{\mathrm{r}}), d', alloc, g_{\mathrm{l}}, fst(alloc(g_{\mathrm{l}}))).$$

From Lemma 10.16 we can conclude

$$vmatch_{\mathrm{gvar}}(sc(c.mem), value_{\mathrm{g}}(c.mem, g_{\mathrm{r}}), d, alloc, g_{\mathrm{r}}, fst(alloc(g_{\mathrm{r}}))).$$

So, looking into the definition of $vmatch_{\mathrm{gvar}}$, it remains to show that the allocated region of g_{l} in $d'.mm$ contains the same values as the allocated region of g_{r} in the old assembly memory $d.mm$. This is trivially true by the precondition on $d'.mm$. q.e.d.

10.3 Conditional

For the correctness proofs of the different $C0$ statements we usually argue in two levels. At the lower level we argue that the generated assembly code manipulates memory and registers in a certain way. These proofs use the verification methodology from Section 6.4. We will mostly omit these proofs here and just mention the statement of the corresponding lemmas. At the upper level we then show that the low-level behavior of the assembly code achieves consistency.

In this section we will exemplarily present the statements of the low-level lemmas in details. Later, we will handle these as briefly as possible. Even in this section, we will omit the proofs of the low-level lemmas. In the following sections we will mostly just summarize these kind of lemmas in the proofs of the top-level lemmas.

Lemma 10.24 (Conditionals low-level: *true* case) *Let the C0 configuration c and the assembly configuration d be consistent with respect to the allocation function alloc. Furthermore, let the first statement in the program rest of c be a conditional statement and let the value of its condition e be true.*

Then, given that some low-level preconditions hold, we will eventually reach an assembly configuration d' in which the program counter points behind the code of the first nop instruction, i.e., to the start of the code for the if-part (cf. Figure 7.8 on page 152 which illustrates the code generation for conditional

statements). Formally, this is stated by the following formula.

$$
\begin{aligned}
&\mathit{consis}(te, ft, c, \mathit{alloc}, d) \wedge c \in \mathit{conf}\surd(te, ft) \wedge (te, ft, gst) \in \mathit{xltbl}_{\mathit{prog}} \\
&\wedge\ \mathit{hd}(\mathit{s2l}(c.\mathit{prog})) = \mathit{Ifte}(e, s_1, s_2) \wedge \delta_{C0}(te, ft, c) = \lfloor c' \rfloor \\
&\wedge\ \mathit{addr}_{\mathit{max}} \leq 2^{32} \wedge \mathit{avail}_{\mathit{mem}}(\mathit{addr}_{\mathit{max}}, te, ft, c) \\
&\wedge\ \mathit{asm}_{\mathit{pre}}(d, \mathit{range}_c, \mathit{code}_{\mathit{stmt}}(\mathit{hd}(\mathit{s2l}(c.\mathit{prog})))) \\
&\wedge\ \mathit{reval}(te, mc, e)(0) = \mathit{Bool}(\mathit{true}) \\
\Longrightarrow \exists t, d' :\quad & d \xrightarrow[\mathit{range}_c, \mathit{range}_a]{t, the(cbase_s(hd(s2l(c.prog))))+4\cdot csize_e(e)+8} d' \\
&\wedge d'.\mathit{mm} = d.\mathit{mm} \\
&\wedge d'.\mathit{spr} = d.\mathit{spr} \\
&\wedge \forall r \in \{r_{\mathit{sbase}}, r_{\mathit{htop}}, r_{\mathit{lframe}}, r_{\mathit{jal}}\} : d'.\mathit{gpr}!r = d.\mathit{gpr}!r
\end{aligned}
$$

Lemma 10.25 (Conditionals low-level: *false* case) *Let the C0 configuration c and the assembly configuration d be consistent with respect to the allocation function alloc, let the first statement in the program rest of c be a conditional, and let the value of condition e be false.*

Then, given that some low-level preconditions hold, we will eventually reach an assembly configuration d′ in which the program counter points behind the code of the second nop instruction, i.e., to the start of the code for the else-part (cf. Figure 7.8 on page 152).

$$
\begin{aligned}
&\mathit{consis}(te, ft, c, \mathit{alloc}, d) \wedge c \in \mathit{conf}\surd(te, ft) \wedge (te, ft, gst) \in \mathit{xltbl}_{\mathit{prog}} \\
&\wedge\ \mathit{hd}(\mathit{s2l}(c.\mathit{prog})) = \mathit{Ifte}(e, s_1, s_2) \wedge \delta_{C0}(te, ft, c) = \lfloor c' \rfloor \\
&\wedge\ \mathit{addr}_{\mathit{max}} \leq 2^{32} \wedge \mathit{avail}_{\mathit{mem}}(\mathit{addr}_{\mathit{max}}, te, ft, c) \\
&\wedge\ \mathit{asm}_{\mathit{pre}}(d, \mathit{range}_c, \mathit{code}_{\mathit{stmt}}(\mathit{hd}(\mathit{s2l}(c.\mathit{prog})))) \\
&\wedge\ \mathit{reval}(te, mc, e)(0) = \mathit{Bool}(\mathit{false}) \\
\Longrightarrow \exists t, d' :\quad & d \xrightarrow[\mathit{range}_c, \mathit{range}_a]{t, the(cbase_s(hd(s2l(c.prog))))+4\cdot csize_e(e)+8+4\cdot the(csize_s(s_1))+8} d' \\
&\wedge d'.\mathit{mm} = d.\mathit{mm} \\
&\wedge d'.\mathit{spr} = d.\mathit{spr} \\
&\wedge \forall r \in \{r_{\mathit{sbase}}, r_{\mathit{htop}}, r_{\mathit{lframe}}, r_{\mathit{jal}}\} : d'.\mathit{gpr}!r = d.\mathit{gpr}!r
\end{aligned}
$$

Theorem 10.26 (Correctness of conditionals)
Let (te, ft, gst) be a translatable C0 program, and let the valid C0 configuration c and the valid assembly configuration d be consistent with respect to the allocation function alloc. Let the head of the current program rest $s = \mathit{hd}(\mathit{s2l}(c.\mathit{prog}))$ be a conditional statement. Furthermore, assume that there is sufficient memory and that the preconditions for the execution of $\mathit{code}_{\mathit{stmt}}(s)$ are fulfilled.

Then, we will eventually reach an assembly configuration d′ which is consis-

tent with the next C0 configuration c'.

$$
\begin{aligned}
& consis(te, ft, c, alloc, d) \wedge c \in conf\surd(te, ft) \wedge (te, ft, gst) \in xltbl_{prog} \\
& \wedge\, is\_Ifte(s) \wedge \delta_{C0}(te, ft, c) = \lfloor c' \rfloor \\
& \wedge\, addr_{max} \leq 2^{32} \wedge avail_{mem}(addr_{max}, te, ft, c) \\
& \wedge\, asm_{pre}(d, range_c, code_{stmt}(s)) \\
\Longrightarrow \exists t, d' : \quad & d \xrightarrow[range_c, range_a]{t, d'.dpc} d' \\
& \wedge consis(te, ft, c', alloc, d') \\
& \wedge d'.spr = d.spr
\end{aligned}
$$

Proof We prove this theorem by case distinction on the value of e. For both cases, the only interesting proof goal is to show control consistency; data consistency is preserved because neither the $C0$ machine nor the assembly machine alter their memory.

Case 1: *reval* (*te*, mc, e)(0) = *Bool* (*true*)
In this case we use Lemma 10.24 to show that eventually the program counter points behind the first *nop* instruction (cf. Figure 7.8 on page 152).

If the if branch of s is not empty we are mostly done; we just have to show that the code for the first statement in this branch starts directly behind the first *nop* instruction.

If the if branch is empty we first show that the jump instruction behind the if branch works correctly, i.e., that the program counter eventually points directly behind the code of s. Then, we use Theorem 10.5 to derive control consistency.

Case 2: *reval* (*te*, mc, e)(0) = *Bool* (*false*)
In this case we use Lemma 10.25 to show that eventually the program counter points behind the second *nop* instruction (cf. Figure 7.8 on page 152).

If the else branch is not empty we are mostly done; the arguments are similar to the corresponding case of the if branch.

If the else branch is empty we apply Theorem 10.5 to show that we eventually reach the code base of the successor statement and can derive control consistency. q.e.d.

10.4 Loop

Similar to the previous section we divide the correctness proof for *Loop* statements into two levels. Two lemmas argue about the low-level correctness of the code in case that the condition is *true* or *false*, respectively. The top-level theorem then combines this into a statement that the code for loops preserves the simulation relation.

Lemma 10.27 (Loops low-level: *true* case) *Assume that the C0 configuration c and the assembly configuration d are consistent with respect to the*

allocation function alloc. Further, let the first statement in the program rest of c be a loop Loop(e, lb), and let the value of the loop condition e be true.

Then, given that some low-level preconditions hold, we will eventually reach an assembly configuration d′ in which the program counter points to the first instruction of the loop body:

$$\begin{aligned}\exists d', t: \quad & d \xrightarrow[range_c, range_a]{t, the(cbase_s(hd(s2l_{ns}(lb))))} d' \\ & \wedge d'.mm = d.mm \\ & \wedge d'.spr = d.spr \\ & \wedge \forall r \in \{r_{sbase}, r_{htop}, r_{lframe}, r_{jal}\} : d'.gpr!r = d.gpr!r\end{aligned}$$

Lemma 10.28 (Loops low-level: *false* case) *With the same preconditions as in the previous lemma except that the value of the loop condition e is false we will eventually reach an assembly configuration d′ in which the program counter points directly behind the code of the loop.*

$$\begin{aligned}\exists d', t: \quad & d \xrightarrow[range_c, range_a]{t, the(cbase_s(hd(s2l_{ns}(lb))))+4 \cdot csize_s(Loop(e,lb))} d' \\ & \wedge d'.mm = d.mm \\ & \wedge d'.spr = d.spr \\ & \wedge \forall r \in \{r_{sbase}, r_{htop}, r_{lframe}, r_{jal}\} : d'.gpr!r = d.gpr!r\end{aligned}$$

Theorem 10.29 (Correctness of loops)
Let (te, ft, gst) be a translatable C0 program, and let the valid C0 configuration c and the valid assembly configuration d be consistent with respect to the allocation function alloc. Let the head of the current program rest s = hd(s2l(c.prog)) be a loop statement. Furthermore, assume that there is sufficient memory and that the preconditions for the execution of code_stmt(s) are fulfilled.

Then, we will eventually reach an assembly configuration d′ which is consistent with the next C0 configuration c′.

$$\begin{aligned}& consis(te, ft, c, alloc, d) \wedge c \in conf\surd(te, ft) \wedge (te, ft, gst) \in xltbl_{prog} \\ & \wedge is\_Loop(s) \wedge \delta_{C0}(te, ft, c) = \lfloor c' \rfloor \\ & \wedge addr_{max} \leq 2^{32} \wedge avail_{mem}(addr_{max}, te, ft, c) \\ & \wedge asm_{pre}(d, range_c, code_{stmt}(s)) \\ \Longrightarrow \exists t, d': \quad & d \xrightarrow[range_c, range_a]{t, d'.dpc} d' \\ & \wedge consis(te, ft, c', alloc, d') \\ & \wedge d'.spr = d.spr\end{aligned}$$

Proof We prove this theorem by case distinction on the value of e. For all cases, the only interesting proof goal is to show control consistency; data consistency is preserved because neither the $C0$ machine nor the assembly machine alter their memory.

Case 1: *reval* (*te*, mc, e)(0) = *Bool* (*true*)
In this case we use Lemma 10.27 to show that eventually the program counter points to the beginning of the code of the first statement in the loop body. From the definition of the $C0$ transition function we know that $hd(s2l_{\text{ns}}(c'.prog)) = hd(s2l_{\text{ns}}(lb))$. Thus, we have control consistency between c' and d'.

Case 2: *reval* (*te*, mc, e)(0) = *Bool* (*false*)
In this case we use Lemma 10.28 to show that eventually the program counter points directly behind the code of the loop. Then, we apply Theorem 10.5 to show that we eventually reach the code base of the successor statement of the loop. This proves control consistency between c' and d'. q.e.d.

10.5 Assignment

The low-level correctness proofs for assignments are split into two groups. In Section 10.5.1 we handle assignments of basic values which fit into processor registers. In Section 10.5.2 we argue about the correct assignment of larger values. Finally, in Section 10.5.3 we combine both results to the high-level correctness of assignments.

ISABELLE In Isabelle, normal assignments (Ass) and assignments of aggregate literals (Ass_{AL}) share the same constructor. There, the correctness proof for assignments is split into *three* parts and there exists only a single top-level theorem for all three kinds of assignments. Here, we present the correctness proof about assignments of aggregate literals separately in Section 10.6.

10.5.1 Low-Level: Assigning Basic Values

Lemma 10.30 (Correct storage of basic values) *Let r_d and r_s be registers different from $r1$ and let $i2n(d.gpr!r_d) + ofs$ be a valid destination address, e.g., it must be inside the address range but not inside the code range. Further, assume that the preconditions for the execution of $code_{store}(n, r_d, r_s, ofs)$ hold in configuration d.*

Then, we will eventually reach an assembly configuration d' in which the value of register r_s has been written to the memory at address $i2n(d.gpr!r_d)+ofs$.

$$
\begin{aligned}
&asm_{pre}(d, range_c, code_{store}(n, r_d, r_s, ofs))\\
&\wedge r_d \neq r1 \wedge r_s \neq r1\\
&\wedge i2n(d.gpr!r_d) + ofs < 2^{32} \wedge 4 \mid i2n(d.gpr!r_d) + ofs\\
&\wedge (i2n(d.gpr!r_d) + ofs, 4) \subseteq range_a \wedge i2n(d.gpr!r_d) + ofs \notin range_c\\
&\wedge 4 \mid fst(range_c) \wedge 4 \mid snd(range_c) \wedge 0 \leq ofs < 2^{32}\\
\Longrightarrow \exists t, d' :\ & d \xrightarrow[range_c, range_a]{t, d.dpc+4\cdot\lvert code_{store}(n,r_d,r_s,ofs)\rvert} d'\\
&\wedge d'.mm = d.mm\,[(i2n(d.gpr!r_d) + ofs) \div 4 := int2cell(d.gpr!r_s)]\\
&\wedge d'.spr = d.spr\\
&\wedge \forall r \notin \{r1\} : d'.gpr!r = d.gpr!r
\end{aligned}
$$

10.5.2 Low-Level: Big Assignments

The code template for big assignments (cf. Figure 10.1) contains a loop which slightly complicates the low-level verification. In Isabelle, the proof is partitioned into several lemmas. Here, we just give a brief overview on these lemmas; only for the last lemma we will additionally present a formal statement.

1. `big_assignment_code_aux_loop_correct_common`
   This lemma argues about the effect of the loop body (the highlighted part in Figure 10.1). It proves that a single word has been copied and that the destination register r_d and the auxiliary register $r1$ have been updated correctly.

2. `big_assignment_code_aux_loop_correct1`
   This lemma argues about one execution of the loop (all instructions except the initial *addi*) in case that the branch *will not* be taken, i.e., when we have $i2n(d.gpr!r1) = 4$. In this case only a single word has to be copied and we will eventually reach a configuration d' where the program counter points behind $code_{\text{cpy}}(r_d, r_s, n, 0)$, the correct word has been updated in the memory, and only registers in $\{r1, r2, r_d, r_s\}$ have been changed.

3. `big_assignment_code_aux_loop_correct2`
   This lemma argues about one execution of the loop in case that the branch *will* be taken, i.e., $i2n(d.gpr!r1) > 4$. In this case we show that we eventually reach a configuration d' where the program counter points to the load instruction at offset 4, the correct word has been updated in the memory, the destination register r_d has been incremented by 4, the destination and source registers r_d and r_s have both been incremented by 4, the auxiliary register $r1$ has been decremented by 4, and no other registers except $r2$ have been altered.

4. `big_assignment_code_aux_loop_correct`
   This lemma proves the overall correctness of the loop by induction on the number i of words which have to be copied (assuming $i2n(d.gpr!r1) = 4 \cdot i$ and $i > 0$). Depending on the value of $i2n(d.gpr!r1)$ we use the previous two lemmas to verify the induction step. We prove, that we eventually reach a configuration where the program counter points directly behind the code template and i word have been copied from and to the right addresses.

5. `big_assignment_code_aux_correct`
   This lemma adds the initial add instruction to the correctness proof.

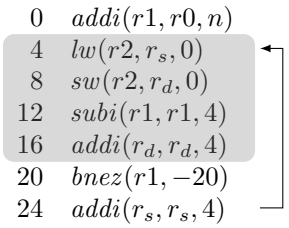
6. `big_assignment_code_correct`
   Finally, this lemma adds the correctness of the offset computation (cf. Definition 7.28 on page 150). Below, we present this lemma in more detail.

Lemma 10.31 (Correct copying of memory regions) *Let n be the number of bytes to be copied (a multiple of four), let r_d and r_s be proper destination and source registers, and let $i2n(d.gpr!r_s)$ and $i2n(d.gpr!r_d) + ofs$ be valid*[2] *source and destination addresses, respectively. Further, assume that the preconditions for the execution of $code_{cpy}(r_d, r_s, n, ofs)$ hold in configuration d.*

Finally, we require that either the source and destination regions do not overlap at all or that they are exactly identical.[3]

[2] i.e., properly aligned, inside the address range, etc.

[3] We need this requirement to show that no parts of the source region which have not yet been copied are being overwritten. Strictly speaking, it would be sufficient to require that the start address of the destination region is not greater than the start of the source region or that it is greater than the last address of the source region ($(i2n(d.gpr!r_d) + ofs) \div 4 \leq$



Listing 10.1: Code for Copying Memory Regions: $code^{\text{loop}}_{\text{cpy}}(r_d, r_s, n)$

Then, we will eventually reach an assembly configuration d' in which the content $data = read_{data}(d.mm, i2n(d.gpr!r_s) \div 4, n \div 4)$ of the source memory region has been copied to the destination region $((i2n(d.gpr!r_d) + ofs) \div 4, n \div 4)$.

$$
\begin{aligned}
&asm_{pre}(d, range_c, code_{cpy}(r_d, r_s, n, ofs)) \\
&\wedge 3 < r_d < 32 \wedge r_s \in fregs_{ini} \wedge r_s \neq r_d \wedge 0 \leq ofs < 2^{32} \\
&\wedge 4 \mid i2n(d.gpr!r_d) + ofs \wedge 4 \mid i2n(d.gpr!r_s) \wedge 4 \mid n \wedge 4 \leq n < 2^{15} \\
&\wedge (i2n(d.gpr!r_d) + ofs, n) \subseteq range_a \wedge (i2n(d.gpr!r_s), n) \subseteq range_a \\
&\wedge i2n(d.gpr!r_s) + n < 2^{32} \wedge i2n(d.gpr!r_d) + ofs + n < 2^{32} \\
&\wedge \begin{pmatrix} (i2n(d.gpr!r_d) + ofs, n) \asymp (i2n(d.gpr!r_s), n) \\ \quad \vee\ i2n(d.gpr!r_d) + ofs = i2n(d.gpr!r_s)) \end{pmatrix} \\
&\wedge 4 \mid fst(range_c) \wedge 4 \mid snd(range_c) \wedge 4 < snd(range_c) \leq i2n(d.gpr!r_d) + ofs \\
&\Longrightarrow \exists t, d' : \quad d \xrightarrow[range_c, range_a]{t, d.dpc + \mid code_{cpy}(r_d, r_s, n, ofs) \mid} d' \\
&\qquad \wedge d'.mm = d.mm\,([(i2n(d.gpr!r_d) + ofs) \div 4, n \div 4] := data) \\
&\qquad \wedge d'.spr = d.spr \\
&\qquad \wedge \forall r \notin \{r1, r2, r3, r_d, r_s\} : d'.gpr!r = d.gpr!r \\
&\qquad \wedge (ofs \neq 0 \longrightarrow d'.gpr!r_d = d.gpr!r_d)
\end{aligned}
$$

10.5.3 High-Level Correctness

The high-level correctness proof for assignments is divided into four parts.

1. In Lemmas 10.32 and 10.33 we show that – given a lot of technical preconditions – the assignment code from Definition 7.29 on page 150 preserves allocation, value, and pointer consistency. We have split this into two lemmas here: one about elementary assignments and one about big assignments. In Isabelle both lemmas are merged into a single one. These lemmas are independent from assignment statements. We will later reuse them in the correctness proofs for function calls and returns.

2. In Lemma 10.34, we specialize the previous lemmas to prove that, starting from data consistent configurations, the execution of the assignment code

$i2n(d.gpr!r_s) \div 4 \vee i2n(d.gpr!r_s) \div 4 + n \div 4 < (i2n(d.gpr!r_d) + ofs) \div 4)$. However, for big assignments of $C0$ programs we can show the stronger assumption from the lemma.

(without the code for expression evaluation) eventually reaches an assembly configuration which is data consistent with the next $C0$ configuration and in which return addresses have not been messed up. In this lemma we assume the expression evaluation to be done correctly.

3. In Lemma 10.35, we combine this lemma with the correct evaluation of the left and right expression of the assignment.

4. In Theorem 10.36, we add a proof for control consistency.

We start with a *general* lemma about the high-level correctness of the assignment code $code_{\text{ass}}$ for the case of elementary assignments. This lemma is general because it is not fixed to assignment *statements* but will be reused in the proofs for function call and return statements.

Lemma 10.32 (Correct assignment code: elementary values) *Let c be a valid $C0$ configuration which is value, pointer, and allocation consistent with assembly configuration d via the allocation function alloc and assume that the preconditions for the execution of the assignment code are fulfilled. Moreover, assume that source register r_s and destination register r_d are valid, that the $C0$ value v matches the content of r_s, and that r_d plus an offset ofs is the allocated address of a reachable elementary g-variable g_l with a type t. Finally, let the content of v be type correct with respect to t and contain only reachable pointers.*

Then, the assembly machine will eventually reach a configuration d' in which the the allocated address of g_l has been updated with the content of r_s, where the program counter points directly behind the assignment code, and which is value, pointer, and allocation consistent with the new configuration where a memory update of g_l with value v has happened. Furthermore, only auxiliary registers and r_s and (if $ofs = 0$) r_d have been changed.

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge memupd(c.mem, g_l, v) = \lfloor m' \rfloor \\
& \wedge consis_v(c, alloc, d) \wedge consis_p(c, alloc, d) \wedge consis_{alloc}(te, ft, c, alloc) \\
& \wedge asm_{pre}(d, range_c, code_{ass}(false, r_d, r_s, asize_t(t), ofs)) \\
& \wedge r_s \in fregs_{ini} \\
& \wedge (ofs = 0 \longrightarrow r_d \in fregs_{ini} \cup \{r3\}) \wedge (ofs \neq 0 \longrightarrow 3 < r_d < 32) \\
& \wedge tmatch_{ass}(ty_g(sc(c.mem), g_l), t) \wedge vmatch_{rval}(alloc, v, d.gpr!r_s) \\
& \wedge g_l \in reachable_g(c.mem) \wedge (fst(alloc(g_l)), asize_t(t)) \subseteq range_a \\
& \wedge i2nd.gpr!r_d + ofs = fst(alloc(g_l)) \\
& \wedge ty\surd(te, sc(c.mem), v, t, 0) \wedge \forall i < size_t(t) : reachable_{mcell}(c.mem, v(i)) \\
& \wedge 4 \mid d.dpc \wedge fst(alloc(g_l)) < 2^{32} \wedge 0 \leq ofs < 2^{32} \\
\Longrightarrow \exists t, d' : \quad & d \xrightarrow[range_c, range_a]{t, d.dpc + 4 \cdot |code_{ass}|} d' \\
& \wedge consis_v(c[mem := m'], alloc, d') \\
& \wedge consis_p(c[mem := m'], alloc, d') \\
& \wedge consis_{alloc}(te, ft, c[mem := m'], alloc) \\
& \wedge memchngd(d.mm, d'.mm, fst(alloc(g_l)) \div 4, snd(alloc(g_l)) \div 4) \\
& \wedge d'.spr = d.spr \\
& \wedge \forall r \notin \{r1, r2, r3, r_d, r_s\} : d'.gpr!r = d.gpr!r \\
& \wedge ofs \neq 0 \longrightarrow d'.gpr!r_d = d.gpr!r_d
\end{aligned}
$$

PROOF To prove this lemma we have to apply the low-level Lemma 10.30. For this, we have to discharge numerous preconditions. After applying the low-level assignment lemmas, we know that the correct memory region of the assembly machine has been updated with the correct value without messing up the registers.

To prove value and pointer consistency we use Lemmas 10.13, 10.14, and Corollary 10.15. This proves that all elementary g-variables (inside and outside of the updated region) are value and pointer consistent after the update.

Finally, we have to show that only the correct part of the assembly memory and only the right registers have been altered. This follows directly from the low-level correctness. q.e.d.

The next lemma is similar to the previous one except that it argues about big assignments.

Lemma 10.33 (Correct assignment code: big values) *Assume the same preconditions as in Lemma 10.32 with the exception that the value to be copied is not elementary. Thus, r_s does not contain the value but the allocated address of a g-variable g_r whose content will be copied.*

Then, the assembly machine will eventually reach a configuration d' in which the the allocated address of g_l has been updated with the content at the allocated address of g_r, where the program counter points directly behind the assignment code, and which is value, pointer, and allocation consistent with the new configuration where a memory update of g_l with value of g-variable g_r. Furthermore, only auxiliary registers and r_s and (if $ofs = 0$) r_d have been changed.

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge memupd(c.mem, g_l, v) = \lfloor m' \rfloor \\
& \wedge consis_v(c, alloc, d) \wedge consis_p(c, alloc, d) \wedge consis_{alloc}(te, ft, c, alloc) \\
& \wedge asm_{pre}(d, range_c, code_{ass}(true, r_d, r_s, asize_t(t), ofs)) \\
& \wedge r_s \in fregs_{ini} \\
& \wedge (ofs = 0 \longrightarrow r_d \in fregs_{ini} \cup \{r3\}) \wedge (ofs \neq 0 \longrightarrow 3 < r_d < 32) \\
& \wedge tmatch_{ass}(ty_g(sc(c.mem), g_l), t) \\
& \wedge g_l \in reachable_g(c.mem) \wedge (fst(alloc(g_l)), asize_t(t)) \subseteq range_a \\
& \wedge g_r \in reachable_g(c.mem) \wedge (fst(alloc(g_r)), asize_t(t)) \subseteq range_a \\
& \wedge i2nd.gpr!r_d + ofs = fst(alloc(g_l)) \wedge i2nd.gpr!r_s = fst(alloc(g_r)) \\
& \wedge 4 \mid d.dpc \wedge fst(alloc(g_l)) + asize_t(t) < 2^{32} \wedge fst(alloc(g_r)) + asize_t(t) < 2^{32} \\
& \wedge 0 \leq ofs < 2^{32} \wedge -2^{15} \leq asize_t(t) < 2^{15} \wedge r_d \neq r_s \\
& \wedge v = value_g(c.mem, g_r) \wedge initialized_g(c.mem, g_r) \\
\Longrightarrow \exists t, d' : \quad & d \xrightarrow[range_c, range_a]{t, d.dpc + 4 \cdot |code_{ass}|} d' \\
& \wedge consis_v(c[mem := m'], alloc, d') \\
& \wedge consis_p(c[mem := m'], alloc, d') \\
& \wedge consis_{alloc}(te, ft, c[mem := m'], alloc) \\
& \wedge memchngd(d.mm, d'.mm, fst(alloc(g_l)) \div 4, snd(alloc(g_l)) \div 4) \\
& \wedge d'.spr = d.spr \\
& \wedge \forall r \notin \{r1, r2, r3, r_d, r_s\} : d'.gpr!r = d.gpr!r \\
& \wedge ofs \neq 0 \longrightarrow d'.gpr!r_d = d.gpr!r_d
\end{aligned}
$$

PROOF To prove the lemma for big assignments we use Lemma 10.31. Besides this, we argue similarly to the previous lemma. q.e.d.

Now, we combine the general lemmas about elementary and big assignments and specialize them for the case of assignment statements. Many of the preconditions from the previous lemmas are now implicitly given by more general preconditions; thus, the list of preconditions is much shorter than before.

Lemma 10.34 (Assignments: data consistency) *Let the valid C0 configuration c be data and return address consistent with assembly configuration d via the allocation function alloc and let the head of the current program rest be a translatable assignment statement. We abbreviate the type of the left expression by*

$$t_l = the(type(te, gst(c.mem), lst_{top}(c.mem), e_l))$$

and the code which is generated for the assignment by

$$il = code_{ass}(\neg elem?_t(t_l), hd(fregs_{ini}), hd(tl(fregs_{ini})), asize_t(lt), 0).$$

Furthermore, assume that there is sufficient memory, that the next transition of the C0 machine does not fail, that the preconditions for the execution of $code_{stmt}(hd(s2l(c.prog)))$ are fulfilled, and that expressions e_l and e_r have already been correctly evaluated into the registers $hd(fregs_{ini})$ and $hd(tl(fregs_{ini}))$, respectively.

Then, the assembly machine will eventually reach a configuration where the program counter points directly behind the assignment code and which is data and return address consistent with c'. Furthermore, the special registers have not been altered.

$$\begin{aligned}
&c \in conf\surd(te, ft) \wedge consis_d(te, ft, c, alloc, d) \wedge consis_{ra}(te, ft, c, d)\\
&\wedge hd(s2l(c.prog)) = Ass(e_l, e_r) \wedge \delta_{C0}(te, ft, c) = \lfloor c' \rfloor\\
&\wedge addr_{max} \leq 2^{32} \wedge avail_{mem}(addr_{max}, te, ft, c)\\
&\wedge Ass(e_l, e_r) \in xltbl_{stmt}(te, ft, gst(c.mem), lst_{top}(c.mem))\\
&\wedge expr_{result}\surd(te, c, alloc, e_l, false, d.gpr!(hd(fregs_{ini})))\\
&\wedge expr_{result}\surd(te, c, alloc, e_r, elem?_t(t_l), d.gpr!(hd(tl(fregs_{ini}))))\\
&\wedge asm_{pre}(d, range_c, il)\\
\Longrightarrow \exists t, d' :\ & d \xrightarrow[range_c, range_a]{t, d.dpc+4\cdot|il|} d'\\
&\wedge consis_d(te, ft, c', alloc, d')\\
&\wedge consis_{ra}(te, ft, c', d')\\
&\wedge d'.spr = d.spr\\
&\wedge \forall r \in \{r_{sbase}, r_{htop}, r_{lframe}, r_{jal}\} : d'.gpr!r = d.gpr!r
\end{aligned}$$

PROOF Before we actually start with the proof we apply Lemma 5.1 on page 69 which allows us to unfold the definition of the $C0$ transition function without agonizing over leading *Skip* statements.

Then, depending on whether we assign elementary or big types we use either Lemma 10.32 or 10.33. In both cases we have to discharge numerous preconditions; e.g., we use Theorem 8.1 to show that the g-variables are reachable which allows use to prove – using Lemma 10.6 – that their allocated base address is a multiple of four.

After applying these lemmas we have value, pointer, and allocation consistency; not for the new configuration c' but for some auxiliary configuration $c[mem := c'.mem]$, e.g., $consis_{\mathrm{v}}(c[mem := c'.mem], alloc, d')$. This is not exactly what we need. However, value and pointer consistency does not depend on the program rest, so we are done for these.

It remains to prove register, frame header, and return address consistency. For return address consistency we first apply Lemma 10.19 which ensures $consis_{\mathrm{ra}}(te, ft, c, d')$. However, the successors of return statements in the program rest as well as the recursion depth have not changed between c and c', thus we can conclude $consis_{\mathrm{ra}}(te, ft, c', d')$. With the same arguments we conclude frame header consistency from Lemma 10.18.

To prove register consistency we mainly need to show (besides the fact that the special registers are not changed) that every reachable g-variable is allocated below the top heap address:

$$\forall g \in reachable_{\mathrm{g}}^{\mathrm{nameless}}(c'.mem) : \\ fst(alloc(g)) + snd(alloc(g)) \leq i2n(d.gpr!r_{\mathrm{htop}}).$$

Since neither the allocation function nor the top heap address have changed in-between d and d' we just need to show that all g-variables g which are reachable in c' have also been reachable in c. This follows from Theorem 8.2 and allows us to conclude the property from register consistency in the old configuration c. q.e.d.

Lemma 10.35 (Assignment including expression evaluation) *Let c be a valid C0 configuration which is consistent with assembly configuration d via allocation function alloc, let (te, ft, gst) be a translatable C0 program, and let the head of the current program rest $s = hd(s2l(c.prog))$ be an assignment statement. Again, we assume that there is sufficient memory, that the next transition of the C0 machine does not fail, and that the preconditions for the execution of $code_{stmt}(s)$ are fulfilled.*

Then, the assembly machine eventually reaches a configuration d' where the program counter points directly behind the code of the assignment statement and which is data and return address consistent with c'. Furthermore, the special registers have not been altered between d and d'.

$$\begin{aligned} & c \in conf\surd(te, ft) \wedge consis(te, ft, c, alloc, d) \wedge (te, ft, gst(c.mem)) \in xltbl_{prog} \\ & \wedge is\_Ass(s) \wedge \delta_{C0}(te, ft, c) = \lfloor c' \rfloor \\ & \wedge addr_{max} \leq 2^{32} \wedge avail_{mem}(addr_{max}, te, ft, c) \\ & \wedge asm_{pre}(d, range_c, code_{stmt}(te, ft, gst(c.mem), lst_{top}(c.mem), s)) \\ & \Longrightarrow \exists t, d' : \quad d \xrightarrow[range_c, range_a]{t, the(cbase_s(s)) + 4 \cdot the(csize_s(s))} d' \\ & \qquad \wedge consis_d(te, ft, c', alloc, d') \\ & \qquad \wedge consis_{ra}(te, ft, c', d') \\ & \qquad \wedge d'.spr = d.spr \\ & \qquad \wedge \forall r \in \{r_{sbase}, r_{htop}, r_{lframe}, r_{jal}\} : d'.gpr!r = d.gpr!r \end{aligned}$$

Proof First, we have to show – using Theorem 9.2 and Lemma 9.1 – that the evaluation of the left and right expression works correctly.

Now, we combine the execution of both expressions using Lemma 6.9 on page 118 and obtain a new (intermediate) assembly configuration d'' in which the results of both expressions are stored in registers. Using Lemma 6.13 we conclude that in d'' the precondition asm_{pre} for the actual assignment code holds.[4]

Next, we apply Lemma 10.34 which provides us with an assembly configuration d' for the existential quantifier in the proof goal (additionally, we instantiate t with the sum of the step numbers for the execution of both expressions and the assignment code). This completes the proof of data and return address consistency and shows that no registers are unexpectedly changed.

It remains to combine using Lemma 6.9 the execution from d to d'' with the execution from d'' to d'. q.e.d.

Theorem 10.36 (Correctness of assignments)
Let (te, ft, gst) be a translatable C0 program, let the valid C0 configuration c be consistent with assembly configuration d with respect to the allocation function alloc, and let the head of the current program rest $s = hd(s2l(c.prog))$ be an assignment statement. Assume that there is sufficient memory, that the next transition of the C0 machine does not fail, and that the preconditions for the execution of $code_{stmt}(s)$ are fulfilled.

Then, the assembly machine eventually reaches a configuration d' which is completely consistent with c'. Furthermore, the special purpose registers have not been altered.

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge consis(te, ft, c, alloc, d) \wedge (te, ft, gst(c.mem)) \in xltbl_{prog} \\
& \wedge is\_Ass(s) \wedge \delta_{C0}(te, ft, c) = \lfloor c' \rfloor \\
& \wedge addr_{max} \leq 2^{32} \wedge avail_{mem}(addr_{max}, te, ft, c) \\
& \wedge asm_{pre}(d, range_c, code_{stmt}(te, ft, gst(c.mem), lst_{top}(c.mem), s)) \\
& \Longrightarrow \exists t, d' : \quad d \xrightarrow[range_c, range_a]{t, d'.dpc} d' \\
& \qquad\qquad \wedge consis(te, ft, c', alloc, d') \\
& \qquad\qquad \wedge d'.spr = d.spr
\end{aligned}
$$

Proof We start by applying Lemma 10.35 which proves data and return address consistency. So, we are left with control consistency which depends on the correctness of control code (from conditionals or while loops) that possibly follows $code_{\text{stmt}}(s)$.

The correctness of this kind of control code has already been verified in Section 10.1.2. Thus, using Theorem 10.5, we can easily conclude that we will eventually reach an assembly configuration which is additionally control consistency with c'. q.e.d.

10.6 Assignment of Aggregate Literals

The high-level parts of the correctness proof for assignments of aggregate literals are similar to those for normal assignments. The low-level proof, however, is

[4] These fine-grained arguments about execution of assembly machines should give the reader some impression about (hidden) work in the formal proof system. In the following proofs we will not mention the usage of lemmas from Section 6.4.3 in as much detail as here.

completely different. The following lemma proves that the execution of the code which is generated for assignments of aggregate literals stores values in the assembly memory which are content consistent with the original $C0$ literal.

The code generation for aggregate literal assignments works by recursion on the structure of the literal (cf. Section 7.4.3 on page 151). Thus, the lemma has been verified by structural induction. However, the code generation is not defined by a single recursive function, but by two mutual recursive functions.[5] To prove properties of mutual recursive functions it is necessary to do the proof simultaneously for all of them. Thus, the claim of the lemma needs a separate part for each function to be inductive. These parts are very similar but differ in some details (e.g., for a structural literal we need to keep track of the additional offset i. However, as we will not give details of this proof, we present here only the main statement for $code_{\mathrm{alit}}$.

Lemma 10.37 (Aggregate assignments: low-level) *Let c be a valid $C0$ configuration and let the assembly code preconditions hold for the code which is generated for an assignment of the valid aggregate literal l to the destination address $i2n(d.gpr!r_d) + ofs$. Additionally, let the usual low-level preconditions hold and abbreviate with $t = type_{alit}(l)$ the type of l.*

Then, we will eventually reach an assembly configuration d' in which the destination memory region is content consistent with the $C0$ value of l for arbitrary g-variables g whose type matches the type of the aggregate literal. Besides the intended change in the destination memory region and two auxiliary registers, no parts of the assembly machine have been changed between d and d'.

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge r_d \in fregs_{ini} \wedge 4 \mid i2n(d.gpr!r_d) + ofs \\
& \wedge\ snd(range_c) \leq i2n(d.gpr!r_d) + ofs \wedge addr_{max} \leq 2^{32} \\
& \wedge\ asm_{pre}(d, range_c, code_{alit}(r_d, ofs, l)) \wedge l \in valid_{alit} \\
& \wedge\ (i2n(d.gpr!r_d) + ofs, asize_t(t)) \subseteq range_a \\
\Longrightarrow\ & \exists t, d' : \quad d \xrightarrow[range_c, range_a]{t, d.dpc+\lvert code_{alit}(r_d, ofs, l)\rvert} d' \\
& \qquad \wedge \forall g : tmatch_{ass}(ty_g(sc(c.mem), g), t) \\
& \qquad\qquad \Longrightarrow vmatch_{gvar}(reval_{alit}(l), d', alloc, g, i2n(d.gpr!r_d) + ofs) \\
& \qquad \wedge memchngd(d.mm, d'.mm, (i2n(d.gpr!r_d) + ofs) \div 4, asize_t(t) \div 4) \\
& \qquad \wedge \forall r \notin [r1, r2] : d'.gpr!r = d.gpr!r \wedge d'.spr = d.spr
\end{aligned}
$$

Proof We prove this lemma by induction on the structure of the literal. For the induction start we combine the correctness of the code for (basic) literal expressions with Lemma 10.30.

Observe that we do not have data consistency during the induction step due to the fact that only a part of the aggregate literal has been copied. There is no corresponding (data consistent) $C0$ configuration for this situation. Although we need data consistency for all other kinds of expressions (cf. Definition 9.3), it is not required for literals and the absence of data consistency poses no problem. The reason for this is simply that the evaluation of literals does not access the memory.

The remaining details of this proof are not interesting, so we omit them here. q.e.d.

[5] In Isabelle, there are *four* such functions.

Lemma 10.38 (Aggregate assignments: high-level) *Let (te, ft, gst) be a translatable $C0$ program, let the valid $C0$ configuration c be consistent with assembly configuration d via allocation function alloc, and let the head of the current program rest $s = hd(s2l(c.prog))$ be an assignment of an aggregate literal. Assume that there is sufficient memory, that the next transition of the $C0$ machine does not fail, and that the preconditions for the execution of $code_{stmt}(s)$ are fulfilled.*

Then, the assembly machine eventually reaches a configuration d' where the program counter points directly behind the code of the assignment statement and which is data and return address consistent with c'. Furthermore, the special registers have not been altered between d and d'.

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge consis(te, ft, c, alloc, d) \wedge (te, ft, gst(c.mem)) \in xltbl_{prog} \\
& \wedge\, is\_Ass_{AL}(s) \wedge \delta_{C0}(te, ft, c) = \lfloor c' \rfloor \\
& \wedge\, addr_{max} \leq 2^{32} \wedge avail_{mem}(addr_{max}, te, ft, c) \\
& \wedge\, asm_{pre}(d, range_c, code_{stmt}(te, ft, gst(c.mem), lst_{top}(c.mem), s)) \\
& \Longrightarrow \exists t, d' : \quad d \xrightarrow[range_c, range_a]{t, the(cbase_s(s)), the(csize_s(s))} d' \\
& \qquad \wedge consis_d(te, ft, c', alloc, d') \\
& \qquad \wedge consis_{ra}(te, ft, c', d') \\
& \qquad \wedge d'.spr = d.spr \\
& \qquad \wedge \forall r \in \{r_{sbase}, r_{htop}, r_{lframe}, r_{jal}\} : d'.gpr!r = d.gpr!r
\end{aligned}
$$

Proof We start out as in Lemma 10.35 by proving that the left expression le of s is correctly compiled and that its allocated address is stored in register $hd(fregs_{\text{ini}})$. Then, we apply Lemma 10.37 – instantiating $ofs = 0$ and $r_d = hd(fregs_{\text{ini}})$ – which gives a step number t and a new assembly configuration d'. Instantiating the g in the lemmas claim with $leval(te, c.mem, le)$ we obtain

$$vmatch_{\text{gvar}}(reval_{\text{alit}}(l), d', alloc, leval(le), fst(alloc(leval(le)))).$$

It remains to prove return address and data consistency. Data consistency follows directly from Theorem 10.22. For return address consistency we use Lemma 10.19. q.e.d.

We combine this lemma with control consistency in exactly the same way as for normal assignments.

Theorem 10.39 (Correctness of assignments of aggregate literals) *Let (te, ft, gst) be a translatable $C0$ program, let the valid $C0$ configuration c be consistent with assembly configuration d with respect to the allocation function alloc, and let the head of the current program rest $s = hd(s2l(c.prog))$ be an assignment statement. Assume that there is sufficient memory, that the next transition of the $C0$ machine does not fail, and that the preconditions for the execution of $code_{stmt}(s)$ are fulfilled.*

Then, the assembly machine eventually reaches a configuration d' which is completely consistent with c'. Furthermore, the special purpose registers have

not been altered.

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge consis(te, ft, c, alloc, d) \wedge (te, ft, gst(c.mem)) \in xltbl_{prog} \\
& \wedge is\_Ass_{AL}(s) \wedge \delta_{C0}(te, ft, c) = \lfloor c' \rfloor \\
& \wedge addr_{max} \leq 2^{32} \wedge avail_{mem}(addr_{max}, te, ft, c) \\
& \wedge asm_{pre}(d, range_c, code_{stmt}(te, ft, gst(c.mem), lst_{top}(c.mem), s)) \\
& \Longrightarrow \exists t, d' : \quad d \xrightarrow[range_c, range_a]{t, d'.dpc} d' \\
& \qquad\qquad \wedge consis(te, ft, c', alloc, d') \\
& \qquad\qquad \wedge d'.spr = d.spr
\end{aligned}
$$

10.7 Allocation of Dynamic Memory

In this section we prove the correct compilation of *PAlloc* statements. For the low-level correctness, we present in Section 10.7.1 only the two most important lemmas; in Isabelle, there are several more Lemmas which argue separately about the correctness of $code^{\text{test}}_{\text{alloc}}$, $code_{\text{zerofill}}$, and $code^{\text{zero}}_{\text{alloc}}$. The correctness proofs for high-level correctness are given in Section 10.7.2.

10.7.1 Low-Level Correctness

We present two low-level lemmas about the correctness of the code for memory allocation on the heap. The first lemma comprises the case that allocation *fails* because of insufficient heap memory. The second lemma argues about the correctness of *successful* allocation. For both lemmas we omit the proofs.

Lemma 10.40 (Low-level correctness: failure) *Let n be the allocated size of the newly allocated g-variable, let register r_d contain the address where the new pointer should be stored (i.e., the address of the left side expression), and let the preconditions for the assembly execution of the heap allocation code hold for configuration d. Additionally, assume that registers r_d and r_{htop} contain multiples of four, and that the allocated memory region for the left side expression is properly located in the memory (e.g., does not overlap with the code region). Finally, assume that allocation fails, i.e., that $d.gpr!r_{htop} + \lceil n \rceil_4$ exceeds the maximum heap address.*[6]

Then, we will eventually reach an assembly configuration d' in which the program counter points directly behind the heap allocation code, the memory at address $d.gpr!r_d$ has been updated with a null pointer, and no registers besides

[6] Observe that we align n to multiples of four to ensure that the top heap address is properly aligned for all types (cf. Section 7.4.6).

the auxiliary registers have been changed.

$$
\begin{aligned}
&asm_{pre}(d, range_c, code^{test}_{alloc}(n, r2) \circ code^{zero}_{alloc}(n, r_d, r2)) \\
&\wedge 4 \mid i2n(d.gpr!r_{htop}) \wedge 4 \mid i2n(d.gpr!r_d) \wedge 0 \leq \lceil n \rceil_4 < 2^{32} \\
&\wedge snd(range_c) \leq i2n(d.gpr!r_d) < 2^{32} \\
&\wedge (i2n(d.gpr!r_d), 4) \subseteq range_a \wedge r_d \in fregs_{ini} \\
&\wedge \lceil n \rceil_4 < abase_{heap} + asize^{max}_{heap} \wedge 0 \leq abase_{heap} + asize^{max}_{heap} < 2^{32} \\
&\wedge i2n(d.gpr!r_{htop}) + \lceil n \rceil_4 \geq abase_{heap} + asize^{max}_{heap} \\
&\Longrightarrow \exists t, d' : \quad d \xrightarrow[range_c, range_a]{t, d.dpc+64} d' \\
&\qquad\qquad \wedge \forall r \notin [r1, r2, r3] : d'.gpr!r = d.gpr!r \\
&\qquad\qquad \wedge d'.mm = d.mm\,[i2n(d.gpr!r_d) \div 4 := int2cell(0)] \\
&\qquad\qquad \wedge d'.spr = d.spr
\end{aligned}
$$

Lemma 10.41 (Low-level correctness: success) *Let, as in the previous lemma, n be the allocated size of the newly allocated g-variable, let r_d contain the destination address, and let the preconditions for the assembly execution of the heap allocation code hold in configuration d. Assume that registers r_d and r_{htop} contain multiples of four, and that the allocated memory regions for the left side expression and for the newly allocated variable are properly located in the memory (e.g., they do not overlap with the code region). Finally, assume that allocation succeeds, i.e., that $d.gpr!r_{htop} + \lceil n \rceil_4$ does not exceed the maximum heap address.*

Then, we will eventually reach an assembly configuration d' in which the program counter points directly behind the heap allocation code, the heap top register has been incremented by $\lceil n \rceil_4$, and the memory has been updated in two steps. First, address $d.gpr!r_d$ has been updated with a pointer to the newly allocated heap variable. We abbreviate this intermediate *memory with $mm' = d.mm\,[i2n(d.gpr!r_d) \div 4 := int2cell(d.gpr!r_{htop})]$. Second, the allocated region of the new heap variable has been initialized with zeroes. Finally, no registers besides r_{htop} and the auxiliary registers have been changed.*

$$
\begin{aligned}
&asm_{pre}(d, range_c, code^{test}_{alloc}(n, r2) \circ code^{zero}_{alloc}(n, r_d, r2)) \\
&\wedge 4 \mid i2n(d.gpr!r_{htop}) \wedge 4 \mid i2n(d.gpr!r_d) \wedge 0 < \lceil n \rceil_4 < 2^{32} \\
&\wedge snd(range_c) \leq i2n(d.gpr!r_d) < 2^{32} \\
&\wedge (i2n(d.gpr!r_d), 4) \subseteq range_a \wedge (i2n(d.gpr!r_{htop}), \lceil n \rceil_4) \subseteq range_a \\
&\wedge snd(range_c) \leq i2n(d.gpr!r_{htop}) \wedge d.gpr!r_{htop} \neq 0 \\
&\wedge i2n(d.gpr!r_{htop}) + \lceil n \rceil_4 < 2^{32} \\
&\wedge 0 \leq abase_{heap} + asize^{max}_{heap} < 2^{32} \wedge r_d \in fregs_{ini} \\
&\wedge i2n(d.gpr!r_{htop}) + \lceil n \rceil_4 < abase_{heap} + asize^{max}_{heap} \\
&\Longrightarrow \exists t, d' : \quad d \xrightarrow[range_c, range_a]{t, d.dpc+64} d' \\
&\qquad\qquad \wedge d'.spr = d.spr \wedge \forall r \notin [r_{htop}, r1, r2, r3] : d'.gpr!r = d.gpr!r \\
&\qquad\qquad \wedge i2nd'.gpr!r_{htop} = i2nd.gpr!r_{htop} + \lceil n \rceil_4 \\
&\qquad\qquad \wedge d'.mm = \\
&\qquad\qquad\quad mm'\,([i2nd.gpr!r_{htop} \div 4, \lceil n \rceil_4] := replicate(\lceil n \rceil_4, int2cell(0)))
\end{aligned}
$$

10.7.2 High-Level Correctness

For the high-level correctness of allocation we need to extend the allocation function with an entry for the new heap variable which contains its allocated base address and size (cf. Definition 10.1).

Using this new allocation function, we will then show in Lemma 10.44 data and return address consistency. For the case that allocation is successful, the proof depends on Lemma 10.42 which argues that sub g-variables of the new heap variable are value and pointer consistent if they are properly initialized with zeroes. Finally, in Theorem 10.45, we add control consistency as in the correctness proof for assignments by using the results from Section 10.1.

Definition 10.1 (Extended allocation function) We define the function $alloc_{\mathrm{xt}} :: (gvar \mapsto (\mathbb{N} \times \mathbb{N})) \times symbolconf \times \mathbb{N} \times \mathbb{N} \mapsto (gvar \mapsto (\mathbb{N} \times \mathbb{N}))$ which extends a given allocation function with values for the newly allocated g-variable.

Let *te* be a type environment, *sc* a symbol configuration, *alloc* the old allocation function, i the index of the new heap g-variable (i.e., the number of variables in the old heap), and b the allocated base address of the new heap variable. Then, we define the new allocation function by

$$alloc_{\mathrm{xt}}(alloc, sc, i, b)(g) = \begin{cases} (b + displ_{\mathrm{g}}(g), asize_{\mathrm{t}}(ty_{\mathrm{g}}(g))) & \text{if } root_{\mathrm{g}}(g) = gvar_{\mathrm{hm}}(i) \\ alloc(g) & \text{otherwise} \end{cases}$$

The following auxiliary lemma shows that sub g-variables of the newly allocated heap variable are value and pointer consistent with the new $C0$ configuration if they are properly initialized with zeroes in the assembly configuration.

Lemma 10.42 *Let d be an assembly configuration, t a valid type, and c a valid $C0$ configuration in which we have sufficient heap memory for the additional allocation of a variable of type t. Further, let g be a sub g-variable of this new heap variable, let g_l be a valid elementary g-variable which does not overlap with g, and assume that the value at the allocated address of g in assembly configuration d is 0.*

We abbreviate the $C0$ memory configuration after heap extension with $m' = extend_{heap}(avail_{heap}, c.mem, t)$ and the extended allocation function with $alloc' = alloc_{xt}(alloc, sc(m'), |hst(c.mem)|, b)$. Finally, we assume that updating g_l in $C0$ memory configuration m' with a pointer to the newly allocated g-variable is successful and gives a new memory configuration m''.

Then, g is value and pointer consistent is configuration d with respect to $C0$

memory m'' and allocation function $alloc'$.

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge root_g(g) = gvar_{hm}(|hst(c.mem)|) \\
& \wedge valid_{ty}(te, t) \wedge avail_{heap}(c.mem, t) \\
& \wedge g_l \in gvars\surd(sc(m')) \wedge size_t(ty_g(sc(m'), g_l)) = 1 \\
& \wedge \neg overlap_g(sc(m'), g_l, g) \\
& \wedge cell2int(d.mm(fst(alloc'(g)) \div 4)) = 0 \\
& \wedge memupd(m', g_l, [Ptr(gvar_{hm}(|hst(c.mem)|))]) = \lfloor m'' \rfloor \\
& \Longrightarrow consis_v^g(m'', alloc', d, g) \wedge consis_p^g(m'', alloc', d, g)
\end{aligned}
$$

Proof The main effort of this proof is to show that the value of g in m'' corresponds to the init value of g, i.e., that

$$value_{\mathrm{g}}(m'', g)(0) = init_{\mathrm{val}}(ty_{\mathrm{g}}(sc(m''), g))!0.$$

Then we know, because 0 matches the init values of all elementary types, that g is value and pointer consistent.

The former is proved as follows.

$$
\begin{aligned}
& value_{\mathrm{g}}(m'', g)(0) \\
& = value_{\mathrm{g}}(m', g)(0) && (g \text{ and } g_{\mathrm{l}} \text{ do not overlap}) \\
& = value_{\mathrm{g}}(m', root_{\mathrm{g}}(g))(ba_{\mathrm{g}}(sc(m'), g) - ba_{\mathrm{g}}(sc(m'), g_{\mathrm{l}})) \\
& = init_{\mathrm{val}}(ty_{\mathrm{g}}(sc(m''), root_{\mathrm{g}}(g)))!(ba_{\mathrm{g}}(sc(m'), g) - ba_{\mathrm{g}}(sc(m'), g_{\mathrm{l}})) \\
& && (\text{correct initialization during } extend_{\mathrm{heap}}) \\
& = init_{\mathrm{val}}(ty_{\mathrm{g}}(sc(m''), g))!0 && (\text{correctness of } init_{\mathrm{val}}) \\
& && \text{q.e.d.}
\end{aligned}
$$

Now, we present a lemma which shows that g-variables which are reachable after allocation are either sub g-variables of the newly allocated variable or have already been reachable in the old configuration. Similar to Theorem 8.2 on page 172, this lemma will be very important in the proof of Lemma 10.44 where we show that reachable g-variables which have not been altered are still value and pointer consistent: we can conclude from the lemma that these variables must have been reachable – and thus consistent – also in the old configuration and because they have not been altered we know that they are *still* consistent.

Lemma 10.43 (Reachable g-variables after allocation) *Let the program rest of the valid C0 configuration c start with an allocation statement and assume that the C0 transition function gives a new configuration c', i.e., that there are no runtime errors. Then, given that g is a reachable g-variable in c', it either has also been reachable in the previous configuration c or it is a sub g-variable of the newly allocated g-variable.*

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge hd(s2l(c.prog)) = PAlloc(e, tn) \wedge \delta_{C0}(te, ft, c) = \lfloor c' \rfloor \\
& \wedge g \in reachable_g(c'.mem) \wedge root_g(g) \neq gvar_{hm}(|hst(c.mem)|) \\
& \Longrightarrow g \in reachable_g(c.mem)
\end{aligned}
$$

Proof We prove this lemma in two steps. The easy part is to show that $g \in reachable_{\mathrm{g}}(extend_{\mathrm{heap}}(avail_{\mathrm{heap}}, c.mem, t))$ implies that g has also been

reachable in c. This can be proved in a similar way as Theorem 8.2 via induction about the definition of reachable g-variables.

For the second part, we abbreviate with $g_\mathrm{l} = leval(te, c.mem, e)$ the g-variable which gets assigned a pointer to the new heap variable, i.e., the left value of the left expression of the allocation statement. We have to prove that every reachable g-variable in the new configuration c' has also been reachable in an intermediate configuration where only the heap has been extended but the pointer assignment has not yet happened; formally: $g \in reachable_\mathrm{g}(c'.mem)$ implies $g \in reachable_\mathrm{g}(extend_\mathrm{heap}(avail_\mathrm{heap}, c.mem, t))$. Here, the idea is to show that every reachable g-variable outside the newly allocated variable is reachable via a path which *does not* contain g_l. Then, the remaining proof is just a simple exercise because changing a g-variable which is not needed for the reachability should not affect it.

However, to prove *formally* that g is reachable in c' without going over g_l is not so easy: e.g., to make the arguments inductive in Isabelle, we are required to introduce a strengthened version of reachability: `reachable_nameless_gvars_strong`. We will not introduce the formal definitions here because the main idea is quite obvious and can be given in a few words: in c', the g-variable g_l points to the newly allocated g-variable and all pointer g-variable inside the new one are initialized with null pointers. Thus, the inductive definition of reachability finds its end here. This means that no g-variable outside the new one is reachable via g_l.

As g is not such a sub g-variable of g_l it must be reachable without going through g_l. Thus, it must have been reachable already in c. q.e.d.

However, observe that there might be g-variables which have been reachable in the old configuration c (via g_l) but are not reachable in c' anymore.

Lemma 10.44 (Data consistency) *Let (te, ft, gst) be a translatable C0 program, let the valid C0 configurtion c be consistent with assembly configuration d via allocation function alloc, and let the head of the current program rest $s = hd(s2l(c.prog))$ be an allocation statement. Further, assume that there is sufficient memory, that the next transition of the C0 machine does not fail, that the maximum heap size is a meaningful number, and that the number of return statements in the program rest corresponds to the recursion depth. Finally, we require that the preconditions for the execution of $code_{stmt}(s)$ are fulfilled.*

Then, the assembly machine eventually reaches a configuration d' which is data and return address consistent with the next C0 configuration c' via some (new) allocation function $alloc'$.

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge consis(te, ft, c, alloc, d) \wedge (te, ft, gst(c.mem)) \in xltbl_{prog} \\
& \wedge\, s = PAlloc(e, tn) \wedge \delta_{C0}(te, ft, c) = \lfloor c' \rfloor \\
& \wedge\, abase_{heap} + asize_{heap}^{max} < addr_{max} \leq 2^{32} \wedge avail_{mem}(addr_{max}, te, ft, c) \\
& \wedge\, \#ret_{top}(s2l(c.prog)) + 1 = |c.mem.lm| \\
& \wedge\, asm_{pre}(d, range_c, code_{stmt}(te, ft, gst(c.mem), lst_{top}(c.mem), s)) \\
\Longrightarrow\ & \exists t, d', alloc' : \quad d \xrightarrow[range_c, range_a]{t, the(cbase_s(s)) + 4 \cdot the(csize_s(s))} d' \\
& \qquad \wedge consis_d(te, ft, c', alloc', d') \wedge consis_{ra}(te, ft, c', d') \\
& \qquad \wedge d'.spr = d.spr
\end{aligned}
$$

PROOF Let in the following t denote the type of the newly allocated variable, and $g_\mathrm{l} = \mathit{leval}(te, c.\mathit{mem}, e)$ the g-variable which corresponds to the left expression. We split the correctness proof of this lemma into two cases: either there is enough free heap memory and the allocation succeeds or there is not enough memory and it fails.

Before we start with the case distinction we show using Theorem 9.2 and Lemma 9.1 that the evaluation of e is correctly compiled, i.e., that we reach an assembly configuration d' in which the allocated base address of g_l is stored in register $hd(\mathit{fregs}_\mathrm{ini})$.

Case 1: $\neg \mathit{avail}_\mathrm{heap}(c.\mathit{mem}, t)$

In this case allocation fails: this makes the proof quite simple because there is only a single memory update with a null pointer and the heap memory remains unchanged.

We use Lemma 10.40 to show that we eventually reach an assembly configuration d'' in which the assembly memory at address $\mathit{fst}(\mathit{alloc}(g_\mathrm{l}))$ has been set to zero. The numerous preconditions of the lemma can be proved using lemmas similar to the ones from Section 10.2.1; we omit the details. This proves the first and the last goal of the theorem. It remains to prove data and return address consistency.

Data consistency follows from Theorem 10.22. Among others, this theorem requires us to prove that g_l is reachable; this follows from Theorem 8.1. We also have to show that the right value of the memory update is type correct and contains only reachable pointers; in this case this is obvious because we update with a null pointer.

For return address consistency we first conclude $\mathit{consis}_\mathrm{ra}(te, \mathit{ft}, c, d')$ using Lemma 10.19. That we have $\mathit{consis}_\mathrm{ra}(te, \mathit{ft}, c', d')$ finally follows from the fact that neither the *Return* statements in the program rest nor the return destinations in the stack have changed between configurations c and c'.

Case 2: $\mathit{avail}_\mathrm{heap}(c.\mathit{mem}, t)$

In this case allocation succeeds and we have to use an extended allocation function $\mathit{alloc}' = \mathit{alloc}_\mathrm{xt}(\mathit{alloc}, \mathit{sc}(m'), |\mathit{hst}(c.\mathit{mem})|, d.\mathit{gpr}!r_\mathrm{htop})$. Additionally, we have to deal with the zero initialization of the new heap variable.

We start using Lemma 10.41 to show that we eventually reach an assembly configuration d'' in which the heap top register has been incremented by $\lceil n \rceil_4$ and where the assembly memory at address $\mathit{fst}(\mathit{alloc}(g_\mathrm{l}))$ has been set to the allocated base address of the new heap variable, which itself has been initialized with zeroes. Again, we use lemmas from Section 10.2.1 and similar ones to discharge the numerous preconditions. This proves the first and the last goal of the theorem and we are, as before, left with data and return address consistency.

In the following, we present only the main ideas of the proof, in order not to distract the reader from the important arguments.

In the previous case we could simply use Theorem 10.22 to prove data consistency. Now, this is harder because we have to simultaneously consider two memory updates: the update of the left side expression with a pointer to the new heap variable and the zero initialization of the new

heap variable. The first update makes the new heap variable reachable; thus, data consistency can only hold if all its sub g-variables are value and pointer consistent. This does not hold before the second memory update is done. So, we have to prove preservation of data consistency in this case in one big-step for both memory updates.

We unpack the definition of data consistency and do the proof bit-by-bit.

- First, we have to show for all reachable elementary g-variables g in the new configuration c' that they are value and pointer consistent. We distinguish three cases.

  If the considered g-variable g is the just updated left side expression, i.e., $g = g_l$, we have to show that the update in the assembly machine correctly sets $mm(fst(alloc(gl)))$ to the address of the new heap variable. This follows directly from Lemma 10.41.

  If g is a sub g-variable of the newly allocated g-variable, value and pointer consistency follows from Lemma 10.42.

  In the third case, i.e., if g is nothing *special*, we first have to show that g was also reachable in the old $C0$ configuration c; this follows from Lemma 10.43. Then, we can conclude that g was value and pointer consistent in c and we use Lemma 10.14 and Corollary 10.15 to show that the memory update of g_l has not destroyed this consistency. It remains to show that the initialization has done no harm to the consistency of g. This is easy to do because the allocated memory region for the new heap variable cannot overlap with the allocated region of any g-variable which was reachable (and thus valid) in the old configuration.
- For frame header consistency, register consistency, and allocation consistency there are no special or interesting proof goals. Thus, we omit the proof details here.

q.e.d.

Isabelle The last requirement of the previous lemma is rather technical. For some proof details we need to know that all return destinations are valid g-variables. We can derive this from the predicate $rdest\surd$ (which is part of valid configurations), but only if there is a return statement in the program rest for every return destination, i.e., if the number of return statements equals the number of return destinations.

Observe, that the (similar) requirement about the number of return statements which is included in the $conf\surd$ predicate only requires that there are not more return statements than return destinations. This weaker condition was chosen to simplify the equivalence proof between big-step and small-step semantics; however, it is not strong enough to formally prove the lemma in Isabelle / HOL.

Now, it remains to combine the data consistency from the previous lemma with control consistency.

Theorem 10.45 (Correctness of allocation)
Assume the same preconditions as in the previous Lemma 10.44. Then, the assembly machine eventually reaches a configuration d' which is completely

consistent with the next C0 configuration c' and in which the special purpose registers have not been altered.

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge consis(te, ft, c, alloc, d) \wedge (te, ft, gst(c.mem)) \in xltbl_{prog} \\
& \wedge is\_PAlloc(s) \wedge \delta_{C0}(te, ft, c) = \lfloor c' \rfloor \\
& \wedge abase_{heap} + asize_{heap}^{max} < addr_{max} \leq 2^{32} \wedge avail_{mem}(addr_{max}, te, ft, c) \\
& \wedge asm_{pre}(d, range_c, code_{stmt}(te, ft, gst(c.mem), lst_{top}(c.mem), s)) \\
& \wedge \#ret_{top}(s2l(c.prog)) + 1 = |c.mem.lm| \\
\Longrightarrow \exists t, d', alloc' : \quad & d \xrightarrow[range_c, range_a]{t, d'.dpc} d' \\
& \wedge consis(te, ft, c', alloc', d') \\
& \wedge d'.spr = d.spr
\end{aligned}
$$

Proof For the proof of this theorem we combine Lemma 10.5 and 10.44 in the same way as in the proof of Theorem 10.36. q.e.d.

10.8 Function Calls

In this section, we prove that *SCall* statements are compiled correctly. We do not strictly separate low-level and high-level proofs here but split up the proofs similar to the structure of the generated code (cf. Section 7.4.7). We start the proofs in Section 10.8.1 by showing that parameter passing works correctly and preserves value and pointer consistency. We continue in Section 10.8.2 with a low-level correctness proof for the code which sets up the new stack frame and conclude in Section 10.8.3 with a combination of the partial results.

Before we start with the actual proofs we define a function which adjusts an allocation function to a given stack. When extending the runtime stack during function calls we have to adapt the allocation function. We have to extend the domain for the local g-variables in the new stack frame.

Definition 10.2 (Updated allocation function) We introduce a function $alloc_{\text{upd}} :: (gvar \mapsto (\mathbb{N} \times \mathbb{N})) \times functableT \times tenv \times symbolconf \mapsto (gvar \mapsto (\mathbb{N} \times \mathbb{N}))$ which updates a given allocation function with regards to a given symbol configuration. This means that we set the allocation function for *named* g-variables to their allocated base address and allocated size, respectively, and we leave it unchanged for *heap* g-variables. This adapts the allocation function to changes in the stack and in the global symbol table. However, we use it only for the extension of the stack during function calls.

Let *ft* be a function table, *te* a type environment, *sc* a symbol configuration, and *alloc* the old allocation function. Then, we define the updated allocation function by

$$
alloc_{\text{upd}}(alloc, ft, te, sc)(g) = \begin{cases} alloc(g) & \text{if } mem_{\text{g}}(g) = hm \\ \big(abase_{\text{g}}(te, ft, sc, g), asize_{\text{t}}(ty_{\text{g}}(sc, g))\big) & \text{otherwise} \end{cases}
$$

10.8.1 Parameter Passing

The formal verification of parameter passing was somewhat complicated. The main reason for this are the strong preconditions of Lemmas 10.32 and 10.33

which are used to verifiy the correct transfer of a *single* parameter. These lemmas have to be used inside an induction proof to derive the correctness of parameter passing of *all* parameters. Thus, we have to keep the neccessary preconditions available during the induction which requires us to keep track of them inside the induction invariant. This makes the invariant rather verbose.

In the following Lemma 10.46 we present the actual correctness lemma for parameter passing, including the complex induction invariant. To make the lemma inductive we simultaneously argue about two $C0$ configurations c and c'. Configuration c is the initial configuration before the parameter passing has started. Configuration c' is an intermediate $C0$ configuration in which a certain prefix of the parameters has already been copied. According to the $C0$ semantics of function calls (cf. Section 4.4.3 on page 64) all function parameters are evaluated in the old configuration c. In order to be able to apply the correctness theorem for expression evaluation we require data consistency for the old configuration c in addition to value, pointer, and allocation consistency for the intermediate configuration c'.

We will prove Lemma 10.46 by induction on the number of parameters. But instead of using the *fixed* parameter list from the function table we will use a *variable pl* which initially equals the parameter list from the function table. This allows to consume the list of parameter expressions *el* (from the function call statement) and the list of parameters *pl* synchronously during the induction. Figure 10.3 illustrates the different $C0$ and assembly intermediate configurations and their relation.

Lemma 10.46 *Let (te, ft, gst) be a translatable $C0$ program and let the valid $C0$ configurtion c (the* original *configuration before start of the parameter passing) be data consistent with assembly configuration d via allocation function alloc. Assume that extending the memory configuration of c with a new frame nlm (whose symbol table is the function symbol table for f) gives the memory of c'. Let the* intermediate $C0$ *configuration c' (after a certain prefix of the parameter list has already been copied) be valid and value, pointer, and allocation consistent with assembly configuration d via the updated allocation function $alloc' = alloc_{upd}(alloc, ft, te, sc(c'.mem))$. Further, assume that the types of the expression list el match the types of the parameter list pl, and that copying all parameters gives a new $C0$ memory configuration m'.*

We require all elements of the parameter list to be present in the parameters of the function f (i.e., that pl is a sub list of the parameter list of f). We require for all parameter expressions that they are valid and translatable, that there are enough registers for their evaluation, and that the allocated size of their type is not too large. Finally, we assume the usual requirements regarding sufficient memory and the preconditions for execution of the generated code.

We abbreviate with $fs_{cur} = \lceil asize_{st}(lst, 12)\rceil_4$ the allocated size of the current frame and with gst and lst the symbol tables of the global memory and the top most local memory of c, respectively. Additionally, we abbreviate by $il = code_{passing}(te, f, gst, lst, tl(fregs_{ini}), fs_{cur}, el, map(fst, pl))$ the code which is generated for parameter passing.

Then, we will eventually reach an assembly configuration d' which is value, pointer, and allocation consistent with the $C0$ memory configuration m' (i.e., after copying all parameters) and in which the memory outside the new stack frame has not been changed. Additionally, d' is data consistent with the old $C0$

configuration c (this is neccessary to make the statement inductive).

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge c' \in conf\surd(te, ft) \wedge consis_d(te, ft, c, alloc, d) \\
& \wedge consis_v(c', alloc', d) \wedge consis_p(c', alloc', d) \wedge consis_{alloc}(te, ft, c', alloc') \\
& \wedge c'.mem = c.mem[lm := c.mem.lm \circ nlm \wedge nlm.st = st_{fun}(f) \\
& \wedge tmatch_{para}(map(type(te, gst, lst), el), map(snd, pl)) \\
& \wedge copy_{para}(c'.mem, map(the(reval(te, mc)), el), pl) = \lfloor m' \rfloor \\
& \wedge \forall p \in pl : p \in st_{fun}(f) \\
& \wedge \forall e \in el : \; e \in valid_{expr}(te, gst, lst) \\
& \qquad \wedge e \in xltbl_{expr}(te, ft, gst, lst) \\
& \qquad \wedge enough_{regs}(te, gst, lst, elem?_t(type(te, gst, lst, e)), |fregs_{ini}| - 2, e) \\
& \qquad \wedge -2^{15} \leq asize_t(type(te, gst, lst, e)) < 2^{15} \\
& \wedge (te, ft, gst) \in xltbl_{prog} \wedge 4 \mid d.dpc \\
& \wedge addr_{max} \leq 2^{32} \wedge avail_{mem}(addr_{max}, te, ft, c) \wedge avail_{stack}(te, ft, c') \\
& \wedge asm_{pre}(d, range_c, il) \\
& \Longrightarrow \exists t, d' : d \xrightarrow[range_c, range_a]{t, d.dpc+4\cdot|il|} d' \\
& \qquad \wedge consis_d(te, ft, c, alloc, d') \\
& \qquad \wedge consis_v(c'[mem := m'], alloc', d') \\
& \qquad \wedge consis_p(c'[mem := m'], alloc', d') \\
& \qquad \wedge consis_{alloc}(te, ft, c'[mem := m'], alloc') \\
& \qquad \wedge unchngd_{mem}(d.mm, d'.mm, 0, abase_{lm}(sc(c'.mem), |c.mem.lm|) \div 4) \\
& \qquad \wedge unchngd_{mem}(d.mm, d'.mm, abase_{heap} \div 4, i2n(d.gpr!r_{htop}) \div 4) \\
& \qquad \wedge d'.spr = d.spr \\
& \qquad \wedge \forall r \in hd(fregs_{ini}) \# \{r_{sbase}, r_{htop}, r_{lframe}, r_{jal}\} : d'.gpr!r = d.gpr!r
\end{aligned}
$$

Proof We prove this lemma by induction on the list *el* of parameter expressions and simultaneously strip down the list *pl* of function parameters.

Case 1: induction start: $el = [\,]$ and $pl = [\,]$
This case is trivial. We instantiate $t = 0$ and $d' = d$ and are done.

Case 2: induction step: $el = e \# el'$ and $pl = p \# pl'$
For the induction step we have to

1. show that copying the first parameter works correctly and
2. combine this with the induction hypothesis which argues that copying the remaining parameters also works correctly.

Figure 10.3 illustrates the different intermediate configurations.

We first show using Theorem 9.2 that the first parameter e is correctly evaluated. Then we apply Lemmas 10.32 and 10.33 to prove that the assignment of this parameter works as expected. We instantiate among others c with c' and *alloc* with the extended allocation function $alloc'$. We use r_{lframe} as destination register and $fs_{\text{cur}} + displ_{\text{v}}(12, lst, p)$ as offset. To apply these lemmas we have to discharge a lot of low-level preconditions. We present the most involved ones.

- For non-elementary parameters we have to show that their $C0$ value and reachability does not change between configurations c and c'; this allows to inherit most of the necessary preconditions from properties of expression evaluation in c.
- For elementary parameters we have to show that their value v fits the register value from expression evaluation. We conclude

$$\begin{aligned} &vmatch_{\mathrm{rval}}(alloc, v, d.gpr!hd(tl(fregs_{\mathrm{ini}}))) \qquad \text{(Theorem 9.2)} \\ &= vmatch_{\mathrm{rval}}(alloc', v, d.gpr!hd(tl(fregs_{\mathrm{ini}}))). \end{aligned}$$

  This conclusion is correct because we know from the definition of $alloc_{\mathrm{upd}}$ and from allocation consistency that the allocated base address of all g-variables which have been reachable in c is unchanged.
- We have to show that the destination region of the assignment is contained inside the new frame. This follows from the instantiation of r_{lframe} as destination register and $fs_{\mathrm{cur}} + displ_{\mathrm{v}}(12, lst, p)$ as offset and from the definition of the allocated range of the new frame.

We conclude from these two lemmas and the correctness of expression evaluation via Lemma 6.9 on page 118 that we eventually reach an assembly configuration d'' which is value and pointer consistent with the $C0$ memory $m'' = the(memupd(c'.mem, gvar_{\mathrm{lm}}(|c.mem.lm|, p), reval(c.mem, e))$ after copying the first parameter e (cf. Definition 4.34 on page 64). Additionally, we know that between d and d'' only the allocated region for the new frame has been changed in the assembly memory.

Now, we instantiate the induction hypothesis with $c' = c'[mem := m'']$, $d = d''$, $el = el'$, and $pl = pl'$. After some easy proof steps, including the proof that we have data consistency $consis_{\mathrm{d}}(te, ft, c, alloc, d'')$ and allocation consistency $consis_{\mathrm{alloc}}(te, ft, c'[mem := m''], alloc')$ with these instantiations, we are able to conclude that passing the remaining parameters works as expected. This shows that we eventually reach an assembly configuration d' in which – among others – the following holds

$$\begin{aligned} &consis_{\mathrm{d}}(te, ft, c, alloc, d') \\ &consis_{\mathrm{v}}(c'[mem := m'], alloc', d') \\ &consis_{\mathrm{p}}(c'[mem := m'], alloc', d') \end{aligned}$$

Now, it only remains to instantiate t in the lemma's statement with the corresponding step number to complete the proof of this lemma. q.e.d.

To simplify later the usage of the previous lemma we remove those preconditions which are only used for the induction proof and obtain the following corollary.

Corollary 10.47 *Let (te, ft, gst) be a translatable $C0$ program, let the valid $C0$ configuration c be data consistent with assembly configuration d via allocation function $alloc$, and let the head of the current program rest $s = hd(s2l(c.prog))$ be a function call statement. Further, assume that the function (fn, f) is defined in the function table, that the types of the expression list el match the parameter types of f, and that extending the stack with a frame for f gives a new $C0$*

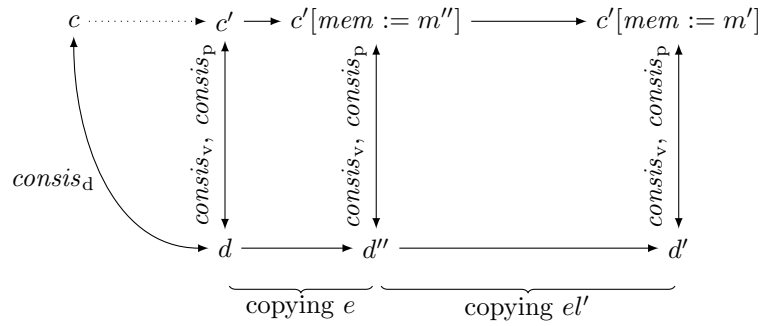


Figure 10.3: Correctness of Parameter Passing: Induction Step

memory configuration m'. Finally, we require the same properties for the list of expressions and the same abbreviations as in the previous lemmas and some more technical conditions.

To keep formulas readable, we abbreviate the code for parameter passing by $il = code_{passing}(te, f, gst, lst, tl(fregs_{ini}), fs_{cur}, el, map(fst, f.params))$.

Then, we will eventually reach an assembly configuration d' which is value, pointer, and allocation consistent with the C0 memory configuration m' (i.e., after copying the parameters) and in which the memory outside the new stack frame has not been changed.

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge consis_d(te, ft, c, alloc, d) \wedge (fn, f) \in ft \\
& \wedge\, hd(s2l(c.prog)) = SCall(vn, fn, pl) \\
& \wedge\, \exists p : s2l(p) = s2l(f.body) \circ tl(s2l(c.prog)) \\
& \wedge\, tmatch_{para}(map(type(te, gst, lst), el), map(snd, f.params)) \\
& \wedge\, extend_{stack}(te, c.mem, vn, f, el) = \lfloor m' \rfloor \\
& \wedge \forall e \in el :\ e \in valid_{expr}(te, gst, lst) \\
& \qquad\qquad \wedge e \in xltbl_{expr}(te, ft, gst, lst) \\
& \qquad\qquad \wedge enough_{regs}(te, gst, lst, elem?_t(type(te, gst, lst, e)), |fregs_{ini}| - 2, e) \\
& \qquad\qquad \wedge -2^{15} \leq asize_t(type(te, gst, lst, e)) < 2^{15} \\
& \wedge\, (te, ft, gst) \in xltbl_{prog} \wedge 4 \,|\, d.dpc \\
& \wedge\, addr_{max} \leq 2^{32} \wedge avail_{mem}(addr_{max}, te, ft, c) \wedge avail_{stack}(te, ft, c') \\
& \wedge\, asm_{pre}(d, range_c, il) \\
& \Longrightarrow \exists t, d' : d \xrightarrow[range_c, range_a]{t, d.dpc+4\cdot|il|} d' \\
& \qquad \wedge\, consis_v(c[mem := m'], alloc', d') \\
& \qquad \wedge\, consis_p(c[mem := m'], alloc', d') \\
& \qquad \wedge\, consis_{alloc}(te, ft, c[mem := m'], alloc') \\
& \qquad \wedge\, unchngd_{mem}(d.mm, d'.mm, 0, abase_{lm}(sc(c'.mem), |c.mem.lm|) \div 4) \\
& \qquad \wedge\, unchngd_{mem}(d.mm, d'.mm, abase_{heap} \div 4, i2n(d.gpr!r_{htop}) \div 4) \\
& \qquad \wedge\, d'.spr = d.spr \wedge d'.gpr!hd(fregs_{ini}) = d.gpr!hd(fregs_{ini}) \\
& \qquad \wedge \forall r \in \{r_{sbase}, r_{htop}, r_{lframe}, r_{jal}\} : d'.gpr!r = d.gpr!r
\end{aligned}
$$

10.8.2 Setting up the New Stack Frame

In this Section we present a lemma about the low-level correctness of the code which sets up the new stack frame and jumps to the beginning of the called function.

Lemma 10.48 *Let c be a valid C0 configuration and let the preconditions for the execution of $code_{newfr}$ hold in assembly configuration d. Besides some additional (technical) preconditions, abbreviate with $s = SCall(vn, fn, pl)$ the function call statement, with fs_{cur} and fs_{new} the allocated size of the current and the new stack frame, respectively, with $f = the(map\text{-}of(ft, fn))$ the called function, with $dist = dist_{scall}(te, ft, gst(c.mem), lst_{top}(c.mem), s)$ the relative jump distance to its start address which itself is abbreviated with $cb = the(cbase_s(hd(s2l_{ns}(f.body))))$. Finally, let $fbase_{new} = i2n(d.gpr!r_{lframe}) + fs_{cur}$ denote the base address of the newly created stack frame.*

Then, the execution of $code_{newfr}(fs_{new}, fs_{cur}, s)$ results after t steps in a new assembly configuration d' in which the r_{lframe} register has been correctly updated, the frame header fields of the new frame are properly initialized, and the special registers r_{sbase} and r_{htop} have not been changed.

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge asm_{pre}(d, range_c, code_{newfr}(fs_{new}, fs_{cur}, s)) \\
& \wedge 0 \leq fs_{cur} < 2^{32} \wedge fs_{cur} > 4 \wedge 0 \leq fs_{new} < 2^{32} \\
& \wedge 4 \mid fs_{cur} \wedge 4 \mid i2n(d.gpr!r_{lframe}) \\
& \wedge i2n(d.gpr!r_{lframe}) + fs_{cur} + 12 < 2^{32} \\
& \wedge (i2n(d.gpr!r_{lframe}) + fs_{cur}, 12) \subseteq range_a \wedge snd(range_c) \leq i2n(d.gpr!r_{lframe}) \\
& \wedge cb \in range_c \wedge cb < 2^{32} \wedge d.dpc + 4 \cdot |code_{newfr}(fs_{new}, fs_{cur}, s)| + dist = cb \\
& \Longrightarrow t, d' : d \xrightarrow[range_c, range_a]{t, cb} d' \\
& \qquad \wedge i2n(d'.gpr!r_{lframe}) = fbase_{new} \\
& \qquad \wedge d'.mm = \\
& \qquad\qquad d.mm \left[\begin{array}{l} (fbase_{new}) \div 4 := int2cell(n2i(d.dpc + 4 \cdot |code_{newfr}|)), \\ (fbase_{new} + 4) \div 4 := int2cell(d.gpr!hd(fregs_{ini})), \\ (fbase_{new} + 8) \div 4 := int2cell(d.gpr!r_{lframe}) \end{array} \right] \\
& \qquad \wedge d'.gpr!r_{sbase} = d.gpr!r_{sbase} \wedge d'.gpr!r_{htop} = d.gpr!r_{htop} \\
& \qquad \wedge d'.spr = d.spr
\end{aligned}
$$

Proof We omit the straightforward proof of this lemma.

10.8.3 Putting it All Together

In this section we combine the results from the previous sections to a complete correctness proof for function call statements. We start with a lemma which shows that function calls do not create new reachable heap g-variables. Then, we prove that the code for function calls preserves data and return address consistency. We conclude with Theorem 10.51 which shows that the code for function calls preserves consistency.

Lemma 10.49 *Let c be a valid C0 configuration and let the head of the program rest be a function call to some existing function f. Further, assume that*

extending the stack for this function call succeeds and gives a new memory configuration m'.

Then, all heap g-variables which are reachable in m' have already been reachable in the old memory configuration c.mem.

$$
\begin{aligned}
& g \in reachable_g^{nameless}(m') \\
& \wedge\, c \in conf\surd(te, ft) \wedge (fn, f) \in ft \\
& \wedge\, hd(s2l(c.prog)) = SCall(vn, fn, pl) \\
& \wedge\, \exists p : s2l(p) = s2l(f.body) \circ tl(s2l(c.prog)) \\
& \wedge\, extend_{stack}(te, c.mem, vn, f, el) = \lfloor m' \rfloor \\
& \Longrightarrow g \in reachable_g^{nameless}(c.mem)
\end{aligned}
$$

Proof We present only the main idea of the proof. We proceed in two steps. First we show that extending the stack with a new frame does not create new reachable heap variables. Intuitively, this holds because all local variables are uninitialized and cannot be on the reachability path for some g-variables. In the second step we prove using Theorem 8.2 on page 172 by induction on the number of parameters that copying the parameters does not create new reachable g-variables. q.e.d.

Lemma 10.50 *Let (te, ft, gst) be a translatable C0 program, let the valid C0 configuration c be consistent with assembly configuration d via allocation function alloc, and let the head of the current program rest be a function call statement which calls function $f = the(map\text{-}of(ft, fn))$. Further, assume that there is sufficient memory, that the next transition of the C0 machine does not fail, that the number of Return statements in the program rest corresponds to the recursion depth, and, finally, that the preconditions for the execution of the compiled code for the function call statement are fulfilled.*

Then, the assembly machine will eventually reach a configuration d' which is data and return address consistent with the next C0 configuration c' via some (new) allocation function $alloc'$.

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge consis(te, ft, c, alloc, d) \wedge (te, ft, gst(c.mem)) \in xltbl_{prog} \\
& \wedge\, s2l(c.prog) = SCall(vn, fn, pl) \wedge \delta_{C0}(te, ft, c) = \lfloor c' \rfloor \\
& \wedge\, addr_{max} \leq 2^{32} \wedge avail_{mem}(addr_{max}, te, ft, c) \wedge avail_{stack}(te, ft, c') \\
& \wedge\, asm_{pre}(d, range_c, code_{stmt}(te, ft, gst(c.mem), lst_{top}(c.mem), s2l(c.prog))) \\
& \wedge\, \#ret_{top}(s2l(c.prog)) + 1 = |c.mem.lm| \\
& \Longrightarrow \exists t, d', alloc' : \quad d \xrightarrow[range_c, range_a]{t, the(cbase_s(hd(s2l_{ns}(f.body))))} d' \\
& \qquad\qquad \wedge consis_d(te, ft, c', alloc', d') \\
& \qquad\qquad \wedge consis_{ra}(te, ft, c', d') \\
& \qquad\qquad \wedge d'.spr = d.spr
\end{aligned}
$$

Proof The code for function call statements consists of three parts: the evaluation of the return destination, code for parameter passing, and code to set up the new stack frame and jump to the called function (cf. Section 7.4.7 on page 154). Thus, the proof is also divided into three parts.

First, we apply Theorem 9.2 to show that the allocated address of the return variable is corrected computed in register $hd(fregs_{\text{ini}})$. Then, we apply

Corollary 10.47 to show that parameter passing works as expected. This gives us an intermediate assembly configuration d' which is value, pointer, and allocation consistent with the new $C0$ configuration c'. Finally, we apply Lemma 10.48 to prove that we correctly set up the new stack frame and jump to the start of the code of function f. Combining these three steps gives us a new assembly configuration d'' and a step number t which we use to instantiate the existential quantifiers for d'' and t. We instantiate the existential quantifier for $alloc'$ with $alloc_{\mathrm{upd}}(alloc, ft, te, sc(c'.mem))$.

It remains to show that d' is data and return address consistent with c' via $alloc'$. We easily prove that only memory inside the new stack frame has been modified in-between the original assembly configuration d and configuration d''. We prove the five components of data consistency separately.

Allocation consistency does not depend on the program rest; thus we can directly conclude allocation consistency for c' from Corollary 10.47. Value and pointer consistency for d'' follow from value and pointer consistency for d' because only memory inside the frame header of the new stack frame has been changed in between; thus, assembly values of g-variables have not changed.

For frame header consistency $consis_{\mathrm{fh}}(te, ft, c', alloc', d'')$ we have to show for all $0 < i < |c'.mem.lm|$ that previous stack pointer and return destination fields in the frame header of the i-th frame are correct. For the new frame, i.e., $i = |c.mem.lm|$, this follows from Lemma 10.48 and for all other frame from $consis_{\mathrm{fh}}(te, ft, c, alloc, d)$ as neither the frame headers nor the allocated base of the frames or the allocated address of the return destinations have changed.

We know from the precondition that $consis_{\mathrm{r}}(te, ft, c, alloc, d)$ holds and have to show $consis_{\mathrm{r}}(te, ft, c', alloc', d'')$. There is only one demanding goal here: to show that all reachable heap g-variables in c' are allocated below $i2n(d''.gpr!r_{\mathrm{htop}})$. Because $i2n(d''.gpr!r_{\mathrm{htop}}) = i2n(d.gpr!r_{\mathrm{htop}})$ we just have to show that the set of reachable heap g-variables has not been extended from c to c' which follows from Lemma 10.49.

This completes the proof of data consistency and the only remaining proof goal is to show return address consistency $consis_{\mathrm{ra}}(te, ft, c', d')$. For return address consistency (cf. Definition 8.13 on page 177) we have to show for all $i < |c'.mem.lm|$ that there exists some function call statement s such that the successor statement of s follows the i-th return statement $Return(re)$ in the program rest of c' and the return address $fh_{\mathrm{ra}}(te, ft, sc(c'.mem), d'.mm, |c'.mem.lm| - (i+1))$ points directly behind the code of s. We distinguish here between the return address of the newly created frame ($i = 0$) and the return addresses in the old frames.

For $i = 0$ we instantiate the existential quantifier in the definition of $consis_{\mathrm{ra}}$ with the return statement which we just execute. We know from Lemma 10.48 that the return address in the new stack frame correctly points behind the code of this statement. Additionally, we conclude from $c \in conf\surd(te, ft)$ that $successors\surd(s2l_{\mathrm{ns}}(c.prog), ft)$ holds, which implies that the function call statement is followed by its successor statement. The $C0$ transition function puts the body of f in front of this statement; thus, the first return statement is followed by the successor of the current function call and return address consistency for $i = 0$ is proved.

For $i > 0$ return address consistency holds because neither the frame header of the i-th frame nor the structure of the program rest behind the first return statement have been changed. q.e.d.

Theorem 10.51
Assume the same preconditions as in the previous lemma and abbreviate the current head of the program rest by $b = hd(s2l(c.prog))$*. Then, the assembly machine will eventually reach a configuration* d' *which is completely consistent with the next C0 configuration* c'*.*

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge consis(te, ft, c, alloc, d) \wedge (te, ft, gst(c.mem)) \in xltbl_{prog} \\
& \wedge is\_SCall(s) \wedge \delta_{C0}(te, ft, c) = \lfloor c' \rfloor \wedge addr_{max} \leq 2^{32} \\
& \wedge avail_{mem}(addr_{max}, te, ft, c) \wedge \#ret_{top}(s2l(c.prog)) + 1 = |c.mem.lm| \\
& \wedge asm_{pre}(d, range_c, code_{stmt}(te, ft, gst(c.mem), lst_{top}(c.mem), s)) \\
& \Longrightarrow \exists t, d', alloc' : \quad d \xrightarrow[range_c, range_a]{t, d'.dpc} d' \\
& \qquad\qquad \wedge consis(te, ft, c', alloc', d') \wedge d'.spr = d.spr
\end{aligned}
$$

Proof The extension of Lemma 10.50 to complete consistency is for function calls easier than for most other statements. Function calls jump directly to the beginning of the first statement in the called function. We know from the $C0$ semantics of function calls that this statement will be the head of the program rest in the new configuration c'. Thus, there is no need to execute additional control code and to apply lemmas from Section 10.1.

The last proof obligation is code consistency, i.e., we have to show that the program code has not been modified. This follows directly from the definition of assembly code execution (cf. Definition 6.15 on page 114). q.e.d.

10.9 Return

The induction step for *Return* statements is relatively simple such that we can omit special low-level lemmas. We prove in a single lemma the preservation of data and return address consistency and add in a second step control consistency.

Lemma 10.52 (Data consistency) *Let* (te, ft, gst) *be a translatable C0 program, let the valid C0 configuration* c *be consistent with assembly configuration* d *via allocation function alloc, and let the head of the current program rest* $s = hd(s2l(c.prog))$ *be a return statement. Further, assume that there is sufficient memory, that the next transition of the C0 machine does not fail, that the number of Return statements in the program rest corresponds to the recursion depth, and, finally, that the preconditions for the execution of* $code_{stmt}(s)$ *are fulfilled.*

Then, the assembly machine will eventually reach a configuration d' *which is data and return address consistent with the next C0 configuration* c'*.*

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge consis(te, ft, c, alloc, d) \wedge (te, ft, gst(c.mem)) \in xltbl_{prog} \\
& \wedge hd(s2l(c.prog)) = Return(re) \wedge \delta_{C0}(te, ft, c) = \lfloor c' \rfloor \\
& \wedge addr_{max} \leq 2^{32} \wedge avail_{mem}(addr_{max}, te, ft, c) \\
& \wedge asm_{pre}(d, range_c, code_{stmt}(te, ft, gst(c.mem), lst_{top}(c.mem), Return(re))) \\
& \wedge \#ret_{top}(s2l(c.prog)) + 1 = |c.mem.lm| \\
& \Longrightarrow \exists t, d' : \quad d \xrightarrow[range_c, range_a]{t, fh_{ra}(te, ft, sc(c.mem), d.mm, |c.mem.lm|-1))} d' \\
& \qquad\qquad \wedge consis_d(te, ft, c', alloc, d') \wedge consis_{ra}(te, ft, c', d') \\
& \qquad\qquad \wedge consis_{code}(te, ft, c', d') \wedge d'.spr = d.spr
\end{aligned}
$$

PROOF We present only the main idea of this proof. We divide the execution of the return code into three phases: the expression evaluation, the assignment of the return value, and the update of the stack.

Observe that allocation consistency is preserved from c to c' because all valid g-variables in c' have also been valid in the original configuration[7] and their ($C0$) address and size has not changed by execution of the return statement.

The correctness of expression evaluation follows from Theorem 9.2.

We prove for the code which copies the return value that it preserves value and pointer consistency and that the frame headers are not changed. We name the memory configuration after assignment of the return value (but before the removal of the top most stack frame) m'. We combine the low-level correctness lemma for load word instructions[8] with Lemmas 10.32 and 10.33 where we instantiate $ofs = 0$ and (if the return expression is not elementary) $g_r = leval(re)$. For the preconditions of this lemma we argue that the types of expression and return destination match because the predicate $rdest\surd$ holds in configuration c. For the same reason we know that the return destination is a valid g-variable. That the assembly destination address corresponds to the allocated address of the return destination follows from frame header consistency. Finally, we conclude from correct expression evaluation (cf. Theorem 9.2 on page 189) that the correct *value* (for elementary return expressions) or the correct *address* (for big return expressions) of the right expression is stored in the source register. The lemma guarantees value, pointer, and allocation consistency of m' with the new assembly configuration. Simple arguments show that these are preserved also after removing the top most stack frame. The lemma also ensures that the memory of the frame headers is not changed during the update because allocated region of valid g-variables and frame headers do not overlap.

Finally, it remains to show that we correctly update register r_{lframe} with the value from the frame header of the previous frame and that we correctly jump to the return address. Assume, that the corresponding code executes from assembly configuration d^i to d^j. We get from the low-level correctness of the code

$$\begin{aligned} d^j.mm &= d^i.mm \\ d^j.dpc &= n2i(fh_{\text{ra}}(te, ft, sc(c.mem), d^i.mm, |c.mem.lm| - 1)) \\ d^j.pcp &= d^j.dpc + 4 \\ d^j.gpr!r_{\text{lframe}} &= n2i(fh_{\text{psp}}(te, ft, sc(c.mem), d^i.mm, |c.mem.lm| - 1)) \end{aligned}$$

Value and pointer consistency have not changed between d^i and d^j or by removing the top most frame from the stack because we have not created new reachable g-variables (thus, we have for all *current* g-variables value and pointer consistency also in c) and have also not updated there value in the assembly or $C0$ machines.

We ensure the remaining properties of data consistency in the following way.

- Register consistency $consis_{\text{r}}$ requires mainly the correctness of r_{lframe} which follows from $consis_{\text{fh}}$. The rest follows from $consis_{\text{r}}$ in the old

[7] Observe that the reverse is not true because the local variables from the current stack frame in c are not valid any more after the return statement has been executed

[8] The corresponding lemma is in the style of Lemma 6.7 on page 116. Its concrete formulation is not used here, so we omit it.

configuration and the fact that the set of reachable heap variables has not been extended.

- For frame header consistency $consis_{\mathrm{fh}}$ we just have to show that $abase_{\mathrm{lm}}$ and frame headers have not changed from c to c'.
- Return address consistency $consis_{\mathrm{ra}}$ holds because the first return of $c.prog$ and the corresponding entry in the stack have been removed; for all other return addresses in the frame headers, the situation has not changed. q.e.d.

Theorem 10.53 (Correctness of returns)
Assume the same preconditions as in the previous lemma. Then, the assembly machine will eventually reach a configuration d' which is completely consistent with the next $C0$ configuration c'.

$$
\begin{aligned}
& c \in conf\surd(te, ft) \wedge consis(te, ft, c, alloc, d) \wedge (te, ft, gst(c.mem)) \in xltbl_{prog} \\
& \wedge\, hd(s2l(c.prog)) = Return(re) \wedge \delta_{C0}(te, ft, c) = \lfloor c' \rfloor \\
& \wedge\, addr_{max} \leq 2^{32} \wedge avail_{mem}(addr_{max}, te, ft, c) \\
& \wedge\, asm_{pre}(d, range_c, code_{stmt}(te, ft, gst(c.mem), lst_{top}(c.mem), Return(re))) \\
& \wedge\, \#ret_{top}(s2l(c.prog)) + 1 = |c.mem.lm| \\
\Longrightarrow\ & \exists t, d' : \quad d \xrightarrow[range_c, range_a]{t, d'.dpc} d' \\
& \qquad \wedge consis(te, ft, c', alloc, d') \\
& \qquad \wedge d'.spr = d.spr
\end{aligned}
$$

Proof The main part of the theorem follows from Lemma 10.52. It only remains to prove control consistency.

From Lemma 10.52 we know that the program counters point to the address ra which was stored in the return address field of the old frame header. We instantiate return address consistency $consis_{\mathrm{ra}}(te, ft, c', d')$ with $i = 0$. Then, we know that there exists some function call statement s such that the successor statement of s follows then original return statement $Return(re)$ in the program rest of c and the return address ra points directly behind the code of s. Now, we can apply Theorem 10.5 – instantiating with the function call statement and not with the original return – to show that we will finally reach the beginning of the next statement in the program rest. q.e.d.

Chapter 11

Implementation Correctness

In this chapter we briefly sketch the correctness proof of the compiler implementation which has been done by E. Petrova in [Pet07]. The goal of this presentation is to give the reader some intuition how the combination of Petrova's result with our simulation theorem fits the Verisoft context.[1]

Roughly speaking, Petrova has proved that her compiler implementation and our compiling specification produce for a given $C0$ program the same assembly code. This is illustrated in Figure 11.1.

More formally speaking, she proves that the diagram in Figure 11.2 commutes, i.e., that when starting in the left upper corner both ways gives the same result, namely $code_{\mathrm{s}}(abs_{\mathrm{C0}}(p)) = abs_{\mathrm{asm}}(code_{\mathrm{i}}(p))$. In this diagram

- $code_{\mathrm{i}}$ represents the compilation via the compiler implementation,
- $code_{\mathrm{s}}$ represents the compiling specification from Chapter 7, and
- abs_{C0} and abs_{asm} are so-called abstraction functions (cf. [MN03]) which abstract $C0$ and assembly programs which are represented as $C0$ data structures in the memory of the compiler implementation to corresponding $C0$ and assembly programs in Isabelle / HOL where they are represented by abstract data types.

Petrova's verification of the compiler implementation has been finished roughly one year before the verification of the simulation theorem from this thesis had been completed. However, the code generation algorithm has been improved to some extent after her final version. Restricted project resources have prevented us from adapting her large proofs to the current version of the code generation algorithm. Instead, we have proved the simulation theorem in two versions: one for the old code generation algorithm (which perfectly fits Petrova's results) and one for the new algorithm.

In this thesis we have presented the new code generation algorithm. However, differences to the old version are restricted to just a few places.

- Code generation for unary operators, address-of, and dereferencing of pointers has been slightly improved and needs fewer assembly instructions.

[1] This combination of both results has been formalized recently by E. Petrova.

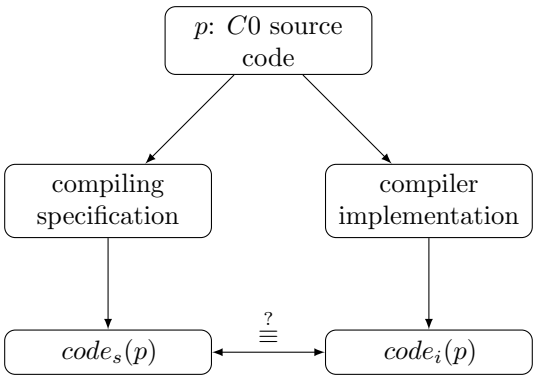


Figure 11.1: Correctness of the Compiler Implementation

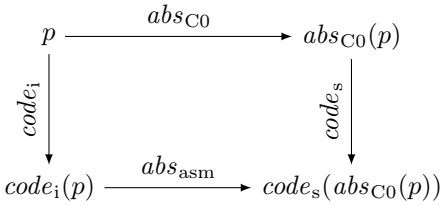


Figure 11.2: Correctness Theorem for the Compiler Implementation

- We have filled a delay slot in the center code for lazy binary operators with useful code instead of a nop instruction.
- A deprecated field in the frame headers has been removed.
- The code generation for assignments has been improved and unified. In particular, assignments, returns, and function calls can now all use the same code templates.
- The semantics of heap allocation have been improved. Now, the code returns a null pointer in case that there is no more heap memory available. In the old version, this run-time failure was not observable by the $C0$ program. For this reason, the new simulation theorem uses a meaningful instantiation for the $avail_{\text{heap}}$ predicate (cf. Section 8.3 on page 181) while the old version just used the predicate which always returns true.

Chapter 12

Summary and Future Work

We have presented in this thesis the formal verification of a simple, non-optimizing compiler for the C-like language $C0$; this includes the definition of a small-step semantics for the $C0$ language. All results have been formalized and mechanically checked in the interactive theorem prover Isabelle / HOL.

In contrast to most related work, the work presented in this thesis is geared to usage in the context of pervasive system verification. Accordingly, the $C0$ language has been designed to be powerful enough to implement system software while remaining 'neat' to allow for efficient formal verification of medium sized applications. Additionally, both the $C0$ semantics and the compiler correctness proof have been done in a small-step manner, allowing for the application of the compiler correctness theorem also for interleaving and non-terminating programs.

Furthermore, our compiler correctness theorem shows *total* correctness of the compiled code. That is, for every state of the source program's computation, the target machine will eventually reach a corresponding state, in which the values of all reachable $C0$ variables are properly represented. Without a total correctness theorem, it would not be possible to transfer functional correctness properties of source programs down to the compiled code. The correctness theorem also includes lifting resource restrictions from the target layer to the $C0$ layer. If these resource restrictions have been discharged at the $C0$ layer, we can be sure that they also hold for the compiled code.

To the best of our knowledge, our work is the first compiler correctness result which *simultaneously* proves a small-step simulation theorem between source programs in a non-toy input language and compiled code for a realistic target architecture, considers resource restrictions and other special requirements of pervasive verification, and argues about the complete compilation chain from source to target language (both represented by abstract data types in the theorem prover) instead of focusing on just a few parts of the compiler (like optimization patterns).

Our results have been successfully used as basis for several other pieces of work in the Verisoft project. We sketch the most important examples.

- As noted in Section 4.4.3, the small-step compiler correctness theorem is being used for the integration of (verified) inline assembly code into (verified) $C0$ applications. Without such a 'semantics' for inline assembly

code, the system layer software in Verisoft (and the libraries which allow invocation of operating system services from user applications) could not have been implemented in a high-level programming language as $C0$. In [ST08], the authors give several examples for the verification of so-called CVM primitives which combine $C0$ and inline assembly code.

- The property transfer from abstract to concrete system layers according to the Verisoft approach to system verification heavily depends on the compiler correctness theorem. The formal pervasive verification of a paging mechanism gives a concrete example of this approach [ASS08]. This result provides strong confidence that our notion of compiler correctness is indeed appropriate for the targeted field of application.
- The simulation theorem from this thesis and the correctness theorem about the compiler implementation from [Pet07] have been formally combined by E. Petrova.

Finally, we discuss possible directions of future work.

Integration of a Garbage Collector. The compiler correctness theorem presented in this thesis only argues about reachable memory objects. This simplifies the integration of a garbage collector like the copying garbage collector which has been verified by E. Petrova using the $C0$ Hoare logic environment.

The compiler correctness theorem and the correctness statement for the garbage collector have not yet been formally combined. However, this is a interesting direction of future work as no formally verified integration of a garbage collector into a verified compiler is known to us.

Memory Usage in the Presence of a Garbage Collector. Pavlova analyzes in [BPS05] heap memory consumption by counting how many memory objects are being allocated. In this thesis we sketched ideas how this idea can be realized using our compiler correctness theorem and the $C0$ verification environment from [Sch06]. However, both methods ignore deallocation of memory (regardless of whether this is done manually or using a garbage collector).

Without a garbage collector, a heap overflow occurs once the size of *all* allocated heap objects exceeds the heap size. In the presence of a garbage collector, a heap overflow only occurs if the size of the *reachable* heap objects exceeds the heap size.

At the layer of the $C0$ small-step semantics, we can easily track the reachable heap objects because all stack frames (and thus all root pointers for the reachability analysis) are visible. Here, we can precisely analyze heap usage even in the presence of a garbage collector. However, formally proving at the small-step layer that a program never uses too much heap memory would be quite costly because a lot of work from the program's functional verification in the $C0$ Hoare logic would have to be repeated. The same holds for proofs regarding the maximum recursion depth of a program (which is closely connected to its maximal stack usage).

The development of a methodology to discharge these kinds of properties already at the Hoare logic layer is interesting future work. Obviously, such a methodology has to keep track of the program's recursion depth and of the

root pointers for the reachability analysis. How exactly this information is best represented and how the results can be transferred down the semantic stack has to be elaborated.

Unified Framework for Verified Optimizing Compilers. The Verifix project [GZ99] and Leroy's Clight compiler [BDL06, Ler06] present nicely structured frameworks for compiler verification. However, their work is not completely compatible with the requirements of pervasive verification. In contrast, our work matches these requirements, while it does not address the issue of layered compilation or optimization.

We propose[1] to establish a standardized framework for compiler verification which allows to separately verify different parts of compilers (optimization patterns, front end, instruction selection, ...) and is able to deal with different notions of correctness (adjusted to different fields of application). To make the latter manageable it is obviously necessary to develop a (small) set of standard correctness statements and to make sure that all components which are verified with respect to a given correctness statements fit seamlessly together. By building a compiler from components which all respect such a correctness statement one could obtain a compiler which also respects this correctness statement.

Such a framework could considerably increase the impact of formally verified compilers because efforts invested by different groups into the verification of different parts of a compiler could be easily integrated into a complete product.

[1] inspired by discussions with Andrew Appel and Helmut Seidl

Appendices

Appendix A

Concrete C0 Syntax

In this appendix we print the grammar for the concrete *C*0 syntax. Section A.1 lists the *C*0 grammar based on these tokens. In Section A.2 you can see the rules used by the *C*0 lexer to map sequences of characters to tokens.

A.1 C0 Grammar

The start rule of the *C*0 grammar is `translation_unit`.

```
translation_unit:
   external_declaration
   | translation_unit external_declaration

external_declaration:
   function_definition
   | function_declaration
   | EXTERN function_declaration
   | declaration

declaration:
   var_declaration
   | EXTERN var_declaration
   | type_definition
   | struct_id_definition

function_definition:
   type_specifier function_name '(' parameter_list ')' function_body
   | type_specifier function_name '(' ')' function_body

function_declaration:
   type_specifier function_name '(' parameter_list ')' ';'
   | type_specifier function_name '(' ')' ';'

function_name:
   ID
   | '*' ID

function_body:
   '{' '}'
```

```
   | '{' statement_list '}'
   | '{' var_declaration_list '}'
   | '{' var_declaration_list statement_list '}'

compound_statement:
   '{' '}'
   | '{' statement_list '}'

var_declaration_list:
   var_declaration
   | var_declaration_list var_declaration

statement_list:
   statement
   | statement_list statement

type_definition:
   TYPEDEF type_specifier identifier ';'

struct_id_definition:
   STRUCT ID '{' var_declarations '}' ';'
   | STRUCT ID ';'

struct_type:
   STRUCT ID

identifier_list:
   identifier
   | identifier_list ',' identifier

identifier:
   identifier2
   | '*' identifier

identifier2:
   ID
   | identifier2 '[' numeric_constant_expression ']'

parameter_spec:
   type_specifier identifier

parameter_list:
   parameter_spec
   | parameter_list ',' parameter_spec

type_declaration:
   type_specifier identifier_list ';'

constant:
   TRUE
   | FALSE
   | NIL
   | NUMERIC_CONSTANT
   | UNSIGNED_NUMERIC_CONSTANT
```

```
   | CHAR_CONSTANT

struct_literal:
  '.' ID '=' complex_literal
   |   '.' ID '=' rexpression

numeric_constant_expression:
   '-' numeric_constant_expression %prec UNARY_MINUS
   | numeric_constant_expression '+' numeric_constant_expression
   | numeric_constant_expression '*' numeric_constant_expression
   | numeric_constant_expression '-' numeric_constant_expression
   | numeric_constant_expression SHIFTLEFT numeric_constant_expression
   | numeric_constant_expression SHIFTRIGHT numeric_constant_expression
   | NUMERIC_CONSTANT
   | UNSIGNED_NUMERIC_CONSTANT
   | CHAR_CONSTANT
   | '(' numeric_constant_expression ')'

struct_literal_list:
  struct_literal
   | struct_literal_list ',' struct_literal

constant_list:
   rexpression
   | constant_list ',' rexpression

complex_literal:
   '{' complex_literal_list '}'
   | '{' constant_list '}'
   | '{' struct_literal_list '}'
   | STRING_LITERAL

complex_literal_list:
   complex_literal
   | complex_literal_list ',' complex_literal

statement:
   lexpression '=' rexpression ';'
   | lexpression '=' complex_literal ';'
   | lexpression '=' NEW '(' type_specifier ')' ';'
   | lexpression  '=' ID '(' rexpr_list ')' ';'
   | lexpression  '=' ID '(' ')' ';'
   | ID '(' rexpr_list ')' ';'
   | ID '('  ')' ';'
   | WHILE '(' rexpression ')' compound_statement
   | FOR '(' lexpression '=' rexpression ';'
             rexpression ';'
             lexpression '=' rexpression
         ')'
        compound_statement
   | RETURN rexpression ';'
   | if_statement
   | ASM '{' asm_instruction_list ASMEND
   | ';'
```

```
if_statement:
   IF '(' rexpression ')' compound_statement
   | IF '(' rexpression ')' compound_statement ELSE compound_statement
   | IF '(' rexpression ')' compound_statement ELSE if_statement

rexpr_list:
   rexpression
   | rexpr_list ',' rexpression

rexpression:
   basic_rexpr %prec BASICREXPR
   | constant
   | rexpression '|' rexpression
   | rexpression '&' rexpression
   | BITWISENEGATION rexpression
   | rexpression SHIFTLEFT rexpression
   | rexpression SHIFTRIGHT rexpression
   | rexpression XOR rexpression
   | rexpression LOGICALOR rexpression
   | rexpression LOGICALAND rexpression
   | rexpression EQUALS rexpression
   | rexpression NOTEQUALS rexpression
   | rexpression '<' rexpression
   | rexpression '>' rexpression
   | rexpression LE rexpression
   | rexpression GE rexpression
   | rexpression '+' rexpression
   | rexpression '-' rexpression
   | rexpression '*' rexpression
   | rexpression '/' rexpression
   | rexpression '%' rexpression
   | '-' rexpression %prec UNARY_MINUS
   | '!' rexpression
   | '&' lexpression %prec ADDRESSOF
   | INT '(' rexpression ')'
   | UNSIGNED '(' rexpression ')'
   | CHAR '(' rexpression ')'

lexpression:
   ID
   | '*' lexpression %prec DEREFERENCE
   | lexpression '[' rexpression ']'
   | lexpression '.' ID
   | lexpression ARROW ID
   | '(' lexpression ')'

basic_rexpr:
   ID
   | '*' lexpression %prec DEREFERENCE
   | basic_rexpr '[' rexpression ']'
   | basic_rexpr '.' ID
   | basic_rexpr ARROW ID
   | '(' rexpression ')'
```

```
type_specifier:
   elementary_type
   | struct_type
   | ID

elementary_type:
   INT
   | BOOL
   | UNSIGNED INT
   | CHAR

var_declarations:
   var_declaration
   | var_declaration var_declarations

var_declaration:
   type_declaration

asm_gpr:
   R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7 | R8 | R9
   | R10 | R11 | R12 | R13 | R14 | R15 | R16 | R17 | R18 | R19
   | R20 | R21 | R22 | R23 | R24 | R25 | R26 | R27 | R28 | R29
   | R30 | R31

asm_spr:
   SR | ESR | ECA | EPC | EDPC | EDATA | RM
   | IEEEF | FCC | PTO | PTL | EMODE | MODE

asm_constant:
   '-' asm_constant
   | NUMERIC_CONSTANT
   | UNSIGNED_NUMERIC_CONSTANT
   | CHAR_CONSTANT

asm_immediate:
    asm_constant
   | ASM_SIZEOF '(' rexpression ')'
   | ASM_OFFSET '(' ID ')'
   | ASM_OFFSET_LO '(' ID ')'
   | ASM_OFFSET_HI '(' ID ')'

asm_instruction_list:
    asm_instruction
   | asm_instruction_list asm_instruction

asm_instruction:
   ASM_ADD '(' asm_gpr ',' asm_gpr ',' asm_gpr ')' ';'
   | ASM_ADDI '(' asm_gpr ',' asm_gpr ',' asm_immediate ')' ';'
   | ASM_SUB '(' asm_gpr ',' asm_gpr ',' asm_gpr ')' ';'
   | ASM_SUBI '(' asm_gpr ',' asm_gpr ',' asm_immediate ')' ';'
   | ASM_AND '(' asm_gpr ',' asm_gpr ',' asm_gpr ')' ';'
   | ASM_ANDI '(' asm_gpr ',' asm_gpr ',' asm_immediate ')' ';'
   | ASM_OR '(' asm_gpr ',' asm_gpr ',' asm_gpr ')' ';'
```

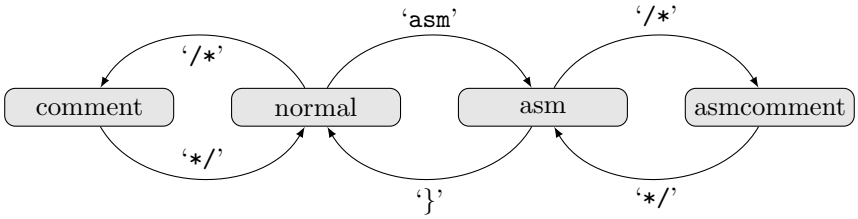


Figure A.1: Operation Modes of the *C*0 Lexer

```
| ASM_ORI '(' asm_gpr ',' asm_gpr ',' asm_immediate ')' ';'
| ASM_XOR '(' asm_gpr ',' asm_gpr ',' asm_gpr ')' ';'
| ASM_XORI '(' asm_gpr ',' asm_gpr ',' asm_immediate ')' ';'
| ASM_BNEZ '(' asm_gpr ','  asm_immediate ')' ';'
| ASM_BEQZ '(' asm_gpr ','  asm_immediate ')' ';'
| ASM_TRAP '(' asm_immediate ')'  ';'
| ASM_MOVI2S '(' asm_spr ','  asm_gpr ')' ';'
| ASM_MOVS2I '(' asm_gpr ','  asm_spr ')' ';'
| ASM_LW '(' asm_gpr ',' asm_gpr ',' asm_immediate ')' ';'
| ASM_SW '(' asm_gpr ',' asm_gpr ',' asm_immediate ')' ';'
| ASM_SEQ '(' asm_gpr ',' asm_gpr ',' asm_gpr ')' ';'
| ASM_SGE '(' asm_gpr ',' asm_gpr ',' asm_gpr ')' ';'
| ASM_SGR '(' asm_gpr ',' asm_gpr ',' asm_gpr ')' ';'
| ASM_SLE '(' asm_gpr ',' asm_gpr ',' asm_gpr ')' ';'
| ASM_SLS '(' asm_gpr ',' asm_gpr ',' asm_gpr ')' ';'
| ASM_SNE '(' asm_gpr ',' asm_gpr ',' asm_gpr ')' ';'
| ASM_SLSI '(' asm_gpr ',' asm_gpr ',' asm_immediate ')' ';'
| ASM_SLL '(' asm_gpr ',' asm_gpr ',' asm_gpr ')' ';'
| ASM_SLLI '(' asm_gpr ',' asm_gpr ',' asm_immediate ')' ';'
| ASM_SRL '(' asm_gpr ',' asm_gpr ',' asm_gpr ')' ';'
| ASM_SRLI '(' asm_gpr ',' asm_gpr ',' asm_immediate ')' ';'
| ASM_RFE '(' ')' ';'
```

A.2 C0 Lexer

The *C*0 lexer distinguishes four modes of operation: *normal* for normal *C*0 code, *asm* for inline assembly code, *comment* for multi-line comments, and *asmcomment* for multi-line comments in inline assembly code. If the lexer encounters an ASM token in normal mode it switches to asm mode; it switched back to normal mode when hitting a closing brace (cf. Figure A.1). The character sequence '`/*`' starts a multi-line comment in assembly or normal mode. Multi-line comments end with the following '`*/`'.

A double slash introduces a comment which ends with the current line. White spaces (space, new line, tabulator) are ignored by the lexer except of course for detecting token boundaries.s

In Table A.1 you see the regular expressions which match complex tokens like identifiers. In Tables A.2 and A.3 we list the mapping of characters to tokens in normal and assembly mode, respectively.

Table A.1: Regular Expressions for Some Complex Tokens

Regular Expression	Token
`"(\\"|[^\n"]*)*"`	STRING_LITERAL
`[a-zA-Z][a-zA-Z0-9_]*`	ID
`(0|([1-9][0-9]*))`	NUMERIC_CONSTANT
`(0|([1-9][0-9]*))u`	UNSIGNED_NUMERIC_CONSTANT
`'.'`	CHAR_CONSTANT
`'\\[0-3][0-7]{2}'`	CHAR_CONSTANT

Table A.2: Mapping of Characters to Tokens in Normal Mode

Character String	Token
`int`	INT
`unsigned`	UNSIGNED
`bool`	BOOL
`char`	CHAR
`typedef`	TYPEDEF
`struct`	STRUCT
`NULL`	NIL
`if`	IF
`else`	ELSE
`for`	FOR
`while`	WHILE
`true`	TRUE
`false`	FALSE
`return`	RETURN
`new`	NEW
`extern`	EXTERN
`->`	ARROW
`<=`	LE
`>=`	GE
`==`	EQUALS
`!=`	NOTEQUALS
`&&`	LOGICALAND
`||`	LOGICALOR
`<<`	SHIFTLEFT
`>>`	SHIFTRIGHT
`~`	BITWISENEGATION
`^`	XOR
`asm`	ASM

Table A.3: Mapping of Characters to Tokens in Assembly Mode

Character String	Token
`emode`	EMODE
`mode`	MODE
`asm_sizeof`	ASM_SIZEOF
`asm_offset`	ASM_OFFSET
`asm_offset_lo`	ASM_OFFSET_LO
`asm_offset_hi`	ASM_OFFSET_HI
`addi`	ASM_ADDI
`add`	ASM_ADD
`andi`	ASM_ANDI
`and`	ASM_AND
`subi`	ASM_SUBI
`sub`	ASM_SUB
`xori`	ASM_XORI
`xor`	ASM_XOR
`seq`	ASM_SEQ
`sge`	ASM_SGE
`sgr`	ASM_SGR
`sle`	ASM_SLE
`sls`	ASM_SLS
`sne`	ASM_SNE
`slsi`	ASM_SLSI
`sll`	ASM_SLL
`slli`	ASM_SLLI
`srl`	ASM_SLE
`srli`	ASM_SRLI
`rfe`	ASM_RFE
`ori`	ASM_ORI
`or`	ASM_OR
`lw`	ASM_LW
`sw`	ASM_SW
`bnez`	ASM_BNEZ
`beqz`	ASM_BEQZ
`trap`	ASM_TRAP
`movi2s`	ASM_MOVI2S
`movs2i`	ASM_MOVS2I
`sr`	SR
`esr`	ESR
`eca`	ECA
`epc`	EPC
`edpc`	EDPC
`edata`	EDATA
`rm`	RM
`ieeef`	IEEEF
`fcc`	FCC
`pto`	PTO
`ptl`	PTL
`r0`	R0
...	...
`r31`	R31
`}`	ASMEND

Appendix B

Mapping to Lemmas in Isabelle / HOL

This section gives a mapping from lemmas, corollaries, and theorems in this thesis to the corresponding formal proofs in the Isabelle theories.

Name	Page	Name in Isabelle
Lemma 5.1	69	`delta_first_statement_executes_invariant`
Lemma 5.2	79	`eval_lval_get_dataslice_correct`
Lemma 5.3	84	`type_correct_aux_base_transform`
Lemma 5.4	84	`type_correct_aux_invariant`
Lemma 5.5	84	`type_correct_memory_invariant`
Lemma 5.6	84	`type_correct_heap_invariant`
Lemma 5.7	84	`init_value_type_correct_aux`
Lemma 5.8	85	`initialize_content_type_correct_aux`
Corollary 5.9	85	`initialize_vars_type_correct`
Lemma 5.10	85	`apply_unop_type_correct`
Lemma 5.11	85	`get_dataslice_type_correct_aux`
Lemma 5.12	86	`eval_prim_val_type_correct`
Lemma 5.13	86	`eval_complex_val_type_correct`
Theorem 5.14	87	`eval_type_correct`
Theorem 5.14	87	`eval_content_valid_ptrs`
Lemma 5.15	92	`stmt_in_proctable_valid_successors`
Theorem 5.16	92	`delta_aux_valid_programrest`
Theorem 5.17	96	`delta_valid_conf`
Axiom 6.1	103	`data2cell2data`
Axiom 6.2	103	`cell2data2cell`
Lemma 6.3	104	`instr2cell2instr`
Lemma 6.4	105	`intwd2natwd2intwd`
Lemma 6.5	105	`natwd2intwd2natwd`
Lemma 6.6	108	`ASMcore_consis`
Lemma 6.7	116	`addi_execution_woi`
Lemma 6.8	117	`beqz_addi_execution2_woi`
Lemma 6.9	118	`asm_executes_woi_append`
Lemma 6.10	118	`asm_exec_precond_woi_take`
Corollary 6.11	119	`asm_exec_precond_woi_hd`
Corollary 6.12	119	`asm_exec_precond_woi_append2`
Lemma 6.13	119	`asm_exec_precond_woi_append`

Name	Page	Name in Isabelle
Corollary 6.14	119	asm_exec_precond_woi_tl
Lemma 7.1	127	relative_code_base_of_sub_stmt
Lemma 7.2	131	asize_type_dvd_4
Lemma 7.3	134	abase_correct_local_global
Theorem 8.1	171	eval_lval_reachable_gvars
Theorem 8.2	172	mem_update_reachable_nameless_gvars_subset
Theorem 8.3	183	compiler_correct
Lemma 9.1	188	correctly_compiled_expression_asm_executes
Theorem 9.2	189	expression_correctly_compiled
Lemma 9.3	190	unsigned_mult_eq_signed_mult
Lemma 9.4	190	eq_regs_same_eq_ptr
Lemma 10.1	195	control_code_last_in_while_correct
Lemma 10.2	195	control_code_last_stmt_in_cond_aux
Lemma 10.3	197	control_code_last_stmt_in_cond
Theorem 10.4	197	control_code_non_return_correct
Theorem 10.5	198	control_consistency_no_return
Lemma 10.6	199	fst_alloc_dvd_4
Lemma 10.7	199	fst_alloc_ge_code_range
Lemma 10.8	200	reachable_gvars_allocated_in_mem
Lemma 10.9	200	alloc_address_range_range_contains
Lemma 10.10	200	g_vars_frame_header_not_overlap
Lemma 10.11	200	elementary_gvar_no_sub_gvars_not_range_overlap
Lemma 10.12	201	gvar_not_range_overlap_or_same_alloc
Lemma 10.13	201	content_consistent_mem_update_gvar_e_p_consistent_invariant_inside
Lemma 10.14	202	content_consistent_mem_update_gvar_e_consistent_invariant_outside
Corollary 10.15	203	content_consistent_mem_update_gvar_p_consistent_invariant_outside
Lemma 10.16	204	get_dataslice_content_consistent
Corollary 10.17	204	mem_part_word_update_e_p_consistent_invariant
Lemma 10.18	205	mem_part_word_update_gvar_fh_consistent_invariant
Lemma 10.19	205	mem_part_word_update_gvar_ra_consistent_invariant
Lemma 10.20	206	mem_update_heap_consistent_invariant
Lemma 10.21	206	mem_update_ws_consistent_invariant
Theorem 10.22	207	content_consistent_mem_update_d_consistent_invariant
Corollary 10.23	207	mem_part_word_update_d_consistent_invariant
Lemma 10.24	208	ifte_correct_true_case
Lemma 10.25	209	ifte_correct_false_case
Theorem 10.26	209	ifte_correct
Lemma 10.27	210	loop_correct_true_case

Name	Page	Name in Isabelle
Lemma 10.28	211	`loop_correct_false_case`
Theorem 10.29	211	`loop_correct`
Lemma 10.30	212	`write_to_memory_code_behavior_correct_aux`
Lemma 10.31	213	`big_assignment_code_correct`
Lemma 10.32	215	part of `assignment_code_correct_aux`
Lemma 10.33	216	part of `assignment_code_correct_aux`
Lemma 10.34	217	`assign_assignment_code_correct`
Lemma 10.35	218	`assign_correct_aux`
Theorem 10.36	219	`assign_correct`
Lemma 10.37	220	`complex_val_assign_code_correct`; the inductive version of the lemma is `complex_val_assign_code_correct_aux`
Lemma 10.38	221	integrated into `assign_correct_aux`
Theorem 10.39	221	integrated into `assign_correct`
Lemma 10.40	222	`heap_alloc_code_correct_failure`
Lemma 10.41	223	`heap_alloc_code_correct_success`
Lemma 10.42	224	combination of `palloc_gvar_e_consistent_new_hm_var` and `palloc_gvar_e_consistent_new_hm_var`
Lemma 10.43	225	`delta_palloc_reachable_gvars_subset`
Lemma 10.44	226	combination of `palloc_correct_aux_failure` and `palloc_correct_aux_success`
Theorem 10.45	228	`palloc_correct`
Lemma 10.46	230	`parameter_passing_code_correct_aux`
Corollary 10.47	232	`parameter_passing_code_correct`
Lemma 10.48	234	`SCall_misc_code_correct`
Lemma 10.49	234	`extend_stack_reachable_nameless_gvars_subset`
Lemma 10.50	235	`scall_correct_aux`
Theorem 10.51	237	`scall_correct`
Lemma 10.52	237	`return_correct_aux`
Theorem 10.53	239	`return_correct`

Bibliography

[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.

[AU73] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume II: Compiling. Prentice-Hall, 1973.

[BBM+07] Jörg Bormann, Sven Beyer, Adriana Maggiore, Michael Siegel, Sebastian Skalberg, Tim Blackmore, and Fabio Bruno. Complete formal verification of TriCore2 and other processors. In *DVCon 2007*, February 2007.

[BD96] Egon Börger and Igor Durdanovic. Correctness of compiling Occam to Transputer code. *The Computer Journal*, 39(1):52–92, 1996.

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer-Verlag, 2006.

[Ber03] Stefan Berghofer. *Proofs, Programs and Executable Specifications in Higher Order Logic*. PhD thesis, Technical University of Munich, 2003.

[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*, volume 4085 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2006.

[BGL04] Yves Bertot, Benjamin Grégoire, and Xavier Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In Jean-Christophe Filliâtre, Christine Paulin-Mohring,

and Benjamin Werner, editors, *TYPES*, volume 3839 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 2004.

[BGZ03] Clark Barrett, Benjamin Goldberg, and Lenore Zuck. Run-time validation of speculative optimizations using CVC. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[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.

[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.

[BKLP04] Christoph Berg, Michael Klein, Dirk Leinenbach, and Wolfgang Paul. Formal operational semantics of C0. Verisoft, Internal Technical Report #8, 2004.

[BMW92] Deryck F. Brown, Hermano Moura, and David A. Watt. Actress: an action semantics directed compiler generator. In *Compiler Compilers 92*, volume 641 of *Lecture Notes in Computer Science*, 1992.

[BN02] Stefan Berghofer and Tobias Nipkow. Executing higher order logic. In *TYPES '00: Selected papers from the International Workshop on Types for Proofs and Programs*, volume 2277 of *Lecture Notes in Computer Science*, pages 24–40, London, UK, 2002. Springer.

[Boa96] Ariane 501 Inquiry Board. Ariane 5: Flight 501 failure. http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf, July 1996.

[BPH07] Jan O. Blech and Arnd Poetzsch-Heffter. A certifying code generation phase. *Electronic Notes in Theoretical Computer Science*, 190(4):65–82, 2007.

[BPS05] Gilles Barthe, Mariela Pavlova, and Gerardo Schneider. Precise analysis of memory consumption using program logic. In Bernhard Aichernig and Bernhard Beckert, editors, *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005), 5-9 September 2005, Koblenz, Germany*, pages 86–95, 2005.

[BZ07] Nick Benton and Uri Zarfaty. Formalizing and verifying semantic type soundness of a simple compiler. In *PPDP '07: Proceedings of the 9th ACM SIGPLAN international symposium on Principles*

and practice of declarative programming, pages 1–12, New York, NY, USA, 2007. ACM.

[CLN+00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 95–107, New York, NY, USA, 2000. ACM.

[CM86] Laurian M. Chirica and David F. Martin. Toward compiler implementation correctness proofs. *ACM Transactions on Programming Languages and Systems*, 8(2):185–214, 1986.

[Cur92] Paul Curzon. A verified compiler for a structured assembly language. In Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*. IEEE Computer Society Press, 1992.

[Cur94] Paul Curzon. The verified compilation of Vista programs. Presented at the 1st ProCoS Working Group Meeting, Gentofte, Denmark., January 1994.

[Dal06] Iakov Dalinger. *Formal Verification of a Processor with Memory Management Units*. PhD thesis, Saarland University, Computer Science Department, July 2006.

[Dav03] Maulik A. Dave. Compiler verification: A bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6), 2003.

[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.

[Die96] Stephan Diehl. *Semantics-Directed Generation of Compilers and Abstract Machines*. PhD thesis, Universität Saarbrücken, 1996.

[dRHH+01] Willem-Paul de Roever, Ulrich Hanneman, Jozef Hooman, Yassine Lakhnech, Mannes Poel, Job Zwiers, and Frank de Boer. *Concurrency Verification*. Cambridge University Press, Cambridge, UK, 2001.

[DV01] Axel Dold and Vincent Vialard. A mechanically verified compiling specification for a Lisp compiler. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Sience*, volume 2245 of *Lecture Notes in Computer Science*, pages 144–155. Springer, 2001.

[Eme05] Pavel Emeliyanenko. Automatic verification of conditions for absence of interrupts. Diploma thesis, Saarland University, Computer Science Department, 2005.

[GB03] Sabine Glesner and Jan O. Blech. Classifying and formally verifying integer constant folding. *Electronic Notes in Theoretical Computer Science*, 82(2), 2003.

[GGZ99] Wolfgang Goerigk, Thilo Gaul, and Wolf Zimmermann. Correct programs without proof? On checker-based program verification. In R. Berghammer and Y. Lakhnech, editors, *Tool Support for System Specification and Verification*, Springer Series Advances in Computing Science, pages 108–123, 1999.

[GH93] Yuri Gurevich and James K. Huggins. The semantics of the C programming language. In Egon Börger, Gerhard Jäger, Hans Kleine Büning, Simone Martini, and Michael M. Richter, editors, *CSL '92*, volume 702 of *Lecture Notes in Computer Science*, pages 274–308. Springer, 1993.

[GH98] Wolfgang Goerigk and Ulrich Hoffmann. Rigorous compiler implementation correctness: How to prove the real thing correct. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullmann, editors, *Applied Formal Methods – FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 122–136, 1998.

[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.

[GL01] Wolfgang Goerigk and Hans Langmaack. Compiler implementation verification and trojan horses. *IJDEA International Journal of Differential Equations and Applications*, 3(3), 2001.

[Gle03] S. Glesner. Using program checking to ensure correctness of compiler implementations. *Journal of Universal Computer Science*, 9(3):191–222, 2003. Special Issue on Compiler Optimization meets Compiler Verification.

[GZ99] Gerhard Goos and Wolf Zimmermann. Verification of compilers. In E.-R. Olderog and B. Steffen, editors, *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 201–230. Springer, 1999.

[GZ00] Gerhard Goos and Wolf Zimmermann. Verifying compilers and ASMs. In Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele, editors, *Abstract State Machines, Theory and Applications*, volume 1912, pages 177–202. Springer, Apr 2000.

[GZB05] Benjamin Goldberg, Lenore Zuck, and Clark Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electronic Notes in Theoretical Computer Science*, 132(1):53–71, 2005.

[HEK+07] Gernot Heiser, Kevin Elphinstone, Ihor Kuz, Gerwin Klein, and Stefan M. Petters. Towards trustworthy computing systems: Taking microkernels to the next level. *SIGOPS Oper. Syst. Rev.*, 41(4):3–11, 2007.

[HGG+99] Andreas Heberle, Thilo Gaul, Wolfgang Goerigk, Gerhard Goos, and Wolf Zimmermann. Construction of verified compiler front-ends with program-checking. In *Proceedings of PSI '99: Andrei Ershov Third International Conference on Perspectives Of System Informatics*, volume 1755 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[HIP05] Mark Hillebrand, Thomas 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.

[HJLT05] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 116–128, New York, NY, USA, 2005. ACM Press.

[HJS93] C. A. R. Hoare, He Jifeng, and Augusto Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30(9):701–739, 1993.

[Hoa03] Tony Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[HTS02] Michael Hohmuth, Hendrik Tews, and Shane G. Stephens. Applying source-code verification to a microkernel: The VFiasco project. In *Proc. 10th ACM SIGOPS*, pages 165–169. ACM Press, 2002.

[IK05] Thomas 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.

[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.

[Jav96] The JavaOS web-page at sun microsystems. http://java.sun.com/developer/products/JavaOS/, 1996.

[Joy89] Jeffrey J. Joyce. Totally verified systems: Linking verified software to verified hardware. In Miriam Leeser and Geoffrey Brown, editors, *Hardware Specification, Verification and Synthesis*, volume 408 of *Lecture Notes in Computer Science*, pages 177–201. Springer, 1989.

[KHCP00] Steve King, Jonathan Hammond, Rod Chapman, and Andy Pryor. Is proof more cost-effective than testing? *IEEE Transactions on Software Engineering*, 26(8):675–686, 2000.

[KN03] Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, 2003.

[KP07] 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.

[Kre06] Verena Kremer. Formally verified evaluation of expressions in a simple compiler. Bachelor's thesis, Saarland University, Computer Science Department, 2006.

[Kro01] Daniel Kroening. *Formal Verification of Pipelined Microprocessors*. PhD thesis, Saarland University, Computer Science Department, 2001.

[KSK06] Aditya Kanade, Amitabha Sanyal, and Uday Khedker. Structuring optimizing transformations and proving them sound. In *Proceedings of the 5th International Workshop on Compiler Optimization meets Compiler Verification (COCV'06)*, Vienna, Austria, April 2006.

[Lei02] Dirk Leinenbach. Implementierung eines maschinell verifizierten Prozessors. Diploma's thesis, Saarland University, Computer Science Department, 2002.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[Lev04] Raya Leviathan. *Validation of Translation to Optimized Machine Code*. PhD thesis, Weizmann Institute of Science, Rehovot, Israel, 2004.

[Lie95] Jochen Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating systems principles*, pages 237–250. ACM Press, 1995.

[LJWF02] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *POPL*, pages 283–294, 2002.

[LMC03] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically Proving the Correctness of Compiler Optimizations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2003)*, San Diego, California, June 2003.

[LMRC05] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Conference Record of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*, Long Beach, California, January 2005.

[LP04] Dirk Leinenbach and Wolfgang Paul. Translation from C0 to DLX. Verisoft, Internal Technical Report #5, 2004.

[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.

[LPP05] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In Bernhard Aichernig and Bernhard Beckert, editors, *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005), 5-9 September 2005, Koblenz, Germany*, pages 2–11, 2005.

[MIR04] The Motor Industry Research Association (MIRA), Ltd., UK. *MISRA-C:2004 — Guidelines for the use of the C language in critical systems*, 2004.

[MN03] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *CADE'03*, volume 2741 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2003.

[MO97] Markus Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, New York, 1997.

[Moo88] J S. Moore. Piton: A verified assembly level language. Technical Report 22, Comp. Logic Inc. Austin, Texas, 1988.

[Moo03] J S. Moore. A grand challenge proposal for formal methods: A verified stack. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *Lecture Notes in Computer Science*, pages 161–172. Springer, 2003.

[MP67] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 33–41. American Mathematical Society, 1967.

[MP00] Silvia M. Müller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[Mül07] Christian Müller. Verification of a simple cache system. Master's thesis, Saarland University, Computer Science Department, 2007.

[NBB+97] Peter Naur (ed.), J.W. Backus, F.L. Bauer, J. Green, C. Katz, J. McCarthy, A.J. Perlis, H. Rutishauser, K. Samelson, B. Vaquois, J.H. Wegstein, A. van Wijngaarded, and M. Woodger. Revised report on the algorithmic language ALGOL 60. In Peter O'Hearn and Robert D. Tennent, editors, *Algol-like Languages*, chapter 1, pages 19–49. Birkhäuser, 1997. First appeared in Communications of the ACM 6(1):1-17, The Computer Journal 5:349-67, and Numerische Mathematik 4:420-53, 1963.

[Nec97] George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[Nec98] George C. Necula. *Compiling with proofs*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1998.

[Nec00] George C. Necula. Translation validation for an optimizing compiler. In *PLDI'00: SIGPLAN Conference on Programming Language Design and Implementation*, pages 83–95. ACM, 2000.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 333–344, New York, NY, USA, 1998. ACM.

[NN99] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992, revised online version: 1999.

[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 Klaus Schneider and Jens Brandt, editors, *TPHOLs*, volume 4732 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 2007.

[OG76] Susan Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

[Ohe01a] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.

[Ohe01b] David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

[ORW95] Dino P. Oliva, John D. Ramsdell, and Mitchell Wand. The VLISP verified PreScheme compiler. *Lisp and Symbolic Computation*, 8(1/2):111–182, 1995.

[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.

[Pal92a] Jens Palsberg. An automatically generated and provably correct compiler for a subset of ADA. In *IEEE International Conference on Computer Languages*, 1992.

[Pal92b] Jens Palsberg. *Provably Correct Compiler Generation*. PhD thesis, Computer Science Department, Aarhus University, 1992. xii+224 pages.

[Pap98] Nikolaos S. Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, 1998.

[Pau82] Lawrence Paulson. A semantics-directed compiler generator. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 224–233, New York, NY, USA, 1982. ACM.

[Pen06] Hristo Pentchev. Verification of expression evaluation. Bachelor's thesis, Saarland University, Computer Science Department, 2006.

[Pen07] Hristo Pentchev. Verified expression evaluation for multiplication and division. Master's thesis, Saarland University, Computer Science Department, 2007.

[Pet07] Elena Petrova. *Verification of the C0 Compiler Implementation on the Source Code Level*. PhD thesis, Saarland University, Computer Science Department, 2007.

[Pol81] Wolfgang Polak. Compiler specification and verification. In J. Hartmanis Gerhard Goos, editor, *Lecture Notes in Computer Science*, volume 124 of *Lecture Notes in Computer Science*. Springer, 1981.

[PSS98a] Amir Pnueli, Ofer Shtrichman, and Michael Siegel. Translation validation for synchronous languages. In *ICALP '98: Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 235–246, London, UK, 1998. Springer-Verlag.

[PSS98b] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In Bernhard Steffen, editor, *TACAS '98*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998.

[PSS98c] Amir Pnueli, Ofer Strichman, and Michael Siegel. The code validation tool CVT: Automatic verification of a compilation process. *Software Tools for Technology Transfer*, 2(2):192–201, 1998.

[Rin99] Martin Rinard. Credible compilation. Technical Report MIT-LCS-TR-776, MIT Laboratory for Computer Science, March 1999.

[Riv04] Xavier Rival. Symbolic transfer functions-based approaches to certified compilation. In Xavier Leroy, editor, *31st Symposium on Principles of Programming Languages*, pages 1–13. ACM, janvier 2004.

[Riv05] Xavier Rival. *Traces Abstraction in Static Analysis and Program Transformation*. PhD thesis, École Normale Supérieure, Computer Science Department, 2005.

[RM99] Martin Rinard and Darko Marinov. Credible compilation with pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

[Sam75] Hanan Samet. *Automatically proving the correctness of translations involving optimized code*. PhD thesis, Stanford University, May 1975.

[Sam78] Hanan Samet. Proving the correctness of heuristically optimized code. *Communications of the ACM*, 21(4):570–582, 1978.

[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.

[SSB01] Robert Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine*. Springer, 2001.

[ST08] Artem Starostin and Alexandra Tsyban. Correct microkernel primitives. In *3rd intl Workshop on Systems Software Verification (SSV08), to appear*. Elsevier Science B. V., 2008.

[Str02] Martin Strecker. Formal verification of a Java compiler in Isabelle. In *Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77. Springer Verlag, 2002.

[Tch04] Andrei Tchaltsev. *Self-Validating Compilation based on Phase Semantics*. PhD thesis, The University of Manchester, Computer Science Department, 2004.

[TH92] Steven W. K. Tjiang and John L. Hennessy. Sharlit – a tool for building optimizers. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 82–93, New York, NY, USA, 1992. ACM Press.

[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.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[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.

[Tzi07] Hristo Tzigarov. Extended oracle proof methods for Isabelle / HOL: Reflection and SMV support. Diploma's thesis, Saarland University, Computer Science Department, 2007.

[Ver03] The Verisoft project. http://www.verisoft.de/, 2003.

[You89] William D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5(4):493–518, 1989.

[ZD03] Wolf Zimmermann and Axel Dold. A framework for modelling the semantics of expression evaluation with abstract state machines. In Elvinia Riccobene Egon Börger, Angelo Gargantini, editor, *Abstract State Machines – Advances in Theory and Applications 10th International Workshop, ASM 2003*, volume 2589 of *Lecture Notes in Computer Science*, pages 391–406. Springer Verlag, 2003.

[ZG97] Wolf Zimmermann and Thilo Gaul. On the construction of correct compiler back-ends: An ASM-approach. *Journal of Universal Computer Science*, 3(5):504–567, May 1997.

[Zim06] Wolf Zimmermann. On the correctness of transformations in compiler back-ends. In *Leveraging Applications of Formal Methods First International Symposium*, volume 4313 of *Lecture Notes in Computer Science*, pages 74–95, 2006.

[ZPF+02] Lenore Zuck, Amir Pnueli, Yi Fang, Benjamin Goldberg, and Ying Hu. Translation and run-time validation of optimized code. *Electronic Notes in Theoretical Computer Science*, 70(4), 2002.

[ZPFG02] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC: A translation validator for optimizing compilers. In Jens Knoop and Wolf Zimmermann, editors, *1st International Workshop on Compiler Optimization meets Compiler Verification COCV2002*, volume 65 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[ZPFG03] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.

Index