Verification of the C0 Compiler Implementation on the Source Code Level

Dissertation

zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultten
der Universität des Saarlandes

Elena Petrova

petrova@cs.uni-sb.de

Saarbrücken, Mai 2007

Tag des Kolloquiums:	4. Mai 2007
Dekan:	Prof. Dr.-Ing. Thorsten Herfet
Vorsitzender des Prüfungsausschusses:	Prof. Dr. Reinhard Wilhelm
1. Berichterstatter:	Prof. Dr. Wolfgang J. Paul
2. Berichterstatter:	Prof. Dr. Andreas Podelski
akademischer Mitarbeiter:	Dr. Mark Hillebrand

Acknowledgements

First and foremost I would like to thank my thesis advisor Prof. Dr. W.J.Paul for his guidance during this research. I highly appreciate the opportunity I had to join his chair.

I owe special thanks to my husband for his never ended support and encouragement that made this thesis possible.

I am gratefully thankful to all the members of the Prof. Paul's chair and, especially, to Dirk Leinenbach for his priceless advice in technical writing and presentation.

I would like to thank Norbert Schirmer for answering my countless questions.

Last but not least, I thank my family for their support over all years of my study.

Abstract

This thesis concerns practical application of two methods for program verification. The programming language we consider is a C dialect, called C0, which supports dynamic memory allocation, recursion, pointer types, etc.

First, we verify a program using a formalization of small-step semantics of C0. The example we study is a small loop program, which allocates a linked list of the given length on the heap.

Second, we describe the verification of a compiler implementation in a Hoare Logic in the sense of partial correctness. The source and implementation language of the compiler is C0.

The correctness statement is divided into independent parts: i) the correctness of the compilation algorithm with respect to the target machine and ii) the correctness of the implementation with respect to the specified algorithm. This thesis considers the second task.

We give the formal specification of the compilation algorithm and develop the connection of the implementation data structures to the abstract types used in the specification. Finally, we show the correctness of the compiler implementation with respect to the specification.

Zusammenfassung

Die vorliegende Arbeit befasst sich mit der praktischen Anwendung von zwei Methoden zur Programmverifikation. Die Programmiersprache, die dabei betrachtet wird, ist C0, ein C Dialekt, der unter anderem dynamische Allokation von Speicher, Rekursion und Pointer unterstützt.

Zuerst beweisen wir die Korrektheit eines Programms mit Hilfe der formalen Small-Step Semantik von C0. Das Beispiel, das wir untersuchen, ist ein kleines Programm, das in einer Schleife eine gelinkte Liste gegebener Länge auf dem Heap alloziert.

Danach beschreiben wir die Verifikation einer Compiler-Implementierung mittels einer Hoare Logik für partielle Korrektheit. Die Quell- und Implementierungssprache des Compilers ist C0.

Die Korrektheitsaussage ist in unabhängige Aufgaben aufgeteilt: i) die Korrektheit des Compileralgorithmus bezüglich der Zielarchitektur und ii) die Korrektheit der Implementierung bezglich dieses Algorithmus. Die vorliegende Arbeit beschftigt sich mit der zweiten Aufgabe.

Wir beschreiben die formale Spezifikation des Compileralgorithmus und entwickeln eine Verbindung zwischen den Datenstrukturen der Implementierung und den abstrakten Typen, die in der Spezifikation benutzt werden. Schließlich zeigen wir die Korrektheit der Compilerimplementierung bezüglich der Spezifikation.

Extended Abstract

The goal of this thesis is to show that formal verification of non-toy programs written in a programming language, that provides such features as dynamic memory allocation, recursion, pointer data structures, can be feasible. As the source language for this work the C0 programming language is taken. This is a dialect of standard C with some restrictions that made formalization of its semantics compact.

In the first part of the thesis we shortly describe the formal small-step semantics of the source language (developed in [1]). We study a feasibility of verification directly in the small-steps semantics. As an example we verify a small loop program, which constructs a linked list on the program heap. We develop a high-level specification connecting the example program constructs and states of the corresponding C0-machine before and after the procedure execution. The proof that execution of the program in the frame of the semantics satisfies the developed specification is given. The proof combines the simulation of the procedure execution on the C0-machine with arguing on the predicates that abstract the program variables to memory states of the machine.

The second part of the thesis concerns programs verification using Hoare Logic and is a part of the Verisoft project [2]. The work is done supported by a verification environment created by Norbert Schirmer [3] for the Isabelle/HOL theorem prover [4]. This verification environment allows application of the Hoare rules (function calls are supported) to some program in an automated fashion being provided with the program specification. The idea of the second part is to show that even large programs can be successfully verified using this environment. As an application example we use a simple non-optimizing compiler for the C0 language with VAMP assembler [5] as the target language.

First, we present the general idea of the verification goal dividing it into two tasks: i) verification of the compilation algorithm independently from the implementation, ii) verification of the implementation in C0 against the specified algorithm. The task of the presented work is to accomplish the latter. We present the set of functions specifying the compiler behaviour although the proof that this specification is correct is the part of Dirk Leinenbach's thesis [1]. We only inherit the formalism and some properties of this model.

The verification of the source code includes three main tasks we concentrate on in details. First, the description of the pointer data structures used to organize data in the implementation has to be done. For every data structure we find a so called abstraction function which states its relation to the abstract data types developed for the algorithm specification. Second, for each procedure included in the implementation its specification in the form of pre-/postconditions is developed. Finally, we carry out the proofs that these specifications are indeed satisfied by the implementation. The verification of the main implementation procedure embodies the whole implementation correctness proof.

All the formal models that this work is based on are developed in the frame of the theorem proving assistants Isabelle/HOL and, partially, in PVS [6](for the first verification example of the work). All the proofs done during the work on this thesis are carried out with their support as well.

Erweiterte Zusammenfassung

Das Ziel dieser Doktorarbeit ist es zu zeigen, dass formale Verifikation auf Programmen mit industriell relevanter Größe anwendbar ist. Die Programme sind in der Programmiersprache C0 geschrieben die dynamische Speicher Allokation, Rekursion und Zeiger Datenstrukturen unterstützt. C0 ist ein Dialekt von Standard C mit einigen Restriktionen die eine kompakte Formalisierung der Semantik ermoglichen.

In dem ersten Teil der Doktorarbeit wird die formale Small-Step Semantik der Quellsprache beschrieben (siehe auch [1]). Wir studieren die Durchführbarkeit der Verifikation in eben dieser Semantik. Als Beispiel verifizieren wir ein kleines Schleifen Programm, welches eine verkette Liste auf dem Programm Heap alloziert. Wir entwickeln eine abstrakte Spezifikation, die das Beispielprogramm mit dem Zustand der entsprechenden C0 Maschine vor und nach der Ausführung der Prozedur in Beziehung setzt. Es wird bewiesen, dass die Ausführung dieses Programms entsprechend der C0 Semantik der entwickelten Spezifikation genügt. Im Beweis wird die Simulation der Prozedur Ausführung in der C0 Maschine mit Argumenten über spezielle Prädikate kombiniert, welche die Programm Variablen zu einem Speicher Zustand in der C0 Maschine abstrahieren.

Im zweiten Teil der Doktorarbeit geht es um Programm Verifikation mittels Hoare Logik wie sie auch im Verisoft Projekt [2] eingesetzt wird. Die getane Arbeit nutzt die Verifikationsumgebung von Norbert Schirmer [3] welche in den Isabelle/HOL Theorem Beweiser [4] integriert wurde. Diese Verifikationsumgebung erlaubt die Anwendung der Hoare Regeln (Funktions Aufrufe werden unterstützt) auf ein Programm in einer automatisierten Art und Weise basierend auf einer gegebenen Programm Spezifikation. Die Idee des zweiten Teils ist es zu zeigen, dass große Programme erfolgreich in dieser Umgebung verifiziert werden konnen. Als ein Anwendungsbeispiel benutzten wir einen einfachen nicht optimierenden Compiler für die Sprache C0, mit VAMP Assembler Code [5] als Zielsprache.

Zuerst präsentieren wir die generellen Ideen hinter dem Verifikations-Ziel und unterteilen es in zwei Bereiche: i) Verifikation des Compiler Algorithmus unabhängig von der Implementierung ii) Verifikation der C0 Implementierung gegen den spezifizierten Algorithmus. Ziel der Doktorarbeit ist es letzteres durchzuführen. Wir präsentieren eine Menge von Funktionen die das Compiler Verhalten spezifizieren; der Beweis, dass diese Spezifikation korrekt ist, ist Teil von Dirk Leinenbachs Doktorarbeit [1]. Wir übernehmen lediglich die Formalisierung und einige Eigenschaften dieses Modells.

Die Verifikation des Quellcodes umschließt drei Hauptaufgaben auf die wir uns im Detail konzentrieren. Zuerst werden die Zeiger Datenstrukturen zur Repräsentation der Daten in der Implementierung beschrieben. Für jede Datenstruktur finden wir eine so genannte Abstraktionsfunktion welche die Relation zu dem abstrakten Datentyp angibt, der für die Spezifikation des Algorithmus entwickelt wurde. Zweitens, für jede Prozedur aus der Implementierung wird die zugehörige Spezifikation in Form von Vor-/Nachbedingungen entwickelt. Schließlich wird gezeigt dass diese Spezifikation von der Implementierung erfüllt wird. Die Verifikation der Haupt-Implementierungsprozedur umschließt den gesamten Implementierungs Korrektheitsbeweis.

Alle formalen Modelle auf die diese Doktorarbeit sich beziehen wurden in der Verifikationsumgebung des Theorem Beweisers Isabelle/HOL und teilweise auch in PVS [6] (für das initiale Beispiel dieser Doktorarbeit) formalisiert. Alle Beweise die während der Arbeiten an dieser Doktorarbeit entstanden wurden ebenfalls innerhalb dieser Umgebungen durchgeführt.

Contents

List of Figures

Chapter 1

Introduction

Motivation As long as it is possible to write programs controlling computer behaviour, these programs contain bugs noticed by their authors or users only after programs fail or give incorrect results. In the modern world connected by networks a large help in searching of mistakes are hackers, showing how the applications created by software giants can be easily attacked. With growing size of programs the number of hidden mistakes grows as well, giving work to armies of debuggers, testers, programmers. Catching errors often requires even more time than the program development itself and as the direct consequence more money.

Reliability becomes a key issue for computer systems, especially for those used in e.g. automotive engineering, banking, space missions etc. Unfortunately, all experimental means cannot guarantee that the final version of a program is stable and free of errors. The more complex a program is the more combinations of input data and working scenarios are needed to be tested. Even an extensive testing cannot cover all of them. That makes such a method for justifying program correctness unsafe. In this situation the absence of errors in a program can be proven only mathematically.

Nowadays the mathematical arguing is needed to be not only highly formal, but also provable by some proving assistant. This aspect becomes more important with a growing size of models we prove the correctness of, as it is often impossible to present a model and related mathematical proofs up to the last detail in a compact and clear way on paper. The influence of the human factor can easily be the cause of mistakes and gaps in a model. This dramatically decreases when developing and proving a theory with support of proof tools. Even things, which are claimed as obvious by a human, need to be shown formally and precisely in the frame of a proving assistant.

There is a large number of formalized semantics for various programming languages and theories that allow to formally argue about program execution and correctness. Actually, it is a very established field covered by numerous textbooks (e.g. [7]). So far, despite the well established theory, practical program verification is still not just an engineering task. Verification of non-toy programs written in realistic languages is often considered as very laborious and even infeasible. Thus, verification of programs, which have several thousand lines of code and are written in languages featuring function calls, recursion, and dynamic data structures is an issue of the day. Correctness of programs is not only interesting as an independent problem but rather in the context of pervasive system verification. A program also needs to be correct as a part of the system, i.e. concurrently running with other programs. This issue stresses a problem of verification based on small-step semantics, that allow to argue about interleaving and non-terminating execution. However, even a verified program does not imply the correctness of its object code executed by a processor. This can only be ensured by *verified* compilation from the source program to the object code of the target architecture. Thus, compiler correctness is a very important question in correctness statements about entire computer systems.

Problems This thesis addresses the following problems:

1. Verification of a small program directly in the frame of small-step semantics for a C-like language.

2. Verification of a C-like program of realistic size (thousands of lines of code), which extensively uses non-trivial dynamic data structures and function calls, in terms of any semantics.

3. A proof that a compiler for a C-like language, which is implemented in that language itself, is correct. According to the works of Chirica and Martin [8], and Goerigk et al [9], the compiler correctness task can be split into several subproblems. These are: i) formal specification and correctness proof of the compilation algorithm; ii) verification of an implementation with respect to the specification; iii) correct bootstrapping. The latter is also known as *the bootstrapping problem* i.e. how do we get a first verified executable of the compiler verified on the source code level.

Framework Work on this thesis is mostly done in the frame of the Verisoft project [2], whose task is pervasive verification of entire computer systems (aimed in [10], [11]) including hardware as well as software. The lowest layer of the system is presented the by the formally verified VAMP processor, a 32-bit RISC CPU with DLX instruction set [5]. Other system layers are a microkernel, a simple operating system, and several applications (including an email client and signature software).

The high-level language of applications is the C0 programming language. C0 is a type-safe subset of C with some restrictions that make it more close to Pascal and allow us to keep the language semantics small. This is very important, since too complicated semantics make formal correctness proof of software infeasible (compare results in [12], [13]). However, despite these restrictions, C0 is powerful enough to be used for the implementation of real computer system applications. It provides function calls, recursion, derived and pointer data types, and dynamic data allocation.

Results In the first part of the thesis we present verification of a small C0 program directly in the frame of a C0 small-step semantics. The program we have verified is a loop program that creates a linked list data structure on the heap. We use the semantics for C0 that has been developed in [1] and formalized in PVS [6] and in the Isabelle/HOL theorem prover [4].

The second part of the thesis concerns verification of the implementation of a simple non-optimizing compiler. The compiler handles the C0 language with an extensive type system, large collection of expressions and statements, including dynamic allocation of memory. The compiler implementation is also written in C0, the target language is the assembler language of the VAMP. The verification concerns only a compilation algorithm, we ignore the parsing phase and transformation of assembler programs to object code.

The correctness proof is carried out with support of the verification environment which is developed as a part of Verisoft by Schirmer [14]. The environment is based on a Hoare Logic and integrated in Isabelle/HOL. It offers a convenient framework for verification of programs written in imperative languages.

The compiler implementation is a realistic application, which operates with non-trivial pointer data structures, dynamically allocated data (that makes it harder to argue about the stability of the memory) and includes recursive procedures. Thus, one of the main messages of the thesis is that verification of such programs is not only feasible, but can be done with a reasonable effort.

Despite the successful verification using the small step semantics, we outline some aspects that make its use strongly inconvenient for larger programs. However, the small-step semantics allows to argue about interleaving program execution, non-terminating programs, and memory management, which is impossible in the Hoare Logic based on big-step semantics. Since it seems to be infeasible to handle large programs only using the small step semantics, the combination of the mentioned methods can solve the problem. Portions of the code (as large as possible) are to be handled in the Hoare Logic. Then the relevant results are to be transferred to the more detailed (small-step) semantics using equivalence theorems between the semantics. The rest of the properties about a program execution need to be shown in the frame of the small-step semantics. If necessary, some properties can be transferred back to the Hoare Logic framework via equivalence theorems.

The second part of the thesis, being considered together with some other tasks of the Verisoft project, is able to solve the third problem mentioned above. The correctness of the compiler specification (i.e. the compilation algorithm) is shown by Leinenbach [1]. One of the ongoing works in Verisoft is the development of a translation validation tool for the compiler. The translation validation approach (see Section 1.1) allows to justify the correctness of an individual run of a compiler for a specific C0 program. In case the work on the tool is successful, we solve the third subtask of the compiler verification problem. We need translation validation only to show the correctness of the application of the C0 compiler on its own source code.

Outline In the reminder of Chapter 1 we summarize the related work to this thesis, shortly introduce the Verisoft project, the language C0, and the Isabelle/HOL theorem prover. We also introduce some basic notations we use in the following chapters of the thesis.

In Chapter 2 we highlight the most important parts of the formal semantics of the C0 programming language.

Chapter 3 describes the example of verification of a small procedure creating a linked list in the frame of the given semantics.

Chapter 4 introduces the verification environment for programs written in an imperative language.

Chapters 5-8 concern verification of the compiler implementation. We describe some implementation details, modelling of the involved data structures, connection of the implementation with its specification, and details of the correctness proof.

In Chapter 9 we summarize the results and outline work building on this thesis.

1.1 Related Work

Related work for this thesis can be summarized into two main groups: i) work about practical verification of programs, ii) work about compiler verification on the abstract level as well as verification of their implementation.

Program Verification examples In the literature there are a few published verification examples of program code that are carried out in a theorem prover and rely on the underlying language semantics. We point out some of the interesting cases.

A large collection of verified program code (mostly in low-level languages) is one of the results of the famous CLI project [10]. The main goal of this project was a stack of verified components, starting from the hardware level up to applications written in a high level language.

In the frame of the CLI project a simple program for iterative multiplication was verified. The program is written in a subset of Gypsy, a high-level language called Micro-Gypsy which supports simple types, one dimensional arrays, if and loop statements, recursive procedures [15]. This example illustrates the approach to verifying programs in the semantics for a Micro-Gypsy. Pointer structures and memory allocation are out of consideration. The semantics was developed as part of the work on verification of the compiler specification from Micro-Gypsy to the target assembler language Piton for the FM8502 processor, which is also a part of the CLI project.

Based on the framework developed in the CLI project, Wilding [16] verified an application which generates moves for the game Nim. This application is written in Piton, a high-level assembler language with typed data and recursive procedures. Verification is carried out with the Boyer-Moore theorem prover [17], also known as Nqthm. The Piton implementation has about 300 line of code. Wilding points out that verification is very time consuming and difficult, thus assembler-like languages are not a proper means for verification of applications that can be implemented in a high level language.

One of the important results in the CLI project is the verification of an operating system microkernel called KIT [18]. The implementation of the kernel is very simple and only includes some basic services of the kernel (process scheduling, error handling, message passing, and an interface to asynchronous devices). The microkernel is implemented in about 350 lines of the assembler language of an artificial target machine with a von Neumann architecture. The main result of the verification is the theorem which demonstrates that the behavior of a single task running under the kernel implements an abstract definition of a process. KIT and its specification are defined in the Boyer-Moore logic, and the proof is done with the Boyer-Moore theorem prover [17].

Another great proof effort in the CLI project is the verification of the Berkeley C string library. Based on the techniques developed for verification of Piton programs, Yu has verified 21 of 22 subroutines of the library on the binary code level [19]. The only function left out needed the formalization of IO, so it was excluded. The correctness proofs were carried out with respect to the operational semantics covering about 80% of instructions of the MC68020 microprocessor. The machine code for the MC68020 was produced by the standard `gcc` compiler with optimization. The model of the processor was developed and the proofs were carried out in Nqthm. This work covers about 200 lines of C source code, which corresponds to about 500 instructions in assembler and 1300 in binary code, respectively.

In [20] Abrial presented the B method for developing safety critical software, that was then successfully applied in the industry. It is aimed to development of fault free software from the beginning, rather than verifying an existent implementation. First, an abstract model of the software and its properties are formalized and have to be proven. Proof obligations are mechanically generated, the proofs can be carried out using automatic and interactive tools. Then, the abstract model is manually transformed into the concrete model written in B0, which is a subset of the B language. During this phase abstract data are linked to concrete types (integer, boolean, and their arrays), the abstract operations must be expressed using basic control structures (sequence, alternative, loop). The following step is the validation of the concrete model with respect to the abstract one. This is again carried out using the proof obligation generator, by the automatic and interactive provers.

It was first applied in a large project by Siemens Transportation Systems (formerly Matra Transport) for the development of the safety critical software of Paris underground metro Line 14 [21]. The initial version of the provided modelling and proving toolkit was far from being applied in a large project. It allowed only rather small models to be proven. Thus, the method first needed to be tuned to answer the industrial needs. Industrialization lasted 4 years and was aimed to: i) enhancing the toolkit performances to handle large volumes of modelling, ii) improving the proof techniques. The nature of the language (no structure or pointer types) allowed a high degree of automatization (application of model checking techniques). The method was extended by the automatic translation of the concrete model to the implementation in the Ada programming language, required by the host systems used in the project. The resulting technique (also mentioned as Siemens B Method) was applied in the project.

Thus, the proofs were done for the concrete model and then Ada source code was generated from it. The verification included i) showing properties of the system with respect to the abstract model in B; ii) the correctness proof of the concrete model with respect to the abstract one. In [21] authors report on 80% automatic proof coverage, 10% of proofs, required additional rules to be provided, and 2254 lemmas left to be proven interactively. The final amount of produced Ada code is 86.000 lines. The large size of the implementation is due to copies of template code for different blocks, and a special way of implementing Ada code including flattening of complex expressions via intermediate variables. The convincing results of this project resulted in the further development of Siemens B method and recent use of it in Roissy VAL project [22], where the automatization degree reached 97%.

Another example of successful use of program code verification in industry is the SPARK language in the SHOLIS project of *Praxis Critical System* [23]. SPARK is a subset of Ada provided with a commercial toolset that allows to do data flow analysis and prove partial correctness of the programs. SPARK has a large number of severe restrictions compared with original Ada. The most significant difference to C0: no recursion and no derived and pointer data types. The size of the application is about 27,000 lines with ratio of declarations/statements about 50/50. The automatization degree in discharging verification conditions (about 9000) produced by the tools is about 75 percent.

The feasibility of the mechanized verification of programs with pointer data structures by means of Hoare Logic were explicitly shown by work of Mehta and Nipkow [24]. Their paper presents a verification method for a general imperative language with pointers, which can model a family of programming languages. The authors apply the method for the verification of a Schorr-Waite algorithm implementation in a Hoare Logic framework in the Isabelle/HOL theorem prover and showed that this approach is scalable to larger programs.

The verification environment we use in this work was previously tested on several simpler study cases. Implementation of the libraries (collection of basic operations) for such data structures as doubly linked lists and strings were verified with the use of the verification environment in the frame of the Verisoft project. A study case for verification of BDD algorithms was carried out by Ortner [25]. These examples obviously deal with much simpler data structures than the compiler implementation.

One of the projects that have an intention for verification of a real-sized system is the work in NICTA on verification of the L4 microkernel [26], which is written in C++ and assembler languages. They present an extension of Schirmer's verification environment with a low-level heap memory model. It allows to mix up the untyped low-level heap view with the typed multiple heaps and to argue on the word level when necessary. This model allows to avoid type safety restrictions in pointer programs and to verify programs written in C rather than in a subset with type-safety requirements. They present a case study example for their environment, which is a function from L4, but they have not proven the correctness of much code so far.

One of the approaches for software verification is connected with the recent formalism of separation logic, which facilitates reasoning about code with pointers [27]. Separation logic simplifies reasoning that certain data structures sharing the heap stay disjoint, which was the most wearisome part of the work we have done. The separation logic approach has proved successful in a number of studies, including the Schorr-Waite graph marking algorithm [28] and an abstract version of a copying garbage collector [29]. So far, there is not much code proved with mechanized tool support. For example, work of Weber [30] just does a step on the way to integrate separation logic in the verification framework based on Isabelle/HOL.

The presented work is a step towards the verification of large software systems. It considers an implementation language without very severe limitations (e.g. being compared to the languages that are used in the industrial projects mentioned above). We have verified a medium-sized (approximately 1500 lines of C0 code, organized in 60 procedures) program, which is not an artificial example but actually used in the Verisoft project (e.g. to compile the kernel and operating system code). Since the compiler uses about 20 different pointer data structures, our work required formalization of every one of them, whereas completed examples presented above were concerned with much smaller number of data structures.

Correctness of the compiler implementation Compiler correctness is not a new issue (reports on this topic started to appear in the 1960s); there are a number of researches and scientific projects that were carried out in this field. Most of the papers in this field (especially early ones) considered only verification of a compilation algorithm rather than verification of its implementation. They claim the correct translation between different state machines. One of the earliest works in this field is a work of McCarthy [31] which only focuses on compilation of arithmetic expressions, not a whole set of language constructs. Among the recent works is a work of Strecker and Nipkow on verification of a compilation algorithm from Java to Java byte code and formal byte code verification [32,33]. In the latter the authors consider the semantics (big-step) and compilation of an object-oriented language, which is a substantial subset of Java. Their work only concentrates on the language analysis without arguing on the implementation correctness of the compiler.

As part of the work on the CLI stack, Moore [34] pointed out that to finish the correctness proof of the compiler (from Piton language to machine code of FM8502), the verification of the compiler implementation has to be done. His work however, as well as the VLisp project [35], covers only verification of the compiler specification.

Verification of the implementation of compilers for non-toy languages is often mentioned as infeasible in the literature [36]. Therefore, most of the works in the field of compiler correctness aimed to avoid verification of the implementation itself.

One of the approaches that aim to produce trusted compilers is compiler generation. In this case the source of the compiling program is created with the use of a special tool from a formal description of semantics for input/output languages and translation between them. Most papers in this direction (especially early ones, e.g. [37, 38]) have the disadvantage, that the compiler produces the object code that runs much slower than the code produced by their handwritten analogs. Moreover, only some of them (e.g. [39]) proved the compilation algorithm behind their systems to be correct. One of the successful works (for a realistic language) in this area is the Cantor compiler generator [40,41]. This was developed to generate code for the compiler from a realistic source language (a subset of Ada) and to real target machines (HP Precision and SPARK). The great advantage is that the generated compiler can be easily adjusted to new semantics requirements. Despite this, the main weak points in this approach are that the implementation of the generation tool (written in Perl) is not verified and hence cannot be completely reliable, the algorithm was proven correct only manually.

One of the works close to the Verisoft compiler subtask is by Blazy, Dargaye, and Leroy [42, 43]. They present two-step translation from a subset of C called Clight first to the intermediate language Cminor and then to the PowerPC assembler language (with some optimizations). Clight incorporates more features of standard C language than C0, e.g. two more loop constructs, `break`, `continue` statements, prefix/postfix operation, pointer arithmetic, etc. The semantics of the source and target languages (big-step) as well as the translation between them have been specified in the Coq proof assistant. An executable compiler was obtained by automatic extraction of executable code from the Coq specification rather than independently written and proven correct. There are several points in what their work differs from our project. The clear advantage of theirs is an optimizing compiler and extended set of statements. However, they use the big step semantics, that makes it harder to argue about interleaving execution of communicating processes on the C machine. Currently they do not deal with dynamic data structures. Moreover, implementation correctness of their compiler is not formally stated. The extraction step is not verified, that makes the correctness issue of the running compiler weaker.

The weak point of the techniques mentioned above is that they rely on correct execution of some unverified software, e.g. certifiers, compiler generators etc.

Along with the verification of implementation, which allows to state the correctness of compilation for any input program, there is an alternative method that allows to verify a particular run of a compiler.

The *translation validation* approach was proposed by Pnueli et al [44] and has the following idea: the source and target program, which is the output from some compiler/translator, are provided to a so called analyzer, that automatically checks whether the target program correctly implements the source program. The analyzer either establishes the equivalence between inputs or produces a counter-example. A counter-example is produced not only if there exists a compilation bug but also if the analyzer could not decide whether two inputs are equivalent (such a situation is often called a false alarm). The equivalence check is based on a refinement relation between corresponding semantics, the result of this step is a proof script checkable by some proof checker. The paper presents theoretical insights on construction of such a tool. The approach is rather general, suitable for a large range of input/output languages. Authors illustrate it with a small example, where the input language is a synchronous data-flow language called Signal and the output is sequential C.

A comparable approach for compiler verification presented in [45] is called *credible compilation*. That work concentrates on validation of a variety of standard code optimizations performed by compilers. The authors present theoretical foundations of their approach, where the equivalence check is derived from Floyd-Hoare rules. The equivalence is checked between programs in control-flow graph representation before and after optimizing transformations. The parsing and code generation phases are not considered, the authors only outline how the framework can be extended to handle code generation. Moreover, in [46] the method is extended by optimizations concerning pointer analysis.

In [47, 48] the original translation validation approach results in practical applications. The former paper presents a translation validation tool (CVT) and its use in the context of the SACRES project. The tool performs automatic validation of translation from designs in DC+ synchronous language into C and Ada. The source and target programs follow severe limitations, in particular, they are restricted to single-loop programs. The tool was successfully tested on an industrial-sized program of a few thousand lines long. It was partitioned manually into 5 units every of which were separately compiled and validated. A very few cases could not be validated by the current implementation of the tool.

The second paper presents a methodology for validation of optimizing compilers. The authors distinguish between structure preserving and modifying optimizations and present theory that allows to handle both of them. Moreover, they describe a prototype tool called VOC-64 for validation of the optimizations performed by the SGI Pro-64 compiler. The input/output language for the validation is the intermediate language WHIRL of the compiler. The tool handles most standard optimizations (including loop unrolling) and generates verification conditions to be provided into CVT. The coupling of tools and extension of VOC to handle other loop optimization is still to be done.

The work of Necula [49] presents the prototype translation validation infrastructure (TVI) for the GNU C compiler, that deal with optimizations applied to a program. TVI, based on symbolic evaluation technique, compares the program in the intermediate format, which is used in the GNU C compiler, before and after each compiler pass. The key point of the work is the real, expressive language without additional simplifications. The tool handles only the intermediate phases of compilation, ignoring the parser and the code generator. The tool is able to handle most optimizations applied by *gcc*. It was tested on real-sized software systems, such as the compiler itself and the Linux kernel. The current drawback is a relatively large number of false alarms during validation.

Thus, besides the high automation of this approach, it was also applied for optimizing compilers. However, its main limitation is that a bug in translation can only be revealed when the compiler is run on a program that triggers the bug. Our approach, i.e. the total verification, ensures the correct translation for any input. The translation validation appears to be the solution for the bootstrapping problem, i.e. allows to get the verified executable of the compiler, verified on the source code level.

The Verifix project [50], initiated in Universities of Karlsruhe, Ulm, and Kiel, considered different topics concerning correct compiler construction. Goerigk and Simon showed that the full compiler correctness includes tree subtasks: namely correctness of the compiling specification, the compiler as a high level program, and the compiler executable. Gaul and Zimmermann [51] present an elegant theory for the translation of intermediate languages to the machine languages. This theory was partially formalized in the PVS theorem prover. Moreover, the implementation of the compiler for ComLisp (a subset of Common Lisp) into the binary machine code of the Inmos Transputer computer was verified on the machine code level by a manual syntactical check [9]. The compiler implementation $\pi_{\mathcal{C}}$ is compiled with an existing unverified compiler for CommonLisp into executable $m_{\mathcal{C}}$. Then a syntactical check of $m_{\mathcal{C}}$ against the code for $\pi_{\mathcal{C}}$ expected from the verified compilation algorithm is performed. The authors mention that the second subtask, i.e. the correct implementation of π_C on the source code level is to be shown formally.

1.2 C0 Programming Language

In this section we present a short summary about the C0 programming language. Its concrete syntax and visibility rules for variables are very similar to the standard C. Operational semantics, though, is similar to Pascal. The main language restrictions in comparison to C are:

1. No prefix and postfix arithmetic operations, e.g. i++
2. No pointer arithmetic
3. Every function has only one `return` statement as the last statement of its body
4. No pointers to local variables
5. No pointers to functions
6. The size of arrays has to be statically defined
7. Side effects are forbidden (e.g. no function calls as a part of expressions)

These restrictions simplify formal definition of the language semantics, although the restriction of the language functionality is not crucial. The language is still powerful enough to implement any application we need in the frame of the Verisoft project.

C0 provides some basic types as well as some complex type constructors. The supported types are:

- Simple (or basic) types
  - `int`, 32-bit signed integers with value range $\{-2^{31}, \ldots, 2^{31}-1\}$
  - `unsigned int`, 32-bit unsigned integers with value range $\{0, \ldots, 2^{32}-1\}$
  - `char`, 8-bit signed integers with value range $\{-128, \ldots, 127\}$
  - `bool`, boolean type with value range $\{true, false\}$
- Complex types (based on simple ones)
  - Pointers (typed, there is no empty `void` type)
  - Arrays (of fixed, statically defined size)
  - Structs (a tuple type with named components)

So called complex constants, which are constant instances of a struct or an array type, are also provided. The usage of the complex constants is restricted, they are allowed to appear only as the right hand side of an assignment.

In Table 1.1 we summarize the expressions supported by C0. In the column "type" we have for arithmetic expressions keywords depending on types that are allowed for the expression: *bool* - only for the boolean type; *int* - for the integer type; *arith* - for the integer and unsigned integer types; *num* for the integer, unsigned integer, and char types; *elem* for the types noted by *num* and pointer types. If the types for both sides of an arithmetic expression are equal, we write it once. Operations 20-22 are the type casts to convert between the numerical types.

id	expr	type	id	expr	type
0	|| (logical or)	bool	15	* (times)	arith
1	&& (logical and)	bool	16	/ (divides)	arith
2	== (equal)	elem	17	! (logical not)	bool
3	!= (not equal)	elem	18	(bitwise negation)	arith
4	<	num	19	- (unary minus)	int
5	>	num	20	unsigned()	num
6	<=	num	21	int()	num
7	>=	num	22	char()	num
8	<< (left shift)	arith, num	23	* (pointer dereferencing)	
9	>> (right shift)	arith, num	24	&(address of)	
10	— (bitwise or)	arith	25	(struct field access)	
11	& (bitwise and)	arith	26	(array element access)	
12	$\wedge$ (bitwise xor)		27	(constant)	
13	+ (plus)	arith	28	(variable access)	
14	- (minus)	arith			

Table 1.1: C0 expressions

The language provides the following statements:

- assignment (allowed for simple types and structs, also possible for arrays if the right side of the assignment is a complex constant)
- `while`-loop
- conditional statement `if (e) then {...} else {...}`
- function call (the result of a function call is allowed to be assigned only to a variable)
- return from a function (only as the last statement of a program)
- a PASCAL style new statement. It uses a type t and a pointer p (with matching type) as arguments. It creates on the heap a data object with type t and makes p point to that object.

1.3 Isabelle/HOL

Isabelle/HOL is an interactive theorem prover, which is one of the tools we use to support model developing and carrying out of correctness proofs in the Verisoft project. The results of this work (models and proofs) are fixed as mathematical theories there.

The work presented in the thesis is done completely in the theorem prover without creating a paper-and-pencil version first. It connects several formal models mentioned in Section **??**. The formal models which we present in the thesis are kept close to the their description in Isabelle/HOL, although there might exist another ways to formulate them on paper.

Moreover, using a theorem prover clearly restricts mathematical machinery we are used to form paper-and-pencil proofs. That restriction is connected with the concrete implementation of mathematical objects in the theorem prover. Moreover, the way we describe a function predefines the set of rules, lemmas, and techniques we can apply while proving a statement about it.

The first restriction we confront is the type system, where types are fully recursive, all functions are total and predicate subtypes are not allowed. So, we cannot specify a type for an integer in some range or for lists of a particular length. Object properties of such a kind are needed to be given additionally by a predicate over the type we want to refine.

By technical reasons in Isabelle/HOL it is much easier to use natural *primitive* recursion to formulate recursive functions (compared with specification of *totally* recursive functions). For the former, the recursive function call being applied to an object of a inductive type is only allowed for its direct subobject, e.g. to the tail of a list. Such a scheme allows to prove the termination of recursion automatically. For the latter, we additionally need to specify a measure function, whose value decreases with each recursion step. The monotonicity of this function is also needed to be proved.

List is a build-in type, polymorphic (i.e. lists of any type can be defined), has a large number of proved properties presented as lemmas, and is widely used to define tables, mappings, sequences, strings, bit vectors etc., also in the models presented in the thesis.

Since the proofs which are carried out for the thesis do not exist in the paper-and-pencil form, the sketches of them presented here are extracts from the complete formal versions fixed in the Isabelle/HOL theories. Showing the way how something is proven we omit a number of rewriting and conclusion steps, which are seen as obvious by a human being. These steps are still needed to be explicitly done in the theorem prover to make the sequence of conclusions confirmed as a proof. In the presented proofs, which are shown based on case distinction, we normally present only one of the cases and notice that the others can be done analogously. Of course, in the theorem prover one actually needs to carry out the very similar proofs for each of the cases. Besides the lemmas we present in the theses there is a large number of lemmas we do not mention here since they are trivial from the human point of view but are necessary when carrying out proofs in Isabelle/HOL. Also, all natural properties of the build-in data types we mention in the thesis (e.g. lists and natural numbers) are actually lemmas in the theorem prover, which we use implicitly in the thesis.

1.4 Basic Notation

In this section we introduce some notation we use in the thesis. $\mathbb{N}$ denotes the set of natural numbers (including zero), $\mathbb{Z}$ denotes the set of integers and $\mathbb{B}$ - the set of boolean values.

We use polymorphic constants ϵ and α to define some error and arbitrary value of an appropriate (derived from the context) type.

We use $\Longrightarrow$ as "implies" to separate assumptions of a lemma from its conclusion.

Definition 1.4.1 Let $f, f' \in A \to B$ be functions. We denote with $f' := f[x := a]$ the function update

$$f'(i) := \begin{cases} a & \text{if } i = x \\ f(i) & \text{if } i \neq x \end{cases}, \quad \text{for all } i \in A$$

In the same way we denote update of component $r.x$ of some record $r = (\ldots, x, \ldots)$ with value a, i.e. $r[x := a]$.

Definition 1.4.2 Let A be a set. $2^A = \{B \mid B \subseteq A\}$ is the **power set** of A.

Definition 1.4.3 Let $c = (x, y)$ be a pair. Then $fst(c) = x$ and $fst(c) = y$ are access functions to the elements of c.

1.4.1 Lists

In this section we present the polymorphic list type and some functions working with lists.

Definition 1.4.4 Let T be a type. Then we define the list type T^* with components of type T inductively:

$$T^* = [] \mid (T \times T^*),$$

where a list is either an empty list $[]$ or a pair $(x, xs) \in T \times T^*$. The first pair component x is the **head** of the list and xs is its **tail**, which is again a list of type T^*

The induction scheme for lists is connected with the definition, where list l is either an empty list $l = []$ or a pair $l = (x, xs)$ with head x and tail xs and has the following form:

$$P([]) \wedge \big(P(xs) \longrightarrow P((x, xs))\big) \Longrightarrow P((x, xs))$$

The property P is true for all lists (x, xs) if i) it is true for an empty list , and ii) validity of P for list (x, xs) follows from $P(xs)$.

Functions working with lists are defined recursively, we present here some of them and their properties we use in the following chapters. Conversion from $x \in T$ to the list with one element is defined by $[x] = (x, [])$. Assembling of several single elements to a list is denoted with $[x, y, z] = (x, (y, (z, [])))$.

Definition 1.4.5 Let $l \in T^*$ be a non-empty list. We access the head and the tail through the function applications $hd(l)$ and $tl(l)$, respectively. The elements of l can be enumerated (starting with 0) and access to the i-th element of the list, where $i \in \mathbb{N}$, is denoted by l_i.

Definition 1.4.6 $\circ$, list concatenation Let $l, l' \in T^*$ be lists, then their is denoted by $l \circ l'$.

Definition 1.4.7 Function $rev : T^* \to T^*$ returns for any list l its **reverse**.

$$rev(l) = \begin{cases} [] & \text{if } l = [] \\ rev(xs) \circ [x] & \text{if } l = (x, xs) \end{cases}$$

Definition 1.4.8 Function $mem : T^* \times T \to \mathbb{B}$ tests whether some element $x \in T$ is a member of list $l \in T^*$. We denote $mem(l, x)$ with $x \in_* l$.

Definition 1.4.9 Function $set : T^* \to 2^T$ returns **set of elements** of a list. We denote $set(l)$ with $\{l\}$.

Definition 1.4.10 Let $l \in T^*$ be a list and $f \in T \to T'$ be a mapping function. Then $map : (T \to T') \times T^* \to T'^*$ maps a list of type T to a list of type T' via function f.

$$map(f, l) = \begin{cases} [] & \text{if } l = [] \\ (f(x), map(f, xs)) & \text{if } l = (x, xs) \end{cases}$$

Definition 1.4.11 Let $l \in T^*$ be a list. Then function $length \in T^* \to \mathbb{N}$ defines the length of the list.

$$length(l) = \begin{cases} 0 & \text{if } l = [] \\ 1 + length(xs) & \text{if } l = (x, xs) \end{cases}$$

Below we denote $length(l)$ with $|l|$.

Definition 1.4.12 Let $l \in A^*$ be a list. The list l is **distinct** if all its elements are different. The function $distinct(l)$, which is defined recursively, has the following property we are interested in:

$$distinct(l) = \forall i, j < |l|.\ i \neq j \longrightarrow l_i \neq l_j$$

Definition 1.4.13 Let $l \in (T \times T')^*$ be a list of pairs. List l is **unique** if the first components of the list elements are distinct.

$$unique(l) = distinct(map(fst, l))$$

Definition 1.4.14 Let $l \in T^*$ be a list and $f \in T \rightarrow \mathbb{B}$ be a function that checks some property of an instance of type T. Then function

$$pfx(f, l) = \begin{cases} [] & \text{if } l = [] \\ [] & \text{if } l = (x, xs) \wedge f(x) \\ (x, pfx(xs)) & \text{if } l = (x, xs) \wedge \neg f(x) \end{cases}$$

takes a prefix of the list until the first element where property f holds.

Definition 1.4.15 Let $l \in T^*$ be a list and $f \in T \rightarrow \mathbb{B}$ be a function that checks some property of element of type T. Then function

$$sfx(f, l) = \begin{cases} [] & \text{if } l = [] \\ xs & \text{if } l = (x, xs) \wedge f(x) \\ sfx(f, xs) & \text{if } l = (x, xs) \wedge \neg f(x) \end{cases}$$

takes a suffix of the list after the first element where property f holds.

It can be shown that these functions are connected as given below:

Lemma 1.4.16 $\mathit{distinct}(xs) \longrightarrow \mathit{rev}(pfx(P, xs)) = sfx((\lambda x.\ \neg P(x)), \mathit{rev}(xs))$

Definition 1.4.17 Let $l \in T^*$ be a non-empty list. Then $\mathit{last}(l)$ returns the last element of the list. We use the following property:

$$l \neq [] \Longrightarrow \mathit{last}(l) = l_{|l|-1}$$

Definition 1.4.18 The following function defines update of list l at position i with value a.

$$\mathit{update}(l, i, a) = \begin{cases} [] & \text{if } l = [] \\ (a, xs) & \text{if } l = (x, xs) \wedge i = 0 \\ (x, \mathit{update}(xs, i - 1, a)) & \text{if } l = (x, xs) \wedge i \neq 0 \end{cases}$$

We denote $\mathit{update}(l, i, a)$ with $l[i := a]$.

Lemma 1.4.19 Update of two appended lists changes only one of them according to the updated position.

$$\begin{aligned} &(i < |l_1| \longrightarrow (l_1 \circ l_2)[i := a] = l_1[i := a] \circ l_2) \wedge \\ &(i \geq |l_1| \longrightarrow (l_1 \circ l_2)[i := a] = l_1 \circ (l_2[i - |l_1| := a])) \end{aligned}$$

Definition 1.4.20 Let $l \in (T \times T')^*$ be a list of pairs and x be some variable of type T. Function $\mathit{map\_of}(l, x)$

- converts l to a function f of type $T \rightarrow T' \cup \epsilon$ such that

$$f(a) = \begin{cases} b & \text{if } (a, b) \in_* l \\ \epsilon & \text{if } a \notin_* \mathit{map}(\mathit{fst}, l) \end{cases}$$

- returns application $f(x)$

If the first list components are distinct the result is clearly defined. ϵ denotes an undefined/error result.

$$\mathit{unique}(l) \wedge (x, y) \in_* l \Longrightarrow \mathit{map\_of}(l, x) = y$$

1.4.2 Choice Operator

We use the choice operator $\varepsilon : (X \rightarrow \mathbb{B}) \rightarrow X$ to select an arbitrary element from the subset of X defined by some choice condition of type $X \rightarrow \mathbb{B}$. If the choice condition is given by a lambda abstraction $\lambda x.t$, we denote $\varepsilon((\lambda x.t))$ by $\varepsilon x.t$. The important property about the choice operator, that if only one element satisfying the choice condition exists, it will be returned.

Lemma 1.4.21 Let $p \in X$ be an element, that satisfy property $P \in X \rightarrow \mathbb{B}$ and there is no other $y \in X$, satisfying this property. Then $\varepsilon(P)$ returns p.

$$P(p) \wedge (\forall y.\ P(y) \longrightarrow x = y) \implies \varepsilon(P) = p$$

1.4.3 Inductive Abstract Types

Let us present syntax of inductive types. The general type scheme of some inductive type T is the following:

$$T = C_0 : (T_{00} \times \ldots \times T_{0k}) \mid \ldots \mid C_n : (T_{n0} \times \ldots \times T_{nm}),$$

where C_i are type constructors, which produce instances of type T from parameters of types T_{ij} (which are optional, so a constructor can have no parameters). Inductive types can be recursive, i.e. parameters of constructors can be of type T as well.

Any instance of type T is an object formed with help of one of the constructors. This means, performing a case distinction on some $p \in T$, one will get $(n + 1)$ cases what this p can be, by the number of constructors included in the type. If p is an instance created using constructor C_0, then there exists some tuple $(p_{00}, \ldots, p_{0k})$ such that it matches type $(T_{00} \times \ldots \times T_{0k})$ (i.e. the parameter type of the constructor) and p is equal to $C_0(p_{00}, \ldots, p_{0k})$. The further cases are analogous.

Let us consider a pair of examples. An inductive type Week which uses constructors without parameters:

$$\mathit{Week} = \mathit{Mon} \mid \mathit{Tue} \mid \mathit{Wed} \mid \mathit{Thu} \mid \mathit{Fri} \mid \mathit{Sat} \mid \mathit{Son}$$

Case distinction scheme on $\mathit{day} \in \mathit{Week}$ will create seven cases to consider, i.e. either $\mathit{day} = \mathit{Mon}$, or $\mathit{day} = \mathit{Tue}$, etc.

Another modification of the Week type:

$$\mathit{working} = \mathit{Mon} \mid \mathit{Tue} \mid \mathit{Wed} \mid \mathit{Thu} \mid \mathit{Fri}$$

$$\mathit{weekend} = \mathit{Sat} \mid \mathit{Son}$$

$$Week = WorkingDay : working \mid WeekendDay : weekend$$

The case distinction for $day \in Week$ will produce two cases: either day is some working day $workd$, i.e. $day = WorkingDay(workd)$; or it is some weekend day $wndd$, i.e. $day = WeekendDay(wndd)$. To consider more cases on what day day actually is, one need to perform further case distinction on variables $workd$ and $wndd$.

To make introduction of inductive types shorter, we will represent them by enumerating possible variants of their instances. For example, the new notation for the type presented above is:

$$p = C_0(p_{00}, \ldots, p_{0k}) \mid \ldots \mid C_n(p_{n0}, \ldots, p_{nm}).$$

Declaring the types of parameters p_{ij} is, of course, necessary.

1.4.4 C0 Type Notation

We introduce the following notation for syntax of C0 types to be used in the following chapters. Every C0 type t is either of:

- simple type $t \in \{int, nat, char, bool\}$, where nat is used to denote unsigned integer values
- pointer type $t = t'*$, where t' is the type of the target object
- array type $t = t'[n]$, where n is number of array elements and t' is their type
- structure type $t = struct\{f_1 : t_1, \ldots, f_n : t_n\}$, where for all $1 \leq i \leq n$ pair $f_i : t_i$ denotes an i-th component of the structure with the name f_i and of type t_i

Chapter 2

Small Step Semantics of C0

In this section we present formal semantics for the C0 programming language, which has been developed by Leinenbach and Paul [1] in the frame of the Verisoft project. We give here only basic definitions without concentrating on details too much. The complete definitions and correctness proofs of lemmas, which are mentioned here, are part of Leinenbach's thesis, which is to appear.

We are especially interested in the syntax of programs, since in the following chapters we consider the compiler of the C0 language as the verification example, and therefor we need to have the formal definition of its input.

We also present the memory model, which is crucial for carrying out proofs directly in the semantics framework as we will show in Chapter 3.

2.1 Abstract Syntax

We use abstract types $nm_{\mathcal{T}}$, nm_v, nm_c, and $nm_{\mathcal{P}}$ to model names of types, variables, structure components, and procedures, respectively.

2.1.1 Types

Definition 2.1.1 We define C0 types by inductive type $\mathcal{T}$. Let $n \in \mathbb{N}$ be a number, $t' \in \mathcal{T}$ be a type, $tn \in nm_{\mathcal{T}}$ be a type name, and $sc \in (nm_c \times \mathcal{T})^*$ be a list of structure components. Then **type** $t \in \mathcal{T}$ is:

$$t = BoolT \mid IntT \mid CharT \mid UsgnT \mid Arr(n, t') \mid Str(sc) \mid Ptr(tn) \mid NullT$$

Types $BoolT$, $IntT$, $CharT$, and $UsgnT$ are called **elementary** types.
We define a **type environment** $tenv$ as a list, whose elements are pairs consisting of a type and its name, i.e. $tenv \in (nm_{\mathcal{T}} \times \mathcal{T})^*$.

To access the content of a complex type we use the following selectors:
$the\_Arr \in \mathcal{T} \to (\mathbb{N} \times \mathcal{T})$, $the\_Str \in \mathcal{T} \to (nm_c \times \mathcal{T})^*$, and $the\_Ptr \in \mathcal{T} \to nm_{\mathcal{T}}$, which are defined in the following way $the\_Arr(Arr(n, t')) = (n, t')$, $the\_Str(Str(sc)) = sc$, $the\_Ptr(Ptr(tn)) = tn$. The result of applying the selectors to the other constructors is undefined.

The predicates $is\_T(t) \in \mathbb{B}$ return true if type t is defined by constructor T, e.g. $is\_Ptr(Ptr(tn)) = True$ and $is\_Arr(Ptr(tn)) = False$.

Thus, a type can be one of the elementary types, an array type with n elements of any type t', a structure type with fields (components) modelled as a list of pairs of names and types sc, a pointer type or a special type $NullT$ to identify the type of the null pointer. The pointer type constructor takes a type name instead of a type itself to provide the possibility to define a pointer to a type, which was not defined before, what is used to create self-linking or mutual types (e.g. lists). The null pointer type constructor is needed to keep any expression including the null pointer constant well-typed.

To fix the correct syntax for types (e.g. to prohibit empty arrays and structures) we need to set some limitations on that general type.

Definition 2.1.2 Let $t \in \mathcal{T}$ be a type. Let tt be a type environment. We define t to be valid with respect to tt if the following predicate holds:

$$valid_{\mathcal{T}}(t) = \begin{cases} True & \text{if } t \text{ is elementary} \vee \\ & \quad t = NullT \\ n > 0 \wedge valid_{\mathcal{T}}(t') & \text{if } t = Arr(n, t') \\ tn \in_* map(fst, tt) & \text{if } \;= Ptr(tn) \\ sc \neq [] \wedge unique(sc) \wedge & \\ \forall c \in_* sc.\ valid_{\mathcal{T}}(snd(c)) & \text{if } t = Str(sc) \end{cases}$$

Definition 2.1.3 Let $t \in \mathcal{T}$ be a type. We define the **relative size** of t by the following function:

$$tsize(t) = \begin{cases} 1 & \text{if } t \text{ is elementary} \vee t = NullT \\ n * tsize(t') & \text{if } t = Arr(n, t') \\ \sum_{i=0}^{|sc|-1} tsize(sc_i) & \text{if } t = Str(sc) \end{cases}$$

2.1.2 Variables

Program variables are defined by their name and type.

Definition 2.1.4 Let $vn \in nm_v$ be a variable name and $t \in \mathcal{T}$ be a type. Then

$$v = (vn, t) \in (nm_v \times \mathcal{T})$$

be a **variable declaration**. A **symbol table** is a list of variable declarations.

Set of the global variables of a program and local variables of every procedure are modelled as symbol tables.

Definition 2.1.5 Let $st \in (nm_v \times \mathcal{T})^*$ be a symbol table. A correctly declared symbol table is defined by:

$$valid_{ST}(st) = unique(st) \wedge \forall v \in_* st.\ valid_{\mathcal{T}}(snd(v))$$

$$\begin{aligned} op_b \; = \; & plus \mid minus \mid times \mid divide \mid bw\_or \mid bw\_and \mid bw\_xor \mid sh\_left \mid sh\_right \mid \\ & greater \mid less \mid equal \mid greaterequal \mid lessequal \mid notequal \end{aligned}$$

$$op_l = log\_and \mid log\_or$$

$$op_u = un\_minus \mid bw\_neg \mid log\_not \mid to\_int \mid to\_unsigned \mid to\_char$$

Figure 2.1: Expression operators

2.1.3 Constants

We consider constants of elementary types.

Definition 2.1.6 Constants are defined by inductive type $\mathcal{C}$. Let $n \in \mathbb{N}$, $i \in \mathbb{Z}$ be numerical values and $b \in \mathbb{B}$ be a boolean value. Then constant c of type $\mathcal{C}$ is defined in the following way:

$$c = Bool(b) \mid Int(i) \mid Char(i) \mid Unsg(n) \mid Nil,$$

where Nil defines the null pointer constant.

Since constant values are infinite (caused by the Isabelle/HOL type system), we have to introduce some predicates setting restrictions on the value ranges. Thus, we need to establish the ranges of constant values that can be stored in the memory with cells of some finite size. Thus, for all numerical types predicates $is\_valid\_\{int, nat, char\}$ check whether the number is in the appropriate range. We use is_valid_nat predicate for testing unsigned integers.

2.1.4 Expressions

The categories of expression operators that are provided by C0 are presented in Figure 2.1, where op_b is the type for binary operators, op_u for unary operators, op_l for so called "lazy" operators (prefix $log\_$ stays for logical operations compare with $bw\_$ for bitwise), which are differently evaluated (for details see Section 6.2). Currently we have only logical AND an OR operations as "lazy" operators.

Definition 2.1.7 Expressions are defined by inductive type $\mathcal{E}$. Let $vn \in nm_v$ and $cn \in nm_c$ be a variable name and structure component name respectively, $cst \in \mathcal{C}$ be a constant, $bo \in op_b$ be a binary operator, $uo \in op_u$ be a unary operator, $lo \in op_l$ be a "lazy" binary operator, and $e_1, e_2 \in \mathcal{E}$ be expressions. Then expression $e \in \mathcal{E}$ is constructed recursively in the following way:

$$\begin{aligned} e \; = \; & VarAcc(vn) \mid Lit(cst) \mid ArrAcc(e_1, e_2) \mid StrAcc(e_1, cn) \mid \\ & BinOp(bo, e_1, e_2) \mid LazyBinOp(lo, e_1, e_2) \mid \\ & UnOp(uo, e_1) \mid AddrOf(e_1) \mid Deref(e_1), \end{aligned}$$

i.e. an expression can be a variable, array, or structure access; a constant; a binary, unary, or "lazy" operator; "address-of" operator or a pointer dereferencing.

Definition 2.1.8 Let $e, e' \in \mathcal{E}$ be expressions. Then predicate $sub\_expr(e, e')$ recursively checks whether e' is a subexpression of e.

It is clear that expressions can be presented as trees with variable accesses and constants as leaves. Since types of leaf nodes can be simply defined, we can reconstruct the type of any expression recursively.

Definition 2.1.9 Let $tenv$ be a type environment, gst, lst be global and local symbol tables. The type of expression e is computed by the function

$$type_{\mathcal{E}}(tenv, gst, lst, e) \in \mathcal{T} \cup \epsilon.$$

We need to provide this function with symbol tables to be able to determine types of variable accesses by the variable name and with the type environment to be able to find the type of a pointer dereferencing expression, since pointer types are only provided with type names. If the accessed variable name is not present in the given symbol tables or the type environment does not include the name referred by a pointer type, or expression is just ill-typed, the type of the expression cannot be defined and the function returns an undefined value ϵ.

The given expression representation allows to construct not well-typed expressions and expressions, which are not part of the C0 language. Therefore we need additional constraints to fix the legal syntax. Validity of expressions is obviously defined with respect to an environment, which includes a type environment and symbol tables. Predicate $valid_{\mathcal{E}}(tenv, gst, lst, e)$ states that expression e is valid with respect to the type environment $tenv$, and global and local symbol tables gst, lst. The whole definition of expression validity can be found in [1]. Let us give some examples what the criterion of correct expression are.

Variable Access $e = VarAcc(vn)$ Variable name must be defined at least in one of symbol tables:

$$valid_{\mathcal{E}}(tenv, gst, lst, e) \equiv vn \in_* map(fst, gst) \vee vn \in_* map(fst, lst)$$

Binary operation "plus" $e = BinOp(plus, e_1, e_2)$ Subexpressions of a binary operation must be valid and have the same numerical type (the operand types which are supported by certain operators are presented in Table 1.1). Let $t_1 = type_{\mathcal{E}}(tenv, gst, lst, e_1)$, $t_2 = type_{\mathcal{E}}(tenv, gst, lst, e_2)$ be the types of the subexpressions.

$$\begin{aligned} valid_{\mathcal{E}}(tenv, gst, lst, e) \equiv{} & valid_{\mathcal{E}}(tenv, gst, lst, e_1) \wedge valid_{\mathcal{E}}(tenv, gst, lst, e_2) \wedge \\ & t_1 = t_2 \wedge t_1 \neq \epsilon \wedge t_1 \in \{IntT, UsgnT\} \end{aligned}$$

Pointer Dereferencing $e = Deref(e_1)$ The expression to be dereferenced must be valid and of a valid pointer type:

$$\begin{aligned} valid_{\mathcal{E}}(tenv, gst, lst, e) \equiv{} & valid_{\mathcal{E}}(tenv, gst, lst, e_1) \wedge \\ & \exists tn \in nm_{\mathcal{T}}.\ t_1 = Ptr(tn) \wedge tn \in_* map(fst, tenv), \\ & \text{where } t_1 = type_{\mathcal{E}}(tenv, gst, lst, e_1) \end{aligned}$$

2.1.5 Statements

The formal syntax for C0 statements is provided by representation of provided by:

Definition 2.1.10 Statements are presented by the inductive type $\mathcal{S}$. Let $e, e_1 \in \mathcal{E}$ be expressions, $s_1, s_2 \in \mathcal{S}$ be statements, $tn \in nm_{\mathcal{T}}$ be a type name, $vn \in nm_{\mathcal{T}}$ be a variable name, $pn \in nm_{\mathcal{P}}$ be a procedure name, $el \in \mathcal{E}^*$ be an expression list, and $id \in \mathbb{N}$ be a number. Then **statement** $s \in \mathcal{S}$ is defined recursively as:

$$\begin{aligned} s \;=\;& Skip \mid Comp(s_1, s_2) \mid Ass(e, e_1, id) \mid Alloc(e, tn, id) \mid \\ & Call(vn, pn, el, id) \mid Return(e, id) \mid Ifte(e, s_1, s_2, id) \mid Loop(e, s_1, id) \end{aligned}$$

The additional argument id for statements is used to give a unique identifier for each statement of a program and allows us to distinguish between occurrences of the same statement in different points of a program. The constructor $Alloc$ defines the new-operator with parameters: tn describing the type, for which memory is allocated; and e is the pointer which points to the newly allocated portion of memory. The procedure call statement is allowed to return the resulting value only to a variable and not to any other kind of expressions. As the second parameter it takes the name of the procedure which is invoked.

Similarly to expressions we declare properties of the statement syntax by predicate $valid_{\mathcal{S}}(tenv, pt, gst, lst, s)$ with procedure environment pt (see Section 2.1.6) as additional parameter. Let us consider some cases in details:

Sequential computation $s = Comp(s_1, s_2)$ Both substatements have to be valid:

$$\begin{aligned} valid_{\mathcal{S}}(tenv, pt, gst, lst, s) \;\equiv\;& valid_{\mathcal{S}}(tenv, pt, gst, lst, s_1) \wedge \\ & valid_{\mathcal{S}}(tenv, pt, gst, lst, s_2) \end{aligned}$$

Assignment $s = Ass(e, e_1, id)$ Expressions and their types have to be valid. Let $t = type_{\mathcal{E}}(tenv, gst, lst, e)$ and $t_1 = type_{\mathcal{E}}(tenv, gst, lst, e_1)$.

$$\begin{aligned} valid_{\mathcal{S}}(tenv, pt, gst, lst, s) \;\equiv\;& valid_{\mathcal{E}}(tenv, gst, lst, e) \wedge valid_{\mathcal{E}}(tenv, gst, lst, e) \wedge \\ & t \neq \epsilon \wedge t_1 \neq \epsilon \wedge (t = t_1 \vee is\_Ptr(t) \wedge is\_NullT(t_1)) \end{aligned}$$

Other cases also include validity of subexpressions, substatements, type matching when assigning values, existence of names which are accessed, etc.

Any statement can be transformed into a list of substatements, whose execution has a non trivial effect (i.e. one excludes $Skip$ and $Comp$).

Definition 2.1.11 Let $s \in \mathcal{S}$ be a statement. Then function

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

produces the list of its non-trivial substatements.

We define the predicate $\mathit{distinct}_{\mathcal{S}}(s) \in \mathbb{B}$ to state that all non-trivial substatements of s are different (based on the parameter id of the constructors).

Analogous to expressions $\mathit{sub\_stmt}(s, s') \in \mathbb{B}$ tests whether s' is a subexpression of s.

2.1.6 Procedures

The procedure table includes all information about procedures of a program.

Definition 2.1.12 We define a **procedure declaration** $f \in \mathcal{P}$ as the following tuple:

$$f = (\mathit{body}, \mathit{par}, \mathit{loc}, \mathit{rt})$$

where components are

- $f.\mathit{body} \in \mathcal{S}$ - the procedure body presented as a statement
- $f.\mathit{par} \in (nm_v \times \mathcal{T})^*$ - list of procedure parameters
- $f.\mathit{loc} \in (nm_v \times \mathcal{T})^*$ - local symbol table (including parameters)
- $f.\mathit{rt} \in \mathcal{T}$ - type of the procedure result

List of pairs of procedure names and their declarations form a **procedure environment** $pt \in (nm_{\mathcal{P}} \times \mathcal{P})^*$.

The validity predicate of some procedure declaration f with respect to type and procedure environments, global symbol table includes:

$$\begin{aligned}
&\mathit{valid}_{\mathcal{P}}(\mathit{tenv}, pt, \mathit{gst}, f) \equiv \\
&\mathit{valid}_{\mathcal{S}}(\mathit{tenv}, pt, \mathit{gst}, f.\mathit{loc}, f.\mathit{body}) \wedge \\
&\mathit{valid}_{ST}(\mathit{tenv}, f.\mathit{loc}) \wedge \mathit{valid}_{ST}(\mathit{tenv}, f.\mathit{par}) \wedge \\
&\exists e, id.\ \mathit{last}(\mathit{s2l}(f.\mathit{body})) = \mathit{Return}(e, id) \wedge f.\mathit{rt} = \mathit{type}_{\mathcal{E}}(\mathit{tenv}, \mathit{gst}, \mathit{lst}, e)
\end{aligned}$$

Definition 2.1.13 Let pt be a procedure environment. We say that pt is valid with respect to the type environment tenv and the symbol table gst if the following predicate holds.

$$\begin{aligned}
&\mathit{valid}_{PT}(\mathit{tenv}, \mathit{gst}, pt) \equiv \\
&(\forall f \in_* pt.\ \mathit{valid}_{\mathcal{P}}(\mathit{tenv}, pt, \mathit{gst}, \mathit{snd}(f))) \wedge \mathit{unique}(pt) \wedge \mathit{distinct}_{PT}(pt)
\end{aligned}$$

The predicate $\mathit{distinct}_{PT}(pt)$ states that all statements are distinct within the procedure table pt, so the following holds:

$$\mathit{distinct}_{PT}(pt) \longrightarrow \forall f \in_* pt.\ \mathit{distinct}_{\mathcal{S}}(\mathit{snd}(f).\mathit{body})$$

2.2 Abstract C0 Machine Configuration

Let us very shortly present C0 machines (describing a set of possible states of some program) and the semantics of a program execution. The sections hereafter are written with respect to semantics version formulated using PVS, so it can differ from the current theories in Isabelle/HOL. The configuration $c \in Conf$ describes a state of a C0 machine and includes the following components:

- $c.pr \in \mathcal{S}$ - statement to be executed (called the program rest)
- the environment, describing the C0 machine state, including the following components
  - $c.tt \in (nm_{\mathcal{T}} \times T)^*$ - the type environment
  - $c.pt \in (nm_p \times \mathcal{P})^*$ - the procedure environment
  - $c.m$ - memory configuration (presented below)

Components $c.tt$ and $c.pt$ of a configuration are statical whereas $c.pr$ and $c.m$ are changed during the execution of the program rest.

2.2.1 Memory

The memory configuration construction is nested. The content of a memory is based on the type $Cell$ that models a memory cell, that can contain one object of an elementary type or of a pointer type. To store (read) an object to (from) a cell, we have for every type t, which can be stored in one cell, conversion functions $t2mem \in t \to Cell$ and $mem2t \in Cell \to t$. These functions are such that for any $a \in t$ we have $mem2t(t2mem(a)) = a$.

Definition 2.2.1 Let $i, base \in \mathbb{N}$ be numbers. An **address** $a = (mem, base) \in Addr$ includes memory identifier $a.mem$ and base address in the mentioned memory $a.base$. The **memory identifier** is either gm (for global memory) , hm (for heap) memory, or a pair (lm, i), which denotes the i-th local memory in the stack of local memories.

Definition 2.2.2 Let $a \in Addr$ be an address. A pointer value is defined as

$$p = NP \mid a,$$

where p is either the null pointer (NP) or a non-empty pointer to address a in the memory.

A memory $m = (cnt, st, size) \in Mem$ includes i) its content $m.cnt \in \mathbb{N} \to Cell$, which is a mapping from addresses to the corresponding cells; ii) symbol table $m.st \in (nm_v \times \mathcal{T})^*$, which is a declaration of the variables that are kept in this memory; iii) the number of the adjacent occupied memory cells (starting from $cnt(0)$) $m.size$.

A memory configuration $c.m$ is a tuple (gm, lm, hm) with components:

- $c.m.gm \in Mem$ - global memory

- $c.m.lm \in Mem^*$ - stack of local memories
- $c.m.hm \in Mem$ - program heap

Definition 2.2.3 Let c be a configuration. Then predicate $valid_C(c) \in \mathbb{B}$ checks for the semantical and syntactical consistence of components of c.

The validity of a configuration includes the validity of the type table, procedure table, and symbol tables in all memory components. Moreover, the predicate includes the memory correctness statement, which provides the non-overlapping location of variables; consistence of the recursion depth, local stack length, and the number of returns in the program rest, etc (for details see [1]).

2.3 Program execution

2.3.1 Expression Evaluation

Expression evaluation as a part of program execution is modelled recursively by function $eval \in (Conf \times \mathcal{E}) \to (Data \cup \epsilon)$, where ϵ denotes a run-time error during the evaluation. The evaluation result is presented by type $Data$, such that $d \in Data$ (which we call a data slice) is the following tuple $d = (ad, type, intm, data)$, where the components are:

- $d.ad \in Addr$ - the address of the evaluation result if it is located in the memory
- $d.type \in \mathcal{T}$ - type of the returned data
- $d.intm \in \mathbb{B}$ - a flag, stating either the data are in the memory (0) or an intermediate result (1)
- $d.data \in \mathbb{N} \to Cell$ - a block of memory cells keeping the result of the evaluation (at addresses from 0 to $tsize(d.type) - 1$)

Definition 2.3.1 Let $x, y \in Data$ be data slices. Then predicate

$$\begin{aligned} non\_overlap(x, y) \quad = \quad & x.ad.mem \neq y.ad.mem \vee \\ & x.ad.base + tsize(x.type) \leq y.ad.base \vee \\ & y.ad.base + tsize(y.type) \leq x.ad.base \end{aligned}$$

states that x and y do not overlap in the memory.

For the evaluation of expressions containing variable accesses we use a help function which binds the program variable names to particular addresses in the memory. We do not present the function definition (given in [1]), since we are only interested in one of the properties connected with it. Namely, that two different variables are stored in distinct memory regions.

Lemma 2.3.2 Two different variables, whose names are declared in some symbol table of a valid configuration c(stated by the predicate *declared*), have non-overlapping memory regions.

$$\begin{aligned}&valid_C(c) \wedge declared(c, v) \wedge declared(c, w) \wedge v \neq w \Longrightarrow \\ &non\_overlap(eval(c, VarAcc(v)), eval(c, VarAcc(w)))\end{aligned}$$

Definition 2.3.3 Let c be a configuration, d be a data slice. The following predicate states that a data slice is consistent with the memory region defined by the address and type fixed in d.

In case $d.ad.mem = gm$ we have

$$\begin{aligned}mem\_ds(c, d) \equiv \neg d.intm \wedge d.ad.base + tsize(d.type) < c.m.gm.size \wedge \\ \forall 0 \leq i < tsize(d.type).\ d.data(i) = c.m.gm.cnt(d.ad.base + i)\end{aligned}$$

The remained cases (for local and heap memory) are defined analogously.

Definition 2.3.4 Let c be a configuration and d be a data slice. We define function $deref\_ds(c, d) \in (Data \cup \epsilon)$ which dereferences data slice d. Let address a is equal to $mem2ptr(d.data(0))$. Then data slice $d' = deref\_ds(c, d)$ has the following properties (for the case $a.mem = gm$):

$$\begin{aligned}&is\_Ptr(d.type) \wedge d.ad.base + tsize(d.type) < c.m.gm.size \wedge a \neq NP \Longrightarrow \\ &d' \neq \epsilon \wedge d'.ad = a \wedge \neg d'.intm \wedge \\ &\forall\ 0 \leq i < tsize(d.type).\ d'.data(i) = c.m.gm.cnt(a.base + i)\end{aligned}$$

The cases for $a.mem \neq gm$ are defined analogously.

As the consequence of such properties we have that dereferencing result always consistent with the memory.

Lemma 2.3.5 $deref\_ds(c, d) \neq \epsilon \Longrightarrow mem\_ds(c, deref\_ds(c, d))$

2.3.2 Memory Update

Definition 2.3.6 Let $cnt \in \mathbb{N} \rightarrow Cell$ be a memory content, $d \in Data$ be a data slice, and a be a number. Then function *copy_mem* updates the memory content starting with address a by copying the data from d.

$$copy\_mem(cnt, d, a)_i = \begin{cases} cnt_i & \text{if } i < a \vee a + tsize(d.type) \leq i \\ d.data(i - a) & \text{otherwise} \end{cases}$$

Function $mem\_update(c, addr, d) \in Conf$ is based on the *copy_mem* function and updates the memory of configuration c at the address $addr \in Addr$ by data from data slice d.

The properties of *mem_update* function we are interested in:

Lemma 2.3.7 Let the evaluation of an expression e after a memory update at the address $eval(c,e).ad$ by data slice d produce some data slice. Then the content of the evaluated data slice equals to $d.data$.

$$\begin{aligned}&eval(c,e) \neq \epsilon \wedge tsize(eval(c,e).type) = tsize(d.type) \Longrightarrow\\ &eval(mem\_update(c, eval(c,e).ad, d), e).data = d.data\end{aligned}$$

Lemma 2.3.8 Update of one from two non-overlapping memory regions does not change the other.

$$\begin{gathered}c' = mem\_update(c, d.ad, d') \wedge mem\_ds(c,d) \Longrightarrow\\ \forall x \in Data.\ mem\_ds(c,x) \wedge non\_overlap(d,x) \longrightarrow mem\_ds(c',x)\end{gathered}$$

2.3.3 Next State Computation

Program execution is modelled by function $\delta \in (Conf \cup \epsilon) \rightarrow (Conf \cup \epsilon)$, where ϵ is used to model an error state of computations. The next configuration for some configuration c is defined recursively with respect to the program rest to be executed.

We consider in details only the statements we will use for the verification example in the next chapter.

- For $c.pr = \epsilon$ next computation $\delta(c)$ always returns ϵ, i.e. once the error state is reached, it is not possible to leave it.

- $c.pr = Skip$
  An empty statement has no effect on the configuration: $\delta(c) = c$

- $c.pr = Comp(s_1, s_2)$
  If $s_1 = Skip$ then $\delta(c) = c[pr := s_2]$ else we recombine the sequence of statements to be executed, namely $\delta(c) = c'[pr := Comp(c'.pr, s_2)]$, where $c' = \delta(c[pr := s_1])$

- $c.pr = Ass(e, e_1, id)$
  Let the evaluation of both expressions e and e_1 be successful, i.e. $d_1 = eval(c,e) \neq \epsilon$ and $d_2 = eval(c, e_1) \neq \epsilon$. Performing the assignment operation we update the memory at the address $d_1.ad$ by data $d_2.data$ calling the function $mem\_update(c, d_1.ad, d_2)$.

- $c.pr = Alloc(e, tn, id)$
  Let $d = eval(c,e)$ be the expression evaluation result, $t = map\_of(c.tt, tn)$ be the allocated type. If $is\_Ptr(d.type)$ holds and $t \neq \epsilon$, i.e. we have a valid allocation statement, then the $c.pr$ will not cause a run-time error. The configuration is updated as follows. If $c.m.hm.size + tsize(map\_of(c.tt, tn)) \geq 2^{32}$ we assign a null pointer to the expression e. In the other case to update the memory region of e we need to construct some data slice rd, where the sufficient fields are filled as defined below:

  - $rd.intm = 1$ - it is an intermediate value

  - $rd.type = d.type$ - the value is of the same type as the variable where we store it
  - the data we need to store equals $nv = ptr2mem((hm, c.m.hm.size))$. Thus, the newly allocated address is taken after the last occupied address on the heap, which is $c.m.hm.size - 1$. The data component is the following mapping:

$$rd.data = \lambda i.\ \text{if } i = 0 \text{ then } nv \text{ else } \alpha,$$

    recall that α denotes an arbitrary value.

  The heap memory is obviously needed to be extended by the number of cells corresponding to the allocated type, i.e. the size of new heap hm' is equal to

$$hm'.size = hm.size + tsize(map\_of(c.tt, tn)).$$

  Finally, we update the configuration with the modified heap memory $c' = c[m := c.m[hm := hm']]$ with respect to the constructed data slice, namely $c'' = mem\_update(c', d.ad, rd)$. The latter update writes the newly allocated address at the address in the memory corresponding to expression e.

- $c.pr = Loop(e, s_1, id)$
  To avoid a run-time error here the evaluation of e has to return some data slice $d = eval(c, e)$ and moreover, the result needs to be of the boolean type: $is\_Bool(d.type)$. As the next step we analyze the value stored in d, namely: $b = mem2bool(d.data(0))$. The statement under consideration is a control statement, i.e. it changes only the program rest:

$$c'.pr = \begin{cases} Comp(s_1, Loop(e, s_1, id)) & \text{if } b \\ Skip & \text{otherwise} \end{cases}$$

  The memory part of the configuration stays unchanged: $c'.m = c.m$.

Definition 2.3.9 We define function $trace \in (Conf \cup \epsilon \times \mathbb{N}) \rightarrow Conf \cup \epsilon$ such that $trace(c, n)$ recursively applies the δ function n times to a configuration c.

Chapter 3

Program Verification Using Small Step C0 Semantics

Our first goal is to estimate whether it is feasible to carry out proofs directly in the frame of the C0 small-step semantics. This kind of test is absolutely necessary if we intend to use the semantics for proving programs that cannot be verified using other means (e.g. based on big-step semantics). The example of such programs are non-terminating and concurrent programs. Moreover, a verified non-trivial program implicitly confirms that the developed model describes the expected functionality.

In this chapter we present the verification of a small program which constructs an empty linked list of a given length on the program heap.

3.1 The Example Program

Figure 3.1 presents the source code (written in the C0 language) of the example program. Let k be a constant, which denotes the given length of a list we need to construct. Variable `head` is the pointer to the head of the constructed list; `elt` is a temporary variable used to allocate and insert a new element to the list; `n` keeps the current length of the list.

```
struct list{
struct list *next;
int cont };
    ...

head = NULL; n =0;
while ( n < k ){
elt = new(struct list);
elt->next = head ;
head = elt; n = n + 1};
```

Figure 3.1: The source code of the example program

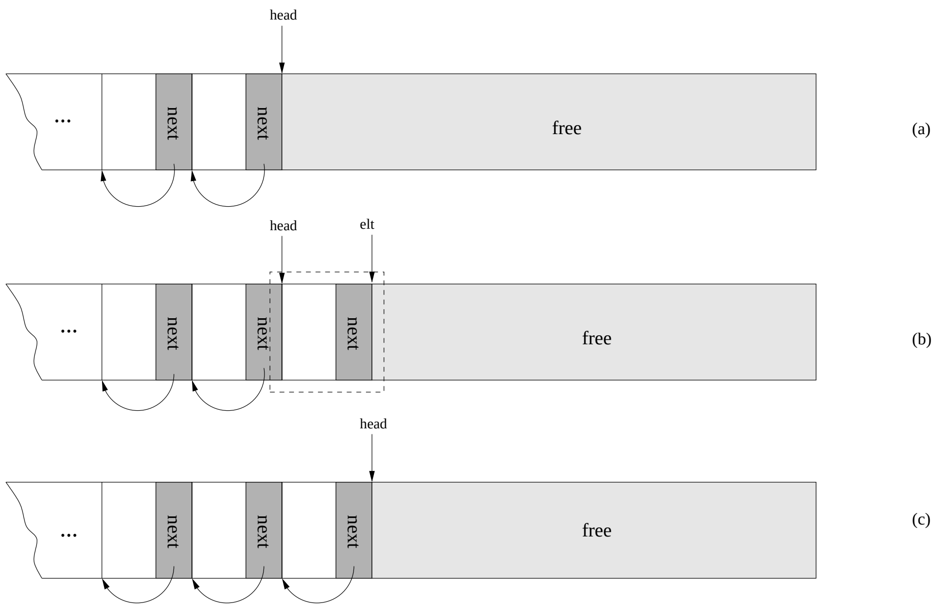


Figure 3.2: Stages of a loop iteration: (a) before entering the loop; (b) after the new statement; (c) at the end of the loop

Figure 3.2 illustrates changes of the program memory state and the variables during one loop iteration.

From the source code we construct the program dependent configuration components, namely *c.pr*, *c.tt*, and *c.pt*. The procedure table includes only one given procedure, so is of no interest. The type table and program rest are depicted in Figure 3.3. So, the type table includes three types: the integer, a structure of two components realizing linked list, and the pointer type to this structure. We extract the body of the loop (*body*) to be able to refer to it as to a separate statement.

3.2 Verification

The verification process consists of two parts:

1. we verify the loop body as a sequential program. Stating its correctness we abstract the program that will be executed afterwards

2. we use the lemma about one loop iteration in the induction proof about the while-loop to show the correctness of the whole procedure

Obviously, the correct execution of the program depends on particular properties of the configuration environment. So, to prove the correctness theorem for the given example we need to rely on the assumption, that we have enough memory to allocate k list elements.

To specify the result of the program execution in some state (and hence, to formulate the correctness theorem about the program), we need to state the presence of a linked list in the configuration memory.

$$\begin{array}{rcl}
c.tt & = & [(int\_t, IntT), \\
 & & (list, str), \\
 & & (list\_ptr, Ptr(list))] \\
str & = & Str([(next, Ptr(list)), (data, IntT)]) \\
 & & \\
prg & = & Comp(Ass(VarAcc(head), Lit(Nil), 0), \\
 & & Comp(Ass(VarAcc(n), Lit(Int(0)), 1), \\
 & & Loop(BinOp(greater, VarAcc(n), Lit(Int(k))), body, 2)) \\
 & & \\
body & = & Comp(Alloc(elt, list\_ptr, 3), \\
 & & Comp(Ass(StrAcc(Deref(VarAcc(elt)), next), VarrAcc(head), 4), \\
 & & Comp(Ass(VarAcc(head), VarAcc(elt), 5), \\
 & & Ass(VarAcc(n), BinOp(plus, VarAcc(n), Lit(Int(1))), 6)))) \\
 & & \\
c.pr & = & prg
\end{array}$$

Figure 3.3: Formal representation of the example program in C0 semantics

3.2.1 Linked Lists

First, we present the general concept of linked lists on pointers. Second, we adapt it to the memory model of the semantics we have presented in the previous chapter.

Definition 3.2.1 Let *ptr* be an abstract type for pointers and constant $Null \in ptr$ be a null pointer. Let $f \in ptr$ be a pointer, $next \in ptr \to ptr$ be a function returning for some pointer the following pointer in the list, and $n \in \mathbb{N}$ be a number. We state that pointer f points to a **linked list** of length n if the following predicate holds :

$$list(f, next, n) = \begin{cases} f = Null & \text{if } n = 0 \\ \exists \pi \in \mathbb{N} \to ptr.\ \pi(0) = f \wedge & \\ (\forall 0 \le i \le n-1.\ \pi(i+1) = next(\pi(i))) \wedge & \text{otherwise} \\ next(\pi(n-1)) = Null & \end{cases}$$

Now, this general concept needs to be adapted to the considered memory model. The role of the *ptr* type in the C0 semantics is played by data slices, which have their *type* field of any pointer type. To specify the pointer data slices and, in particular, data slices which contain null pointers, we introduce the following predicates.

Definition 3.2.2 Let $d \in Data$ be a data slice. A predicate $ptr\_ds \in Data \to \mathbb{B}$ checks, whether a data slice contains pointer data, and $null\_ptr\_ds \in Data \to \mathbb{B}$ tests for a null pointer.

$$ptr\_ds(d) \equiv is\_Ptr(d.type)$$

$$null\_ptr\_ds(d) \equiv ptr\_ds(d) \wedge (mem2ptr(d.data(0)) = NP)$$

A data slice which satisfies predicate ptr_ds we call a *pointer data slice.* To model the computation of the next pointer in the list we use a function $next \in Data \to Data$. It extracts the next pointer data from a *pointed* data slice and obviously depends on the type of elements that build up the list.

A linked list in the memory is defined as follows.

Definition 3.2.3 Let $c \in Conf$ be a configuration, $f \in Data$ be a pointer data slice, $next \in Data \to Data$, $n \in \mathbb{N}$. The predicate $c0\_list(c, f, next, n)$ states the presence of a linked list of length n pointed to by f in the program memory.

$$c0\_list(c, f, next, n) = \begin{cases} null\_ptr\_ds(f) & \text{if } n = 0 \\ \exists \pi \in \mathbb{N} \to Data.\ list\_func(c, f, next, n, \pi) & \text{if } n > 0 \end{cases}$$

The $list\_func$ predicate defines π as a permutation on a linked list:

$$\begin{aligned} list\_func(c, f, next, n, \pi) \quad \equiv \quad & \pi(0) = f \wedge (\forall i < n.\ deref\_ds(c, \pi(i)) \neq \epsilon) \wedge \\ & (\forall\ 0 \leq i \leq n-1.\ next_i = \pi(i+1)) \wedge \\ & null\_ptr\_ds(next_{n-1}), \\ & \text{where } next_i = next(deref\_ds(c, \pi(i))) \end{aligned}$$

Thus, we use the idea presented above and introduce a linked list as a permutation π. Each element $\pi(i)$ is a pointer, which points to the next element and the last member is a null pointer. Since function $next$ works on a pointer data slice, then in order to get the next element for some element $\pi(i)$ we first need to dereference it.

Definition 3.2.4 Function

$$\begin{aligned} & deref\_nth(c, f, next, i) = \\ & \begin{cases} f & \text{if } i = 0 \\ next(deref\_ds(c, deref\_nth(c, f, next, i-1))) & \text{otherwise} \end{cases} \end{aligned}$$

recursively computes the i-th element of the list.

Clearly, if we have any list permutation π, then it is equivalent to $deref\_nth$ function.

Lemma 3.2.5

$$list\_func(c, f, next, n, \pi) \Longrightarrow \forall i < n.\ \pi(i) = deref\_nth(c, f, next, i)$$

The proof is obvious induction on i.

3.2.2 Loop Iteration Theorem

We divide the theorem about one loop iteration into two parts: i) we argue about changed and (that is even more important) unchanged components of the configuration; ii) based on (i) we show that these changes mean the insertion of a new list component to the existing list on the heap.

Let us introduce some notations that we will use to formulate the theorem. For any configuration c_i we use $c_{i+1} = \delta(c_i)$. Shortcut mm_i denotes memory component mm in the configuration c_i, so formally: $c_i.m.mm$. For any variable name v we denote by v_i the corresponding data slice in the configuration c_i, i.e. $eval(c_i, VarAcc(v))$.

The necessary initial conditions to show the correct execution of one iteration of the loop are: i) static validity of the configuration (that implies the correct expression evaluation and the execution of statements without run-time errors); ii) the evaluation of the loop condition to *true*; iii) enough memory on the heap to place one more list element.

Theorem 3.2.6 (**Configuration after Loop Iteration**)

Let c_0 be the configuration before the loop execution, $et = map\_of(c_0.tt, list)$ be the type of the allocated memory chunk for list elements (see Figure 3.3 for the type environment), $cond = BinOp(greater, VarAcc(n), Lit(Int(k)))$ be a loop condition, $prg = Comp(Loop(cond, body, 2), p)$ be a program rest.

We state that after execution of one iteration of the while loop (which requires one more step, than the execution of the loop body):

- the configuration is in a valid state
- the occupied memory on the heap increases by the size of the allocated type
- variable *head* points to a newly allocated element, which starts at the address, where the heap before the body execution ended
- the memory cell, where *head* points, keeps at the place of the next pointer the value of *head* before the execution
- value of variable n is increased by one
- data slices outside the memory regions occupied by variables $head, elt, n$ and the new element on the heap are not changed
- the program rest to be executed next is the same as the program rest before the iteration.

Formally:

$$
\begin{array}{ll}
(P) & valid_C(c_0) \wedge c_0.pr = prg \wedge is\_valid\_nat(hm_0.size + tsize(et)) \wedge \\
& mem2bool(eval(c, cond).data(0)) = true \\
& \Longrightarrow \\
& \exists l.\ trace(c_0[pr := body)], l-1).pr = [] \wedge c_l \neq \epsilon \wedge valid_C(c_l) \wedge \\
& hm_l.size = hm_0.size + tsize(et) \wedge \\
& mem2ptr(head_l.data(0)) = (hm, hm_0.size) \wedge \\
& hm_l.cnt(hm_0.size) = head_0.data(0) \wedge \\
& mem2nat(n_l.data(0)) = mem2nat(n_0.data(0)) + 1 \wedge \\
& (\forall x \in Data. \\
& \quad mem\_ds(c_0, x) \wedge non\_overlap(head_l, x) \wedge \\
& \quad non\_overlap(elt_l, x) \wedge non\_overlap(deref\_ds(c_l, head_l), x) \wedge \\
& \quad non\_overlap(x, n_l) \\
& \quad \longrightarrow mem\_ds(c_l, x)) \wedge \\
& c_l.pr = c_0.pr
\end{array}
$$

Proof: To prove the functional part of the goal (changes in the configuration) we just consecutively unfold the definitions of the step function δ and the expression evaluation function *eval*. The absence of the run-time errors and hence, the validity of the final configuration, are implied by the construction of the statical configuration components and the validity of the initial configuration.

Executing the *eval* function we show for any step i that type properties of the evaluated expressions (i.e. tests like $is\_Bool(eval(c_i, cond).type)$, $is\_Ptr(head_i)$, etc.) indeed hold. To show such facts we use the invariant properties of the statical configuration components during the execution, such as $\forall i.\ c_i.tt = c_0.tt$. For example, to show the correct execution of the *Alloc* statement we argue, that since *list* is a name of the allocated type included in *c.tt*, then the corresponding type $map\_of(c_0.tt, list) \neq \epsilon$ and it is obviously equal to $elt_0.type$ (what follows from the validity of the statement we execute).

The changes of the variables content after assignments and memory allocation follow with definitions of *mem_update*, Lemma 2.3.7. The stability of the memory content outside of the variables that were changed follows with Lemmas 2.3.2, 2.3.8. For the data slices located in the heap memory outside the newly allocated region it is true based on the following observations. They are located at the addresses less than $hm_l.size - tsize(et) = hm_0.size$; that are exactly the only addresses that can satisfy predicate $mem\_ds(c_0, x)$ for the case $x.ad.mem = hm$.

3.2.3 Abstract List Theorem

The second part of the theorem states that the abstract list has grown.

Let us concretize the *next* function which is used in the abstract *list* predicate. This obviously depends on the type of list elements. For any given data slice (that has the type of a list element) it needs to return a pointer data slice constructed as follows:

$$next \in Data \to Data$$

- $ds.type = str$

$$next(ds).k = \begin{cases} Ptr(list) & \text{if } k = type \\ \lambda x.\ \text{if } x = 0 \text{ then } ds.data(0) \text{ else } \alpha & \text{if } k = data \\ ds.k & \text{otherwise} \end{cases}$$

- $ds.type \neq str$

$$next(ds) = \alpha$$

Recall that by α we refer to some arbitrary value. Note, that the *data* field of the new data slice includes only the first field (i.e. next pointer) of the given data slice.

Theorem 3.2.7 Let c_0 be the configuration before the loop execution, *next* be the function we have described above. We state that if before the execution of one iteration of the loop:

- pointer *head* points to a linked list
- variable n keeps the length of the list
- each element of the list is located in the heap memory

then after one loop iteration:

- the linked list pointed by *head* contains one more element
- $(i + 1)$-th element of the updated list is the same as i-th element of the list before the execution
- each element of the updated list is located in the heap memory

$$\begin{array}{ll}
 & P \wedge num = mem2nat(n_0) \wedge \\
(i1) & c0\_list(c_0, head_0, next, num) \wedge \\
(i2) & \forall k < num.\ deref\_ds(c_0, deref\_nth(c_0, head_0), next, k).ad.mem = hm \\
 & \Longrightarrow \\
 & \exists l.\ trace(c_0[pr := body], l - 1).pr = [] \wedge \\
(list) & c0\_list(c_l, head_l, next, num + 1) \wedge \\
(tail) & (\forall 1 \leq k < num.\ deref\_nth(c_l, head_l, next, k + 1) = \\
 & \qquad deref\_nth(c_0, head_0, next, k)) \wedge \\
(hm) & \forall k < num + 1.\ deref\_ds(c_l, deref\_nth(c_l, head_l), next, n).ad.mem = hm
\end{array}$$

By assumption P of the theorem we refer to the assumptions of Theorem 3.2.6.

Proof: Based on Theorem 3.2.6 we conclude that we can finish one loop iteration with l steps. Moreover, dereferencing of the new value of the *head* variable is valid (i.e. its result is not equal to ϵ) and produces the following data slice:

$$deref\_ds(c_l, head_l).k = \begin{cases} (hm, hm_0.size) & \text{if } k = ad \\ str & \text{if } k = type \\ head_0.data & \text{if } k = data \\ false & \text{if } k = intm \end{cases} \qquad (*)$$

Let us consider two cases:

1. $num = 0$

From preconditions P, $(i1)$ and *c0_list* definition we conclude that predicate $null\_ptr\_ds(head_0)$ holds. To prove claim $(list)$ we need to show the existence of π satisfying predicate $list\_func(c_l, head_l, next, 1, \pi)$ (see Definition 3.2.3). We construct permutation π such that

$$\pi(0) = \begin{cases} head_l & \text{if } i = 0 \\ \alpha & \text{otherwise} \end{cases},$$

which clearly turns *list_func* to true as: i) $\pi(0) = head_l$ (by construction), ii) $ptr\_ds(head_l)$ holds, iii) $next(deref\_ds(c_l, head_l))$ is the null pointer data slice (by construction of *next*, (*) and $null\_ptr\_ds(head_0)$).

Since the new list includes only one element, we do not need to show subgoal $(tail)$ of the theorem.

As $deref\_ds(c_l, deref\_nth(c_l, head_l, next, n)) = deref\_ds(c_l, head_l)$ (by def. of *deref_nth*), the claim (hm) of the theorem is true as well.

2. $num > 0$

From $(i1)$ and the definition of *c0_list* we conclude that there exists some π' such that $list\_func(c_0, head_0, next, num, \pi')$ holds. To show the correctness of the list predicate we construct a permutation π such that

$$\pi(i) = \begin{cases} head_l & \text{if } i = 0 \\ next(deref\_ds(c_l, head_l)) & \text{if } i = 1 \\ \pi'(i-1) & \text{if } 2 \le i \le num \end{cases},$$

which has to satisfy predicate $list\_func(c_l, head_l, next, num + 1, \pi)$ included in *c0_list*.

To show the validity of the claim we need the following observation.

$$1 \le i \le num.\ deref\_ds(c_l, \pi(i)) = deref\_ds(c_0, \pi'(i-1)) \qquad (**)$$

1. $i = 1$

By construction of π we have

$$deref\_ds(c_l, \pi(1)) = deref\_ds(c_l, next(deref\_ds(c_l, head_l))).$$

Using definition of $next$ and (*) we can show the equality

$$next(deref\_ds(c_l, head_l)).data(0) = head_0.data(0).$$

Considering the fact that $\pi'(0) = head_0$ (from definition of the $c0\_list$ predicate), we only need to show that dereferencing of pointer data slices with equal data components is the same in both configurations c_0 and c_l. From assumption $(i2)$ the equality $deref\_ds(c_0, \pi'(0)).ad.mem = hm$ follows. Thus, the data of $deref\_ds(c_0, \pi'(0))$ are placed outside the changed variables and the new element. Since the $mem\_ds(c_0, deref\_ds(c_0, \pi'(0)))$ predicate holds (by definition of $list\_func$ and Lemma 2.3.5)), then following Theorem 3.2.6 the content of that data slice is unchanged in configuration c_l, i.e. $deref\_ds(c_0, \pi'(0)) = deref\_ds(c_l, \pi'(0))$. From definition of $deref\_ds$ obviously follows

$$d.data(0) = d'.data(0) \Longrightarrow deref\_ds(c, d) = deref\_ds(c, d').$$

Using this for $d = deref\_ds(c_l, \pi'(0))$ and $d' = deref\_ds(c_l, \pi(1))$ we finish the claim.

2. $i > 0$
   By Lemma 3.2.5 and $(i2)$ the equality $\pi'(i-1).ad.mem = hm$ follows. Using arguments analogous to the previous case we can show that $deref\_ds(c_0, \pi'(i-1)) = deref\_ds(c_l, \pi'(i-1))$ and by construction of π it equals to the data slice $deref\_ds(c_l, \pi(i))$.

The statement (**) is the main helping mean to prove the theorem. Let us consider the proof of $(list)$ part of the claim in more details.

We need to show that $list\_func(c_l, head_l, next, num + 1, \pi)$ holds. The first conjunct of this predicate, i.e. $\pi(0) = head_l$, is obviously true. The second part $\forall i < num + 1. deref\_ds(c_l, \pi(i)) \neq \epsilon$ follows with (*) for $i = 0$ and with $(**), (i1)$, and the definition of the $c0\_list$ predicate for all others.

The part of the $list\_func$ definition that provides the chaining of the list:

$$\forall\ 0 \leq i \leq n-1.\ next_i = \pi(i+1)$$

is also easily shown. Case $i = 0$ follows by construction of π, for others we have

$$\begin{array}{lcl}
next_i & = & \text{(notation of } next_i\text{)} \\
next(deref\_ds(c_l, \pi(i))) & = & (**) \\
next(deref\_ds(c_0, \pi'(i-1))) & = & \text{(i1), def. } c0\_list \\
\pi'(i) & = & \text{(construction of } \pi\text{)} \\
\pi(i+1) & &
\end{array}$$

The proof of the rest follows similar arguments □.

3.3 Main Theorem

We formulate the correctness theorem about the program as follows:

Theorem 3.3.1 Let c_0 be the configuration before the program execution.

$$valid_C(c_0) \wedge is\_valid\_nat(hm_0.size + k * tsize(et)) \wedge c_0.pr = prg \Longrightarrow$$
$$\exists l.\ trace(c_0, l).pr = [] \wedge c0\_list(c_l, next, head_l, k)$$

Proof: By induction on k.

For $k = 0$ we instantiate the existence quantifier in the claim with $l = 3$. This number follows from the observations below. To get out of the loop we need to go through two initializing assignments (for *head* and n) and one while-loop application. Expanding definitions of δ and *eval* we can show that $mem2ptr(head_1) = NP$ and $mem2nat(n_2) = 0$. Since $mem2bool(eval(c_2, cond))$ is clearly equals to $false$ and we have nothing in the program rest after the *Loop* statement, we leave the program after three steps. The presence of the list in the memory follows with $c_3.m = c_2.m$ (by *Loop* execution) and $outside(n_2, head_2)$ so that $null\_ptr\_ds(head_3)$ holds.

To prove the induction step $k \rightarrow k + 1$ we instantiate the existence quantifier with $2 + m * k + 1$, where m is a number of steps needed for one loop iteration. The first two steps correspond to initial assignments, and the last step is needed to execute the while statement the last time before leaving the procedure. The induction step is performed with the theorems proven above. $\square$

3.4 Conclusion

Despite the successful verification of the presented example in the frame of the described C0 semantics, this approach is not actually suitable for proving the correctness of large programs. One of the most time consuming steps during the proof is the routine unfolding of the step and evaluation functions. That is needed to show the changes of configuration as well as impossibility to end up in the error state.

From the mathematical point of view, it involves simple argumentation. Unfortunately, the automatic simplification and rewriting provided by Isabelle/HOL is not very helpful here, often the result of the step-function execution depends on the evaluation done not in the previous steps but in earlier ones. To use the results of such an evaluation we need to take the non-overlapping of memory regions into account and explicitly show that evaluation results are still valid in the current state. This requires, of course, quantifier instantiation and applying the number of lemmas stating the invariant properties of the configuration, and properties of the program variables allocation in the memory. That part of the proof was just shortly mentioned in Theorem 3.2.6.

As we have just mentioned, showing that the evaluation result still the same in the current state can rarely be done without using additional arguments about the memory configuration. The very detailed and involved memory model makes the proof, what parts of the memory are not changed, tedious. Obviously, that becomes more complex with the growing number of the accessed variables (including the nameless ones on the heap) during the program execution.

Another step, which is obvious for the human being but is still required to be explicitly done in the theorem prover, is the checking of the semantic consistency when evaluating an expression. Clearly, the type properties of an expression data slice are statical and can be proved once at the beginning of the verification. Using these facts as lemmas (to show the stability of the types in the different configurations) requires a large amount of purely mechanical work, since the expression evaluation is most frequent action in any program.

Thus, the deep embedding of the expressions and the memory model, where the stability of untouched memory regions after a memory update is needed to be additionally shown, makes an effort for the verification of even a simple program unnecessarily laborious.

Moreover, the considered example concerns a simple data structure. Usually, programs deal with more complex data structures. Establishing the predicates which connect program variables with corresponding abstract entities and arguing on them in the frame of this memory model can be a really hard task. In the following we show, that the verification environment based on Hoare Logic allows to verify programs with complex data structures much more effectively.

This case study has some positive outcomes as well. Despite all complications we were able to carry out this proof at all, what points to the reasonable size of the semantics' model. The use of the semantics has also several advantages. First, its realistic memory model allows to argue about memory management in detail. Second, this approach (as opposite to big-step semantics) can be used to prove non-terminating, concurrent programs. If we extend the presented semantics with support of inline assembler statements, it will be very useful in verification of low-level parts of system software.

Chapter 4

Verification of Programs Using Hoare Logics

In this section we shortly describe another way for the verification of programs written in a sequential programming language. The presented method was used for carrying out the second verification example which we consider in the thesis, namely the verification of the compiler for the C0 language.

The ideas of formal program verification proposed by Floyd, Hoare, and others resulted in approaches known as *(Floyd-)Hoare logic* [7]. The main point of those is to describe a program state by a predicate called an assertion. The logic is defined for some programming language and includes rules how every type of statements of the language modifies the assertion.

In the current work we use the Hoare logic version formalized and proven sound and complete in the Isabelle/HOL by Schirmer [3]. The verification environment created by Schirmer is developed for a general language model. This provides all common language features which are peculiar to imperative modern programming languages. In particular, the language model can be easily instantiated by C0 language program constructs. For this language Hoare logic rules for both partial and total correctness were developed. The Hoare Logic was extended by the concrete syntax, verification condition generator, applying rules to a program code automatically, and was integrated to the Isabelle/HOL as a very convenient tool to be used for the verification of applications.

Below we summarize some important issues about the verification environment.

4.1 Programming Language

To argue about the correctness of a program we first need to formalize its constructs. In the general language model proposed by Schirmer only statements are fixed, expressions can be any HOL expressions (shallow embedding). Let us first introduce the general language model (called Simpl) and later we show how to transform a C0 program to it.

4.1.1 Language Model

Let Var be a set of all variables of a program to be verified, Val be a set of values they can take, then $\Sigma = \mathit{Var} \to \mathit{Val}$ is a state space of the program (set of all possible program states), i.e. all possible assignments to the program variables. Let P be the set of procedure names appearing in the program.

Definition 4.1.1 Let $p \in P$ be a procedure name, c_1, c_2 be statements, $f, \mathit{init} \in \Sigma \to \Sigma$ and $\mathit{res} \in \Sigma \times \Sigma \to \Sigma$ be update functions over states, and $b \in \Sigma \to \mathbb{B}$ be a boolean expression over state in Σ.

Then the statements c of the language are defined recursively:

$$\begin{aligned} c &= \mathit{Skip} \mid \mathit{Basic}(f) \mid \mathit{Seq}(c_1, c_2) \mid \mathit{Cond}(b, c_1, c_2) \mid \\ &\quad \mathit{While}(b, c_1) \mid \mathit{Call}(\mathit{init}, p, \mathit{res}) \end{aligned}$$

Let us consider each of the statement variants in more detail and give their operational semantics. A judgement $\Gamma \vdash s - c \to t$ means that the execution of command c in state s leads to state t with respect to a procedure environment Γ. Since Γ maps procedure names to procedure bodies, the execution of a call statement depends on it. The operational semantics is a so called "big step" semantics, i.e. the body of a called function is considered to be executed as one statement.

The semantics is defined as a set of inference rules

$$\frac{A_0 \ldots A_n}{C},$$

where $A_0, \ldots, A_n$ (where $0 < n$) are assumptions that are necessary to deduce conclusion C. The execution of program c is defined inductively by introducing rules for each statement:

- Skip does nothing.
- $\mathit{Basic}(f)$ is used to model assignments and memory allocation by applying a state update function $f \in \Sigma \to \Sigma$. Let the program state $\sigma \in \Sigma$ include a variable $a \in \mathit{Var}$. We can model the assignment of a value v to variable a by the function $f(\sigma) = \sigma[a := v]$. The command is executed by applying function f to the current state.

$$\Gamma \vdash s - \mathit{Basic}(f) \to f(s)$$

- $\mathit{Seq}(c_1, c_2)$ is the sequential composition of statements c_1 and c_2.

$$\frac{\Gamma \vdash s - c_1 \to s' \quad \Gamma \vdash s' - c_2 \to t}{\Gamma \vdash s - \mathit{Seq}(c_1, c_2) \to t}$$

  This rule can be read forward as well as backward. Forward: if we know, that execution of program c_1 in state s leads to state s' and moreover, executing c_2 in state s' we end up in t, then we can conclude that execution of sequential composition of c_1 and c_2 starting in state s leads t. Backward, which we can use for deduction: in order to show, that execution of $\mathit{Seq}(c_1, c_2)$ in state s leads to t, we need to show that execution of c_1 leads to some intermediate state s', and c_2 being started in s' leads to t.

- $Cond(b, c_1, c_2)$ is a conditional statement. Depending on b either c_1 or c_2 is executed.

$$\frac{b(s) \qquad \Gamma \vdash s - c_1 \rightarrow t}{\Gamma \vdash s - Cond(b, c_1, c_2) \rightarrow t} \qquad \frac{\neg b(s) \qquad \Gamma \vdash s - c_2 \rightarrow t}{\Gamma \vdash s - Cond(b, c_1, c_2) \rightarrow t}$$

- $While(b, c_1)$ is a loop command, where b is the branching condition and c_1 is the function body. If the loop condition b holds in the current state, the loop body c_1 is executed, followed by the recursive execution of the loop. If the loop condition does not hold, we exit the loop in the same state.

$$\frac{b(s) \qquad \Gamma \vdash s - c \rightarrow s' \qquad \Gamma \vdash s' - While(b, c_1) \rightarrow t}{\Gamma \vdash s - While(b, c_1) \rightarrow t}$$

$$\frac{\neg b(s)}{\Gamma \vdash s - While(b, c_1) \rightarrow s}$$

- $Call(init, p, res)$ is a procedure call of the procedure with name p. Function $init$ provides parameter passing, adapting local variables to the context of the called function. Function res takes as parameters the state before the function call and the state after the function call was executed. It returns a state, where: i) local variables of the caller, which were not affected by the call, are restored to the values before the call; ii) global variables and the heap are updated by values they get during the function call; iii) the result of the function execution is passed back to the caller. The execution of the procedure call proceeds in several steps. First, the parameters are passed by applying the $init$ function to the current state. Then we execute the procedure body, which the procedure environment maps to name p. As the execution of the procedure body ends in the state s', we need to restore local variables of the caller and return the result of the function execution with help of res function.

$$\frac{\Gamma \vdash init(s) - \Gamma(p) \rightarrow s'}{\Gamma \vdash s - Call(init, p, res) \rightarrow res(s, s')}$$

Having such a model of the function call we do not need to model a function stack, since locality of variables is provided by $init$ and res functions.

4.2 State Space

A program state represents heap and memory content mapped to the variables at some point of the program execution. In this section we introduce a concrete model for a state space, which is used in the verification environment.

We model the state space as a record (named tuple)

$$\sigma = (g_0, \ldots, g_k, l_0, \ldots, l_m, h_0, \ldots, h_n),$$

which includes all the global $g_0, \ldots, g_k$ and local $l_0, \ldots, l_m$ variables appearing in the program as well as so called heap functions $h_0, \ldots, h_n$ (described in Section 4.2.1), which model the heap. Each component v of the record has the format

$$v = n : t,$$

where n is the name of variable (or a heap function) and t is its type.

It is clear that we need to model all possible C0 types with Isabelle/HOL types. Thus, any variable v of integer or char type is modelled with $v \in \mathbb{Z}$ and of unsigned integer type with $v \in \mathbb{N}$. Pointers to nameless variables on the heap are represented with a special type *Ref* considered in the next section. We refer to the types mentioned above (namely, integer, char, and unsigned int) and pointers by *simple* types. Arrays of simple types are modelled as lists, e.g. variable v of type $int[5]$ is modelled as $v \in \mathbb{Z}^*$. As a direct consequence of such a model we loose a part of the type information, which we need. In Section 8.4.3 we show how this information can still be integrated in the verification process.

Variables of a C0 program can also be of any structure type. We call structure types and array types, whose elements are not of a simple type, *complex* types. C0 variables of complex types are represented by a number of Simpl variables, where each variable is of a simple type or an array with elements of a simple type.

Let us present a function flt, that flattens a complex type to the number of simple subtypes. It returns the information about a subtype including the "path" from the original type to a particular subtype as a string. We give the definition of $flt \in \mathcal{T} \to (string \times \mathcal{T})^*$ in the terms of the C0 semantics. The function is defined recursively and returns lists of pairs (s, t). The first element s presents the selector ("path", stored as a string) of the subtype t.

$$flt(t) =$$
$$\begin{cases} ("", t) & \text{if } t \text{ is simple } \vee \\ & t = Arr(n, T) \wedge T \text{ is simple} \\ map(fln\_c, sc) & \text{if } t = Struct(sc) \\ map((\lambda(a,b).(a, Arr(n, b))), flt(T)) & \text{if } t = Arr(n, T) \wedge T \text{ is not simple,} \end{cases}$$
$$\text{where } fln\_c((x, y)) = map((\lambda(a,b).(x + "\_" + a, b)), flt(y))$$

The ""-symbol denotes an empty string, + denotes concatenation of strings. Thus, for a structure type we return a list of its flattened components using function fln_c. Note, that for a structure component we extend the selectors of the subtypes returned by flt with the name of the component. We represent an array with elements of a complex type by a number of arrays of its simple subtypes.

Thus, C0 variable v of a complex type t, such that $ftn(t) = ((s_0, t_0), \ldots, (s_n, t_n))$ is modelled by Simpl variables: $v\_s_0 : t_0, \ldots, v\_s_n : t_n$. Therefore, access to any component of v is just an access to a corresponding variable.

Let us demonstrate how variable v given below, which is an array of structures, will be represented in the state space.

```
struct s{ int a; int b;};
struct t{ struct s c; bool d;};
 ...
struct t v[2];
```

According to the scheme given above, the state space will include the following components.

$$\sigma = (\ldots, v\_c\_a : \mathbb{Z}^*, v\_c\_b : \mathbb{Z}^*, v\_d : \mathbb{B}^*, \ldots)$$

We notice, that σ is constructed for each concrete program we are going to verify, so it is finite and completely defined. The state representation for any program to be verified can be obviously generated from the program source code. In the frame of Verisoft a tool called `c0_check` was developed. One of its tasks is to perform translation of C0 programs to their Simpl representation (provided with the state space). This tool was also applied for translation of the compiler source code.

4.2.1 Modelling Pointers and Structures

Pointers and memory allocation In the program model a safe version of pointers not supporting address arithmetic is considered. Thus, pointers will not be presented as numerical addresses, but as an abstract type *Ref*, which is isomorphic to $\mathbb{N}$. In this model pointers are not typed, so all variables of a pointer type will be modelled as variables of type *Ref*. The null pointer is the constant $Null \in Ref$. Dynamic memory allocation is modelled by providing a new pointer and is realized as the function

$$new : 2^{Ref} \rightarrow Ref$$

$$new(A) = \varepsilon x.x \notin \{Null\} \cup A$$

The function takes a set of pointers and by means of the Hilbert's choice operator returns a pointer, which is neither the null pointer nor in the given set. So, providing this function with the set of already allocated pointers, we obtain a new one. For finite sets we have two important lemmas about pointer allocation:

Lemma 4.2.1 (new_neq_Null) A is a finite set $\Longrightarrow new(A) \neq Null$

Lemma 4.2.2 (new_not_in_alloc) A is a finite set $\Longrightarrow new(A) \notin A$

Heap To be able to manipulate pointers to structure objects the verification environment incorporates a model of program heap.

The natural way of presenting the heap is a function which maps addresses (in our case - objects of type *Ref*) to objects that are located on the heap. This approach has a drawback that we additionally need to show that updating one of the fields of some structure object does not affect the others.

The heap model, that is actually used in the verification environment, is the adapted split heap approach that goes back to Burstall [52] and was successfully applied by Bornat [53], Metha and Nipkow [24]. The main advantage of this heap model is that it excludes by construction overlapping between different fields of structures.

For every type t, which appears in the program, such that

$$flt(t) = ((s_0, t_0), \ldots, (s_n, t_n))$$

we introduce heap functions $s_i : Ref \rightarrow t_i$ for all $i \leq n$.

Summarizing the approach presented above, each field of a structure (flattened up to elementary types) gets a separate heap in the state space and an access to

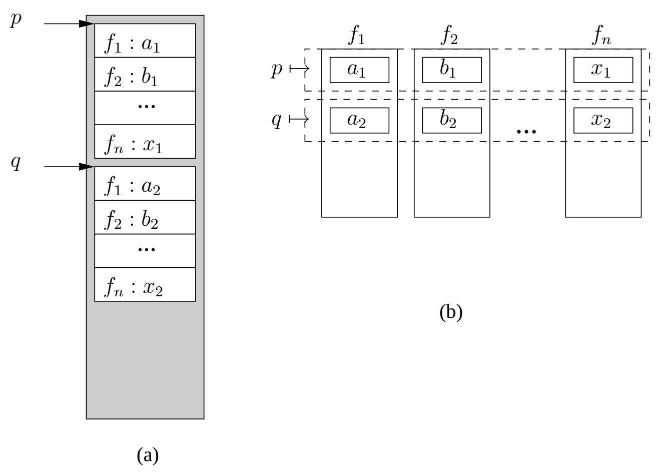


Figure 4.1: Split heap approach: (a) the heap with record objects; (b) the corresponding split heap model

one structure field does not touch values of the others. A pointer access to a field is modelled as a function application. Let p be a pointer to a structure type including field with hame f, then the value stored in the field f via pointer p is equal to $f(p)$.

Figure 4.1 illustrates this idea: (a) shows the heap with two structures that are referenced by pointers p and q; (b) shows this heap modelled as a set of functions that define pointer access to a field as a function application.

We need to emphasize a problem with the described naming for heap functions. For each field f_i of some structure we create a heap function with the same name. This can cause a name collision (and hence, destroy the advantage of the approach) if two structures include fields with the same name. To exclude such a situation and to keep uniqueness of heap functions, for every structure s and its field f_i we can generate heap function s_f_i or use some other collision-free naming convention. This functionality is incorporated in the translation tool.

Let us give an example of a state space. Let a program include the following data type, which is used to organize linked lists:

$$list = struct\{cont : int, next : list*\}$$

This structure contains two components: content of type int and a pointer to the next list element. The program state space σ then includes two heap functions produced for the $list$ type:

$$\sigma = (\ldots,\ cont : Ref \to \mathbb{Z},\ next : Ref \to Ref, \ldots).$$

Figure 4.2 illustrates the special case of the split heap model. Part (a) depicts location of the data for the example list of this type on the heap, where the list corresponds to the abstract list $[1, 5, -2, \ldots]$; (b) shows the corresponding heap split modelling.

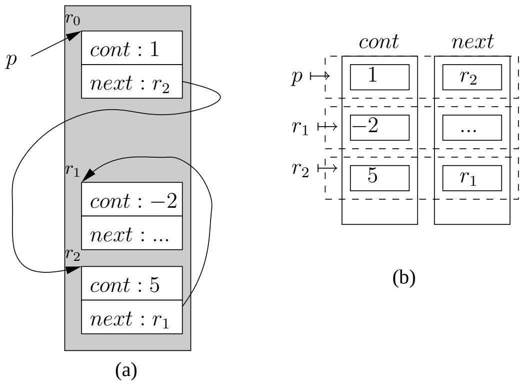


Figure 4.2: Split heap example: (a) heap with objects of *list* type; (b) the corresponding split heap model

Additionally to heap functions for structure values kept on the heap, one also needs to include heap functions for elementary types, e.g. if we have a variable p of type $int*$ we introduce the heap function $int : Ref \to int$ and model dereferencing of p as the function application $int(p)$.

Heap functions are marked as global in the state space, i.e. they are the same for the whole program and changing them during some procedure call, we have them changed after the return from the call. Thus, they model real heap memory behaviour.

Memory Management To model memory allocation we need to keep track of allocated references during the program execution. For that purpose we add an auxiliary component $\sigma.alloc \in Ref^*$ to the state space σ. Since the memory allocation modifies the current statement and does not need any other parameters it is modelled using the *Basic* statement:

$$Basic(\lambda s.s[p := new(\{s.alloc\}), alloc := (new(\{s.alloc\}), s.alloc)])$$

Thus, this statement (we use $p := NEW(alloc)$ as a shortcut for it), being applied to some state, leads to a state where a new reference is allocated and assigned to variable p; the list of allocated references is extended with the new reference. Since lists in the Isabelle/HOL are finite [4] we will always get a new pointer according to Lemma 4.2.2.

4.3 Hoare Logic

Let us look to the theory that is the basis of the verification environment. Most issues we cover in this section concern Hoare logic in general, although we use the language model presented above to illustrate them.

Any variant of Hoare logic deals with so called Hoare triples:

$$P\ c\ Q,$$

including statement c, precondition P (properties of a state space which hold before execution of the program), and postcondition Q (properties that hold after execution of the program).

Since the language model used in the verification environment is quite general and is defined for a polymorphic state space, variables and their types are not fixed until we consider the program needed to be verified. Thus, pre- and postcondition representation is polymorphic as well, and defined as a set of states (where the pre- or postcondition is true).

We only concentrate on the Hoare Logic version for partial correctness, which means that we do not say whether an executed statement terminates but only: if it terminates then some properties hold.

Let $P, Q \in 2^{\Sigma}$ be pre- and postconditions correspondingly, $s, t \in \Sigma$ be states, Γ be the procedure environment (see 4.1), and Θ is a set of Hoare triples that we can assume, then semantics of the Hoare triple is given by two equivalences:

$$\Gamma \models P\ c\ Q \iff \forall s, t.\ \Gamma \vdash s - c \rightarrow t \longrightarrow s \in P \longrightarrow t \in Q$$

The triple $P\ c\ Q$ is valid with respect to Γ (validity is denoted by $\models$) iff the execution of command c starting with s and leading to t implies: if the precondition P holds for s, then postcondition Q holds for t.

The validity with respect to both Γ and Θ is defined as:

$$\Gamma, \Theta \models P\ c\ Q \iff (\forall (P\ c\ Q) \in \Theta.\ \Gamma \models P\ c\ Q) \longrightarrow \Gamma \models P\ c\ Q$$

Thus, if the assumptions from Θ are valid, then the triple $P\ c\ Q$ will be also valid.

The Hoare logic is a set of deductive rules on Hoare triples:

$$\frac{A_0 \dots A_n}{\Gamma, \Theta \vdash P\ c\ Q},$$

where conclusion is that triple $P\ c\ Q$ can be deduced with respect to Γ from assumptions $A_0 \dots A_n$ and from Θ. The rules are recursive and given for every constructor of the statement type similar to the semantics rules.

We do not present the rules in the original form, since they are only used for proving of soundness and completeness of the logic. In the next subsection we introduce the modification of the rules, which is actually used for programm verification and can be derived from the original ones.

The considered Hoare logic version was proven to be both sound and complete in [3].

Theorem 4.3.1 (Soundness) We can only derive (deduce) a valid triple (with respect to the context Γ and Θ)

$$\Gamma, \Theta \vdash P\ c\ Q \longrightarrow \Gamma, \Theta \models P\ c\ Q$$

Theorem 4.3.2 (Completeness) Every valid triple can be derived from the empty context

$$\Gamma, \Theta \models P\ c\ Q \longrightarrow \Gamma, \{\} \vdash P\ c\ Q$$

4.3.1 Verification Condition Generator

Since the Hoare logic is defined inductively by the rules for every language constructor, we can apply them backwards to a statement in order to decompose it to the atomic ones (*Skip* and *Basic*).

The idea of the Verification Condition Generator (VCG) is the following: for the judgement $\Gamma, \Theta \vdash P\ c\ Q$, that we want to prove, we automatically (backwards) apply the Hoare rules until the program c is completely eliminated and a purely logical proof claim $P \subseteq Q'$ remains. Q' is a weakest precondition that is computed from the given postcondition Q by backward rules application. So, the weakest precondition is a set of states which has only the properties, which are necessary in order to guarantee, that execution of the program will end in a state, where the postconditions hold.

In order to be able to apply all the rules automatically we need a version of the rules that can be applied to any Hoare triple and can compute its weakest precondition. Therefor for every language statements we provide a rule of the following format:

$$\frac{P \subseteq WP \quad T_1 \ldots T_i}{\Gamma, \Theta \vdash P\ c\ Q},$$

where $T_1, \ldots, T_i$ are side conditions needed to compute the weakest precondition WP. Moreover, for each modified rule there exist a proof, that it can be deduced from the original ones.

- Let us consider transformation of the original rule for *Basic* as an example:

$$\frac{}{\Gamma, \Theta \vdash \{s.f(s) \in Q\}\ Basic(f)\ Q}$$

  The postcondition can be actually applied to any state, the precondition is the weakest precondition for *Basic* statement, but we can not expect that the triple we want to prove exactly matches the precondition. Thus, we transform to the the following one:

$$\frac{P \subseteq \{s \mid f(s) \in Q\}}{\Gamma, \Theta \vdash P\ Basic(f)\ Q}$$

  This rule has an appropriate format and can be applied to any *Basic* statement.

- The rule for sequential composition combines pre- and postcondtions for both substatements, where R is the weakest precondition for Q and c_2.

$$\frac{\Gamma, \Theta \vdash P\ c_1\ R \quad \Gamma, \Theta \vdash R\ c_2\ Q}{\Gamma, \Theta \vdash P\ Seq(c_1, c_2)\ Q}$$

- For the conditional statement the following rule will be used by the VCG:

$$\frac{\begin{array}{c} P \subseteq \{s \mid (b(s) \longrightarrow s \in P_1) \wedge (\neg b(s) \longrightarrow s \in P_2)\} \\ \Gamma, \Theta \vdash P_1\ c_1\ Q \quad \Gamma, \Theta \vdash P_2\ c_2\ Q \end{array}}{\Gamma, \Theta \vdash P\ Cond(b, c_1, c_2)\ Q}$$

If P_1, P_2 are the weakest preconditions for both branches c_1 and c_2, then the weakest precondition for the conditional combines them with the value of the branching condition b: $\{s \mid (b(s) \longrightarrow s \in P_1) \wedge (\neg b(s) \longrightarrow s \in P_2)\}$. So, if state s satisfies the condition b, then the precondition P_1 have to hold in it; and if s does not satisfy b, then it has to belong to P_2.

- For handling loops we need a rule which allows us to introduce an invariant. Since it cannot be computed by the rules from a while loop, it must be provided by the user. The statement $WhileI(I, e, c)$ introduces a while loop with the annotated invariant. Since the invariant does not have any influence on the deduction, it is semantically defined as a simple while loop $WhileI(I, e, c) = While(e, c)$. We need the invariant annotation only for the rule for the VCG:

$$\frac{P \subseteq I \quad \Gamma, \Theta \vdash (I \cap \{s \mid b(s)\})\ c\ I \quad I \cap \{s \mid \neg b(s)\} \subseteq Q}{\Gamma, \Theta \vdash P\ WhileI(I, b, c)\ Q}$$

  To prove a triple for the while loop by deduction we have to show three subgoals: i) the precondition P must imply the invariant I; ii) the invariant is maintained while the loop is being executed; iii) the invariant and the negated loop condition must imply the postcondition Q.

- The idea of the rule for a procedure call is to simulate any call of the procedure named p, which appears in a program $Call(init, p, res)$ with a call of the procedure specification $\forall x_1 \ldots x_n.\ P'(x_1, \ldots, x_n)\ Call(i, p, r)\ Q'(x_1, \ldots, x_n)$. Logical variables $x_1 \ldots x_n$ are used to parameterize pre- and postconditions, whereas P' and Q' are functions describing the corresponding assertions depending on the parameters $x_1 \ldots x_n$.

$$\frac{\begin{array}{c} P \subseteq \{s \mid \exists y_1 \ldots y_n.\ init(s) \in P'(y_1, \ldots, y_n) \wedge \\ (\forall t.\ t \in Q'(y_1, \ldots, y_n) \longrightarrow res(s, t) \in Q)\} \\ \forall x_1 \ldots x_n.\ P'(x_1, \ldots, x_n)\ Call(i, p, r)\ Q'(x_1, \ldots, x_n) \\ i = (\lambda s.s) \quad \forall s\ t\ .\ res(s, (r(init(s), t)) = res(s, t) \end{array}}{\Gamma, \Theta \vdash P\ Call(init, p, res)\ Q}$$

The final goal to be proved (after application of the VCG to a program) is in the form of set inclusion: $\{\sigma \mid P(\sigma)\} \subseteq \{\sigma' \mid Q(\sigma')\}$. This can be transformed to implication of terms describing the characteristic functions of the sets: $P(\sigma) \longrightarrow Q(\sigma')$, which is easier to work with.

4.3.2 Modelling C0 Language

In this section we present the usage of the verification environment for the C0 programming language. Namely we show how the standard C0 language constructs are described using the general language model. Moreover, we present the concrete syntax for the statements to keep the implementation of procedures considered in Chapter 5 readable.

Let a be a variable in the state space σ, then the variable access is the access to the record field $\sigma.a$, which we denote only by a in the program text.

C0	Simpl	pretty print
`s ; t`	$Seq(s, t)$	$s; t$
`a := b`	$Basic(\lambda s.\ s[a := s.b])$	$a := b$
`if b then {s} else{t}`	$Cond(b, s, t)$	$IF\ \ b\ \ THEN\ \ s\ \ ELSE\ \ t\ \ FI$
`if b then {s}`	$Cond(b, s, Skip)$	$IF\ \ b\ \ THEN\ \ s\ \ FI$
`while b {s}`	$While(b, s)$	$WHILE\ \ b\ \ DO\ \ s\ \ OD$

Table 4.1: Instantiation of the general language model with the C0 language

Table 4.1 sets major correspondence between C0 language constructs, the general language, and the pretty print syntax.

More remarks on instantiating the general language for the C0 case are given below.

- **Arrow access via pointer:** Arrow access `p->f`, where `p` is a pointer to some type t is modelled as application of the generated heap function t_f, i.e. $t\_f(p)$. The pretty print syntax provides the following notation $p \to t\_f$ when presenting a procedure in the verification environment.

- **Point access:** Expressions of a kind `(*p).f` are modelled exactly as `p->f`.

- **Variable of a Structure Type** We denote variables of a structure type by a bold font where it is used as the whole entity in the program, not dividing it to its components. For example, the assignment of two variables v and w of type $t = struct\{f_1 : t_1, \ldots, f_n : t_n\}$ we denote by **v** := **w** instead of assigning each component $\mathsf{v\_f}_1 := \mathsf{w\_f}_1; \ldots; \mathsf{v\_f}_n := \mathsf{w\_f}_n$.

- **Array access:** Access to an array component `a[i]` in C0 corresponds to access to a list element $s.a_i$ in the general language since arrays are modelled as lists. Notation $a[i]$ used as concrete syntax for such an expression in the verification environment.

- **Procedure Definition:** Concrete syntax for defining a procedure F is of the format: *signature* = *body*, where *signature* is:

  $$F(\ list\ of\ parameters \mid list\ of\ resulting\ variables).$$

  The list of resulting variables is used to keep the result of the procedure execution. We use several variables instead of one to be able to model the return of an variable of a structure type, which is flattened to its components.

- **Procedure Call:** A function call is modelled by *Call* statement, the value of functions *init*, *res* depend on concrete program, called procedure, and function call parameters. They are computed for each call case separately (see their meaning in Section 4.1.1). The pretty print syntax for function call `a=F(p0, ..., pn)` is

  $$CALL\ F(p_0, \ldots, p_n, a).$$

Notice, that $p_0, \ldots, p\_n$ are representations of C0 expressions `p0, ..., pn` in the verification environment.

- **Return:** Return statement `return e` is modelled by the assignment of the appropriate values to the resulting variable (or variables), which are mentioned in the procedure signature after the |-symbol.

4.4 Data Abstraction

The approach to specify pre-/postconditions of any statement sequence basically follows the one successfully applied in [24]. During the verification of a very simple program, e.g. performing elementary mathematical computation we can write preconditions and postcondition directly specifying values of variables. For example:

$$\forall \sigma.\ \Gamma \vdash \{\sigma\} \quad a := a * b \quad \{\tau \mid \tau.a = \sigma.a * \sigma.b\}$$

Thus, the program can start in any state (shown by $\{\sigma\}$) without any side condition and it ends in the state, where value of variable a is the product of the values of variables a and b in the state σ.

The approach of assigning values to variables is obviously not enough to deal with large programs working with complex data structures. The way to specify any complicated pre-/postcondition is to define a relation from variables and heap functions to HOL abstract data types. Then we can specify properties of data represented by variables and heap functions on the level of abstract objects.

Let us consider a list abstraction:

Definition 4.4.1 Let $x, y \in Ref$ be references, $h \in Ref \rightarrow Ref$ be a heap function, and $ps \in Ref^*$ be a list of references. We define by the relation

$$Path(x, h, y, ps) = \begin{cases} x = y & \text{if } ps = [] \\ x = q \wedge x \neq Null \wedge Path(h(x), h, y, qs) & \text{if } ps = (q, qs) \end{cases}$$

that there exists a **path** which connects references x, y by the list of references ps, that can be obtained out of the heap h starting with the reference x, following the references in h up to y.

Thus, we have an object twice in the relation: once as the program data structures used to implement it (from the program state), i.e. x, h and y, and second as an abstract object ps. The goal of the abstraction function is to connect these together.

Now using the path relation we can define a list as a path finishing with the null pointer:

Definition 4.4.2 Let $p \in Ref$ be a reference, $h \in Ref \rightarrow Ref$ be a heap function, and $ps \in Ref^*$ be a list of references, then

$$List(p, h, ps) \equiv Path(p, h, Null, ps)$$

claims that there exists a **list** of references ps starting with x so that following references through heap function h it ends with $Null$.

In a similar way we can abstract any objects, which the state space of a program contains. Implicitly we do it when saying that we have lists, tables, or other objects in the program memory. In general, an abstraction function defining some relation between variables, heap functions and abstraction objects looks in the following way:

$$A(v_1, \dots, v_n, h_1, \dots, h_k, a_1, \dots, a_m),$$

i.e. we abstract from variables $v_1, \dots, v_n$ and heap functions $h_1, \dots, h_k$ to one or several abstract objects $a_1, \dots, a_m$ of HOL types.

Thus, by means of abstraction functions we formally give the interpretation to the variables and the heap values of a program.

In the rest of the thesis we use the regular font to distinguish between state space components and logical variables (written in italic), which are used to denote abstract objects, quantified variables etc. We denote access to a state variable by the superscript, i.e. $\sigma.a$ will be denoted by a^σ. Moreover, we will omit the state mark wherever the state is unambiguously defined by the context.

We use the mentioned notation specifying programs in the following way:

$$\forall \sigma.\ \Gamma \vdash \{\sigma \mid \mathsf{b} > 0\} \quad a := a/b \quad \{\tau \mid \mathsf{a} = \mathsf{a}^\sigma/\mathsf{b}^\sigma\}$$

Thus, we omit the state index if we access the current state. only by a component name and access to another state by superscript.

4.5 Application

Verifying a program which consists of more than one procedure we have a so called verification pyramid. We start with procedures, which do not contain any procedure calls in their body and hence, are at the lowest level. Thus, to each procedure we give the specification in the form of a Hoare triple, where the specified program is the corresponding procedure call. Then we use specifications of verified procedures (automatically through the corresponding VCG rule) to verify procedures invoking them. Verification of recursive procedures in the verification environment is realized in the way that such a procedure is verified under assumption that the recursive invocation of this functions satisfies the stated specification. Proceeding this way we get the whole application verified.

In addition to the lemma that the specification is satisfied, for every procedure one needs to prove the lemma, which specifies global state components (heap functions and global variables) that can be changed during the procedure execution. Heap functions not mentioned there are automatically considered as unchanged. However, such lemmas only state which components are changed but do not specify how. So, in the procedure specification for the changed heap functions it is necessary to state explicitly for what inputs they stayed unchanged. Such information can be needed to show that some other data sharing the same heap functions were not corrupted after calling the procedure.

To show that the verification environment is successfully applicable not only for toy examples, it was tested on verification of the real programs which were developed in the Verisoft project. The first results were the verification of the procedure libraries providing data structures and essential operations for some basic data types namely doubly linked lists and strings.

Chapter 5

Verification of the Compiler Implementation

Since it is became clear that the verification environment described in Chapter 4 showed very impressive results being used for the verification of a number of procedures, the next step was to test it on some program which is closer to real ones in the size and complexity and find out if the verification environment is an appropriate means for such kinds of tasks. In the Verisoft project the first such (larger) program was a non-optimizing compiler from the C0 language to the assembler language of the VAMP processor. The VAMP is an out-of-order 32 bit RISC CPU with DLX instruction set defined in the classical Hennesy/Patterson textbook [54].

The size of the implementation whose correctness needs to be shown is large enough (approximately 1500 lines of code); it deals with several complex data types, what implies creating much more complex abstraction functions in comparison to linked lists; the single procedures of the compiler implementation are also more complex.

5.1 Compilation System

The system that performs compilation of C0 programs contains the following components: the front-end, pre-/postprocessor, and the compiler core which realizes a very simple compilation algorithm (see Figure 5.1). The pre-/postprocessor and the compiler core form the compiler back-end. Let us first describe functions of the components.

- **the front-end** (written in the standard C/C++ programming language) parses an incoming C0 program (in a text format), checks its syntactical correctness with respect to the C0 language, elaborates statements and expressions, and produces a program representation in the form of a syntax tree data structure, that can be used by any other application.
- **the compiler pre-/postprocessor** (written in C/C++) converts the data delivered by the front-end to the internal format of the C0 compiler. There are several reasons for having an internal format: i) the front-end outputs

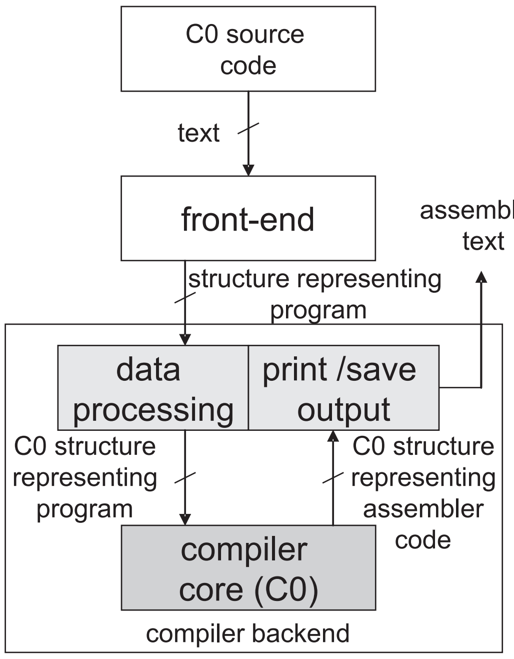


Figure 5.1: Compilation pipeline

data structures that are not compatible to C0 (since they are implemented in C++); ii) the output data structure produced by the front-end can be strongly simplified. Some information stored in the syntax tree data structure is redundant for the compilation algorithm we are using (e.g. braces, some grammar tokens).

The postprocessor includes a procedure to save an assembly program to a disk in a text format.

- **the compiler core** (written in C0) generates the assembly code processing input data in the internal format by a two-pass approach and delivers the produced assembly program back to the compiler pre-/postprocessor.

Thus, the compiler core starts working with a pointer to the data located in the memory, which represent a program to be compiled, placed there by the preprocessor. After compilation is finished, the compiler core returns to the postprocessor the pointer to the data representing a list of assembly instructions, which is the result of compilation.

Of course such a distribution of compiler functions only allows it to function as a back-end. Such an extraction of the input/output operations from the compiler core is due to fact that there are not yet input/output procedures working with a file system written in C0.

If such procedures would exist, then the part of the compiler system that is placed in the postprocessor could be moved into the compiler core. However, the preprocessor would be still necessary, since it serves not only for reading the syntax tree data but also changes their format. The latter, of course, needs to be written in C/C++ and cannot be included into the compiler core. In order to make the compiler core operate as a single program, one would need to modify the preprocessor. This requires the preprocessor to store syntax tree data in the internal format as a file. The compiler core would need to include the procedure call reading data from a file of the required format.

Note, that the distribution in the system described above stays so that parts performing parsing, syntax check, etc. are not written in C0. First, it would be very complicated to implement such a functionality in the C0 language. Moreover, our intention is to verify the the compilation algorithm rather than parsing of the input. So it is enough to have only the compiler core written in C0, which is suitable to be used in the verification environment.

5.1.1 Correctness Statement

We say that the compiler is correct, if for every C0 program P the execution of P on an abstract C0 machine is simulated by the virtual DLX machine running the assembler program, which is the result of the compilation.

We divide the correctness proof into two parts:

- the correctness of compiler specification (i.e. compilation algorithm) with respect to the execution of the compiled code by the processor

- correctness of the implementation with respect to that algorithm

Such a partition is reasonable, since i) proving correctness of a program using the verification environment we only specify the result of the program execution with respect to some formal description; ii) the correctness statement is obviously a step-by-step simulation theorem, which can be shown independently from the implementation based only on the formal description of the algorithm it implements.

In this thesis we will treat only the part of the proof, which concerns the implementation correctness. Moreover we restrict our attention to proving the correctness of the compiler core only. As it was mentioned above we are (presently) omitting the verification of parts of the compilation process such as loading a program from a file, parsing etc. Below we will refer to the compiler core simply by the word 'compiler'.

Thus, to specify the result of the compiler execution in the verification environment we need to define an abstract compilation function, which formally describes a program which is executed by a virtual DLX machine.

Definition 5.1.1 Let $Instr$ be the set of the DLX assembler instructions [54,55]. Let

$$compile(tenv, gst, pt) \in Instr^*$$

be the result of the abstract compiler function with the following input parameters: $tenv$ is a type environment, gst is a global symbol table, and pt is a procedure table.

It is clear, that input parameters of the function fully define a program to be compiled.

To show the implementation correctness we obviously need to show that if the state before the compiler execution (i.e. the content of variables and the heap) can be abstracted to the input parameters of the abstract compiler function *compile*, then the state after the execution can be abstracted to the result of the abstract compiler function.

Theorem 5.1.2 Let $\sigma \in \Sigma$ be a state before the compiler execution, $p \in Ref$ be a pointer to the compiler input data in state σ. Let σ' be a state after the compiler program is executed and $res \in Ref$ be a pointer to the data structure representing a list of assembler instruction in state σ'. If the input data represents some program defined by components $tenv, gst, pt$ then the result of the compiler execution represents assembler program $compile(tenv, gst, pt)$.

$$\begin{gathered}\forall \sigma, tenv, gst, pt. \{\sigma \mid CProg(\mathsf{p}, \sigma, tenv, gst, pt)\} \\ \mathsf{res} := \mathsf{CALL\ compile(p)} \\ \{\sigma' \mid AsmProg(\mathsf{res}, \sigma', compile(tenv, gst, pt))\}\end{gathered}$$

The abstraction functions $CProg$ and $AsmProg$ set relations between the program memory states σ and σ' of the implementation to abstract data, to which we interpret the values of the program variables in these states.

As we said before, the correctness of the compilation algorithm can be shown independently from its implementation. Execution of programs on the abstract machines (C0 and DLX) is defined by means of step functions δ_c and δ_d, respectively. A step function computes the next machine configuration by executing one statement (for C0 machine) or one assembler instruction (for DLX machine). We consider a sequence of configurations of an abstract C0-machine $(C_0, C_1, ...)$ which we have to relate with a sequence of configurations on a DLX machine $(D_0, D_1, ...)$. To describe this relation we use a consistency function $consis(C, alloc, D)$, where the function $alloc$ connects variables of the C0-machine in configuration C with addresses where they are located on the DLX machine in configuration D. The consistency between two machines defined by two aspects stated formally in [56]. Let us shortly describe them:

- control consistency: the program counter of the DLX machine points to the first instruction of the code generated for the statement which is executed next on the C0-machine

- data consistency: addresses of all variables of the C0-machine (including nameless variables on the heap) in the memory of the DLX machine are defined by the *alloc* function. The data are consistent with respect to both machines if

  - for variables of elementary types the value of a variable of the C0-machine is equal to the data stored at the corresponding (via function *alloc*) address in the DLX machine

- for variables of compound types the values of all their subvariables (e.g. array elements and structure fields, flattened up to objects of elementary type where needed) are required to be consistent with values in the memory of the DLX machine
- for pointer variables (including nameless ones on the heap) it is required that the value stored at the allocated address of the pointer contains the allocated address of the target. This defines an isomorphism of the reachable portions of the heap of both the C0 and DLX machines

Note that the allocation function can change during the execution of a program (e.g. because of allocation of new variables on the heap). Thus, we deal with the sequence of allocation functions $alloc_0, \ldots, alloc_i, \ldots$.

The correctness of the compilation algorithm is stated by the following theorem:

Theorem 5.1.3 Let p be a program. Then the execution of p on the C0-machine is simulated by execution of $compile(p)$ on DLX, starting with consistent configurations.

$$\begin{gathered} C_0.pr = p \wedge consis(C_0, alloc_0, D_0) \Longrightarrow \\ \forall i \in \mathbb{N}.\ consis(C_i, alloc_i, D_{s(i)}) \qquad , \end{gathered}$$

where $s(i)$ defines number of instructions needed to be executed on the DLX machine in order to simulate i instructions on the C0-machine (configuration C_i denotes $trace(C_0, i)$).

Now we can reformulate the top most theorem about the compiler implementation combining it with the correctness statement we presented above to state the correct execution of the compiled code on the DLX machine (we present the rough scheme omitting some details). Note that $D_0.m$ and $D_0.PC$ denote memory of the DLX machine and the program counter, respectively. The notation $D_0.m[a : b]$ denotes a range of memory cells from address a to address b. The first element of procedure table pt, namely pt_0, describes `main` function of the compiled program, which has to be executed first. Function $bin \in Instr^* \to \mathbb{Z}^*$ is used to convert an assembler instruction list to its binary representation.

$$\begin{gathered} \forall \sigma, tenv, gst, pt, C_0, D_0 . \{\sigma \mid CProg(\mathsf{p}, \sigma, tenv, gst, pt) \wedge \\ C_0.tt = tenv, C_0.m.gm.st = gst \wedge C_0.pt = pt \wedge \\ C_0.pr = snd(pt_0).body \wedge C.m.lm_0.st = snd(pt_0).loc \wedge \\ (\exists alloc_0\ .\ consis(C_0, alloc_0, D_0)) \wedge \\ D_0.m[D_0.PC : D_0.PC + |compile(tenv, gst, pt)| = bin(compile(tenv, gst, pt))\} \\ \mathsf{res} := \mathsf{CALL\ compile(p)} \\ \{\sigma' \mid AsmProg(\mathsf{res}, \sigma', compile(tenv, gst, pt)) \wedge \\ \forall i \in \mathbb{N}.\ consis(C_i, alloc_i, D_{s(i)})\} \end{gathered}$$

5.2 Verification

The verification process of the compiler implementation includes two main tasks:

- Find proper abstraction functions for input/output data structures. The abstraction functions need to describe not only the relation between the implementation data structures and the data types used for the compiler specification, but also all the necessary properties of pointers involved in the data structures.

- Prove the equivalence of the implementation procedures to the corresponding abstract functions used in the compiler specification.

First we translate the source code of the compiler implementation into Simpl representation. This step is done automatically by the translation tool.

The verification process is obviously iterative. We start with verification of procedures, that do not include any functional calls. We create an initial version of abstraction functions for the data structures which are used in these procedures. During the first verification attempts we show lemmas about required properties of abstraction functions.

During the verification process we might find some details that were not included into the abstraction functions or formulated in an inconvenient way or even incorrectly. So the work also includes the refinement and correction of abstraction function. If there is a number of procedures written using the same template, it is reasonable to find out an approach to show the correctness of the general case once and use it for procedures of this type.

Verifying procedures that include function calls we add new abstraction functions for pointer structures not used before. We might again find gaps in the abstraction function created before, since some properties might be not used by the called procedures. The process is repeated until we reach the main procedure of the verified program.

Chapter 6

Abstract Compiler Function

This chapter gives an overview of the specification of the compiler (i.e. the abstract compiler function) developed in the frame of the Verisoft project [1], the compilation algorithm mainly follows concepts given in [57].

The compiler specification is a set of mathematical functions describing the compilation algorithm and the assembler program produced for any program

$$p = (tenv, gst, pt),$$

which is given by its type environment $p.tenv$, a global symbol table $p.gst$, and a procedure environment $p.pt$. The assembler we use as the output language for the compiler is taken up from [55] and summarized in Appendix A.

In the current chapter we present only the definitions and their properties which we will refer to in the following chapters. The full set of the necessary definitions and lemmas can be found in the corresponding Isabelle theories.

6.1 Memory Layout

One of the tasks of the compiler is to define how the variables of a program will be located in the memory of the DLX machine during the execution. During the execution of a compiled program we create a stack of procedure frames, which keep the local variables of procedures for the time the procedures are being executed.

The VAMP processor requires data to be aligned. Alignment in general is putting data and code in memory at addresses that are more efficient for the hardware to access (often at the price of wasting some memory). RISC processors read from memory in multi-byte 'chunks', usually 4 or 8 bytes long, these 'chunks' must begin at addresses that are multiples of the 'chunk' size. Misaligned memory accesses cause interrupts. We say address a is *aligned* by n if $a \bmod n \equiv 0$. VAMP reads from memory by words and hence needs data aligned by 4. Since VAMP has a word access to memory, procedure frames also need to be located at addresses aligned by 4. A frame includes the header and regions where local variables of the corresponding procedure stored. The header consists of four words to store the following parameters when a procedure is invoked:

- the start address of the previous frame in the stack

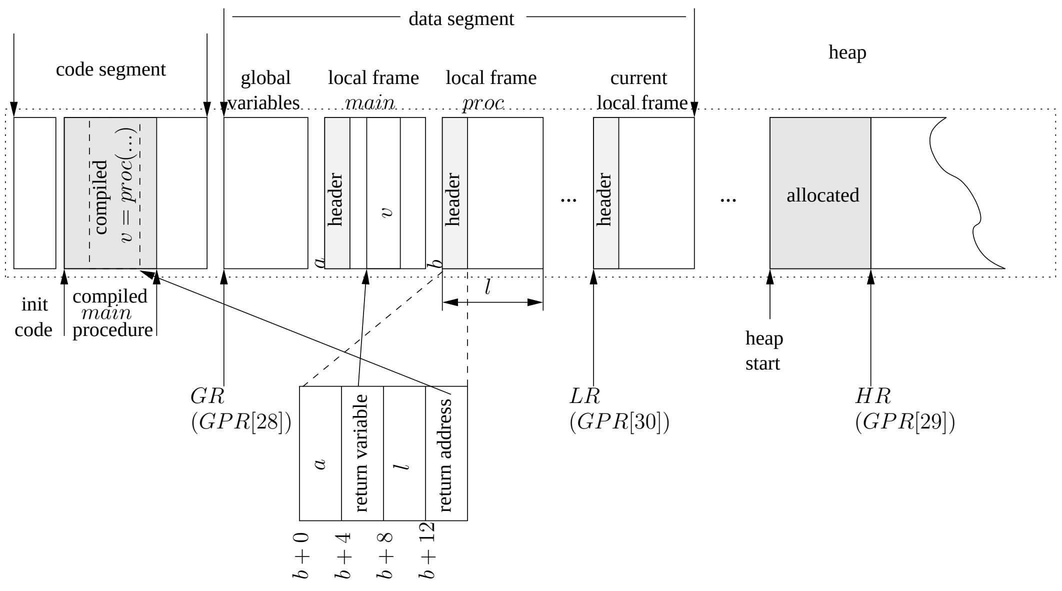


Figure 6.1: Program Memory Layout

- the address of the variable (in the previous frame of the stack), where the result of the current procedure execution has to be returned
- the size of the frame in bytes, which is total size of the header and the data stored in this frame. This information can be used e.g. by low-level software to compute the last occupied address on the stack of local memory frames.
- the program counter in the invoking procedure, i.e. the return address

The frame for global variables is placed below the bottom of the stack, which is the frame of the main function, and it does not include a header (Figure 6.1).

The size of a memory region which is occupied by a variable obviously depends on the amount of memory allocated for its type. Variables are kept aligned in memory as it is required by DLX instruction set architecture. To provide the aligned displacement of variables inside a frame, is one of the tasks of the compiler.

Below we describe formally how alignment of data is supported by the compiler and give the formal definition how amount of memory for allocating of different types is computed.

Alignment and allocation size

Definition 6.1.1 Let $s, d \in \mathbb{N}$ be numbers. Then

$$\lceil s \rceil_d = min\{y \in \mathbb{N} \mid s \leq y \wedge d|y\}$$

is the smallest number greater or equal to s that is divisible by d (denoted by $d|y$).

This number can be easily computed according to the following lemma:

Lemma 6.1.2 For all $d > 0 \Longrightarrow \lceil s \rceil_d = ((s + d - 1)/d) * d$
Note, that / is natural division without remainder, e.g. $5/2 = 2$.

Proof: Let $a = ((s+d-1)/d)*d$. The proof can be split into several subgoals to show that a has all the necessary properties:

i) $s \leq a$ which is true by arguing on arithmetics;

ii) $d|a$ since a is a multiple of d;

iii) a is a minimal number with such properties, i.e. $\forall y.\ s \leq y \wedge d|y \longrightarrow a \leq y$. Let y be a number such that $s \leq y \wedge d|y$. In the case $d|s$ we can easily show that $a = s$, and the claim $a \leq y$ is proven by assumption. Let us consider the second case: $\neg d|s$. From assumption $d|y$ we conclude that exists k such that $y = k \cdot d$ and hence to show the claim we need to prove $a \leq k \cdot d$, i.e. $(s + d - 1)/d \leq k$, which is true, since from the case condition and assumption $s \leq y$ we can show that $s/d < k$, what implies the claim. $\Box$

Now we define how the amount of memory to keep a variable of a particular type is computed.

Definition 6.1.3 Let $t \in \mathcal{T}$ be a type, function $w \in \mathcal{T} \rightarrow \mathbb{N}$ (well defined only for elementary types) defines the number of bytes needed to store one instance of a type, W defines the byte width of the VAMP word. Then mutually recursive functions

$$algn(t) = \begin{cases} w(t) & \text{if } t \text{ is elementary} \\ W & \text{if } t = Ptr(tn) \ \vee \ t = NullT \\ algn(t') & \text{if } t = Arr(n, t') \\ algn\_sc(sc) & \text{if } t = Str(sc) \end{cases}$$

$$algn\_sc(sc) = \begin{cases} 0 & \text{if } sc = [] \\ max(algn(snd(x)), algn\_sc(xs)) & \text{if } sc = (x, xs) \end{cases}$$

define the **type alignment** of t. Function max returns the maximal value between its two arguments.

This definition permits to specify aligned allocation of variables inside a function frame.

For verification of the implementation we need only few properties of the type alignment, whereas arguing on the compilation algorithm includes proof of a large number of lemmas stating properties of this function.

Lemma 6.1.4 Alignment of a valid type is positive:

$$valid_{\mathcal{T}}(t) \Longrightarrow algn(t) > 0$$

Proof: Is straightforward by structural induction on t, using properties stated by the $valid_{\mathcal{T}}$ predicate. Note, that for all elementary types $w(t)$ is obviously larger than zero.

Definition 6.1.5 Let $t \in \mathcal{T}$ be a type. Then mutually recursive functions

$$asize(t) = \begin{cases} algn(t) & \text{if } t \text{ is elementary} \vee t = Ptr(tn) \vee t = NullT \\ n * \lceil asize(t') \rceil_{algn(t')} & \text{if } t = Arr(n, t') \\ asize\_sc(0, sc) & \text{if } t = Str(sc) \end{cases}$$

$$asize\_sc(i, sc) = \begin{cases} i & \text{if } sc = [] \\ i + asize(snd(x)) & \text{if } sc = (x, []) \\ asize\_sc(\lceil i + asize(snd(x)) \rceil_{algn(snd(xs_0))}, xs) & \text{if } sc = (x, xs) \end{cases}$$

define **allocation size** of type t.

The definition computes the number of bytes we need to store a variable of some type in the memory. The size of arrays and structural types pays attention to aligned allocation of the included elements. Note, that to compute the alignment of a structure component, the alignment of the next component (introduced by xs_0, i.e. the first element of the tail in the definition above) is used.

Variable Displacement

Definition 6.1.6 Let v be a variable name and st be a symbol table. Then the **displacement** of the variable with name v inside a frame for st is computed recursively:

$$displ(i, st, v) = \begin{cases} i & \text{if } st = [] \\ \lceil i \rceil_{algn(snd(x))} & \text{if } st = (x, xs) \\ & \wedge fst(x) = v \\ displ(\lceil i \rceil_{algn(snd(x))} + asize(snd(x)), xs, v) & \text{if } st = (x, xs) \\ & \wedge fst(x) \neq v \end{cases}$$

Parameter i in the function definition allows to define the size of the frame header: $displ(16, st, v)$ and $displ(0, st, i)$ for displacement inside a local and the global frame, respectively. Moreover, the function called with $i = 0$ can be used to compute the offset of a structure component inside the component list.

The next lemma presents the dependency between displacement of two adjacent elements in a symbol table:

Lemma 6.1.7 Let $st \in (nm_v \times \mathcal{T})^*$ be a symbol table, then displacement of two consecutive elements are connected as the following:

$$\begin{aligned} &\forall i\ j.\ j + 1 < |st| \wedge unique(st) \longrightarrow \\ &displ(i, st, fst(st_{j+1})) = \\ &\lceil displ(i, st, fst(st_j)) + asize(snd(st_j)) \rceil_{algn(snd(st_{j+1}))} \end{aligned}$$

Proof: By induction on st.

Induction base: $st = []$ Implication holds, since $j + 1 < |st|$ is violated.

Induction step: $st = (x, xs)$. First, we show the case $j = 0$. We expand Definition 6.1.6 on the right side of the claim according to the second case (since $fst(st_0) = fst(x)$). Note, that $st_1 = hd(xs) = xs_0$ and hence, we have

$$displ(i, (x, xs), fst(xs_0)) = \lceil \lceil i \rceil_{algn(snd(x))} + asize(snd(x)) \rceil_{algn(snd(xs_0))} \quad (1)$$

From the other side, expanding definition of *displ* for the left side of the claim with the third case of the same definition (since from $unique((x, xs))$ we can deduce that $fst(x) \neq fst(xs_0)$) we get:

$$displ(i, (x, xs), fst(xs_0)) = displ(\lceil i \rceil_{algn(snd(x))} + asize(snd(x)), xs, fst(xs_0))$$

Expanding it one more time with the second case of the definition (since xs_0 is a head of xs) we end up with (1).

For any $j \neq 0$ we expand the definition at both sides with the third case (since $unique(st) \longrightarrow fst(st_{j+1}) \neq fst(x)$) and instantiating the induction hypothesis (parameter i) with value equal to $\lceil i \rceil_{algn(snd(x))} + asize(snd(x))$ we show the claim. It is clear, that the assumptions of the induction hypothesis are satisfied: $j + 1 < |(x, xs)| \longrightarrow (j - 1) + 1 < |xs|$ and $unique((x, xs)) \longrightarrow unique(xs)$. $\square$

Definition 6.1.8 Let *st* be a symbol table. Then the allocated size of *st* is:

$$asize_{ST}(of, st) = displ(of, st, fst(last(st))) + asize(snd(last(st)))$$

Let us make a pair of remarks about memory allocation. As it can be seen from the definitions above, if we allow parts of the memory word to be allocated for some type, we might have to waste some memory between variables in a frame. The last observation can be illustrated with the following example.

Let us consider two types:

a structure type	$t' = struct\{a : char, b : int, c : bool\}$
an array type	$t = t'[2]$

Let the number of bytes to store the used above elementary types be:

$$w(CharT) = 1;\ w(IntT) = 4;\ w(BoolT) = 1.$$

Let us introduce the following notations:

$$\begin{aligned} C &= (CharN, CharT), \\ I &= (IntN, IntT), \\ B &= (BoolN, BoolT), \end{aligned}$$

where $CharN, IntN, BoolN$ are names for abstract types representing *char*, *int*, and *bool* types, respectively.

Let ft' be the formal representation of the structure type presented above, i.e. $ft' = Struct(C, I, B)$. In the similar way, let ft be the formal representation of the array type, i.e. $t = Arr(2, ft')$.

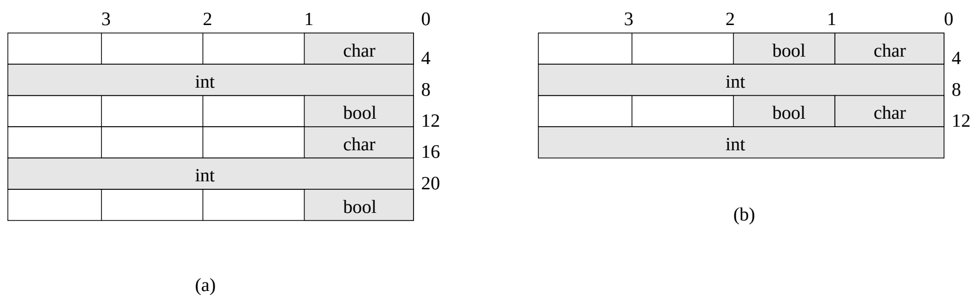


Figure 6.2: Memory allocation for an example type

Computations for the allocation sizes and the displacements are given below.

$$\begin{aligned}
asize(CharT) &= algn(CharT) = 1\\
asize(IntT) &= algn(IntT) = 4\\
asize(BoolT) &= algn(BoolT) = 1\\
algn(ft') &= align\_cs((C, I, B))\\
&= max(algn(CharT), algn\_cs(I, B))\\
&= max(1, max(4, 1)) = 4\\
asize(ft') &= asize\_cs(0, (C, I, B))\\
&= asize(\lceil 0 + asize(CharT)\rceil_{algn(IntT)}, (I, B))\\
&= asize(\lceil\lceil 1\rceil_4 + asize(IntT)\rceil_{algn(BoolT)}, B)\\
&= \lceil 4 + 4\rceil_1 + asize(BoolT)\\
&= 8 + 1 = 9\\
asize(ft) &= 2 * \lceil asize(ft')\rceil_{algn(ft')} = 2 * \lceil 9\rceil_4 = 2 * 12 = 24
\end{aligned}$$

Displacement of the components for a variable of type ft' are computed in the similar way. Figure 6.2(a) demonstrates how such a variable will be placed in memory.

From the definitions above, one can conclude, that the allocation size of a structure type in general depends on the order of its fields. For instance, if we change the order of fields for type ft' in the example above in the way such that $ft' = Struct(C, B, I)$, its allocation size will be $asize(ft') = 8$. Then each instance of type ft will cost one byte less, compare Figure 6.2(a) and (b).

In general, the size of a frame will also depend on the order of the variables which are located within.

6.2 Expression Code Generation

The assembler code for expressions is mostly generated using the Aho-Ullman algorithm [58] for evaluation of an expression syntax tree unless logical AND and OR operators will be evaluated. AND and OR have a fixed evaluation order according to an approach proposed in [59] and which we consider in details below.

Address/Value Evaluation Every expression can be evaluated either to its value or to its address. These cases can appear as the right and left sides of an assignment statement respectively.

In general, we compute a value for expressions e, e_1 in the following cases:

- e is of an elementary or pointer type and at the right side of an assignment or a branching condition for a conditional/loop statement, or a parameter of a procedure call.
- $e \in \mathcal{C}$ - constants can be only evaluated to values
- $\mathit{ArrAcc}(e, e_1)$ - value of e_1 is necessary to compute the address of an array element
- $\mathit{BinOp}(op, e, e_1)$, $\mathit{LazyOp}(op, e, e_1)$, $\mathit{UnOp}(op, e)$ - the direct subexpressions of computational operations

In all other cases we compute the address of expressions.

Address Dereferencing Having an address in the memory we can always *dereference* it to get the value of an elementary type stored at this position.

Definition 6.2.1 Let r be a register where an address to be dereferenced is stored. Let the memory at this address keep a value, which has a type of the size $sz \in \{1, 2, 4\}$. Then the function

$$drf(sz, r) = \begin{cases} [lbu(r, r, 0)] & \text{if } sz = 1 \\ [lhu(r, r, 0)] & \text{if } sz = 2 \\ [lwu(r, r, 0)] & \text{if } sz = 4 \end{cases}$$

generates code for the address dereferencing. The resulting code is one loading assembler instruction depending on the size of data to be loaded. For semantics of assembler instructions see Appendix A.

Code Generation Function The code generation for an expression evaluation requires a number of registers, which are needed to store intermediate values of the evaluation (addresses or values of subexpression). Available registers for code generation do not include: $GPR[0]$, which is of constant value; $GPR[1] - GPR[3]$ which are used by the compiler as auxiliary registers $R_1 - R_3$ respectively; $GPR[28]$ (LR) pointing to the current function frame (current local memory); $GPR[29]$, which is the pointer to the heap last address (heap top); $GPR[30]$ (GR), which is the pointer to the global variables frame in the memory; $GPR[31]$, which used by jal and $jalr$ instructions. We denote the maximal available register list by FR.

Hereafter we use the following shortcuts working with components of program p:

- $valid_{\mathcal{E}}(p, lst, e)$ for $valid_{\mathcal{E}}(p.tenv, p.gst, lst, e)$
- $type_{\mathcal{E}}(p, lst, e)$ as a shortcut for $type_{\mathcal{E}}(p.tenv, p.gst, lst, e)$

- gst in the meaning of $p.gst$
- lst is a list of local variables and depends on the function, which we currently generate the code for

Code generation for an expression in the program p is performed recursively by function

$$code_{\mathcal{E}}(gst, lst, r, d, fr, e) \in Instr^*,$$

which takes as parameters:

- an expression e to be evaluated
- the global symbol table $gst \in (nm_v \times \mathcal{T})^*$
- the current local symbol table $lst \in (nm_v \times \mathcal{T})^*$
- a flag $r \in \mathbb{B}$ saying if expression e has to be evaluated to its address ($r = 0$) or value ($r = 1$)
- a register number $d \in \mathbb{N}$, where we store the result (destination register)
- $fr \in \mathbb{N}^*$ is a list of numbers of the registers that can be used during the evaluation of e (so called free registers).

The result of the function is a list of assembler instructions, performing evaluation of the given expression.

If we are only interested in the size of generated code and not in its content, we use the function

$$csize_{\mathcal{E}}(gst, lst, r, e) \in \mathbb{N},$$

computing the number of instruction in the generated code according to the same algorithm and hence, only provided with parameters which are sufficient to define the number of generated instructions.

Of course, we need a lemma to show the equivalence between the length of the generated code and the result of the $csize_{\mathcal{E}}$ function.

Lemma 6.2.2 For every expression e, which is valid according to type environment $p.tenv$ and symbol tables $p.gst$, lst the following holds:

$$valid_{\mathcal{E}}(p, lst, e) \Longrightarrow csize_{\mathcal{E}}(gst, lst, r, e) = |code_{\mathcal{E}}(gst, lst, r, d, fr, e)|$$

Register Distribution Strategy The strategy of the register distribution between subexpressions is hardcoded in function $code_{\mathcal{E}}$. The way we proceed is described below and illustrated in Figure 6.3.

- The registers we can use for expression evaluation are combined in the free register list $fr = (a, b, c, d, \ldots)$. Initially we have $fr = FR$.
- First we choose the destination register (for simplicity of the formal description in Isabelle/HOL we take the head of the list) and assign it to the root of the expression to be evaluated. The rest is used for evaluation of the subexressions.

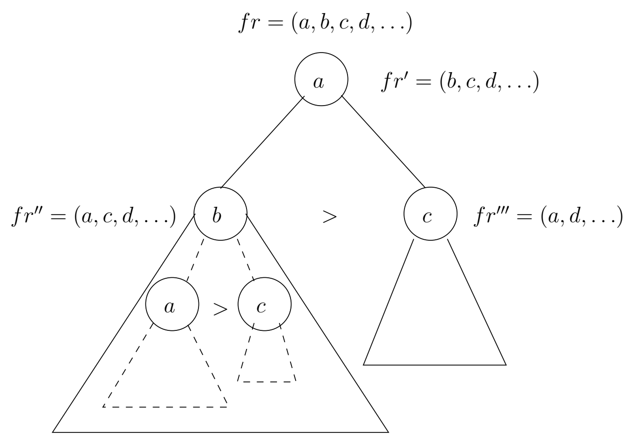

$fr = (a, b, c, d, \ldots)$
$fr' = (b, c, d, \ldots)$
$fr'' = (a, c, d, \ldots)$
$fr''' = (a, d, \ldots)$
a
b
$>$
c
a
$>$
c

Figure 6.3: Register distribution strategy

- The largest subexpression is evaluated (also marked by a register) first. The register is excluded from the free register list and the destination register for the remaining subexpression is being chosen.
- After the registers for subexpression roots are assigned (b and c, respectively), we add register a to the list of remaining free register to be passed to the lower subexpression. It is obvious that it is no problem to rewrite register a evaluating the subexpressions. Notice, that we need two different registers to store the result of subtrees evaluation, in order to avoid that the result of the computation for the second destroys the first result.
- Proceed recursively.

6.3 Expression Code Generation Cases

Below we present the code generation function cases for each kind of expression (function $code_{\mathcal{E}}(gst, lst, r, d, fr, e)$ is denoted by $\mathcal{A}$), and auxiliary functions we use for it. For every code generating function $code_{\ldots}$ we have the corresponding $csize_{\ldots}$ function.

Constant $e = Lit(v)$

$$\mathcal{A} = code_{\mathcal{C}}(d, v) = \begin{cases} [subi(d, 0, 1)] & \text{if } v = Nil \\ [ori(d, 0, i)] & \text{if } v = Char(i) \\ [addi(d, 0, \langle b \rangle)] & \text{if } v = Bool(b) \\ (lhgi(d, c[31:16] \oplus c[15]^{16}), xori(d, d, sext(c[15:0]))) & \text{if } v = Int(i) \lor \\ & \;\; v = Usgn(i) \end{cases},$$

where $\langle b \rangle$ is a number representation of a boolean value b, c is a bit vector representation of numerical constant i, i.e $\langle c \rangle = i$, and $sext$ provides the 32-bit sign extended version of a bit vector (for necessary notations have a look in Application A, for more details see [55]).

Thus, after execution of this code we have the constant value stored in the destination register d: -1 for the null pointer, and 0/1 for boolean constants.

Let us concentrate on the last case (32-bit constant) in more detail. Since the immediate constant, which can be loaded in a general purpose register, is only 16-bit wide, we need to use some trick programming to get a 32-bit constant loaded in a register.

The first thought would be i) to chop the constant into two 16-bit parts, ii) load the upper bits $c[31:16]$ in the upper part of the register using *lhgi* command, iii) perform either plus or bitwise or operation with the lower bits $c[15:0]$. Unfortunately, this does not work, since operations with immediate constants use its sign-extended version. Thus, in the case if $c[15]=1$ instead of having added $c[31:16]0^{16}$ with $0^{16}c[15:0]$ (as we would expect), we actually add $c[31:16]0^{16}$ with $1^{16}c[15:0]$, that, of course, results in an incorrect value loaded in the register.

Let us show that the above presented code actually performs the desired operation. After the first instruction (*lhgi*) we have:

$$GPR[d] = (c[31:16] \oplus c[15]^{16})0^{16}.$$

After the second one:

$$\begin{aligned} GPR[d] &= ((c[31:16] \oplus c[15]^{16})0^{16}) \oplus (c[15]^{16}c[15:0] \\ &= ((c[31:16] \oplus c[15]^{16}) \oplus c[15]^{16})(0^{16} \oplus c[15:0]) \\ &= c[31:16]c[15:0] \\ &= c \end{aligned}$$

Variable access $e = VarAcc(vn)$

First, we generate the code to load the address of the variable into the register (for the case it is not in the range of an immediate constant).

$$code_{ofs}(vn, lst, gst) =$$
$$\begin{cases} code_{\mathcal{C}}(R_1, Usgn(displ(16, lst, vn))) \circ [add(d, LR, R_1)] & \text{if } n \in map(fst, lst) \\ code_{\mathcal{C}}(R_1, Usgn(displ(0, gst, vn))) \circ [add(d, GR, R_1)] & \text{otherwise} \end{cases}$$

Thus, we store to the auxiliary register R_1 the sum of the frame address (global or local) and the variable displacement.

Second, we need to know the type of the variable with name vn

$$t = \begin{cases} map\_of(lst, vn) & \text{if } vn \in map(fst, lst) \\ map\_of(gst, vn) & \text{if } vn \notin map(fst, lst) \wedge vn \in map(fst, gst) \\ \epsilon & \text{otherwise} \end{cases}$$

If the name is not in either of the two symbol tables, the type is undefined. We cannot generate any reasonable code if the variable name is not declared in the symbol tables.

Generating the code we additionally dereference the address of the variable (to get its value) if the flag r is set.

$$\mathcal{A} = \begin{cases} code_{ofs}(vn, lst, gst) \circ drf(asize(t), d) & \text{if } t \neq \epsilon \wedge r \\ code_{ofs}(vn, lst, gst) & \text{if } t \neq \epsilon \wedge \neg r \\ \text{undefined} & \text{otherwise} \end{cases}$$

The previous expressions are the base cases of the recursion and register distribution for them have been already done. The code generation function for the following cases includes the distribution algorithm.

Binary Operators $e = BinOp(op, e_1, e_2)$
Let us introduce the following notations:

$$\begin{aligned} & d_1 = hd(fr), fr_1 = d \circ tl(fr), \\ & d_2 = hd(tl(fr)), fr_2 = d \circ (tl(tl(fr))) \\ & t_1 = type_{\mathcal{E}}(p, lst, e_1), t_2 = type_{\mathcal{E}}(p, lst, e_2), \end{aligned}$$

where d_1 and d_2 are registers, where the results of the subtrees evaluation will be stored; fr_1 and fr_2 are the register lists which can be used for evaluation of the subtrees; t_1 and t_2 are types of the subexpressions.

Thus, providing fr is distinct we can show, that the results of evaluation for e_1 and e_2 will be stored in the different registers. In order to allow larger expressions to be evaluated we can still use the register for the main result (d) in the evaluation of the subtrees.

We choose the order of the evaluation of the subexpression according to their size: the larger subtree is evaluated first. The size of an expression is defined by function $size \in \mathcal{E} \to \mathbb{N}$ which computes the number of inner nodes of the syntax tree of an expression. The code is generated only in case if types of subexpressions are defined: $t_1 \neq \epsilon$ and $t_2 \neq \epsilon$, otherwise the code is undefined.

$$\begin{aligned} & size(e_2) < size(e_1) \\ & \mathcal{A} = code_{\mathcal{E}}(gst, lst, True, d_1, fr_1, e_1) \circ \\ & \qquad code_{\mathcal{E}}(gst, lst, True, d_2, fr_2, e_2) \circ \\ & \qquad code_{op_b}(d, d_1, d_2, t_1, t_2, op) \end{aligned}$$

$$\begin{aligned} & size(e_1) \leq size(e_2) \\ & \mathcal{A} = code_{\mathcal{E}}(gst, lst, True, d_1, fr_1, e_2) \circ \\ & \qquad code_{\mathcal{E}}(gst, lst, True, d_2, fr_2, e_1) \circ \\ & \qquad code_{op_b}(d, d_2, d_1, t_2, t_1, op) \end{aligned}$$

Both subexpression must be evaluated to their value, since binary operations are defined only for elementary numerical types.

Function $code_{op_b}$ generates the code template for binary operator op, which is usually one corresponding assembler instruction, e.g. $add(d, d_1, d_2)$ for the *plus* binary operator. Although, for several operators we need to generate more than one instruction, namely for multiplication and division. Since the VAMP does not include hardware means to fulfill the multiplication and division, we need to emulate them in software.

Another case where we need to emulate the hardware execution is a comparison of unsigned integers. Since the numbers in the processer registers are interpreted as signed integers, then in the case the highest bit of at least one of the compared numbers is 1, the operation will give incorrect answer if we want to interpret the content of the registers as natural numbers. To recognize this case, we provide the $code_{op_b}$ function with the types of the subexpression. The exact instructions and templates, which are generated for the binary operation can be found in Appendix B.1.

Unary operators $e = UnOp(e_1, op)$

The evaluation of the unary operations is done in a similar way, but in this case we have no case distinction in the order of evaluation. We use the same notations as in the previous case.

$$\mathcal{A} = code_{\mathcal{E}}(gst, lst, True, d_1, fr_1, e_1) \circ code_{op_u}(d, d_1, op)$$

Function $code_{op_u}$ generates one corresponding assembler instruction for each operation (the exact instructions which are generated can be found in Appendix B.1).

Array Access $e = ArrAcc(e_1, e_2)$

The evaluation order is the same as in the case of the binary operations, i.e. we evaluate the larger subtree first. We present here only one case ($size(e_2) < size(e_1)$), the second is done analogously.

$$\begin{aligned}\mathcal{A} = {} & code_{\mathcal{E}}(gst, lst, False, d_1, fr_1, e_1) \circ \\ & code_{\mathcal{E}}(gst, lst, True, d_2, fr_2, e_2) \circ \\ & code_{arracc}(d, d_1, d_2, r, t_1)\end{aligned}$$

The result of evaluation of the right subexpression e_2 (to the value) is the number of an element we are going to access. We need to know the start address of an array in the memory in order to compute the position of the element and hence, expression e_1, which has to be of an array type, is evaluated to its address.

Let us denote by $t = snd(the\_Arr(t_1))$ the type of the array elements, then $ofs\_fct = \lceil asize(t) \rceil_{algn(t)}$ is the offset factor of an element in the array.

Function $code_{arracc}$ is defined in as follows:

$$code_{arracc} := \begin{cases} compute\_elt\_addr(d, d_1, d_2, ofs\_fct) & \text{if } \neg r \\ compute\_elt\_addr(d, d_1, d_2, ofs\_fct) \circ drf(asize(t), d) & \text{otherwise} \end{cases}$$

Function $compute\_elt\_addr$ generates the template (i.e. a small assembler program) that implements the operation with the following result:

$$GPR[d] := GPR[d_1] + GPR[d_2] * ofs\_fct$$

Thus, the function computes the address of the accessed element as the sum of the array address and the element displacement. Since currently there is no multiplication operation in VAMP, the software emulation of multiplication is again necessary.

Structure access $e = StrAcc(e_1, cn)$

We use the same notation as before and introduce some additional notations: the type of the component we access is $st = map\_of(the\_Str(t_1), cn)$, and the displacement of the component of such a type inside the memory region for the structure is $dsp = displ(0, the\_Str(t_1), cn)$. The following template is generated for the structure access node of the expression syntax tree:

$$code_{stracc}(d, d_1, r, t_1, cn) = \begin{cases} [addi(d, d_1, dsp)] & \text{if } \neg r \\ (addi(d, d_1, dsp), drf(asize(st), d)) & \text{otherwise} \end{cases}$$

The subexpression of a structure access is evaluated to its address. Register distribution is done as in the case of unary operations.

$$\mathcal{A} = code_{\mathcal{E}}(gst, lst, False, d_1, fr_1, e_1) \circ code_{stracc}(d, d_1, r, t_1, cn)$$

Pointer Dereferencing $e = Deref(e_1)$

The dereferencing code is trivial, since we already have the address of the subexpression in the register determined by the distribution algorithm. Thus, we just copy the value to the destination register and generate address dereferencing code if it is necessary. The code is generated only if the dereferenced type $t = map\_of(p.tenv, the\_Ptr(t_1))$ is valid

$$code_{deref}(d, d_1, r, t_1) = \begin{cases} [add(d, d_1, 0)] & \text{if } t \neq \epsilon \wedge \neg r \\ (add(d, d_1, 0), drf(asize(t), d)) & \text{if } t \neq \epsilon \wedge r \\ \text{undefined} & \text{otherwise} \end{cases}$$

The evaluation scheme for the whole expression is the same as for a unary operator. The subexpression is evaluated to its address.

$$\mathcal{A} = code_{\mathcal{E}}(gst, lst, False, d_1, fr_1, e_1) \circ code_{deref}(d, d_1, r, t_1)$$

Address-of $e = AddrOf(e_1)$

The code generated for this operation is trivial, since we already have the address of the subexpression in the register determined by the distribution algorithm for it and we do not even need to dereference it.

$$\mathcal{A} = code_{\mathcal{E}}(gst, lst, False, d_1, fr_1, e_1) \circ [addi(d, d_1, 0)]$$

"Lazy" Binary Operation $e = LazyBinOp(op, e_1, e_2)$

For expressions of that type we have a fixed evaluation scheme, which does not depend on the size of the subtrees: the left subexpression is evaluated first. It is reasonable, since for the logical operations AND and OR, knowing that the result

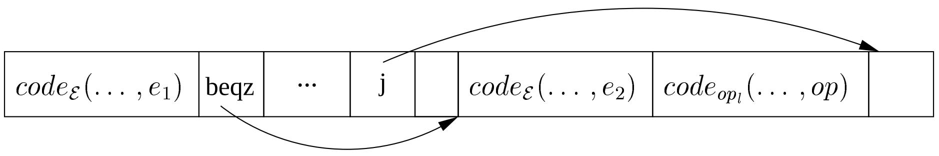

$code_{\mathcal{E}}(\ldots, e_1)$
beqz
...
j
$code_{\mathcal{E}}(\ldots, e_2)$
$code_{op_l}(\ldots, op)$

Figure 6.4: Execution of lazy operator

of the left subexpression has been evaluated to $False$ and $True$ respectively, we do not need to evaluate the right subexpression at all, since the computed values are dominating. To skip the execution of the code generated for the right subexpression we need to insert some additional instructions between them, implementing a jump to the part of the code, which is needed to be executed next.

Let us present this additional code for the logical OR operator

$$\begin{aligned} code_{lazy}(d, d_1, dist, log\_or) &= [beqz(d_1, 16), nop, \\ &\quad addi(d, d_1, 0), \\ &\quad j(dist + 4), nop] \end{aligned}$$

Thus, if the content of $GPR[d_1]$ is zero we have to evaluate the second subexpression and jump to the first position of the corresponding code. Otherwise, we set the content of the destination register d to $true$ and jump over the code generated for the second subexpression and for the root node (Figure 6.4). Thus, the jump distance given by parameter $dist$ is the byte length of the code generated for the right subexpression and for the root of the syntax tree of expression e. Instruction nop after the control commands is due to the delay slot mechanism used in VAMP (see Appendix A, [55]). The entire code generated for e is the following:

$$\begin{aligned} \mathcal{A} = {} & code_{\mathcal{E}}(gst, lst, True, d_1, fr_1, e_1) \circ code_{lazy}(d, d_1, dist, op) \circ \\ & code_{\mathcal{E}}(gst, lst, True, d_2, fr_2, e_2) \circ code_{op_l}(d, d_1, d_2, t_1, t_2, op), \\ \text{where } & dist = 4 \cdot (csize_{\mathcal{E}}(gst, lst, True, e_2) + csize_{op_l}(op)) \end{aligned}$$

Function $code_{op_l}$ generates the corresponding instructions for each of the lazy operators.

Definition 6.3.1 Let gst, lst be the global and a local symbol table, respectively, e be an expression, r be a address/value flag. The predicate $enough(gst, lst, r, e, n)$ states, that n registers are enough for evaluation of e from the program p, with respect to st and flag r.

The predicate is structured in the same way as $code_{\mathcal{E}}$. Let us consider the example case for $e = LazyBinOp(op, e_1, e_2)$:

$$\begin{aligned} enough(gst, lst, r, LazyBinOp(op, e_1, e_2), n) \equiv {} & 2 \le n \wedge \\ & enough(gst, lst, True, e_1, n) \wedge \\ & enough(gst, lst, True, e_2, n - 1) \end{aligned}$$

We can summarize that for an expression with two subexpressions predicate *enough* holds if $n \geq 2$ and *enough* is *true* being tested recursively for the subexpressions (called with parameters n and r defined correspondingly to the definition of $code_{\mathcal{E}}$). If an expression includes only one subexpression, then i) $n \geq 1$ is required and ii) the recursive call of *enough* for the subexpression needs to return *true*. For expressions without subexpressions (e.g. a variable access, a constant) it is always true. Recall that free registers for the evaluation do not include the destination register, so for the last case it is completely enough to have only the destination register to evaluate the expression even if the free register list is empty.

6.4 Statement Code Generation

Let us shortly present a code generation algorithm for statements $s \in \mathcal{S}$ in program p, which is defined recursively by the function

$$code_{\mathcal{S}}(gst, f, s) \in Instr^*.$$

The parameter f is the procedure declaration, in whose body statement s is located, gst is the global symbol table of the program.

The list of free registers fr that we use for evaluation of expressions included in statement s is set to FR (i.e. contains all available free registers). The local symbol table used for evaluation is $snd(f).loc$, which we denote by lst_f below.

Analogously to expressions we have a function $csize_{\mathcal{S}}(gst, f, s) \in \mathbb{N}$ simply computing the length of the generated code in words (or instructions), without producing the compiled instruction list, that corresponds to the number of instructions $|code_{\mathcal{S}}(gst, f, s)|$.

Skip We do not generate any instructions.

Sequence $s = Comp(s_1, s_2)$

We generate the code for both substatements consequently:

$$\mathcal{A} = code_{\mathcal{S}}(gst, f, s_1) \circ code_{\mathcal{S}}(gst, f, s_2)$$

Assignment $s = Ass(e, e_1, id)$

The code generation is only possible provided both expressions are valid. We evaluate both expressions and copy the result. The copying operation depends on the type of expressions in the assignment: the memory region for the complex data types is copied in a loop. The corresponding code is generated by function $code_{ass}(b, sr, d, sz) \in Instr^*$, where parameters are:

- $b \in \mathbb{B}$ - flag, showing either an object of an elementary or pointer type ($b = 0$) or an object of a complex type ($b = 1$) must be copied (the latter e.g. in the case of assigning structures, or passing an array as a parameter to a function call);
- $sr, d \in \mathbb{N}$ - are source and destination registers;

- sz - number of adjacent memory words to be copied.

Function $code_{ass}$ copies sz of adjacent memory cells including the gaps, which possible in memory allocated for structure types due alignment.

In the definitions below we use the following notations for register distribution for expression evaluation : $d_1 = hd(FR)$, $fr_1 = tl(FR)$, $d_2 = hd(tl(FR))$, $fr_2 = tl(tl(FR))$.

$$\begin{aligned}\mathcal{A} = {} & code_{\mathcal{E}}(gst, lst_f, False, d_1, fr_1, e) \circ code_{\mathcal{E}}(gst, lst_f, r, d_2, fr_2, e_1) \circ \\ & \quad code_{ass}(\neg r, d_1, d_2, asize(type_{\mathcal{E}}(p, lst_f, e))), \\ & \text{where } r = \neg(is\_Str(type_{\mathcal{E}}(p, lst_f, e)) \vee is\_Arr(type_{\mathcal{E}}(p, lst_f, e)))\end{aligned}$$

The left expression is evaluated to its address and the right one depending on its type.

Memory allocation $s = Alloc(e, tn, id)$ The execution of the allocation operator includes the following tasks: i) computation of the aligned address a for a new memory region with respect to the previous heap pointer ($GPR[29]$) and the type we allocate the instance of; ii) store address a plus the allocated size for the type as a new value for the heap pointer; iii) assign a to the expression e.

Thus, during the code generation we proceed as follows. We evaluate the expression first and then generate a code template for allocation of the memory on the heap. The template (see Appendix B.2) includes software emulation of division for aligned address computation. It is introduced by function $code_{alloc}(al, as, d) \in Instr^*$, where parameters are: al - alignment of a type to be allocated; as - its allocated size; d - register where the start address of the newly allocated memory chunk must be stored.

$$\begin{aligned}\mathcal{A} = {} & code_{\mathcal{E}}(gst, lst_f, False, d_1, fr_1, e) \circ code_{alloc}(algn(t), asize(t), d_1), \\ & \text{where } t = map\_of(p.tenv, tn)\end{aligned}$$

Since we know the type name we allocate the memory for, we can find its characteristics in the type environment.

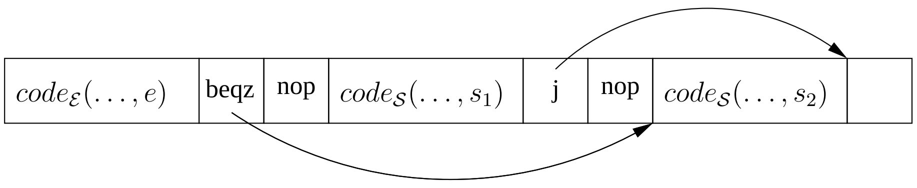
As we mentioned before, execution of the code generated by function $code_{alloc}$ must give the following results:

$$\begin{aligned}GPR[29] &:= \lceil GPR[29] \rceil_{algn(t)} + asize(t) \\ m(GPR[d_1]) &:= \lceil GPR[29] \rceil_{algn(t)}\end{aligned}$$

i.e. allocation on the heap preserves the alignment of data in the memory.

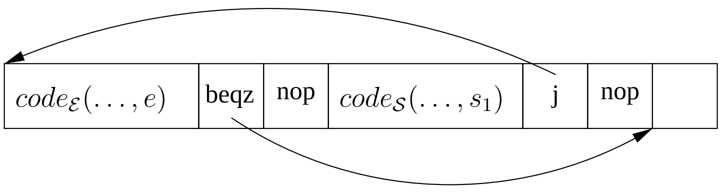
Conditional $s = Ifte(e, s_1, s_2, id)$

We consequently evaluate the conditional expression e, generate two pieces of code corresponding to statements s_1 and s_2, and connect them by assembler



$code_{\mathcal{E}}(\ldots, e)$
beqz
nop
$code_{\mathcal{S}}(\ldots, s_1)$
j
nop
$code_{\mathcal{S}}(\ldots, s_2)$

(a)



$code_{\mathcal{E}}(\ldots, e)$
beqz
nop
$code_{\mathcal{S}}(\ldots, s_1)$
j
nop

(b)

Figure 6.5: Execution of (a) conditional statement; (b) loop

instructions executing the choice between the branches (Figure 6.5(a)).

$$\begin{aligned}\mathcal{A} = {} & code_{\mathcal{E}}(gst, lst_f, True, d_1, fr_1, e) \circ \\ & [beqz(d_1, 4 \cdot csize_{\mathcal{S}}(gst, f, s_1) + 12), nop] \circ \\ & code_{\mathcal{S}}(gst, f, s_1) \circ \\ & [j(4 \cdot csize_{\mathcal{S}}(gst, f, s_2) + 4), nop] \circ \\ & code_{\mathcal{S}}(f, s_2)\end{aligned}$$

Loop $s = Loop(e, s_1, id)$

The approach is similar to the previous case: we evaluate the loop branching expression to a value, then depending on this value we execute the code generated for loop body s_1 or jump to the first instruction after the code corresponding to the loop. Since we repeat the loop execution until the branching condition is valid we generate a jump to the beginning of the loop code and place it after the code for s_1 (Figure 6.5(a)).

$$\begin{aligned}\mathcal{A} = {} & code_{\mathcal{E}}(gst, lst_f, True, d_1, fr_1, e) \circ \\ & [beqz(d_1, 4 \cdot csize_{\mathcal{S}}(gst, f, s_1) + 12), nop] \circ \\ & code_{\mathcal{S}}(gst, f, s_1) \circ \\ & [j(-4 \cdot csz + 4), nop], \\ \text{where } & csz = csize_{\mathcal{E}}(gst, lst_f, True, e) + csize_{\mathcal{S}}(gst, f, s_2)\end{aligned}$$

Function Call $s = Call(vn, fn, el, id)$

First, we generate code for the variable access, i.e. we evaluate variable with name vn to its address (since we need to know where the result of the function call needs to be stored).

Second, we compute the base address of the new local memory frame $lba = \lceil LR + asize_{ST}(16, lst_f)\rceil_L$. It is the first address after the last variable in the current memory frame, aligned by L.

As the next step, we perform parameter passing (the corresponding code is generated by function *par_pass*). That includes the evaluation of every expression el_i from the expression list el and copying the evaluation result to addresses $lba + displ(16, lst', par_i)$ (as by an assignment), where the new local symbol table is $lst' = map\_of(p.pt, fn).loc$ and parameter names par_i are $fst(map\_of(pt, fn).par_i)$. The correct code generation, of course, is only possible in case the called procedure exists in the procedure table: $map\_of(p.pt, fn) \neq \varepsilon$.

As the last action we initialize the header of the new frame, update the stack pointer, and jump to the code piece corresponding to the called procedure (the corresponding code is generated by function *init_frame*, see Application B.2).

So the entire code is assembled as the following:

$$\begin{aligned}\mathcal{A} \;&=\; code_{\mathcal{E}}(gst, lst_f, False, d_1, fr_1, VarAcc(vn)) \circ \\ &\qquad par\_pass(gst, lst_f, fr_1, el) \circ init\_frame(d_1, lba)\end{aligned}$$

Function *init_frame* performs the following operations (the header layout to refer is depicted in Figure 6.1):

- the stack pointer is updated $LR' = LR + lba$
- the first word of the header is the old stack pointer $m(LR') = LR$
- the second word is the register where the address of the result variable is stored $m(LR' + 4) = d_1$
- the third word keeps the aligned allocated size of the new frame $m(LR'+8) = \lceil asize_{ST}(16, lst')\rceil_L$
- the last word of the header keeps the program counter position where the execution will return after the called function is finished (saved by a *jal* instruction in $GRP[31]$): $m(LR' + 12) = GRP[31]$

As we mentioned above, the last two instructions of the *init_frame* function generating the stack initialization template are jump to the called function code, and store the previous value of the program counter (31 - the link register, 28 - the current frame register):

$$[\ldots, jal(dist), sw(R31, R28, 12)].$$

The jump distance *dist* obviously depends on: i) the offset of the called function code the code for whole program; ii) the offset of the caller inside the program code; iii) the size of the code, which was generated before this call statement.

Definition 6.4.1 Let $pl = (x, xs) \in (nm_{\mathcal{P}} \times \mathcal{P})^*$ be a non empty procedure list in program p, gst be its global symboltable, and $f \in (nm_{\mathcal{P}} \times \mathcal{P})$ be a procedure. Then function

$$ba(pl, gst, f) = \begin{cases} 0 & \text{if } f = x \\ 4 * (csize_{\mathcal{S}}(gst, f, snd(f).body) + ba(xs, gst, f)) & \text{otherwise} \end{cases}$$

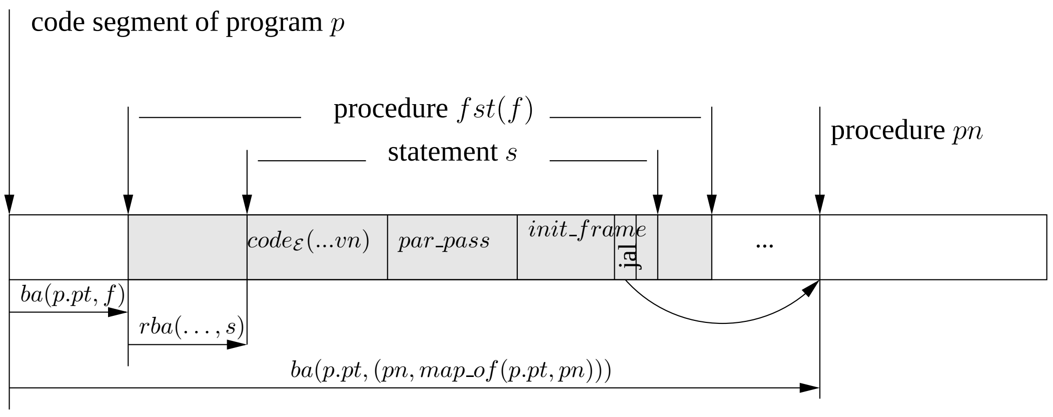


Figure 6.6: Jump distance computation for a call statement

computes the base address of the code for procedure f.

Thus, we arrange code chunks generated for program procedures in the order they appear in the procedure environment.

For any substatement s_2 of statement s_1 (located in the body of procedure f) function $rba(gst, f, s_1, s_2) \in \mathbb{N}$ recursively computes the *relative base address* of the code generated for statement s_2 in the code generated for statement s_1. The definition is simple and done by means of the $csize_{...}$ functions. So, function *rba* is very similar to function *ba*, but the former operates on statements inside a procedure body whereas the latter operates on procedures inside a procedure list.

Obviously, the jump distance *dist* is formally (see Figure 6.6):

$$\begin{aligned} dist &= 4 \cdot (ba(p.pt, gst, (pn, map\_of(p.pt, pn))) \\ &\quad - \\ &\quad (ba(p.pt, gst, f) + rba(gst, f, snd(f).body, s) + \\ &\quad csize_{\mathcal{E}}(gst, lst_f, False, VarAcc(vn)) + csize_{par\_pass}(gst, lst_f, el) + \\ &\quad |init\_frame(lba)| - 1)) \end{aligned}$$

Figure 6.6 illustrates different segments of the compiled code, whose sizes are mentioned in the equation for the jump distance computation.

Return $s = Return(e, id)$

The code for a return statement has the following structure:

$\mathcal{A}$	$=$	$code_{\mathcal{E}}(gst, lst, r, d_1, fr_1, e) \circ$	evaluate the result
		$[lw(R_2, 28, 4)] \circ$	get the address where the resulting value needs to be returned
		$code\_ass(r, d_1, R_2, asize(t)) \circ$	copy the result
		$[lw(R_2, 28, 8),$	get return address in the code of the caller
		$lw(28, 28, 0),$	restore the previous stack pointer
		$jr(R_2), nop]$	jump back

where expression type $t = type_{\mathcal{E}}(p, lst_f, e)$ and flag $r = \neg(is\_Str(t) \vee is\_Arr(t))$ define the size of memory to be copied.

Lemma 6.4.2 For every statement s, which is valid according to type environment $p.tenv$, procedure environment $p.pt$, and symbol tables $p.gst, lst_{pn}$ the following holds:

$$valid_{\mathcal{S}}(p, lst_f, s) \Longrightarrow csize_{\mathcal{S}}(gst, f, s) = |code_{\mathcal{S}}(gst, f, s)|$$

We use predicate $valid_{\mathcal{S}}(p, lst_f, s)$ as a shortcut for $valid_{\mathcal{S}}(p.tenv, p.pt, p.gst, lst_f, s)$.

Let us present some additional lemmas for rba we will mention in the following correctness proof.

Lemma 6.4.3

$$s2l(s) \neq [] \wedge distinct_{\mathcal{S}}(s) \Longrightarrow rba(f, s, hd(s2l(s))) = 0$$

Lemma 6.4.4

$$\begin{aligned}&sub\_stmt(Comp(s_1, s_2), s) \wedge distinct_{\mathcal{S}}(s) \wedge s2l(s_1) \neq [] \wedge s2l(s_2) \neq [] \Longrightarrow\\ &rba(f, hd(s2l(s_2)), s) = rba(f, hd(s2l(s_1)), s) + 4 * csize(f, s_1)\end{aligned}$$

6.5 Program Code Generation

As we already mentioned above, in order to generate assembler code for a program p we consecutively generate the code for every procedure from its procedure environment and append the generated program pieces together.

Definition 6.5.1 Let $pl \in (nm_{\mathcal{P}} \times \mathcal{P})^*$ be a procedure list. Then function

$$code_{PT}(pl) = \begin{cases} [] & \text{if } pl = [] \\ code_{\mathcal{S}}(x, snd(x).body) \circ code_{PT}(xs) & \text{if } pl = (x, xs) \end{cases}$$

generates code for pl with respect to the environment of the program p.

Chapter 7

Abstraction Functions

In Section 4 we mentioned the approach to formulate pre-/postconditions for procedures working with pointer data structures. In this chapter we apply this technique to the data structures which are the input/output data format for the compiler implementation. These data types are set in the relation to the abstract types used to formally specify a program to be compiled (see Section 2.1).

7.1 Basic Data Types

In this section we present two common data structures we use in the program as the skeleton to build more complex ones, namely doubly linked list and binary tree. We define their abstraction functions to HOL data types and some of their properties.

7.1.1 Doubly Linked List

A doubly linked list is a list, whose elements have two pointers: to the previous and to the next element in the list. Therefore we need two fields in the C0 structure type to construct such a list and hence, two corresponding heap functions in the state space of a program:

$$struct\ dlist = \{next : dlist*;\ prev : dlist*, ...\}$$

Each element of a list also includes some content (denoted by dots), which is (in the compiler implementation) usually a pointer to a structure embodying the content of the list element.

Definition 7.1.1 Let $x, y \in Ref$ be references, $n, p \in Ref \to Ref$ be heap functions, and $ps \in Ref^*$ be a reference list, then the relation

$$dList(x, n, p, y, ps) \equiv List(x, n, ps) \wedge List(y, p, rev(ps))$$

is an abstraction function for a **doubly linked list**.

So, starting with the reference x and following the heap function n we obtain the reference list ps and we obtain the reversed list $rev(ps)$ if we start with reference y and follow the heap p. For instance, having such a list in the memory (Figure 7.1.1) we have the following mapping in the heap functions n, p:

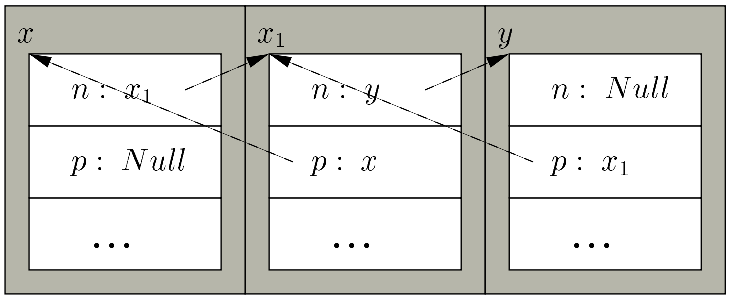


Figure 7.1: Doubly linked list example

n	p
$\dots$	$\dots$
$x \mapsto x_1$	$x \mapsto Null$
$x_1 \mapsto y$	$x_1 \mapsto x$
$y \mapsto Null$	$y \mapsto x_1$
$\dots$	$\dots$

Therefore, the abstract version of the list which satisfies the relation is $ps = [x, x_1, y]$.

Doubly linked lists are used as a basic data structure to create strings, tables, and sequences of objects.

Properties of a Doubly Linked List Here we present some useful properties of the doubly linked list abstraction needed for the correctness proof. Properties of this data structure and procedures to operate with it are verified and the results are presented in [60].

Lemma 7.1.2 Let reference p point to a non-empty list in the memory, then it is not equal to the null pointer and equal to the head of the list.

$$dList(p, nxt, prv, q, dl) \wedge dl \neq [] \Longrightarrow p \neq Null \wedge p = hd(dl)$$

Lemma 7.1.3 Let some reference be not the last element at position i in a non-empty list in the memory, then the following holds:

$$dList(p, nxt, prv, q, dl) \wedge dl \neq [] \wedge i < |dl| \wedge dl_i \neq last(dl) \Longrightarrow$$
$$nxt(dl_i) \neq Null \wedge nxt(dl_i) = dl_{i+1}$$

Lemma 7.1.4 Let a reference be the last element in a non-empty list in the memory, then its component, which points to the next element, is the null pointer:

$$dList(p, nxt, prv, q, dl) \wedge dl \neq [] \Longrightarrow nxt(last(dl)) = Null$$

Lemma 7.1.5 All elements of a doubly linked list are distinct:

$$dList(p, nxt, prv, q, dl) \Longrightarrow distinct(dl)$$

Lemma 7.1.6 There can not be two different lists pointed by the same reference:

$$dList(p, nxt, prv, q, dl) \wedge dList(p, nxt, prv, q, dl') \Longrightarrow dl = dl'$$

Lemma 7.1.7 Let $p \in Ref$ point to some reference list organized by heap functions nxt and prv. If the value of the heap functions is not changed for all list elements, then p points to the same list with respect to the new heap functions nxt', prv'.

$$dList(p, nxt, prv, q, dl) \wedge \forall x \in_* dl.\ nxt'(x) = nxt(x) \wedge prv'(x) = prv(x) \Longrightarrow$$
$$dList(p, nxt', prv', q, dl)$$

7.1.2 Tree

In this section we present another basic abstraction we use in the program. Tree structures are used to organize expression and statement representation in the form of syntax trees. Note, that the syntax trees we use are binary, i.e. they have at most two subtrees.

Definition 7.1.8 Let t be a type. Then T_t is an abstract type for a **tree** of type t (i.e. with nodes of type t) with two constructors:

$$T_t = Tip \mid Node(T_t, t, T_t)$$

Tip is a constructor for the empty tree and $Node$ represents any non empty tree, which is constructed recursively.

Definition 7.1.9 Let tr be a tree. Function tr_mem checks whether $x \in t$ is a node in tree $tr \in T_t$:

$$tr\_mem(t, x) = \begin{cases} False & t = Tip \\ n = x \vee tr\_mem(t_1, x) \vee tr\_mem(t_2, x) & t = Node(t_1, n, t_2) \end{cases}$$

We denote the function application $tr\_mem(tr, x)$ by $x \in_T tr$.

Definition 7.1.10 Let t be a non empty tree. We define $lt(t)$ and $rt(t)$ as access functions yielding the left and right subtrees respectively.

Definition 7.1.11 Let t be a tree. Function $t2s$ transforms a tree to the set of its elements:

$$t2s(t) = \begin{cases} \{\} & t = Tip \\ t2s(t_1) \cup \{n\} \cup t2s(t_2) & t = Node(t_1, n, t_2) \end{cases}$$

We denote the function application $t2s(t)$ by $\{t\}$.

Definition 7.1.12 Let $tr \in T_t$ be a tree and $x \in t$. Then function

$$subtree(x, tr) = \begin{cases} Tip & \text{if } t = Tip \\ tr & \text{if } t = Node(t_1, a, t_2) \wedge x = a \\ subtree(x, t_1) & \text{if } t = Node(t_1, a, t_2) \wedge x \in t_1 \\ subtree(x, t_2) & \text{otherwise} \end{cases}$$

yields the first found subtree of tr with root x. For $x \notin_T tr$ function tr_mem returns the empty tree.

Definition 7.1.13 Let $tr \in T_t$ be a tree. Then function

$$tree\_size(tr) = \begin{cases} 0 & \text{if } t = Tip \\ 1 + tree\_size(t_1) + tree\_size(t_2) & \text{if } t = Node(t_1, a, t_2) \end{cases}$$

computes the number of tree nodes. We denote $tree\_size(tr)$ with $|tr|_T$.

The C0 data structure to build a tree contains two pointers: to the left and to the right subtree, where the null pointer is equivalent to the empty tree:

$$struct\ tree = \{left : tree*;\ right : tree*; ...\}$$

An abstraction function for trees is defined in the following way:

Definition 7.1.14 Let $p \in Ref$ be a reference, $l, r \in Ref \rightarrow Ref$ be heap functions, and $tr \in T_{Ref}$ be a tree of references, then

$$Tree(p, l, r, tr) \equiv$$
$$\begin{cases} p = Null & \text{if } tr = Tip \\ p \neq Null \wedge p = n \wedge Tree(l(p), l, r, t_1) \wedge & \\ Tree(r(p), l, r, t_2) \wedge \{t_1\} \cap \{t_2\} = \varnothing & \text{if } tr = Node(t_1, n, t_2) \end{cases}$$

defines that there exists a **tree** of references tr, starting with reference p and built by means of heap functions l and r.

A tree structure without loops is provided by the condition $\{t_1\} \cap \{t_2\} = \varnothing$.

Analogously to the *dList* abstraction function we can show the uniqueness of the tree pointed by some reference.

Lemma 7.1.15 $Tree(p, l, r, tr) \wedge Tree(p, l, r, tr_1) \Longrightarrow tr = tr_1$

7.2 Complex Data Types

It is easy to notice that abstraction functions for the types presented in the previous section only require two heap functions from a state space as parameters. Of course, for more complicated types the number of heap functions provided to an abstraction function can be counted by tens. To avoid a huge number of parameters writing definitions we supply the whole state instead of several heap functions from it.

7.3 Strings

In the compiler implementation we use strings to keep names of procedures, variables, and structure components. Although we use the string abstraction, some additional functions on it, and lemmas about strings in our correctness prove, we are not interested in the inner design of the string abstraction since we do not modify any strings during the compiler execution. So, we do not give here the complete definition of it, we only notice that the string data structure is based on a doubly linked list. The complete definitions and the verification of procedures operating with strings are presented in [61,62]. The C0 type *c0_string* is a pointer to the data structure used to represent strings. For the same reasons that given above, we do not provide the implementation details.

As it uses six heap functions as parameters, we provide it with a state as was proposed above.

Definition 7.3.1 Let state space Σ includes all the heap functions used to construct the string data structure. Let $\sigma \in \Sigma$ be a state, p be a reference and s be an abstract string, then relation $String(p, \sigma, s)$ stays that p represents the **string** s in state σ.
$String\_cont(p, \sigma) \equiv \varepsilon s.\ String(p, \sigma, s)$ returns some string s that turns the string abstraction function $String(p, \sigma, s)$ to *true* by means of the choice operator.
$is\_String(p, \sigma) \equiv \exists s.\ String(p, \sigma, s)$ returns true if there exists a string starting with reference p.

There are some properties of the abstraction function we will use:

Lemma 7.3.2 There can not be two different strings pointed by the same reference:

$$String(p, \sigma, s) \wedge String(p, \sigma, s') \Longrightarrow s = s'$$

The proof of the lemma is based on the fact that there is no different doubly linked lists starting at the same reference (by Lemma 7.1.6).

Lemma 7.3.3 $String(p, \sigma, s) \Longrightarrow String\_cont(p, \sigma) = s$

The lemma is proven by exploiting the uniqueness of the string starting with the same reference (Lemma 7.3.2), so the only string that can be returned by the choice operator is s.

7.4 Type Table

In this section we consider data types that are used to represent type information of a program.

7.4.1 C0 Data Structure

The data structures which are used to implement the type environment (or in the implementation - type table) are given in Figure 7.2.

$typeT = struct\{$	$id : nat$	type identifier
	$eltN : nat$	number of array elements
	$eltTy : typeT*$	pointer to the type of array elements
	$dsp : nat$	displacement factor of an element in an array
	$strCmp : var\_list*$	pointer to a list of components for struct type
	$ptrTy : typeT*$	the type of elements referenced by a pointer
	$align : nat$	alignment of a type
	$asize : nat\}$	allocated size of a type
$typeT\_list = struct\{$	$nxt : typeT\_list*$	
	$prv : typeT\_list*$	
	$cnt : typeT*\}$	type entry
$varT = struct\{$	$nm : c0\_string$	pointer to a string, which is variable/component name
	$ty : typeT*$	pointer to the entry in type table, which is the type of the variable
	$gl : bool$	global/local variable flag
	$dsp : nat\}$	an offset inside a frame
$varT\_list = struct\{$	$nxt : varT\_list*$	
	$prv : varT\_list*$	
	$cnt : varT*\}$	variable

Figure 7.2: Type table data types

constant	value	constant	value
INT	0	UINT	1
CHAR	2	BOOL	3
PTR	4	ARR	5
STR	6	ELT_NUM	4

Table 7.1: Type identifier constants

The type identifier field *id* can take as value one of the constants given in Table 7.1, whose names are logically connected with the kind of types they code.

The data type *varT* is used to represent variables as well as components of a structure. Complex types are constructed by means of pointers to the subtypes: for an array the type of its elements can be accessed via the field *eltTy*; for a pointer the pointed type - by *ptrTy*; for a structure type the components list is given by the field *strCmp*, where each element points to its type by the *ty* field.

In order to keep all the types as one solid structure, which we call the type table, we connect the data representing them by means of a doubly linked list (data structure *typeT_list*). In the same way we use a doubly linked list to create a variable list or a list of structure components (data structure *varT_list*).

Alignment and allocated size of types are not known before the compiler starts running. For efficiency reasons these values can be computed once and stored in the type structure (fields *align* and *asize* in data structure *typeT*), so they do not need to be recomputed every time they are used (that is very important for complex data types with large induction depth). Analogously, the offset of a variable inside the frame (or displacement of the component inside the structure)

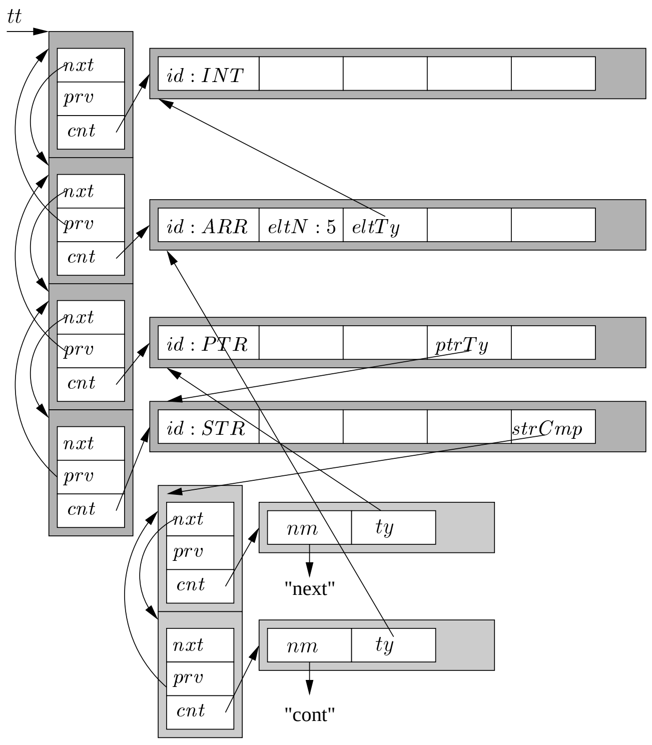


Figure 7.3: An example type table

also can be computed once and kept in the structure of the variable (field *dsp* in data structure *varT*).

The meanings of the other fields are collected in the picture.

The implementation of an example type table is depicted in Figure 7.3. This type table includes an elementary integer type, an array type with 5 elements of the integer type, and a structure type with two components. This structure type represents a linked list: the first component of the structure is the content of the integer type; the second one - the pointer to the next element in the list. The pointer type to this structure is used in the construction of the type and is also included into the type table.

7.4.2 Translation to Abstract Type

Heap functions On the base of the data types presented above the corresponding heap functions for the state space are generated.

Recall that the verification environment requires that different names have to be used to select the components of different structure types. This can be achieved by preprocessing C0 programs and generating names corresponding to the rules we mentioned in Section 4.3.2. As by these rules for any field f from a structure type with name s a heap function with name s_f is generated, it can result in long names which are inconvenient to be used in the following description, e.g. *typeT_list_nxt*.

For the current use we assume the following strategy to choose names for heap functions. We want to keep names short, but taking just the name of a field will cause name clashes in the state space (since e.g. we have several structure types including field *nxt* in the compiler implementation). In order to avoid name clashes and emphasize for which data structure a heap function was actually generated we denote any heap function h, which is generated from structures *typeT* and *typeT_list* with h_t; and by h_v any, which is generated from structures *varT* and *varT_list*. The same approach will be used to name the heap functions for the other compiler structures described below. We will chose a subscript that can be associated with the name of a structure type.

Type names The first gap between the implementation and specification is the absence of type names in the former one. The role of these take pointers to the type table entries. To make use of this fact we introduce the function

$$ref2nm : Ref \to nm_{\mathcal{T}}$$

converting references to names. The only property we need about this function is the injectivity, which we define by a predicate $inj(ref2nm)$, to make sure we map different pointers to different names.

Structure components According to the data type for structure components (*varT* in Figure 7.2), heap function nm_v provides a pointer to a string, which is the component name, whereas ty_v provides a pointer to a type and hence, the type name.

Definition 7.4.1 Let σ be a state, p be a reference, and $cn \in nm_c \times nm_{\mathcal{T}}$ be a pair of a component name and the name of its type. Then the abstraction function

$$Cmp(p, \sigma, cn) \equiv String(\mathsf{nm}_v(p), \sigma, fst(cn)) \wedge snd(cn) = ref2nm(\mathsf{ty}_v(p))$$

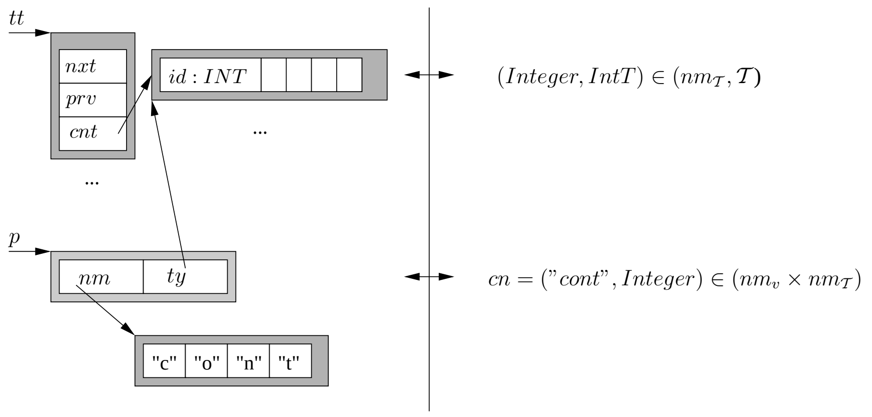
states that reference p is a pointer to a **structure component** in state σ.

The example of the relation between the instances of structure *varT* and abstract data type $nm_c \times nm_{\mathcal{T}}$ is presented in Figure 7.4.

The data structure for a component list is based on the doubly linked list data structure, where the content of each list element is a pointer to a structure component. We define all component names to be unique inside one structure component list and there are no two different list elements pointing to the same component.

Definition 7.4.2 Let σ be a state, p be a reference, dl be a list of references, and $vl \in (nm_c \times nm_{\mathcal{T}})^*$ be an abstract list of structure components. Then

$$\begin{aligned} CmpList(p, \sigma, dl, vl) \equiv\ & (\exists lst.\ dList(p, \mathsf{nxt}_v, \mathsf{prv}_v, lst, dl)) \wedge |dl| = |vl| \wedge \\ & distinct(map(\mathsf{var}_v, dl)) \wedge unique(vl) \wedge \\ & (\forall i < |dl|.\ Cmp(\mathsf{var}_v(dl_i), \sigma, vl_i)) \end{aligned}$$



tt
nxt
prv
cnt
id : INT
...
...
p
nm
ty
"c" "o" "n" "t"
(Integer, IntT) ∈ (nm_T, T)
cn = ("cont", Integer) ∈ (nm_v × nm_T)

Figure 7.4: An example of concrete-abstract relation for a structure component

abstracts p and state σ to lists dl and vl. There are several related functions:
$is\_CmpList(p, \sigma) \equiv \exists\ dl\ vl.\ CmpList(p, \sigma, dl, vl)$ is used to define a pointer to be a pointer to a list of structure components.
$CmpList\_cont(p, \sigma) \equiv \varepsilon\ vl.\ (\exists dl.\ CmpList(p, \sigma, dl, vl))$ is used to get an abstract component list satisfying the relation.
$CmpList\_ref(p, \sigma) \equiv \varepsilon\ dl.\ (\exists vl.\ CmpList(p, \sigma, dl, vl))$ provides a reference list satisfying the relation.

Type abstraction The abstraction from data structures of the type table to the abstract type environment is not straightforward. From the definition of the abstract type it is clear that the type environment is a list of trees, whereas the implementation of it is a graph with pointers in the role of edges. Thus, contrary to the type environment, the type table data structure is defined by non-trivial recursion and it is not simple to map that directly to the type $(nm_{\mathcal{T}} \times \mathcal{T})^*$.

To avoid the complicated direct conversion, we introduce an intermediate type representation *type*, such that its instance $t \in type$ is the following record

$$t = (name, id, eln, elt, pt, cmp),$$

where the components are:

- $t.name \in nm_{\mathcal{T}}$ - type name
- $t.id \in \mathbb{N}$ - type identifier with the same values as in the implementation (Table 7.1)
- $t.eln \in \mathbb{N}$ - number of elements if t represents an array
- $t.elt \in nm_{\mathcal{T}}$ - name of the elements' type if t represents an array
- $t.pt \in nm_{\mathcal{T}}$ - name of the target type if t represents a pointer
- $t.cmp \in (nm_c \times nm_{\mathcal{T}})^*$ - list of components in the case t is a structure type

$$\begin{aligned} Id &= \{INT, UINT, CHAR, BOOL, PTR, ARR, STR\} \\ ElId &= \{INT, UINT, CHAR, BOOL\} \end{aligned}$$

Table 7.2: Identifier sets

The conversion from the implementation data structure to this type is straightforward. The conversion function $ref2type : Ref \to type$ is defined below:

Definition 7.4.3 Let σ be a state, $p \in Ref$ be a reference. Then we define its conversion to $t \in type$ as the following:

$$\begin{aligned} ref2type(p, \sigma) \quad = t \text{ s.t. } \; & t.name = ref2nm(p) \wedge t.id = \mathsf{id}_t(p) \wedge \\ & t.eln = \mathsf{eltN}_t(p) \wedge \\ & t.elt = ref2nm(\mathsf{eltTy}_t(p)) \wedge \\ & t.pt = ref2nm(\mathsf{ptrTy}_t(p)) \wedge \\ & t.cmp = CmpList\_cont(\mathsf{strCmp}_t(p), \sigma) \end{aligned}$$

So, all pointers to the type table entries are converted to type names; numbers are not converted; structure components are obtained from the reference by the choice operator (Definition 7.4.2). It is clear, that we get the whole type table by consequentive conversion of each entry, i.e. we end up with an abstract object of type $type^*$.

Table 7.2 defines some sets of identifiers we use in the definitions below.

To complete the abstraction we need to convert any instance of $type$ to the target type $\mathcal{T}$. In this case the conversion depends not only on the entry which is being converted but also on the whole type table (since for conversion of complex types we need to find their subtypes somewhere in the typetable).

Definition 7.4.4 Let $tn \in nm_{\mathcal{T}}$ be a type name and $tt \in type^*$ be a type table in the intermediate representation. Then function

$$nm2type(tn, tt) = \varepsilon t.\ t \in_* tt \wedge t.name = tn$$

returns from the type table a type with the given name tn.

To keep the recursive definition of the conversion simple to work with (in Isabelle/HOL) we define it using trivial recursion, i.e. for any list we apply recursive function call only to its tail. Thus, if the conversion of any type $t = tt_i$ depends on other types (as in the case of array or structure) they are needed to be placed in the suffix $(tt_{i+1}, \ldots, last(tt))$ of the type table tt (so in the tail of the list starting with tt_i) to keep the recursion trivial.

Definition 7.4.5 Let $n \in \mathbb{N}$ be a number. Then function

$$i2ty(n) = \begin{cases} Bool & \text{if } n = BOOL \\ Int & \text{if } n = INT \\ Usgn & \text{if } n = UINT \\ Char & \text{if } n = CHAR \\ \epsilon & \text{otherwise} \end{cases}$$

converts the identifier value of an elementary type to the corresponding abstract type.

Definition 7.4.6 Let $t \in type$ and $tt \in type^*$ be a type and a non-empty type table in the intermediate representation respectively. Then t is converted to an object from $\mathcal{T}$ as given below:
case a): if $tt = (x, xs) \wedge t = x$

$$type2ty(t, tt) = \begin{cases} i2ty(t.id) & \text{if } t.id \in ElId \\ Ptr(t.pt) & \text{if } t.id = PTR \\ Arr(t.eln, type2ty(nm2type(t.elt, tt), xs)) & \text{if } t.id = ARR \\ Str(f(t.cmp)) & \text{if } t.id = STR \end{cases}$$

where for all $0 \leq i < |t.cmp| - 1$ i-th component of the structure is converted analogously to array elements:

$$f(t.cmp)_i := (fst(t.cmp_i), type2ty(nm2type(snd(t.cmp_i), tt)))$$

case b): if $tt = (x, xs) \wedge t \neq x$

$$type2ty(t, tt) = type2ty(t, xs)$$

Translation of every entry $0 \leq i < |tt| - 1$ of the type table gives the type environment:

$$type\_env(tt)_i := (tt_i.name, type2ty(tt_i, tt))$$

where each element of tt is mapped to a pair of its name and the type from $\mathcal{T}$ to which it will be converted.

The second case of the definition represent a principle of the translation. Translation of elementary types and pointers is straightforward. When translating arrays and structures we need to translate all the type names used to define them (i.e. for elements and components types) to types (from $\mathcal{T}$) as well. Thus, in the case of arrays we first use the name of the type of elements $x.elt$ to find the type itself in the type table and then convert the found type recursively. For the structure case we need to map each pair from the component list to a pair, where the first pair element (component name) stays unchanged, and conversion for the second pair element from the type name to the type from $\mathcal{T}$ is done analogously to the array case.

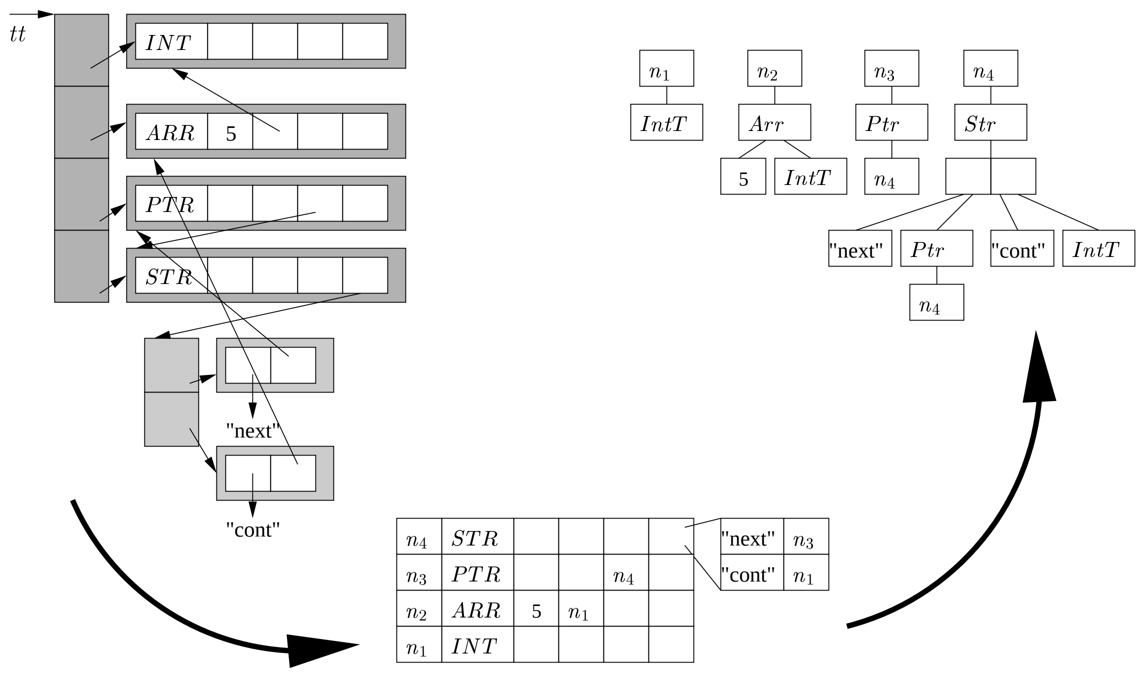


Figure 7.5: Relation between concrete and abstract type table

Now we introduce a wrapper function which combines these two definitions:

$$ref2ty(p, \sigma, tt) = type2ty(ref2type(p, \sigma), tt))$$

Let us now consider several restrictions that hold for the type table and make the translation presented above work. We construct the type table data in the way that we get every type only once. Formally, it is defined in the following way:

Definition 7.4.7 Let $tt \in type^*$ be a type table. Uniqueness of types is defined by the predicate:

$$\begin{aligned} type\_dist(tt) \equiv\ & \forall x, y \in \{tt\}.\ x \neq y \longrightarrow \\ & (x.id \in ElId \longrightarrow x.id \neq y.id) \wedge \\ & (x.id = PTR \longrightarrow x.pt \neq y.pt) \wedge \\ & (x.id = ARR \longrightarrow x.elt \neq y.elt \vee x.eln \neq y.eln) \wedge \\ & (t.id = STR \longrightarrow x.cmp \neq y.cmp) \end{aligned}$$

In the correct type table the type names are also distinct.

Definition 7.4.8 Let $tt \in type^*$ be a type table. The predicate *name_dist* defines the uniqueness of the type names:

$$name\_dist(tt) \equiv \forall x, y \in_* tt.\ x \neq y \longrightarrow x.name \neq y.name$$

While constructing the type table we also keep it well defined in the following sense: if there is a complex type, then the type, which it is based on, must be in the type table. Moreover, we sort the type table so, that types used for the construction of array and structure types are placed below in the type table (except for pointer types). We also need to specify some more obvious things as: identifiers can be only of the presented before values, arrays contain at least one element, pointer types can not depend on itself, there is no two components with the same name inside a structure type.

Definition 7.4.9 Let $tt \in type^*$ be a type table. We say that tt is well-defined if:

$$
\begin{aligned}
&well\_defined(tt) \equiv \\
&\forall x \in_* tt.\ x.id \in Id \wedge \\
&(x.id = ARR \longrightarrow \\
&\quad \exists y.\ y \in_* sfx((\lambda z.z = x), tt) \wedge y.name = x.elt \wedge 0 < x.eln) \\
&(x.id = STR \longrightarrow unique(x.cmp) \wedge \\
&\quad (\forall k \in_* map(snd, x.cmp).\ (\exists y.\ y \in_* sfx((\lambda z.z = x), tt) \wedge y.name = k))) \\
&(x.id = PTR \longrightarrow x.pt \in_* map((\lambda x.\ x.name), tt) \wedge x.pt \neq x.name))
\end{aligned}
$$

Thus, as it was mentioned before, types used for the construction of complex type t are placed in the suffix $sfx((\lambda z.z = t), tt)$ of the type table after t itself.

Type table abstraction function We convert the type table structure kept in the program memory to abstract tt of type $type^*$ and not to the type environment, since it is more natural for the implementation and easier to use in proofs. Moreover, it can be easily converted to the type environment by Definition 7.4.6. We also need to notice, that the type table data structure is needed to be abstracted to reversed tt instead of tt itself. It is necessary, since in the implementation data have the reversed direction of recursion: we start with the simplest types and end with the most complex one. This reversion is connected with the difference between performing an operation on a list when programming in C0 and any functional language (in our case in Isabelle/HOL, which is close to ML). In most cases we start list traversing from the list beginning in C0 whereas doing recursion on abstract lists the first actually processed element will be the last element of the list (see Figure 8.2).

Of course, we can make both models closer by changing the implementation of such operations on lists in the way they start traversing with the last element, adapting the programming style to the verification process. In this work we rather want to show the possibility of the verification of an existing implementation as it is without making any simplifications.

Definition 7.4.10 Let σ be a state, $p \in Ref$ be a reference, $dl \in Ref^*$ be a reference list, and $tt \in type^*$ be a list of types. Then the abstraction function

$Typetable(p, \sigma, dl, tt) =$

(1) $(\exists lst.\ dList(p, \mathsf{nxt}_t, \mathsf{prv}_t, lst, dl)) \wedge |dl| = |tt| \wedge |ElId| < |tt| \wedge$
$name\_dist(tt) \wedge type\_dist(tt) \wedge well\_defined(tt) \wedge$
$(\forall i < |dl|.\ ref2type(\mathsf{cnt}_t(dl_i), \sigma) = rev(tt)_i) \wedge$

(2) $distinct(map(\mathsf{cnt}_t, dl)) \wedge (\forall x \in \{dl\}.\ \mathsf{cnt}_t(x) \neq Null) \wedge$
$(\forall x \in_* map(\mathsf{cnt}_t, dl).\ \mathsf{id}_t(x) = STR \longrightarrow$
$\mathsf{strCmp}_t(x) \neq Null \wedge is\_CmpList(\mathsf{strCmp}_t(x), \sigma)\ ) \wedge$
$(\forall x, y \in_* map(\mathsf{cnt}_t, dl).\ x \neq y \wedge \mathsf{id}_t(x) = \mathsf{id}_t(y) = STR \longrightarrow$
$\{map(\mathsf{cnt}_v, CmpList\_ref(x, \sigma))\} \cap \{map(\mathsf{cnt}_v, CmpList\_ref(y, \sigma))\} = \varnothing)$

sets the relation between reference p and abstract objects dl and tt, so that p points to the **type table** in the memory in state σ. Also we define two additional functions to access abstract components through reference p by means of the choice operator (analogously to Definition 7.4.2) $Typetable\_ref(p, \sigma)$ and $Typetable\_cont(p, \sigma)$ such that

$$Typetable(p, \sigma, dl, tt) \Longrightarrow Typetable\_ref(p, \sigma) = dl \wedge Typetable\_cont(p, \sigma) = tt.$$

The definition includes two parts: (1) is the functional part that says the type table implementation is a list, where each element contains a pointer to a type entry; types are distinct and well defined; and the i-th entry corresponds to an abstract type $rev(tt)_i$. Part (2) of the definition includes technical information on pointers. Thus, e.g. for any structure type there is a list of structure components; lists of structure components inside the type table are disjoint; pointers to the type table entries must be disjoint, i.e. there is no two different pointers pointing to the same entry, etc.

7.4.3 Lemmata

In this section we give some lemmata for the definitions above which are used in the correctness proof in Chapter 8. The following lemma is an example of the approach we use to prove lemmas about functions defined on the base of the choice operator.

Lemma 7.4.11 For any type t from the type table tt

$$name\_dist(tt) \wedge t \in_* tt \Longrightarrow nm2type(t.name, tt) = t$$

Proof: Since $nm2type$ is defined through the choice operator, then according to Lemma 1.4.21 we need to show that i) t satisfies choice condition $\lambda x. x \in tt \wedge x.name = t.name$, which is obvious; ii) for any x such that $x \in tt \wedge x.name = t.name$ the equality $x = t$ holds, which is true since by definition of $name\_dist(tt)$ we have distinct names for all type table entries. $\square$

Lemma 7.4.12 For type $t \in type$ from a non-empty type table tt, such that $t.id = INT, BOOL, CHAR, UINT, PTR$, the corresponding representation in $\mathcal{T}$ is $T = IntT, BoolT, CharT, UsgnT, Ptr(t.pt)$ respectively, i.e. $type2ty(t, tt) = T$
Proof: is simple by induction on tt. □

Lemma 7.4.13 Let $tt \in type^*$ be a non-empty type table including type t s.t. $t.id = ARR$. Then conversion to $\mathcal{T}$ is the following:

$$\begin{aligned}
&tt \neq [] \wedge t \in_* tt \wedge t.id = ARR \wedge name\_dist(tt) \wedge \\
&\exists y.\ y \in_* sfx((\lambda z.z = t), tt) \wedge y.name = t.elt \\
&\Longrightarrow \\
&type2ty(t, tt) = Arr(t.eln, type2ty(nm2type(t.elt), tt), tt),
\end{aligned}$$

i.e. if the type table contains the type of the array elements, then it does not matter whether the translation is done on the part of table $sfx((\lambda z.z = t), tt)$ (which we can get directly from definition of $type2ty$) or on the whole type table.

Proof: by induction on tt.
i) Induction base $tt = []$ does not satisfy the assumptions.
ii) Induction step $tt = (x, xs)$. Let us consider the case $t = x$. Since t is the list head, $sfx((\lambda z.z = t), tt)$ is equal to the list tail xs. Hence, the type of array elements $nm2type(t.elt, tt)$, which is equal to some existing type y by Lemma 7.4.11, belongs to xs. Moreover, y is not equal to x since types are distinct (by $name\_dist(tt)$). By the second case of $type2ty$ definition we get

$$type2ty(t, tt) = Arr(t.eln, type2ty(nm2type(y, tt), xs))\ (*)$$

Also we can show that $type2ty(nm2type(y, tt), xs) = type2ty((nm2type(y, tt), tt)$ by the second case of the definition $type2ty$, since $y \neq x$. Rewriting (*) with the last equation we get the claim.
Case $t \in xs$ is proven by the second case of the function definition, induction hypothesis, and similar argumentation about the type of elements y. □

An analogous lemma needs to be shown for structure types.

Lemma 7.4.14 Let tt be a non-empty type table including type t s.t. $t.id = STR$. Then conversion to $\mathcal{T}$ is the following:

$$\begin{aligned}
&tt \neq [] \wedge t \in_* tt \wedge t.id = STR \wedge name\_dist(tt) \wedge \\
&(\forall k \in_* map(snd, x.cmp).\ \exists y.\ y \in_* sfx((\lambda z.z = t), tt) \wedge y.name = k) \wedge \\
&\forall i < |t.cmp|.\ f(t.cmp_i) = (fst(t.cmp_i), type2ty(nm2type(snd(t.cmp_i), tt), tt)) \\
&\Longrightarrow \\
&type2ty(t, tt) = Str(f(t.cmp))
\end{aligned}$$

Proof: The argumentation for the type of every structure element is similar to the argumentation about the type of array elements in Lemma 7.4.13.

Lemma 7.4.15 Let reference p point to the type table. Then conversion of any type from tt to $\mathcal{T}$ is a valid type.

$$\forall t.\ Typetable(p, \sigma, dl, tt) \wedge t \in_* tt \Longrightarrow valid_{\mathcal{T}}(type\_env(tt), type2ty(t, tt))$$

Proof: by structural induction on type x equal to $type2ty(t, tt)$.

- For the elementary types and $NullT$ the proof is trivial, since they are valid by Definition 2.1.2.
- In case $x = Arr(n, t')$ according to Defintion 2.1.2 we need to show the validity of type t' and $0 < n$. As an additional lemma we can prove by induction, that $type2ty(t, tt) = Arr(n, t') \Longrightarrow t.id = ARR$. Using that, we get $t.elt > 0$ from $well\_defined(tt)$ (included in $Typetable$) and as by Lemma 7.4.13 $n = t.eln$, the second part of the claim is shown. Moreover, from $well\_defined(tt)$ we have that there exists $y \in \{tt\}$ such that $y.name = t.elt$.

  As the induction hypothesis we have that

  $\forall t.Typetable(p, \sigma, dl, tt) \wedge t \in \{tt\} \wedge t' = type2ty(t, tt) \Longrightarrow valid_{\mathcal{T}}(type\_env(tt), t')$

  Instantiating it with $t = y$ and applying the results of Lemma 7.4.13 to t' we have to show

  $$type2ty(nm2type(t.elt, tt), tt) = type2ty(y, tt)$$

  to finish the claim. This is true by Lemma 7.4.11, providing that names in tt are distinct.
- The validity of pointer types can be shown from predicate $well\_defined(tt)$, lemma $type2ty(t, tt) = Ptr(n) \Longrightarrow t.id = PTR$, and the fact that names in the type environment $tenv\_env(tt)$ are equal to names in tt.
- The case $x = Str(cs)$ can be shown analogously considering component types. Additionally we need to prove $unique(t.cmp) \longrightarrow unique(map(fst, cs))$. That is obviously true by the definition of conversion $type2ty$ for the structure case. □

7.5 Variables

C0 Data Type The implementation of a variable list is based on the doubly linked list structure, where each list element includes a pointer to a data structure representing a variable. The C0 data type used to code a variable is the as the one used to represent components of a structure type (Figure 7.2).

Abstraction function In contrast to the type table which is fully defined by itself, a variable (or variable list) depends on the type table, so we provide a pointer to it as a parameter of the abstraction function.

Definition 7.5.1 Let σ be a state, $p \in Ref$ be a reference, $v \in (nm_v \times \mathcal{T})$ be an abstract variable declaration, and reference t be a pointer to the type table. Then relation

$$\begin{aligned} Var(p, t, \sigma, v) \quad \equiv \quad & String(\mathsf{nm}_v(p), \sigma, fst(v)) \wedge \\ & \mathsf{ty}_v(p) \in_* map(\mathsf{cnt}_t, Typetable\_ref(t, \sigma)) \wedge \\ & snd(v) = ref2ty(\mathsf{ty}_v(p), \sigma, Typetable\_cont(t, \sigma)) \end{aligned}$$

describes p as a reference to a variable declaration in state σ.

Thus, by the abstraction function we state $\mathsf{nm}_v(p)$ to be a pointer to the variable name; pointer to its type $\mathsf{ty}_v(p)$ must be pointing to an entry in the type table and conversion of this entry to an abstract type is the type of the represented variable.

Analogously to a list of structure components we define an abstraction function for a variable list (i.e. symbol table).

Definition 7.5.2 Let σ be a state, $p \in \mathit{Ref}$ be a reference, reference t be a pointer to the type table, $dl \in \mathit{Ref}^*$ be a reference list, and vl be an abstract symbol table. Then relation

$$\begin{aligned} \mathit{VarList}(p, t, \sigma, dl, vl) \quad \equiv \quad & (\exists lst.\ \mathit{dList}(p, \mathsf{nxt}_v, \mathsf{prv}_v, lst, dl)) \wedge |dl| = |vl| \wedge \\ & \mathit{unique}(vl) \wedge \mathit{distinct}(\mathit{map}(\mathsf{cnt}_v, dl)) \wedge \\ & \forall i < |dl|.\ \mathit{Var}(\mathsf{cnt}_v(dl_i), t, \sigma, vl_i) \end{aligned}$$

states that p points to a data structure based on list dl implementing a symbol table vl in the memory in state σ. There are functions to get both abstract components of the relation from pointer p via the choice operator: $\mathit{VarList\_ref}(p, \sigma)$ and $\mathit{VarList\_cont}(p, \sigma)$. Thus, the following can be shown (based on the uniqueness of the list representation):

$$\mathit{VarList}(p, t, \sigma, dl, vl) \Longrightarrow \mathit{VarList\_ref}(p, \sigma) = dl \wedge \mathit{VarList\_cont}(p, \sigma) = vl$$

Also, we present two wrapper abstraction functions to distinguish global and local variable lists by exploiting the flag field *glob* in the data structure:

$$\mathit{GVarList}(p, t, \sigma, dl, vl) \equiv \mathit{VarList}(p, t, \sigma, dl, vl) \wedge \forall x \in_* dl.\ \mathsf{glob}_v(\mathsf{var}_v(x))$$

$$\mathit{LVarList}(p, t, \sigma, dl, vl) \equiv \mathit{VarList}(p, t, \sigma, dl, vl) \wedge \forall x \in_* dl.\ \neg\mathsf{glob}_v(\mathsf{var}_v(x))$$

Pointers to variable data are disjoint (described by $\mathit{distinct}(\mathit{map}(\mathsf{var}_v, dl))$) and there are no two variables with the same name in a variable list.

Being connected with the type table, a symbol table represented by this abstraction is always valid.

Lemma 7.5.3 Let t and v be references to the type table and a symbol table respectively. Then

$$\mathit{Typetable}(t, \sigma, tdl, tt) \wedge \mathit{VarList}(v, t, \sigma, vdl, st) \Longrightarrow \mathit{valid}_{ST}(\mathit{type\_env}(tt), st)$$

Proof: The first part of the claim (after expanding definition valid_{ST}) follows directly from the $\mathit{VarList}$ definition. From assumption $\mathit{Typetable}(t, \sigma, tdl, tt)$ we conclude that $\mathit{Typetable\_ref}(t, \sigma)$ returns tdl. Then we have for all $i < |dl|$ that the pointer to the type entry $pt_i = \mathsf{ty}_v(\mathsf{cnt}_v(vdl_i))$ belongs to $\{\mathit{map}(\mathsf{cnt}_t, tdl)\}$. Thus, by Definition 7.4.10 we have $\mathit{ref2type}(pt_i, \sigma) \in_* tt$. Finally Lemma 7.4.15 together with equality $\mathit{Typetable\_cont}(t, \sigma) = tt$ show the goal. $\Box$

$exprT = struct\{$	$id : nat$	expression identifier
	$lt : exprT*$	pointer to the left subexpression
	$rt : exprT*$	pointer to the right subexpression
	$vint : int$	integer/char constant
	$vnat : nat$	unsigned int/bool constant
	$nm : varT*$	variable name/ structure field
	$ty : typeT*\ \}$	pointer to the type table entry

Figure 7.6: Expression C0 data structure

7.6 Expressions

The expression representation in the memory of a program is organized as some kind of syntax tree (see the C0 data structure in Figure 7.6). The identifier field contains a numerical code of an operator according to Table 1.1. Field *ty* defines a type of the expression by pointing to any entry of the type table (set by the preprocessor). Thus, we do not need any additional procedure to define the type of an expression. The null pointer in this field corresponds to the abstract type $NullT$ and combined with identifier equal to 27 (constant) defines a null pointer constant in the program. The constants are coded by combination of fields *vint* and *vnat* and the type of expression. For example, the boolean values are coded with 0 or 1 in the field *vnat* combined with a pointer to the boolean type in the field *ty*. The heap functions in the state space which correspond to the expression data structure are subscribed with index e.

The abstract expression is a tree-like data type and therefore, the mapping from the data structure to abstract type is almost straightforward.

In the data structure binary, lazy and unary operators are coded by the identifier field. For the conversion of any number $n \in \mathbb{N}$ we introduce the following functions connecting the numbers with abstract operators:

$$\begin{aligned} 2binop(n) &\in op_b \cup \epsilon \\ 2lazy(n) &\in op_l \cup \epsilon \\ 2unop(n) &\in op_u \cup \epsilon \end{aligned}$$

To keep the mapping complete we map non-existing identifiers (or the identifiers that code the other operators) to the unknown value ϵ.

In the implementation a constant is also represented by the data structure for expressions in contrast to the abstract side, where constants are coded by a separate abstract type. The conversion from data representing constants to the corresponding object of an abstract data type is done based on the value of the identifier field of structure $exprT$:

Definition 7.6.1 Let $t \in \mathbb{N}$ be a type identifier, $vi \in \mathbb{Z}$ and $vn \in \mathbb{N}$ be possible values of a constant. Then function

$$cst(t, vi, vn) = \begin{cases} Int(vi) & \text{if } t = INT \\ Char(vi) & \text{if } t = CHAR \\ Unsg(vn) & \text{if } t = UINT \\ Bool(True) & \text{if } t = BOOL \wedge vn = 1 \\ Bool(False) & \text{if } t = BOOL \wedge vn = 0 \end{cases}$$

codes these numbers to the abstract constant type.

Definition 7.6.2 Let σ be a state and $p \in Ref$ a pointer to an expression data structure. Then the following function converts p to the corresponding abstract expression.

$$\begin{array}{l} ref2expr(p, \sigma) = \\ \text{let } e_1 = ref2expr(\mathsf{lt}_e(p), \sigma), \\ \quad e_2 = ref2expr(\mathsf{rt}_e(p), \sigma) \text{ in} \\ \begin{cases} LazyBinOp(2lazy(\mathsf{id}_e(p)), e_1, e_2) & \text{if } \mathsf{id}_e < 2 \\ BinOp(2binop(\mathsf{id}_e(p)), e_1, e_2) & \text{if } 2 \leq \mathsf{id}_e < 17 \\ UnOp(2unop(\mathsf{id}_e(p)), e_1) & \text{if } 17 \leq \mathsf{id}_e < 23 \\ Deref(e_1) & \text{if } \mathsf{id}_e = 23 \\ AddrOf(e_1) & \text{if } \mathsf{id}_e = 24 \\ StrAcc(e_1, String\_cont(\mathsf{nm}_v(\mathsf{nm}_e(p)), \sigma)) & \text{if } \mathsf{id}_e = 25 \\ ArrAcc(e_1, e_2) & \text{if } \mathsf{id}_e = 26 \\ Lit(Nil) & \text{if } \mathsf{id}_e = 27 \wedge \mathsf{ty}_e = Null \\ Lit(cst(\mathsf{id}_t(\mathsf{ty}_e(p)), \mathsf{vint}_e(p), \mathsf{vnat}_e(p)) & \text{if } \mathsf{id}_e = 27 \wedge \mathsf{ty}_e \neq Null \\ VarAcc(String\_cont(\mathsf{nm}_v(\mathsf{nm}_e(p)), \sigma)) & \text{if } \mathsf{id}_e = 28 \end{cases} \end{array}$$

Having the conversion defined we can define the abstraction function for the expression data structure:

Definition 7.6.3 Let $p \in Ref$ be a reference to an expression, $t \in Ref$ be a reference to the type table, $g, l \in Ref$ be references to the global symbol table and to some local symbol table respectively in state σ. Then the data structure referenced by p represents abstract expression $expr$ based on reference tree $tr \in T_{Ref}$.

$$\begin{array}{l} Expr(p, t, g, l, \sigma, tr, expr) \equiv \\ Tree(p, \mathsf{lt}_e, \mathsf{rt}_e, tr) \wedge tr \neq Tip \wedge ref2expr(p, \sigma) = expr \wedge \\ \forall x \in_T tr.\ (\mathsf{ty}_e(x) \in_* map(\mathsf{cnt}_t, Typetable\_ref(t, \sigma)) \vee \mathsf{ty}_e(x) = Null) \wedge \\ \qquad \mathsf{id}_e(x) < 29 \wedge \\ \qquad (\mathsf{id}_e(x) = 1 \longrightarrow \ldots) \wedge \ldots \wedge \\ \qquad (\mathsf{id}_e(x) = 28 \longrightarrow \ldots) \end{array}$$

The base of the expression is a non-empty tree. The relation between a pointer and the corresponding abstract expression is done through function $ref2expr$. A pointer in field ty must belong to the type table unless it is the null pointer. The later lines in the definition mention the description of some properties which are different for different kind of expressions. We present these properties in detail for some kind of expressions below.

Analogously to the previous abstraction functions we define the following functions: $is\_Expr(p, t, g, l, \sigma) \in \mathbb{B}$, $Expr\_ref(p, t, g, l, \sigma)$, and $Expr\_cont(p, t, g, l, \sigma)$ with properties:

$$\begin{aligned}&Expr(p, t, g, l, \sigma, tr, expr) \Longrightarrow\\&Expr\_ref(p, t, g, l, \sigma) = tr \wedge Expr\_cont(p, t, g, l, \sigma) = expr,\end{aligned}$$

that can be shown based on the uniqueness of the tree pointed to by p (Lemma 7.1.15) and the property of the choice operator (Lemma 1.4.21).

Similarly to the types described above we introduce an abstraction function for lists of expression, which is used later in the definition of statements: $ExprList(p, t, g, l, \sigma, dl, expr\_list) \in \mathbb{B}$.

Every content reference from reference list $dl \in Ref^*$ pointed to by p is abstracted to an expression using abstraction $Expr$. The remaining parameters are the same as in the previous definition.

Properties of the data structure describe values or relations between pointers of the input data structure. Since the full description of properties for every kind of expression will take a lot of space we only give here several examples, which show of what kind these properties are (the full version can be found in the Isabelle/HOL theories).

Variable Access

$$\begin{aligned}&(\mathsf{id}_e(x) = 28 \longrightarrow\\&\ \mathsf{lt}_e(x) = Null \wedge \mathsf{rt}_e(x) = Null \wedge\\&\ \mathsf{ty}_e(x) \neq Null \wedge \mathsf{ty}_e(x) = \mathsf{ty}_v(\mathsf{nm}_e(x)) \wedge\\&\ (\mathsf{nm}_e(x) \in_* lst \vee \mathsf{nm}_e(x) \in_* gst)),\\&\text{where } gst = map(\mathsf{cnt}_v, VarList\_ref(g, \sigma)),\\&\qquad\quad lst = map(\mathsf{cnt}_v, VarList\_ref(l, \sigma))\end{aligned}$$

It is clear, that a variable access expression does not have any subexpression (hence, empty subtrees in the implementation). The type of expression cannot be the null pointer type and hence, according to the definition of the abstraction function the type pointer shows to some entry in the type table. Moreover, the type of the expression must be equal to the type of the variable we access. The variable we access needs to be placed either in the current local symbol table or in the global symbol table. If there are the same names in both tables we choose the local one, and if we have chosen the global one it implies that there is no variables with the same name in the local symbol table.

Constant

$$\begin{aligned}
&\text{let } T = \mathsf{id}_t(\mathsf{ty}_e(x)) \text{ in}\\
&\mathsf{id}_e(x) = 27 \longrightarrow\\
&\ \mathsf{lt}_e(x) = \mathit{Null} \wedge \mathsf{rt}_e(x) = \mathit{Null} \wedge (\mathsf{ty}_e(x) \neq \mathit{Null} \longrightarrow T \in \mathit{ElId}) \wedge\\
&\ (T = \mathit{INT} \longrightarrow \mathit{is\_valid\_int}(\mathsf{vint}_e(x))) \wedge\\
&\ (T = \mathit{CHAR} \longrightarrow \mathit{is\_valid\_char}(\mathsf{vint}_e(x)))\\
&\ (T = \mathit{UINT} \longrightarrow \mathit{is\_valid\_uint}(\mathsf{vnat}_e(x)))\\
&\ (T = \mathit{BOOL} \longrightarrow \mathsf{vnat}_e(x) = 0 \vee \mathsf{vnat}_e(x) = 1)
\end{aligned}$$

Thus, the data structure coding a constant must not have subtrees, be of an elementary type (see Table 7.1), and include valid values for every type of constant.

Binary Operation ”plus”

$$\begin{aligned}
&\mathsf{id}_e(x) = 13 \longrightarrow\\
&\ \mathsf{lt}_e(x) \neq \mathit{Null} \wedge \mathsf{rt}_e(x) \neq \mathit{Null} \wedge\\
&\ \mathsf{ty}_e(x) \neq \mathit{Null} \wedge \mathsf{ty}_e(\mathsf{lt}_e(x)) = \mathsf{ty}_e(\mathsf{rt}_e(x)) \wedge\\
&\ \mathsf{ty}_e(x) = \mathsf{ty}_e(\mathsf{lt}_e(x)) \wedge \mathsf{id}_t(\mathsf{ty}_e(x)) \in \{\mathit{INT}, \mathit{UINT}\}
\end{aligned}$$

Arithmetic operations must have two subtrees and moreover, both subexpressions and the result of the operation need to be of the same type, which can be either int or unsigned.

Pointer Dereferencing

$$\begin{aligned}
&\text{let } T = \mathsf{ty}_e(\mathsf{lt}_e(x)) \text{ in}\\
&\mathsf{id}_e(x) = 23 \longrightarrow\\
&\ \mathsf{lt}_e(x) \neq \mathit{Null} \wedge \mathsf{rt}_e(x) = \mathit{Null} \wedge\\
&\ \mathsf{ty}_e(x) \neq \mathit{Null} \wedge T \neq \mathit{Null} \wedge \mathsf{ty}_e(p) = \mathsf{ptrTy}_t(T) \wedge \mathsf{id}_t(T) = \mathit{PTR}
\end{aligned}$$

In this case we have only one subtree and the dereferenced expression needs to be of a pointer type.

For any heap function h the non-equality of pointer p to the null pointer needs to be shown explicitly when describing pointer properties, since from existence of $h(p)$ we cannot deduce $p \neq \mathit{Null}$.

In the analogous way we specify relations between all relevant pointers for every kind of expression. To summarize: we mention the number of non-empty subtrees, types of subtrees, and the whole expression (e.g. for comparison operation we have only the boolean type of the resulting expression), restrictions on values and used types.

Moreover, the same properties of several expressions can be combined into groups, e.g. the identity of types of arguments and results for arithmetic expression, absence of null pointers in the fields pointing to subexpressions for all binary operations etc.

These details are used for two main purposes: first, we use them to argue about pointers when verifying any procedure working with the given data structure; second, they are used to show the validity of the abstracted version of an expression. In order to show the latter we first need to prove some auxiliary lemmas about the data structure conversion to an abstract expression.

Lemma 7.6.4

$$\begin{aligned}
&\mathit{Expr}(p,t,g,l,\sigma,tr,\mathit{expr}) \Longrightarrow\\
&(\mathsf{lt}_e(p) \neq \mathit{Null} \longrightarrow \mathit{Expr}(\mathsf{lt}_e(p),t,g,l,\sigma,\mathit{lt}(tr),\mathit{ref2expr}(\mathsf{lt}_e(p),\sigma))) \wedge\\
&(\mathsf{rt}_e(p) \neq \mathit{Null} \longrightarrow \mathit{Expr}(\mathsf{rt}_e(p),t,g,l,\sigma,\mathit{rt}(tr),\mathit{ref2expr}(\mathsf{rt}_e(p),\sigma)))
\end{aligned}$$

Proof: By case distinction on tr.

Case $tr = \mathit{Tip}$ does not satisfy the definition of the Expr abstraction function.

Let the reference tree tr be equal to some node $\mathit{Node}(t_1, a, t_2)$, so $\mathit{lt}(tr) = t_1$ and $\mathit{rt}(tr) = t_2$. The proof for both subtrees is the same. We show that $\mathit{Tree}(\mathsf{lt}_e(p), \mathsf{lt}_e, \mathsf{rt}_e, t_1)$ holds by assumption $\mathit{Tree}(p, \mathsf{lt}_e, \mathsf{rt}_e, tr)$ (expanding the definition of Tree). As all pointer properties hold for all x from tr, they definitely hold for all x from t_1. The relation is valid since $\mathsf{lt}_e(p) \neq \mathit{Null} \longrightarrow t_1 \neq \mathit{Tip}$. Repeating argumentation for $\mathsf{rt}_e(p)$ we finish the proof. $\square$

Let us define a wrapping function getting the type of an expression from the pointer to the data structure.

Definition 7.6.5 Let $p \in \mathit{Ref}$ be a reference in state σ pointing to an expression and $tt \in \mathit{type}^*$ be an intermediate representation of the type table. Then function

$$\mathit{node\_type}(p,\sigma,tt) = \begin{cases} \mathit{NullT} & \text{if } \mathsf{ty}_e(p) = \mathit{Null} \\ \mathit{ref2ty}(\mathsf{ty}_e(p),\sigma,tt) & \text{otherwise} \end{cases}$$

defines the type of the expression.

From the definition of the abstraction function we can derive for every (valid) value of expression identifier the abstract expression, which the data structure is related to, and some of its properties. Let us present here the example for the pointer dereferencing case.

Lemma 7.6.6 Let σ be a state, where reference p points to a pointer dereferencing expression. Then conversion is the following:

$$\begin{aligned}
&\mathit{Typetable}(t,\sigma,tdl,tt) \wedge \mathit{Expr}(p,t,g,l,\sigma,tr,ex) \wedge \mathsf{id}_e(p) = 23 \Longrightarrow\\
&ex = \mathit{Deref}(\mathit{ref2expr}(\mathsf{lt}_e(p),\sigma)) \wedge\\
&\exists tn.\ \mathit{node\_type}(\mathsf{lt}_e(p),\sigma,tt) = \mathit{Ptr}(tn) \wedge\\
&\qquad \mathit{map\_of}(\mathit{type\_env}(tt),tn) = \mathit{node\_type}(p,\sigma,tt)
\end{aligned}$$

Proof: The first conjunct of the claim can be easily shown from assumption Expr by definition of $\mathit{ref2expr}$.

By definition of Expr and properties for $\mathsf{id}_e(p) = 23$ we get that $\mathsf{ty}_e(\mathsf{lt}_e(p))$ is not equal to Null and hence, belongs to the type table entries $\mathit{map}(\mathsf{cnt}_t, \mathit{tdl})$

(since $Typetable(t, \sigma, tdl, tt) \longrightarrow Typetable\_ref(t, \sigma) = tdl$). By $Typetable$ definition we can conclude that exists i such that $\mathsf{ty}_e(\mathsf{lt}_e(p)) = \mathsf{cnt}_t(tdl_i)$ and $T = rev(tt)_i \in_* tt$, where T denotes intermediate type $ref2type(\mathsf{ty}_e(\mathsf{lt}_e(p)), \sigma)$. Therefore, by Lemma 7.4.12 we get that $node\_type(\mathsf{lt}_e(p), \sigma, tt) = type2ty(T, tt)$ is equal to $Ptr(tn)$, with

$$tn = T.pt = ref2nm(\mathsf{ptrTy}_t(\mathsf{ty}_e(\mathsf{lt}_e(p)))),$$

where the last equation is extracted from T notation (expanding definition of $ref2type$).

Moreover, $well\_defined(tt)$ (from $Typetable$) and $T.id = PTR$ (by $Expr$ in case $\mathsf{id}_e(p) = 23$ and definition of $ref2type$) imply $T.pt \in_* map((\lambda x.\ x.name), tt)$. Relation $Expr$ states that pointers $\mathsf{ty}_e(p)$ and $\mathsf{ptrTy}_t(\mathsf{ty}_e(\mathsf{lt}_e(p)))$ are equal (for $\mathsf{id}_e(p) = 23$). Hence, $tn = ref2nm(\mathsf{ty}_e(p))$. The only type with such a name in the intermediate representation of the type table is $ref2type(\mathsf{ty}_e(p), \sigma)$, since the mapping from pointers to names is injective.

According to Definition 7.4.6, the type corresponding to this name in the type environment is

$$map\_of(type\_env(tt), tn) = type2ty(ref2type(\mathsf{ty}_e(p), \sigma), tt),$$

which is equal to $node\_type(p, \sigma, tt)$ by definition of $node\_type$ and the fact, that $\mathsf{ty}_e(p) \neq Null$. $\square$

The complete set of analogous lemmas for all types of expressions can be found in the Isabelle/HOL theories.

The next auxiliary lemma we prove shows the wellfoundness of expression types, which is an important part in the definition of a valid expression.

Lemma 7.6.7 Let σ be a state, where reference p points to an expression, $t \in Ref$ to the type table, and $g, l \in Ref$ point to the global and local symbol table respectively. Then the type of coded expression ex is the same as the converted type from the type pointer:

$$\begin{aligned} &\forall p, ex.\ Typetable(t, \sigma, tdl, tt) \wedge GVarList(g, t, \sigma, gdl, gst) \wedge \\ &\qquad LVarList(l, t, \sigma, ldl, lst) \wedge Expr(p, t, g, l, \sigma, tr, ex) \Longrightarrow \\ &type_{\mathcal{E}}(type\_env(tt), gst, lst, ex) = node\_type(p, \sigma, tt) \end{aligned}$$

Thus, this lemma states that we get the same defined type (not ϵ) by two ways: i) first converting the expression and computing its type afterwards, ii) convert the type of expression presented by field ty in the expression data structure directly (Figure 7.7).

Proof: By induction on tr.

Induction base $tr = Tip$ does not satisfy the $Expr$ abstraction function.

In the induction step $tr = Node(t_1, p, t_2)$ and we need to show that the claim holds for tr if it holds for t_1 and t_2. We make the case distinction on $\mathsf{id}_e(p)$. Let us show some cases in more details.

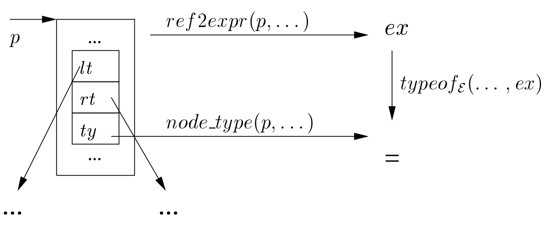


Figure 7.7: Relation between concrete and abstract type table

Binary operation "plus": $\mathsf{id}_e(p) = 13$
By definition of *Expr* and 2*binop* we get an abstract expression

$$ex = BinOp(plus, ref2expr(\mathsf{lt}_e(p)), ref2expr(\mathsf{rt}_e(p), \sigma).$$

Its type $type_\mathcal{E}(type\_env(tt), gst, lst, ex)$ is equal to the type of subexpressions (e.g. left) $type_\mathcal{E}(type\_env(tt), gst, lst, ref2expr(\mathsf{lt}_e(p), \sigma)$. Applying the induction hypothesis to pointer $\mathsf{lt}_e(p)$ and expression $ref2expr(\mathsf{lt}_e(p), \sigma)$ we get

$$\begin{aligned}&Expr(\mathsf{lt}_e(p), t, g, l, \sigma, tr, ref2expr(\mathsf{lt}_e(p), \sigma)) \Longrightarrow\\ &type_\mathcal{E}(type\_env(tt), gst, lst, ref2expr(\mathsf{lt}_e(p), \sigma)) = node\_type(\mathsf{lt}_e(p), \sigma, tt)\end{aligned}$$

(the remaining conditions are implied by the lemmas' assumptions). The left side of the implication is true by the assumptions and Lemma 7.6.4, so to get the claim proved we have to show the equality

$$node\_type(p, \sigma, tt) = node\_type(\mathsf{lt}_e(p), \sigma, tt),$$

which is obviously true, since $\mathsf{ty}_e(p) = \mathsf{ty}_e(\mathsf{lt}_e(p))$ by definition of *Expr*.

Pointer dereferencing: $\mathsf{id}_e(p) = 23$
By Lemma 7.6.6 we have $ex = Deref(ref2expr(\mathsf{lt}_e(p), \sigma))$. It is clear, that the type of ex is the type of the target of the subexpression, which is of some pointer type $Ptr(tn)$. Therefore, the type of ex is the value mapped by the type environment to type name tn, i.e.

$$type_\mathcal{E}(type\_env(tt), gst, lst, ex) = map\_of(type\_env(tt), the\_Ptr(T)), \tag{7.1}$$

where with T we denote $type_\mathcal{E}(type\_env(tt), gst, lst, ref2expr(\mathsf{lt}_e(p), \sigma))$.

Applying the induction hypothesis as in the case above we get

$$T = node\_type(\mathsf{lt}_e(p), \sigma, tt).$$

By Lemma 7.6.6 there exist tn such that $T = Ptr(tn)$ and $map\_of(type\_env(tt), n) = node\_type(p, \sigma, tt)$. As $the\_Ptr(Ptr(tn)) = tn$, the last equation can be substituted in (7.1), what finishes the proof.

All the remaining cases also demand showing correspondence between properties of the data structure in the implementation included in the definition of the abstraction function *Expr* and related properties for the corresponding abstract expression. □

Lemma 7.6.8 Let σ be a state, where reference p points to a data structure presenting an expression, $t \in Ref$ to the type table, and $g, l \in Ref$ point to the global and local symbol table respectively. Then the coded expression ex is a valid expression:

$$\begin{aligned}&\forall p, ex.\ Typetable(t, \sigma, tdl, tt) \wedge GVarList(g, t, \sigma, gdl, gst) \wedge \\ &LVarList(l, t, \sigma, ldl, lst) \wedge Expr(p, t, g, l, \sigma, tr, ex) \Longrightarrow \\ &valid_{\mathcal{E}}(type\_env(tt), gst, lst, ex)\end{aligned}$$

Proof: The lemma is proved again by induction on reference tree tr and case distinction on the identifier. There are two common conditions of validity that need to be shown for any expression: the validity of its subexpression and its type correctness (i.e. the type coded by ty_e match the type of ex).

For expressions with subexpressions the validity of the expressions implies the validity of the subesxpressions; this is shown by the induction hypothesis and Lemma 7.6.4. The second condition of validity that is needed to be shown is type correctness which we have already proved by Lemma 7.6.7.

In addition to common validity conditions there are conditions that depend on the concrete expression type.

Let us present a non-inductive case:

Variable access: $\mathsf{id}_e(p) = 28$

By definitions of $Expr$ and $ref2expr$ we conclude that $ex = VarAcc(n)$, where variable name $n = String\_cont(\mathsf{nm}_v(\mathsf{nm}_e(p), \sigma))$. By the definition of $valid_{\mathcal{E}}$ expression ex is valid if $n \in_* map(fst, gst) \vee n \in_* map(fst, lst)$. Also from $Expr$ we know that $\mathsf{nm}_e(p) \in_* map(\mathsf{cnt}_v, VarList\_ref(l, \sigma))$ or the same for the global variable list. We can show that $VarList\_ref(l, \sigma) = ldl$ and hence, by definition of $VarList$ exists i such that $\mathsf{nm}_e(p) = \mathsf{cnt}_v(ldl_i)$ and relation $Var(\mathsf{nm}_e(p), t, \sigma, lst_i)$ holds and moreover, by definition of Var, the string implementing the variable name (pointed by field nm_v) is abstracted to the name of variable lst_i:

$$String(\mathsf{nm}_v(\mathsf{nm}_e(p)), \sigma, fst(lst_i)).$$

Applying Lemma 7.3.3 we get $n = fst(lst_i)$, which obviously proves the claim. The second case (variable belongs to the global symbol table) is proven in the same way.

For the properties where the exact type must be shown (e.g. Int or $Unsg$ for the operands of "plus") we proceed in the following way. From definition of reference tree $Tree$ we can easily prove that the pointer to the left subtree $\mathsf{lt}_e(p)$ belongs to tr and hence, the type pointer of the left subtree $\mathsf{ty}_e(\mathsf{lt}_e(p))$ is in $map(\mathsf{cnt}_t, Typetable\_ref(t, \sigma))$ (by definition of $Expr$), where $Typetable\_ref(t, \sigma)$ is equal to tdl. Therefore, conversion of $\mathsf{ty}_e(\mathsf{lt}_e(p))$ to an intermediate representation belongs to tt. Using Lemma 7.4.12 and the value of the type identifier $\mathsf{id}_t(\mathsf{ty}_e(\mathsf{lt}_e(p))$ fixed by definition of $Expr$ for the case of "plus" the claim can be shown.

The cases for other kind of expressions are shown based on Lemmas 7.6.7, 7.6.4, and lemmas about individual properties of them (e.g. Lemma 7.6.6). $\square$

$stmtT = struct\{$	$id : nat$	statement identifier
	$lt : stmtT*$	pointer to the left statement
	$rt : stmtT*$	pointer to the right statement
	$elt : exprT*$	pointer to the left expression
	$ert : exprT*$	pointer to the right expression
	$cf : funcT*$	pointer to the procedure structure (in the case of procedure call)
	$par : exprT\_list*\}$	pointer to the list of parameters to be passed in the function call

Figure 7.8: Statement data structure

To show the equivalence of the generated code for a case where we choose the larger subexpression (e.g. binary operations), we need the following lemma.

Lemma 7.6.9 Abstract expression *ex* and its implementation based on reference tree *tr* have the same number of nodes.

$$Expr(p, t, g, l, \sigma, tr, ex) \Longrightarrow size(ex) = |tr|_T$$

Proof: Is straightforward by induction on *tr* and case distinction on $\mathsf{id}_e(p)$. □

7.7 Statements

The C0 data structure coding statements is depicted in Figure 7.8. Similarly to expressions, it is based on the binary tree structure. We mark the heap functions corresponding to its fields with subscript s.

Expressions are part of a statement in the case of assignment (both pointers *elt* and *ert* are used), function call, memory allocation, and in conditionals (only *elt*, the left expression field, is used).

Pointers to the called procedure (in the case of function call) are of type *funcT*, which will be presented in the next section.

Since the abstract version of a statement is also defined with tree-like recursion, the mapping from the implementation to it is done directly without any intermediate types. In the abstract representation of statements we have an additional parameter for all statements except *Skip* and *Comp*, which provides the uniqueness of them within the procedure table. In the implementation, the role of the statement identifier is played by pointers to a statement structure instance since they completely define its position in the function body of a procedure and procedure table. We use this observation to define a function, which is injective mapping from references to numbers:

$$ref2id \in Ref \to \mathbb{N}$$

Let us define the conversion function from a pointer in some state to the corresponding abstract statement. Coding of the identifiers is clearly presented there.

Definition 7.7.1 Let σ be a state, where reference p points to a statement. Then the abstract equivalent of the statement is computed recursively with the following function.

$$\begin{aligned}\text{Let } & e_1 = \mathit{Expr\_cont}(\mathsf{elt}_s(p), \sigma), \quad e_2 = \mathit{Expr\_cont}(\mathsf{ert}_s(p), \sigma)\\ & s_1 = \mathit{ref2st}(\mathsf{lt}_s(p), \sigma), \quad s_2 = \mathit{ref2st}(\mathsf{rt}_s(p), \sigma),\\ & vn = \mathit{String\_cont}(\mathsf{nm}_e(\mathsf{elt}_s(p)), \sigma)\end{aligned}$$

$$\mathit{ref2st}(p, \sigma) = \begin{cases} \mathit{Ass}(e_1, e_2, \mathit{ref2id}(p)) & \text{if } \mathsf{id}_s = 0\\ \mathit{Ifte}(e_1, s_1, s_2, \mathit{ref2id}(p)) & \text{if } \mathsf{id}_s = 1\\ \mathit{Comp}(s_1, s_2) & \text{if } \mathsf{id}_s = 2\\ \mathit{Loop}(e_1, s_1, \mathit{ref2id}(p)) & \text{if } \mathsf{id}_s = 3\\ \mathit{Alloc}(e_1, \mathit{ref2nm}(\mathsf{ty}_e(\mathsf{elt}_s(p))), \mathit{ref2id}(p)) & \text{if } \mathsf{id}_s = 4\\ \mathit{Call}(vn, \mathit{ExprList\_cont}(\mathsf{par}_s(p)), \sigma), \mathit{ref2id}(p)) & \text{if } \mathsf{id}_s = 5\\ \mathit{Return}(e_1, \mathit{ref2id}(p)) & \text{if } \mathsf{id}_s = 6\\ \mathit{Skip} & \text{if } \mathsf{id}_s = 7 \end{cases}$$

The translation is mostly straightforward. Expressions and expression lists are extracted from the corresponding fields of the data structure referenced by p with functions *Expr_cont* and *ExprList_cont*, respectively. In the case of a memory allocation statement, the type name of the type to be allocated is extracted from the left expression type.

The abstraction function for statements is defined analogously to *Expr*, so we do not show the complete definition here, but only properties of nodes defining some types of statements.

Since the function call statement includes a pointer to a procedure to be called, to specify properties of such a statement we need to define that pointer $\mathsf{cf}_s(p)$ belongs to an entry in the procedure table. The approach we used for this purposes before, i.e. providing the abstraction function with a pointer to the procedure table and extracting the reference list (the table is based on) by means of the choice operator, is not suitable in this case. The abstraction function for procedures and hence, for procedure table cannot be defined without the statement abstraction function (to specify the pointer to the function body). So, we have mutual dependency, which is, of course, undesirable, but it is easy to avoid if we provide the statement abstraction function directly with that reference list. Thus, the abstraction function is the following:

Definition 7.7.2 Let $p \in \mathit{Ref}$ be a reference to statement, $t \in \mathit{Ref}$ be a reference to the type table, $g, l \in \mathit{Ref}$ be references to the global symbol table and to some local symbol table respectively in state σ, and $\mathit{fdl} \in \mathit{Ref}^*$ be a list of pointers to the procedure table entries. Then the data structure referenced by p represents abstract statement s based on reference tree tr.

$$\mathit{Stmt}(p, t, g, l, \sigma, \mathit{fdl}, tr, s) \in \mathbb{B}$$

Let us present relations and restrictions on pointers of the data structure for some kinds of statement.

Assignment

$$\begin{aligned}
&\mathsf{id}_s(x) = 0 \longrightarrow\\
&\ \mathsf{elt}_s(x) \neq \mathit{Null} \wedge \mathsf{ert}_s(x) \neq \mathit{Null} \wedge\\
&\ \mathit{is\_Expr}(\mathsf{elt}_s(x), t, g, l, \sigma) \wedge \mathit{is\_Expr}(\mathsf{ert}_s(x), t, g, l, \sigma) \wedge\\
&\ (\mathsf{ty}_e(\mathsf{elt}_s(x)) = \mathsf{ty}_e(\mathsf{ert}_s(x)) \wedge \mathsf{ty}_e(\mathsf{elt}_s(x)) \neq \mathit{Null} \vee\\
&\ \mathsf{ty}_e(\mathsf{elt}_s(x)) \neq \mathit{Null} \wedge \mathsf{id}_t(\mathsf{ty}_e(\mathsf{elt}_s(x))) = \mathit{PTR} \wedge \mathsf{ty}_e(\mathsf{ert}_s(x)) = \mathit{Null})
\end{aligned}$$

Thus, an assignment statement does not have any statement below in a syntax tree and its *elt* and *ert* fields are abstracted to expressions. The types of the expression can be either the same (and not of the null pointer type) or the left - of any pointer type and the right - the null pointer.

Function Call

$$\begin{aligned}
&\mathsf{id}_s(x) = 5 \longrightarrow\\
&\mathit{is\_Expr}(\mathsf{elt}_s(x), t, g, l, \sigma) \wedge \mathsf{id}_e(\mathsf{elt}_s(x)) = 28 \wedge \mathsf{ty}_e(\mathsf{elt}_s(x)) = \mathsf{rty}_f(\mathsf{cf}_s(x)) \wedge\\
&\mathsf{cf}_s(x) \neq \mathit{Null} \wedge \mathsf{cf}_s(x) \in_* \mathit{fdl} \wedge\\
&\mathit{is\_ExprList}(\mathsf{par}_s(x), t, g, l, \sigma) \wedge |el| = |vl| \wedge\\
&\forall i < |el|.\ \mathsf{ty}_e(\mathsf{cnt}_e(\mathsf{el}_i)) = \mathsf{ty}_v(\mathsf{cnt}_v(\mathsf{vl}_i)) \wedge \mathsf{ty}_e(\mathsf{cnt}_e(\mathsf{el}_i)) \neq \mathit{Null} \vee\\
&\qquad\qquad \mathsf{ty}_e(\mathsf{cnt}_e(\mathsf{el}_i)) = \mathit{Null} \wedge \mathsf{id}_t(\mathsf{ty}_v(\mathsf{cnt}_v(\mathsf{vl}_i))) = \mathit{PTR}\\
&\text{where } el = \mathit{ExprList\_ref}(\mathsf{par}_s(x), t, g, l, \sigma),\\
&\qquad\quad vl = \mathit{VarList\_ref}(\mathsf{vpar}_f(\mathsf{cf}_s(x)), t, \sigma)
\end{aligned}$$

We have the following properties: i) the left side of the statement is a variable expression (identifier equal to 28); ii) the type of the expression is equal to the return type of the called procedure (field *rty* in the C0 data structure describing procedure declaration in Figure 7.9); iii) the pointer to the called function entry is in the reference list *fdl*; iv) the pointer to parameters must be abstracted to an expression list. The additional criterion on the parameter list that it has to match to parameter variables in the called procedure (pointed to by the field *vpar*), i.e. the number of passed parameters needs to be the same as the number of receiving variables and they must have matching types (equal, or pointer type/null pointer type pair).

Conditional Statement

$$\begin{aligned}
&\mathsf{id}_s(x) = 2 \longrightarrow\\
&\mathsf{lt}_s(x) \neq \mathit{Null} \wedge \mathsf{rt}_s(x) \neq \mathit{Null} \wedge\\
&\mathit{is\_Expr}(\mathsf{elt}_s(x), t, g, l, \sigma) \wedge \mathsf{ty}_e(\mathsf{elt}_s(x)) \neq \mathit{Null} \wedge \mathsf{id}_t(\mathsf{ty}_e(\mathsf{elt}_s(x))) = \mathit{BOOL} \wedge\\
&\forall y \in_T \mathit{subtree}(x, tr).\ \mathsf{id}_s(y) \neq 6
\end{aligned}$$

A conditional statement has to have its both subtrees non-empty, and the expression of boolean type. The last condition excludes the occurrence of return statement in the subtrees of the conditional.

So far we have formulated the standard properties of several kinds of statements. Properties of the remaining statements are constructed similarly.

Additionally we define abstraction function *CStmt* (C stays for compilable) with the same parameters as *Stmt*, that for every statement including expressions states that $|FR|$ is enough to evaluate it.

Analogous to expressions we can show that pointers to the subtrees of a statement implementation can be abstracted to statements as well.

Lemma 7.7.3

$$\begin{aligned}
&Stmt(p, t, g, l, \sigma, fdl, tr, s) \Longrightarrow \\
&(\mathsf{lt}_s(p) \neq Null \longrightarrow Stmt(\mathsf{lt}_s(p), t, g, l, \sigma, lt(tr), ref2st(\mathsf{lt}_s(p), \sigma))) \wedge \\
&(\mathsf{rt}_s(p) \neq Null \longrightarrow Stmt(\mathsf{rt}_s(p), t, g, l, \sigma, rt(tr), ref2st(\mathsf{rt}_s(p), \sigma)))
\end{aligned}$$

By construction of a reference binary tree, substatements of a statement implementation are always distinct. Function *ref2id* (that is clearly injective, as it corresponds to the correct construction of statement data structure) allows to transfer this property to abstract statements.

Lemma 7.7.4

$$Stmt(p, t, g, l, \sigma, fdl, tr, s) \wedge inj(ref2id) \Longrightarrow distinct_{\mathcal{S}}(s)$$

7.8 Procedures

The C0 data structure used in the compiler to implement procedures is shown in Figure 7.9.

$funcT = struct\{$	$nm : c0\_string$	procedure name
	$vpar : varT\_list*$	parameters
	$loc : varT\_list*$	local variables
	$body : stmtT*$	procedure body statement
	$rty : typeT*$	returned type
	$asize : nat$	allocation size
	$code : asmT\_pair*$	pointer to the code generated for the procedure
	$of : nat\ \}$	offset of the procedure code

Figure 7.9: Procedure data type

The heap functions generated for that data structure are marked by subscript $_f$. The local variables list includes also all parameters. The field *asize* is used to keep the allocation size of the corresponding frame (it is computed once and stored to be not recomputed). The field *of* stores the offset of the code generated for a procedure with respect to the whole program code after the first pass of the compiler to be used for jump distances computations.

Definition 7.8.1 Let $p \in Ref$ be a pointer to procedure data, and $t, g \in Ref$ be pointers to the type table and global symbol table in state σ respectively. Then the following relation connects the concrete data with its abstract variant $f \in \mathcal{P}$.

$$\begin{aligned}
&Func(p, t, g, \sigma, fdl, f) \equiv \\
&\exists ldl.\ LVarList(\mathsf{loc}_f(p), \sigma, ldl, f.par \circ f.loc) \wedge \\
&\exists tr.\ Stmt(\mathsf{body}_f(p), t, g, \mathsf{loc}_f(p), tr, f.body) \wedge \\
&\exists pdl.\ LVarList(\mathsf{vpar}_f(p), \sigma, pdl, f.par) \wedge \\
&\mathsf{rty}_f(p) \in_* Typetable\_ref(t, \sigma) \wedge \\
&ref2ty(\mathsf{rty}_f(p), Typetable\_cont(t, \sigma)) = f.rt \wedge \\
&\{VarList\_ref(\mathsf{loc}_f(p), t, \sigma)\} \cap \{VarList\_ref(\mathsf{par}_f(p), t, \sigma) = \varnothing
\end{aligned}$$

Since the procedure data structure is abstracted to an object of $\mathcal{P}$, we do not have the information about names included. The parameter fdl is used to be passed to the $Stmt$ relation. The reference list, that parameters and local variables are based on is distinct.

To include the information about the procedure names into the abstraction function for a list of procedures, we abstract it to the procedure table (i.e. to type $(nm_{\mathcal{P}} \times \mathcal{P})^*$ and not to $\mathcal{P}^*$).

Definition 7.8.2 Let $p \in Ref$ be a pointer, and $t, g \in Ref$ be pointers to the type table and global symbol table in state σ respectively. Then the relation states that p is the pointer to the procedure table $penv$, whose implementation is based on reference list dl.

$$\begin{aligned}
&FuncList(p, t, g, \sigma, dl, pt) \equiv \\
&\exists l.\ dList(p, \mathsf{nxt}_f, \mathsf{prv}_f, l, dl) \wedge |dl| = |pt| \wedge dl \neq [] \wedge \\
&distinct(map(\mathsf{cnt}_f, dl)) \wedge unique(pt) \wedge \\
&(\forall i < |fdl|.\ String(\mathsf{nm}_f(dl_i), \sigma, fst(pt_i)) \wedge \\
&\qquad\qquad Func(\mathsf{cnt}_f(dl_i), t, g, \sigma, map(\mathsf{cnt}_f, dl), snd(pt_i))) \wedge \\
&\forall x, y \in_* dl.\ x \neq y \longrightarrow \\
&\{tr_x\} \cap \{tr_x\} = \varnothing \wedge \{loc_x\} \cap \{loc_y\} = \varnothing \wedge \{par_x\} \cap \{par_y\} = \varnothing \\
&\text{where } \forall i \in Ref.\ tr_i = Stmt\_ref(\mathsf{body}_f(\mathsf{cnt}_f(i)), t, g, \sigma, map(cnt_f, dl)), \\
&\qquad\qquad\qquad loc_i = VarList\_ref(\mathsf{loc}_f(\mathsf{cnt}_f(i)), t, \sigma) \\
&\qquad\qquad\qquad par_i = VarList\_ref(\mathsf{par}_f(\mathsf{cnt}_f(i)), t, \sigma)
\end{aligned}$$

Thus, we specify that every entry i in reference list dl can be abstracted as pointing to a procedure name (first component of pt_i) and its declaration (the second component). We need to define uniqueness of entries in the procedure environment on the abstract level, since the distinct pointers to procedure names do not imply that property. Also, now we can provide the $Stmt$ relation (through $Func$) with the actual reference list to procedure environment entries namely $map(\mathsf{cnt}_f, dl)$.

Moreover, we state that i) the trees organizing the function bodies and ii) the lists organizing parameters and local variables are distinct among all the procedures of the procedure table.

Based on predicate $CStmt$ we also define $CFunc$ and $CFuncList$ abstraction functions, which are the same as $Func$ and $CFuncList$, except $Stmt$ is replaced with $CStmt$ and $Func$ with $CFunc$, respectively.

Now, the validity for abstracted statements and procedure tables can be shown.

Lemma 7.8.3 For references t, g, l, f pointing to the type table tt, the global symbol table gst, a local symbol table lst, and the procedure table pt of some program, the statement, referenced by pointer p is valid with respect to gst, lst, pt, and type environment $type\_env(tt)$.

$$\begin{aligned}\forall p, s.\ & Typetable(t, \sigma, tdl, tt) \wedge GVarList(g, t, \sigma, gdl, gst) \wedge \\ & FuncList(f, t, g, \sigma, dl, pt) \wedge LVarList(l, t, \sigma, ldl, lst) \wedge \\ & Stmt(p, t, g, l, \sigma, map(\mathsf{cnt}_f, dl), tr, s) \Longrightarrow \\ & valid_{\mathcal{S}}(type\_env(tt), pt, gst, lst, s)\end{aligned}$$

Lemma 7.8.4

$$\begin{aligned}& Typetable(t, \sigma, tdl, tt) \wedge GVarList(g, t, \sigma, gdl, gst) \wedge FuncList(p, t, g, \sigma, dl, pt) \\ & \Longrightarrow valid_{PT}(type\_env(tt), gst, pt)\end{aligned}$$

7.9 Program

The program data structure includes pointers to program components:

$$progT = struct\{ttable : typeT\_list*; gvars : varT\_list*; ptable : funcT\_list*\}$$

The corresponding heap functions are denoted with $_p$.

Definition 7.9.1

$$\begin{aligned}& CProgram(p, \sigma, tenv, gst, pt) \equiv \\ & \exists tdl, gdl, pdl, tt.\ Typetable(\mathsf{ttable}_t(p), \sigma, tdl, tt) \wedge type\_env(tt) = tenv \wedge \\ & GVarList(\mathsf{gvars}_p(p), \mathsf{ttable}_t(p), \sigma, gdl, gst) \wedge \\ & CFuncList(\mathsf{ptable}_p(p), \mathsf{ttable}_t(p), \mathsf{gvars}_p(p), \sigma, pdl, pt) \wedge \\ & \\ & \forall x \in_* pdl, y \in_* tdl.\ \mathsf{id}_t(\mathsf{cnt}_t(y)) = STR \longrightarrow \\ & \{dl_y\} \cap \{gdl\} = \varnothing \wedge \{dl_y\} \cap \{VarList\_ref(\mathsf{loc}_f(\mathsf{cnt}_f(x)), \mathsf{ttable}_t(p), \sigma)\} = \varnothing \\ & \ldots \\ & \text{where } dl_y = CmpList\_ref(\mathsf{StrCmp}_t(\mathsf{cnt}_t(y)), \sigma)\end{aligned}$$

Thus, the abstraction function i) represents pointers as implementation of abstract objects, ii) states the disjointness of pointer structures in the memory: structure components in the type table do not interfere with global variables, local variables, procedure parameters.

7.10 Assembler Instruction List

As a result of the execution the compiler implementation provides a list of assembler instructions.

C0 data type The data type used to organize the instruction list in the memory is defined in Figure 7.10.

$asmT = struct\{$	$id : nat$	instruction identifier
	$opc : char[4]$	opcode
	$rd : nat$	destination register
	$rs1 : nat$	first source register
	$rs2 : nat$	second source register
	$sa : nat$	shift amount
	$imm : int$	immediate constant
$asmT\_list = struct\{$	$nxt : asmT\_list*$	
	$prv : asmT\_list*$	
	$cnt : asmT\}$	instruction
$asmT\_pair = struct\{$	$head : asmT\_list*$	
	$last : asmT\_list*$	
	$length : asmT\}$	length of program piece

Figure 7.10: Assembler instruction list data types

The data type for a assembler instruction is constructed to represent DLX assembler given in [55](for short reference see Appendix A).

While generating instruction lists we perform only two operations with them: concatenating two lists and insertion of a new element to the end of an existing list. The implementation of procedures providing these for doubly linked lists is quite slow since to access its last element we need to go through the whole list. Since these operation occur very often in the compiler implementation, it is desirable to increase speed of their execution. For this purpose we introduce an addition data structure *asmT_pair* to keep both the pointers to the first and to the last elements of the list with an additional field to keep the lists length, which also allows us to check the length of the list without carrying out the recursive procedure over it. Using this data structure we can implement procedures providing quick append and insert-to-the-end operations for lists of instruction.

Translation to an abstract type Analogously to data structures presented before we denote the heap functions generated for assembler implementation with subscript a.

Definition 7.10.1 Let I be the instruction set, $i \in I$ be an instruction. Then $type(i) \in \mathbb{N}$ returns type of instruction i (I, R or J according to [55]) coded as a number.

The information about instruction type is not actually used by the compiler, it is included to unify the following coding of assembler instructions to the binary machine code.

Definition 7.10.2 Let σ be a state and $p \in Ref$ be a reference. Then predicate $instr(p, \sigma, i)$ states that conversion of p is an abstract instruction i.

The relation between a pointer to an instruction data and an abstract instruction i is trivial, it just tests equality of the same fields in both representations and the correspondence of the opcode for each kind of instruction. For example, to state that p points to instruction $andi(RD, RS1, ic)$ the following statement must hold:

$$\mathsf{opc}_a(p) = \text{"}andi\text{"} \wedge \mathsf{rd}_a(p) = RD \wedge \sigma.rs1_a(p) = RS1 \wedge \mathsf{imm}_a(p) = ic.$$

As a part of the abstraction relation representing the concrete side we use data structure $asmT\_pair$, since in the implementation the assembler program is presented by an instance of this type.

Definition 7.10.3 Let $f, l \in Ref$ be references, $ln \in \mathbb{N}$ be a number, $dl \in Ref^*$ be reference lists, and $prg \in I^*$ be an abstract instruction list. Then abstraction function

$$\begin{aligned}
&AsmProg(f, l, ln, \sigma, dl, prg) \equiv \\
&dList(f, \mathsf{nxt}_a, \mathsf{prv}_a, l, dl) \wedge ln = |dl| \wedge |dl| = |prg| \wedge distinct(map(\mathsf{cnt}_a, dl)) \wedge \\
&(\forall i < |dl|.\ instr(\mathsf{cnt}_a(dl_i), \sigma, prg_i) \wedge \mathsf{id}_a(\mathsf{cnt}_a(dl_i)) = type(prg_i)) \wedge \\
&\{dl\} \cap \{map(\mathsf{cmd}_a, dl)\} = \varnothing \wedge \{dl\} \subseteq \{\mathsf{alloc}\} \wedge \{map(\mathsf{cmd}_a, dl)\} \subseteq \{\mathsf{alloc}\}
\end{aligned}$$

states that there exists an instruction list in the memory, representing abstract prg on the base of reference list dl with f and l as the pointers to its first and last element, and ln equals to length of the instruction list.

The condition $\{dl\} \cap \{map(\mathsf{cmd}_a, dl)\} = \varnothing$ is used to argue that changing one of them do not change the other. It is clearly caused by the memory model we use, since really the references in these lists have different types and cannot be affected by each other.

The additional component of the state space $\sigma.alloc$, which keeps track of allocated pointers and is used as the parameter of NEW statement (Section 4.2.1), helps to argue about disjointness of objects sharing the same heap functions.

Let σ contain some instruction list prg:

$$AsmProg(f, l, ln, \sigma, dl, prg).$$

Then after some procedure generating a new instruction list (and not changing data representing prg) we get state σ', where the heap functions related to instruction data structure are obviously changed, i.e. $\mathsf{h}_a^{\sigma'} \neq \mathsf{h}_a^{\sigma}$. We need to prove that the old list is really not changed and is contained in σ', what is often needed during the code generation. Knowing that $\forall x \in \{\mathsf{alloc}^{\sigma}\}$ values of all heap functions related to instructions stay unchanged it is easy to show.

Lemma 7.10.4

$$\begin{aligned}
&AsmProg(f, l, ln, \sigma, dl, prg) \wedge \{\mathsf{alloc}^{\sigma}\} \subseteq \{\mathsf{alloc}^{\sigma'}\} \wedge \\
&\forall x \in \{\mathsf{alloc}^{\sigma}\}, h_a.\ h_a^{\sigma'}(x) = \mathsf{h}_a^{\sigma}(x) \Longrightarrow \\
&AsmProg(f, l, ln, \sigma', dl, prg)
\end{aligned}$$

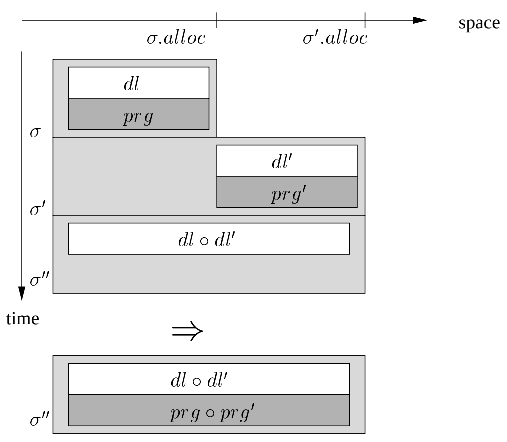


Figure 7.11: Append of two instruction lists

Proof: By Lemma 7.1.7 and the assumption about heap functions we get that relation $dList(f, \mathsf{nxt}_a^{\sigma'}, \mathsf{prv}_a^{\sigma'}, l, dl)$ still holds (since $\{dl\} \subseteq \{\mathsf{alloc}^{\sigma}\}$). The lengths of lists dl, prg which are equal to ln have not changed. As $\mathsf{cnt}_a^{\sigma} = \mathsf{cnt}_a^{\sigma'}$ for all references from dl and abstraction functions depend only on heap functions h_a we can show that $instr(\mathsf{cnt}_a^{\sigma}(dl_i), \sigma, prg_i) \longrightarrow instr(\mathsf{cnt}_a^{\sigma'}(dl_i), \sigma', prg_i)$ holds for all $i < |dl|$. The last line of $AsmProg$ definition follows from consecutive sets inclusion: $\{dl\} \subseteq \{\mathsf{alloc}^{\sigma}\} \wedge \{\mathsf{alloc}^{\sigma}\} \subseteq \{\mathsf{alloc}^{\sigma'}\} \longrightarrow \{dl\} \subseteq \{\mathsf{alloc}^{\sigma'}\}$

Also, we use component alloc to argue on the append operation of two instruction lists. Let us consider the situation, where two lists are consecutively created and then concatenated (Figure 7.11).

Lemma 7.10.5 Let references f, l point to an instruction list based on $dl \in Ref^*$ in σ and f', l' to another instruction list based on dl' in σ'. If state σ'' includes concatenation of dl and dl' referenced by $p \in Ref$, then p can be abstracted to instruction list $prg \circ prg'$.

$$\begin{aligned}
&AsmProg(f, l, ln, \sigma, dl, prg) \wedge AsmProg(f', l', ln', \sigma', dl', prg') \wedge \\
&\forall x \in \{\mathsf{alloc}^{\sigma}\}, h_a.\ h_a^{\sigma'}(x) = h_a^{\sigma}(x) \wedge \{\mathsf{alloc}^{\sigma}\} \subseteq \{\mathsf{alloc}^{\sigma'}\} \wedge \\
&\{dl'\} \cap \{\mathsf{alloc}^{\sigma}\} = \varnothing \wedge \{map(\mathsf{cnt}_a^{\sigma'}, dl')\} \cap \{\mathsf{alloc}^{\sigma}\} = \varnothing \wedge \\
&dList(p, \mathsf{nxt}_a^{\sigma''}, \mathsf{prv}_a^{\sigma''}, q, dl \circ dl') \wedge \\
&\forall h. h \neq \mathsf{nxt}_a \wedge h \neq \mathsf{prv}_a \longrightarrow h^{\sigma''} = h^{\sigma'} \\
&\Longrightarrow AsmProg(p, q, ln + ln', \sigma'', dl \circ dl', prg \circ prg')
\end{aligned}$$

Thus, the conditions we have are i) generation of prg' should not affect the structure implementing prg; ii) append operation on dl, dl' changes only heap functions $\mathsf{nxt}_a, \mathsf{prv}_a$.

Proof: The $dList$ relation from the claim follows directly from the assumptions. About the unchanged content of instruction lists based on dl, dl' in state σ', and hence σ'' (since only dl and dl' are changed) we argue analogously to the previous lemma. $ln + ln' = |dl \circ dl'| = |prg \circ prg'|$ is easily shown from the assumptions.

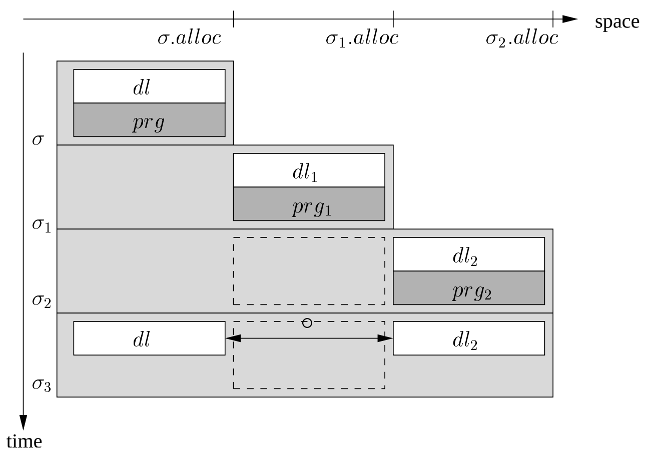


Figure 7.12: Out-of-order append of three instruction lists

The distinctness of the pointers to the content of list elements can be rewritten as follows:

$$\begin{aligned}
&distinct(map(\mathsf{cnt}_a^{\sigma''}, dl \circ dl')) = \\
&distinct(map(\mathsf{cnt}_a^{\sigma'}, dl \circ dl')) = \\
&distinct(map(\mathsf{cnt}_a^{\sigma'}, dl)) \wedge distinct(map(\mathsf{cnt}_a^{\sigma'}, dl')) \wedge \\
&map(\mathsf{cnt}_a^{\sigma'}, dl) \cap map(\mathsf{cnt}_a^{\sigma'}, dl') = \varnothing,
\end{aligned}$$

where the first conjunct is true as $\forall x \in \{dl\}.\ \mathsf{cnt}_a^{\sigma'}(x) = \mathsf{cnt}_a^{\sigma}(x)$ (since $\{dl\} \subseteq \{\mathsf{alloc}^{\sigma}\}$) and assumption $distinct(map(\mathsf{cnt}_a^{\sigma}, dl))$ included in $AsmProg$. The second conjunct follows from $AsmProg$ for prg', and $map(\mathsf{cnt}_a^{\sigma'}, dl') \cap \{\mathsf{alloc}^{\sigma}\} = \varnothing$ whereas $map(\mathsf{cnt}_a^{\sigma'}, dl) \subseteq \{\mathsf{alloc}^{\sigma}\}$ implies that their intersection is empty.

The lists dl' and $map(\mathsf{cnt}_a^{\sigma'}, dl')$ of the second instruction list do not belong to alloc^{σ}. In contrast, lists dl and $map(\mathsf{cnt}_a^{\sigma'}, dl)$ are in $\{\mathsf{alloc}^{\sigma}\}$. Since we know that these pairs do not have common elements (inside each pair), we can conclude that $\{dl \circ dl'\} \cap \{map(\mathsf{cnt}_a^{\sigma'}, dl \circ dl')\} = \varnothing$. The last conjunctions from the claim of the lemma (with $AsmProg$ expanded) can be shown from set inclusion $\{\mathsf{alloc}\} \subseteq \{\mathsf{alloc}^{\sigma'}\} = \{\mathsf{alloc}^{\sigma''}\}$

However, there is the situation (shown in Figure 7.12) where the conditions of the given lemma do not suit to show the claim. We create three pieces of code and then append them to get the structure implementing $prg \circ prg_2 \circ prg_1$. The lemma works for the first append, but trying to apply it while appending prg_1 we get the conditions $\{dl_1\} \cap \{\mathsf{alloc}^{\sigma_3}\} = \varnothing$ and $\{map(\mathsf{cnt}_a, dl_1)\} \cap \{\mathsf{alloc}^{\sigma_3}\} = \varnothing$ violated. For that case we have an additional lemma similar to the previous one, where these conditions are changed to statements showing the distinctness of all participating reference lists explicitly:

$$\begin{aligned}
&\ldots \{dl\} \cap \{dl'\} = \varnothing \wedge \{map(\mathsf{cnt}_a^{\sigma}, dl)\} \cap \{map(\mathsf{cnt}_a^{\sigma'}, dl')\} = \varnothing \wedge \\
&\{dl\} \cap \{map(\mathsf{cnt}_a^{\sigma'}, dl')\} = \varnothing \wedge \{dl'\} \cap \{map(\mathsf{cnt}_a^{\sigma}, dl)\} = \varnothing \ldots
\end{aligned}$$

Chapter 8

Verification Details

Verification of a program code in Hoare logic using abstraction functions for the program data is usually very involved and hard to show in paper-and-pencil proofs. However, in this chapter we try to show the main approach and some concrete details for typical cases of the verification process. We start with lower levels of the verification pyramid for the compiler implementation. Formulating specifications of a procedure behavior we follow syntax notations given in Section 4.3.2 and omit types of logical variables where they can be deduced from the context.

8.1 Compilation Algorithm

The code generation algorithm realized in the compiler implementation proceeds in two passes. During the first pass we do not know the size of the compiled code for all the procedures and cannot immediately compute the jump distances, which are needed for coding function calls. Computing the code size recursively at the position where it is needed (as it is done in the specification) is, of course, possible but ineffective. Thus, during the first pass we produce the compiled code without filling the jump distances. Repeating the whole algorithm for the code generation as the second pass is also ineffective, so during the first pass we store positions of jumps needed to be filled in a special data structure (by pointers to the corresponding instructions data in the memory) together with some additional information. Since, after the first pass the size of the compiled code for every function is known, and therefore, the relative offset for each procedure can be calculated, we can compute jump distances using the information stored during the first pass. Therefore, the compilation algorithm has the following steps:

- Some data structures are initialized (e.g. list of free registers).
- Alignment and allocated size for all entries of the type table are computed.
- Displacement of the global variables inside the global frame is computed.
- For every procedure in the procedure table displacements for local variables are computed.
- First pass: code for each procedure body is generated; code pieces of single procedures are combined to one program.

- Second pass: filling in the jump distances in the code of all call statements.

8.2 Type Table

In this section we show some details of the verification of procedures working with the type table.

Like the compiler specification, the implementation includes three procedures used for that computation: *min_gt_div* computing number $\lceil s \rceil_d$; *compute_align* and *compute_asize* which are equivalent to *algn* and *asize* functions of the specification, respectively.

8.2.1 Alignment computation

The procedure *compute_align* computes the alignment for every type from the type table. It takes a pointer to the type table as a parameter. It does not need any return values, as it changes multiple values directly on the heap, but since we have to return something according to C0 semantics, we just return integer value 0. Recall, that a return statement in C0 is modelled as an assignment of the returned value to a special return variable (res_int for this case).

Implementing this procedure we have two while loops: the outer loop going through the type table and the nested one going through structure component lists (see Figure 8.1). The size of the array sizes is equal to the number ELT_NUM (see Table 7.1) of elementary types (which is four in our case) and each element from the array is equal to the number of bytes needed to store the object of the corresponding type in the memory. Array sizes allows us to change the memory allocation size for the elementary types without changing the function code or the proof of its correctness, just by redefining values stored in the array.

Theorem 8.2.1 If the state before execution of the procedure *compute_align* contains the type table and if the array sizes keeps the number of bytes to store each of the elementary types, then the state after execution contains the same type table, where in the field align_t of every table entry the value of the alignment of the coded type is stored. Recall that *ElId* is the set of the identifiers of elementary types.

$$\begin{gathered}\forall \sigma, dl, tt.\ \Gamma \vdash \{\sigma \mid \mathit{Typetable}(\mathsf{hd}, \sigma, dl, tt) \wedge \mathit{inj}(\mathit{ref2nm}) \wedge \\ (\forall i < |\mathit{ElId}|.\ \mathsf{sizes}[i] = w(i2ty(i)))\} \\ \mathsf{res\_int} := \mathsf{CALL}\ \ \mathsf{compute\_align(hd)} \\ \{\sigma' \mid \mathit{Typetable}(\mathsf{hd}^{\sigma}, \sigma', dl, tt) \wedge \\ \forall x \in_* dl.\ \mathsf{align}_t(\mathsf{cnt}_t(x)) = \mathit{algn}(\mathit{ref2ty}(\mathsf{cnt}_t(x), \sigma', tt))\}\end{gathered}$$

Proof: Since the implementation includes while loops, we need to provide an invariant to each of them for VCG. As align_t is the only heap function changed during the execution of this procedure, values of all other heap functions do not change and so in the proof we can refer to their values in the current state instead of the values in the initial state.

```
**compute_align** ( hd | res_int) =
WHILE hd $\neq$ Null DO
    tid := hd $\rightarrow$ cnt $\rightarrow$ id ;
    IF tid < ELT_NUM THEN
        hd $\rightarrow$ cnt $\rightarrow$ align := sizes[tid]
    ELSE IF tid = ARR THEN
        hd $\rightarrow$ cnt $\rightarrow$ align := hd $\rightarrow$ cnt $\rightarrow$ eltTy $\rightarrow$ align
    ELSE IF tid = STR THEN
        cmps := hd $\rightarrow$ type $\rightarrow$ strCmp; max := 0 ;
        WHILE cmps $\neq$ Null DO
            n := cmps $\rightarrow$ cnt $\rightarrow$ ty $\rightarrow$ align ;
            IF max < n THEN max := n FI ;
            cmps := cmps $\rightarrow$ nxt
        OD ;
        hd $\rightarrow$ cnt $\rightarrow$ align := max
    ELSE IF tid = PTR THEN
        hd $\rightarrow$ cnt $\rightarrow$ align:= PTR_SIZE
    FI FI FI FI ;
    hd := hd $\rightarrow$ nxt
OD ;
res_int := 0
```

$hd : typeT\_list{*};\ \ cmps : varT\_list{*};\ \ tid, n, max : nat;\ \ res\_int : int$

Figure 8.1: Implementation of the procedure to compute type alignment

The invariant $I(\tau)$ for the outer loop in some state τ during the loop execution is the following (recall that with x^{σ} we refer to the initial value of state component x before the procedure execution and with x to the current one):

$$
\begin{array}{ll}
(I_1) & Typetable(\mathsf{hd}^{\sigma}, \tau, dl, tt) \wedge inj(ref2nm) \wedge (\forall i < |ElId|.\ \mathsf{sizes}[i] = w) \wedge \\
(I_2) & (\mathsf{hd} \neq Null \longrightarrow \exists i.\ \mathsf{hd} = dl_i \wedge \\
 & \forall x \in_* (dl_0, \ldots, dl_{i-1}).\ \mathsf{align}_t(\mathsf{cnt}_t(x)) = algn(ref2ty(\mathsf{cnt}_t(x), \tau, tt))) \wedge \\
 & (\mathsf{hd} = Null \longrightarrow Q(\tau))
\end{array}
$$

The part I_1 of the invariant is inherited from the preconditions: hd initially pointed to the beginning of the type table. I_2 states that during the loop execution all the entries, which are placed before the current position of the hd pointer, have the field align_t filled by the value computed for the abstract type corresponding to this entry. As soon as the value of hd becomes the null pointer, postconditions hold (we refer to it with Q). Let us consider the invariant $I'(\tau)$ of the inner loop:

$$\begin{array}{ll}
(I'_1) & I_1(\tau) \wedge \mathsf{hd} \neq Null \wedge \exists i.\ \mathsf{hd} = dl_i \wedge \\
& (\forall x \in_* (dl_0, \ldots, dl_{i-1}).\ \mathsf{align}_t(\mathsf{cnt}_t(x)) = algn(ref2ty(\mathsf{cnt}_t(x), \tau, tt))) \wedge \\
(I'_2) & \exists sc.\ ref2ty(\mathsf{cnt}_t(\mathsf{hd}), \tau, tt) = Str(sc) \wedge \\
& \exists rl.\ rl = CmpList\_ref(\mathsf{strCmp}_t(\mathsf{cnt}_t(\mathsf{hd})), \tau) \wedge \\
(I'_3) & (\mathsf{cmps} \neq Null \longrightarrow \\
& \quad \exists j.\ \mathsf{cmps} = rl_j \wedge \mathsf{max} = algn\_sc((sc_0, \ldots, sc_{j-1}))) \wedge \\
& (\mathsf{cmps} = Null \longrightarrow \mathsf{max} = algn(ref2ty(\mathsf{cnt}_t(\mathsf{hd}), \tau, tt))))
\end{array}$$

If we enter the nested loop, we compute the alignment value for a structure type. The part I'_1 of the invariant is taken from the outer loop invariant. We include the information about alignment values which are already computed, as processing a structure type demands alignment of the subtypes to be known. Part I'_2 states that reference hd points to a structure type entry, which includes a component list based on some list of references rl and corresponds to abstract component list sc. In I_3 we describe the value, which variable max takes during the loop execution. Until cmps becomes the null pointer, it keeps the maximal alignment value among the prefix $sc_0, \ldots sc_{j-1}$ of the abstract component list sc. As soon as cmps becomes equal to $Null$, max is the alignment of the whole component list sc.

Let us consider the goals, which we need to show during the verification of this procedure, in more details.

After applying VCG, verification goes according to the following scheme:

1. The preconditions of the procedure (denoted by function P) imply the invariant of the first loop (in the initial state):

$$P(\sigma) \Longrightarrow I(\sigma)$$

2. In the case when the loop iterations are not finished, the invariant I is maintained (with respect to the case distinction). Moreover, in the case of a structure type the implication of the second invariant I' with initial values of cmps and max must be shown.

$$\begin{array}{l}
I(\tau) \wedge \mathsf{hd} \neq Null \Longrightarrow \\
(\mathsf{tid} < EL\_ID \longrightarrow \\
\ I(\tau[\mathsf{hd} := \mathsf{nxt}_t(\mathsf{hd}), \mathsf{align}_t := \mathsf{align}_t[\mathsf{cnt}_t(\mathsf{hd}) := \mathsf{sizes}[\mathsf{tid}]]])) \wedge \\
(\mathsf{tid} = ARR \longrightarrow \\
\ I(\tau[\mathsf{hd} := \mathsf{nxt}_t(\mathsf{hd}), \\
\qquad \mathsf{align}_t := \mathsf{align}_t[\mathsf{cnt}_t(\mathsf{hd}) := \mathsf{align}_t(\mathsf{eltTy}_t(\mathsf{cnt}_t(\mathsf{hd})))]])) \wedge \\
(\mathsf{tid} = PTR \longrightarrow \\
\ I[\mathsf{hd} := \mathsf{nxt}_t(\mathsf{hd}), \mathsf{align}_t := \mathsf{align}_t[\mathsf{cnt}_t(\mathsf{hd}) := PTR\_SIZE]])) \wedge \\
(\mathsf{tid} = STR \longrightarrow \\
\ I'(\tau[\mathsf{cmps} := \mathsf{strCmp}_t(\mathsf{cnt}_t(\mathsf{hd})), \mathsf{max} := 0])
\end{array}$$

Variable hd and heap function align_t are updated according to case distinction inside the loop.

3. The second invariant is maintained during the execution of the inner loop. According to the IF-construct two subcases are needed to be shown.

$$I'(\tau) \wedge \mathsf{cmps} \neq Null \Longrightarrow$$
$$(\mathsf{max} < n \longrightarrow I'(\tau[\mathsf{cmps} := \mathsf{nxt}_v(\mathsf{cmps}); \mathsf{max} := n])) \wedge$$
$$(\mathsf{max} \geq n \longrightarrow I'(\tau[\mathsf{cmps} := \mathsf{nxt}_v(\mathsf{cmps})]),$$

   where $n = \mathsf{align}_t(\mathsf{ty}_v(\mathsf{cnt}_v(\mathsf{cmps})))$

4. The invariant I' after finishing the inner loop (in some state τ) implies the invariant I (where τ updated according to the case of a structure type):

$$I'(\tau) \wedge \mathsf{tid} = STR \wedge \mathsf{cmps} = Null \Longrightarrow$$
$$I(\tau[\mathsf{hd} := \mathsf{nxt}_t(\mathsf{hd}), \mathsf{align}_t := \mathsf{align}_t[\mathsf{cnt}_t(\mathsf{hd}) := \mathsf{max}]]$$

5. The invariant I after finishing the outer loop implies the postconditions:

$$I(\tau) \wedge \mathsf{hd} = Null \Longrightarrow Q(\tau)$$

Let us consider the details of the proof for each subgoal:

Goal 1. Part I_1 of the invariant is obviously implied by the preconditions. List dl is not empty since $|dl| = |tt| \geq |ElId|$ (by definition of $Typetable$) and hence, $\mathsf{hd}^\sigma \neq Null$ and $\mathsf{hd}^\sigma = dl_0$ (by Lemma 7.1.2), so we have $i = 0$ satisfying the invariant.

Goal 2. The proof of this goal has some claims that need to be shown independently of the case (pointer, array, etc.) we choose. Let us prove them first. Since the only changed heap function is align_t,

$$Typetable(\mathsf{hd}^\sigma, \tau, dl, tt) \longrightarrow Typetable(\mathsf{hd}^\sigma, \tau', dl, tt)$$

holds, where with τ' we denote the updated (respective the case) state τ. The rest of I_1 also hold, not being affected by the changes of align_t.

To show I_2 we consider the following cases: i) hd does not point to the last element in the type table, i.e. $\mathsf{hd} \neq last(dl)$, which provides $\mathsf{nxt}_t(\mathsf{hd}) \neq Null$ (by Lemma 7.1.3); ii) $\mathsf{hd} = last(dl)$ giving $\mathsf{nxt}_t(\mathsf{hd}) = Null$ according to Lemma 7.1.4. Hence, both variants of part I_2 of the invariant are covered.

i) we need to show that

$$\exists i.\ \mathsf{nxt}_t(\mathsf{hd}) = dl_i \wedge$$
$$\forall x \in_* (dl_0, \ldots, dl_{i-1}).\ align'(\mathsf{cnt}_t(x)) = algn(ref2ty(\mathsf{cnt}_t(x), \tau', tt)), \quad (8.1)$$

where $align'$ is the updated heap function align_t with respect to the case distinction. As by assumption $I(\tau)$ there exists i such that $\mathsf{hd} = dl_i$, we have $\mathsf{nxt}_t(\mathsf{hd}) = dl_{i+1}$ according to Lemma 7.1.3. Instantiating the existence quantifier in (8.1) with $i + 1$ we can easily show that the first conjunct of this goal is true.

The proof of the second conjunct depends on the particular case.

Elementary types We have to show

$$\forall x \in_* (dl_0, \ldots, dl_{(i+1)-1}).\ align'(\mathsf{cnt}_t(x)) = algn(ref2ty(\mathsf{cnt}_t(x), \tau', tt)),$$

where $align' = \mathsf{align}_t[\mathsf{cnt}_t(\mathsf{hd}) := \mathsf{size}[\mathsf{tid}]]$.

For all x from dl until dl_i it is true by assumption

$$\forall x \in_* (dl_0, \ldots, dl_{i-1}).\ \mathsf{align}_t(\mathsf{cnt}_t(x)) = algn(ref2ty(\mathsf{cnt}_t(x), \tau, tt))$$

from $I(\tau)$ and the following observations. First, we have $ref2ty(\mathsf{cnt}_t(x), \tau, tt) = ref2ty(\mathsf{cnt}_t(x), \tau', tt)$. Second, $align'$ is equal to align_t for all elements from dl until dl_i. The latter follows from the disjointness of fields $\mathsf{cnt}_t(x)$ for all x from dl (by the definition of $Typetable$), that implies the update of the heap function align_t at position $\mathsf{cnt}_t(dl_i)$ does not change its value for any $x \neq dl_i$.

For reference x equal to $dl_i = \mathsf{hd}$ we can write

$$align'(\mathsf{cnt}_t(x)) = \mathsf{size}[\mathsf{tid}] = w(i2ty(tid)) = algn(ref2ty(\mathsf{cnt}_t(\mathsf{hd}), \tau, tt))$$

The last equality in the chain is valid, as by definition of $Typetable$ there is an intermediate type representation

$$rev(tt)_i = ref2type(\mathsf{cnt}_t(\mathsf{hd}), \tau) \tag{8.2}$$

such that type $type2ty(rev(tt)_i, tt)$ is elementary (by Lemma 7.4.12) and hence, its alignment is equal to $w(type2ty(rev(tt)_i, tt))$ (by Definition 6.1.3).

Pointer The proof for a pointer type is similar. The only condition we need additionally: constant $\mathsf{PTR\_SIZE}$ is equal to W.

Array For an array type, the value which updates the heap function align_t after one loop iteration is equal to $\mathsf{align}_t(\mathsf{eltTy}_t(\mathsf{cnt}_t(\mathsf{hd})))$. Hence, we need to show that

$$\mathsf{align}_t(\mathsf{eltTy}_t(\mathsf{cnt}_t(\mathsf{hd}))) = algn(ref2ty(\mathsf{cnt}_t(\mathsf{hd}), \tau, tt)) \tag{8.3}$$

From (8.2) and predicate $well\_defined$ for tt (from the definition of $Typetable$) there exists type y such that its name is $y.name = rev(tt)_i.elt$ and y belongs to the part of the type table defined by $sfx((\lambda z.z = rev(tt)_i), tt)$. By rewriting this type table part is:

$$\begin{array}{lcr}
sfx((\lambda z.z = rev(tt)_i), tt) & = & (\text{def. } well\_defined\ ) \\
pfx((\lambda z.z \neq rev(tt)_i), rev(tt)) & = & (\text{ L. 1.4.16 }) \\
(rev(tt)_0, \ldots, rev(tt)_{i-1}) & & (\text{ def. } pfx\ )
\end{array}$$

From (8.2) and definition of the conversion function $ref2type$ we have

$$rev(tt)_i.elt = ref2type(\mathsf{cnt}_t(\mathsf{hd}), \tau).elt = ref2nm(\mathsf{eltTy}_t(\mathsf{cnt}_t(\mathsf{hd}))) \tag{8.4}$$

Thus, we can rewrite:

$$\begin{array}{lcr} \mathit{ref2ty}(\mathsf{cnt}_t(\mathsf{hd}), \tau, tt) & = & (\ \mathsf{def}.\mathit{ref2ty}\ ) \\ \mathit{type2ty}(\mathit{ref2type}(\mathsf{cnt}_t(\mathsf{hd})), \tau), tt) & = & (8.2) \\ \mathit{type2ty}(\mathit{rev}(tt)_i, tt) & & \end{array}$$

and

$$\begin{array}{lcr} \mathit{algn}(\mathit{type2ty}(\mathit{rev}(tt)_i, tt)) & = & (\ \text{L. } 7.4.13\ ) \\ \mathit{algn}(\mathit{Arr}(\mathit{rev}(tt)_i.\mathit{eln}, \mathit{type2ty}(\mathit{nm2type}(\mathit{rev}(tt)_i.\mathit{elt}), tt))) & = & (\ (8.4),\ \mathit{algn}\ ) \\ \mathit{algn}(\mathit{type2ty}(\mathit{nm2type}(\mathit{ref2nm}(\mathsf{eltTy}_t(\mathsf{cnt}_t(\mathsf{hd}))), tt))) & = & (\ \mathsf{L.\ 7.4.11}\ ) \\ \mathit{algn}(\mathit{type2ty}(y, tt)) & & \end{array}$$

We can show that type y must be equal to $\mathit{ref2type}(\mathsf{eltTy}_t(\mathsf{cnt}_t(\mathsf{hd})), \tau)$ and moreover, since $y \in_* (\mathit{rev}(tt)_0, \ldots, \mathit{rev}(tt)_{i-1})$, the reference $\mathsf{eltTy}_t(\mathsf{cnt}_t(\mathsf{hd}))$ can only be placed in prefix $(dl_0, \ldots, dl_{i-1})$ of the list. It can be shown from the fact that the mapping of references to type names is injective, type names are unique, and values $\mathsf{cnt}_t(x)$ for all $x \in_* dl$ are distinct.

Since $\mathsf{eltTy}_t(\mathsf{cnt}_t(\mathsf{hd})) \in_* (dl_0, \ldots, dl_{i-1})$, we can use the assumption about already computed values of align_t for that part of dl and immediately get

$$\mathsf{align}_t(\mathsf{eltTy}_t(\mathsf{cnt}_t(\mathsf{hd}))) = \mathit{algn}(\mathit{ref2ty}(\mathsf{eltTy}_t(\mathsf{cnt}_t(\mathsf{hd})), \tau, tt)) \tag{8.5}$$

Combining (8.5) with the rewriting steps given above and the value of the intermediate type $y = \mathit{ref2type}(\mathsf{eltTy}_t(\mathsf{cnt}_t(\mathsf{hd})), \tau)$ we prove subgoal (8.3).

Structure The last claim of the current goal is the case of a structure type. The part I'_1 of the invariant I' is obvious and follows directly from $I(\tau)$ and $\mathsf{hd} \neq \mathit{Null}$. I'_2 follows from the definition of *Typetable* and Lemma 7.4.14 about the correct conversion of structure types. The rest is proven analogously to Goal 1 as the reference cmps is the head of list rl before the execution of the inner loop.

ii) in the case $\mathsf{hd} = \mathit{last}(dl)$ argumentation is analogous to the previous case after the instantiation of the quantifier: after updating the last element in the type table we have alignment for all the types computed.

Goal 3. The case distinction is organized with the reference cmps and reference list rl in the same way as for hd and dl in Goal 2 covering both cases of the implication in I'. Argumentation about alignment of structure components is similar to the array case, so we need to show that their types are located in the part of the type table where value in the align_t field is already computed. Additionally we need to show that variable max always contains the maximal value among the components for both cases we get from IF-statement.

Analogously to Goal 2 we instantiate the existential quantifier with $j + 1$ in the invariant after one iteration. Also, similarly to the array case it can be shown that

$$\mathsf{align}_t(\mathsf{ty}_v(\mathsf{cnt}_v(\mathsf{cmps}))) = \mathit{algn}(\mathit{snd}(sc_j)) \tag{8.6}$$

Suppose, the current value of max is less than $\mathsf{align}_t(\mathsf{ty}_v(\mathsf{cnt}_v(\mathsf{cmps})))$. We need to show that the new value of max is equal to $\mathit{algn\_sc}((sc_0, \ldots, sc_{(j+1)-1}))$.

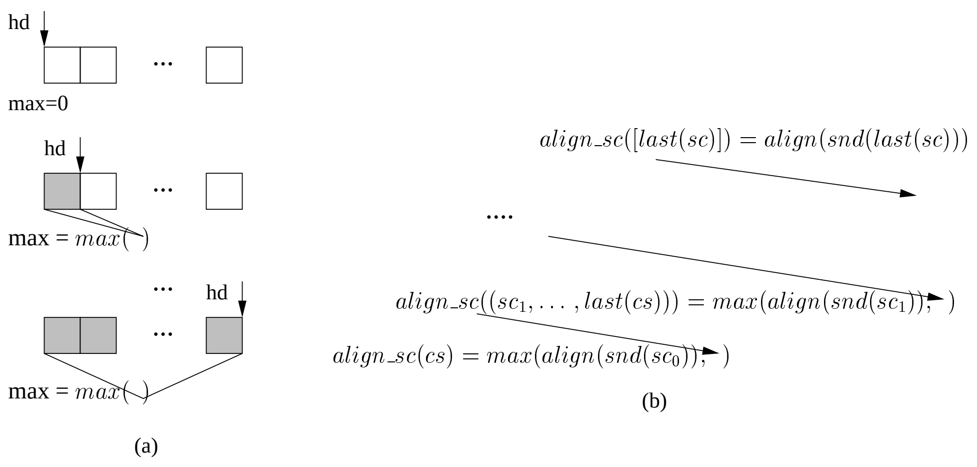

hd
max=0
hd
max = $max($ $)$
hd
max = $max($ $)$
(a)
$align\_sc([last(sc)]) = align(snd(last(sc)))$
....
$align\_sc((sc_1, \dots, last(cs))) = max(align(snd(sc_1)),$ $)$
$align\_sc(cs) = max(align(snd(sc_0)),$ $)$
(b)

Figure 8.2: Computation sequence (a) implementation (b) specification

From the assumption $I'(\tau)$ we get $\mathsf{max}^{\tau} = algn\_sc((sc_0, \dots, sc_{j-1}))$, whereas by expanding the definition of $algn\_sc$ in the claim $I'(\tau')$ we get

$$\mathsf{max}^{\tau'} = max(algn(snd(sc_0)), algn\_sc((sc_1, \dots, sc_{(j+1)-1})),$$

so we cannot use the induction hypothesis.

It is easy to see that recursion in the procedure and in the corresponding abstract function goes differently, namely in the reversed way (see Figure 8.2).

So, we need an additional lemma:

Lemma 8.2.2 Alignment of a structure component list is the same as the alignment of its reverse:

$$algn\_sc(xs) = algn\_sc(rev(xs))$$

That can be easily shown by induction on xs.

Now we can rewrite:

$$\begin{array}{lll}
\mathsf{max}^{\tau'} & = & \\
\mathsf{align}_t(\mathsf{ty}_v(\mathsf{cnt}_v(\mathsf{cmps}))) & = & \\
max(\mathsf{align}_t(\mathsf{ty}_v(\mathsf{cnt}_v(\mathsf{cmps}))), \mathsf{max}^{\tau}) & = & /(8.6),\ I_2'(\tau)\ / \\
max(algn(snd(sc_j)), algn\_sc((sc_0, \dots, sc_{j-1}))) & = & /\ \text{def.}rev\ / \\
max(algn(snd(sc_j)), algn\_sc(rev(sc_{j-1}, \dots, sc_0))) & = & /\ \text{L. 8.2.2}\ / \\
max(algn(snd(sc_j)), algn\_sc((sc_{j-1}, \dots, sc_0))) & = & /\ \text{def. } algn\_sc\ / \\
algn\_sc((sc_j, (sc_{j-1}, \dots, sc_0))) & = & /\ \text{L. 8.2.2}\ / \\
algn\_sc((sc_0, \dots, sc_{(j+1)-1})) & &
\end{array}$$

The case, where value of max is greater than or equal to the alignment of the current component can be shown analogously.

Goal 4. Part I_1 of the claimed invariant follows directly. The rest of the proof is similar to Goal 2. Thus, the values of $\mathsf{align}_t^{\tau'}$ are not changed for all elements from $(dl_0, \ldots, dl_{j-1})$. Since the nested loop is finished, the value updating the align_t heap function is equal to max, i.e. it contains the value of the alignment of the current structure type. This is enough to finish the goal.

Goal 5. The postcondition follows directly from the invariant.

On the example of this procedure we have presented the approach of verification of a procedure with two while-loops based on a list traversal. The form of an invariant and case distinction we have presented is similar for loops of such a kind.

8.2.2 Procedure *min_gt_div*

Procedure *min_gt_div*(s, d) computing the number $\lceil \mathsf{s} \rceil_{\mathsf{d}}$ is realized according to Lemma 6.1.2.

Theorem 8.2.3 For all states, where the value of variable d is positive the call for *min_gt_div*(s, d) returns $\lceil s \rceil_d$.

$$\begin{gathered} \forall \sigma.\ \Gamma \vdash \{\sigma \mid \mathsf{d} > 0\} \\ \mathsf{n} := \mathsf{CALL}\ \ \mathsf{min\_gt\_div(s,d)} \\ \{\sigma' \mid \mathsf{n} = ((\mathsf{s}^\sigma + \mathsf{d}^\sigma - 1)/\mathsf{d}^\sigma) * \mathsf{d}^\sigma\} \end{gathered}$$

8.2.3 Computation of allocated size

The procedure *compute_asize* computing the allocated size for all entries in the type table is organized in a similar way to *compute_align* (see Figure 8.1). It must be called after *compute_align* has been executed, since it requires the values of align_t. Except for allocated size, it also stores:

- into field dsp_t: the offset factor of array elements inside the array memory $\lceil asize(t') \rceil_{algn(t')}$ if a type table entry represents type $Arr(n, t')$
- into field dsp_v (for every structure component): offset of each structure component inside the structure memory region if a type table entry represents type $Str(sc)$

Having values in dsp_t and dsp_v precomputed we can use them directly when generating code for access to a structure field or an array element.

We introduce a new abstraction function for the *initialized* type table:

Definition 8.2.4 Let σ be a state, $p \in Ref$, $dl \in Ref^*$, and $tt \in type^*$ be an abstract type table representation. Then the type table with initialized values for alignment and allocated size of the types is defined as follows:

$$\begin{array}{l}
\mathit{FTypetable}(p,\sigma,dl,tt) \equiv \\
\mathit{Typetable}(p,\sigma,dl,tt) \wedge \\
\forall x \in_* \mathit{map}(\mathsf{cnt}_t, dl). \\
\quad \mathsf{align}_t(x) = \mathit{algn}(\mathit{ref2ty}(x,tt)) \wedge \mathsf{asize}_t(x) = \mathit{asize}(\mathit{ref2ty}(x,tt)) \wedge \\
\quad (\mathsf{id}_t(x) = \mathit{ARR} \longrightarrow \mathsf{dsp}_t(x) = \lceil \mathit{asize}(t') \rceil_{\mathit{algn}(t')}) \wedge \\
\quad (\mathsf{id}_t(x) = \mathit{STR} \longrightarrow \\
\qquad \forall y \in_* \mathit{map}(\mathsf{cnt}_v, \mathit{CmpList\_ref}(\mathsf{strCmp}_t(x),\sigma)). \\
\qquad\quad \mathsf{dsp}_v(y) = \mathit{displ}(0, \mathit{fst}(y), \mathit{CmpList\_cont}(\mathsf{strCmp}_t(x),\sigma))), \\
\text{where } t' = \mathit{snd}(\mathit{the\_Arr}(\mathit{ref2ty}(x,\sigma,tt)))
\end{array}$$

Theorem 8.2.5 If the state before execution of the function *compute_asize* contains a type table with computed alignment for all its entries then the state after execution contains the same type table, where in the field asize_t of every table entry the value of the allocated size for the corresponding type is stored.

$$\begin{array}{c}
\forall \sigma, dl, tt.\ \Gamma \vdash \{\sigma \mid \mathit{Typetable}(\mathsf{hd},\sigma,dl,tt) \wedge \mathit{inj}(\mathit{ref2nm}) \wedge \\
\forall x \in_* \mathit{map}(\mathsf{cnt}_t, dl).\ \mathsf{align}_t(x) = \mathit{algn}(\mathit{ref2ty}(x),\sigma,tt))\} \\
\mathsf{res\_int} := \mathsf{CALL}\ \ \mathsf{compute\_asize}(\mathsf{hd}) \\
\{\sigma' \mid \mathit{FTypetable}(\mathsf{hd}^{\sigma},\sigma',dl,tt)\}
\end{array}$$

Proof: The proof proceeds according to the scheme presented in Theorem 8.2.1. However, there are some points we would like to emphasize.

Outer loop The invariant of the outer loop is constructed in a similar way as the one from Theorem 8.2.1 (extended with information about fields dsp_t, dsp_v according to the definition of *FTypetable*). Analogously to that proof we find the values of alignment and allocated size for the type of an array element among the already passed entries and store the newly computed value into field asize_t (and dsp_t) without changing any other entries.

As computation of *asize* demands the call of *min_gt_div*, applied to alignment values, we need to show that its precondition holds, i.e. that alignment values are always positive. By Lemma 7.4.15 all types in the type table are valid and by Lemma 6.1.4 all valid types have positive alignment.

Inner loop Displacement of a component is computed in the same way as variable displacement inside a symbol table (Figure 8.3), with hd replaced by $\mathsf{strCmp}_t(\mathsf{cnt}_v(\mathsf{hd}))$. However, the initial value of displ is 0 in this case.

Allocated size of a component and its location inside the component list are connected according to the following lemma:

Lemma 8.2.6 Let $sc \in (nm_c \times \mathcal{T})^*$ be a structure component list with unique component names and $v \in (nm_c \times \mathcal{T})$ be a component in this list. Then displacement of v inside sc can be computed as follows:

$$unique(sc) \wedge v \in_* sc \Longrightarrow$$
$$displ(0, fst(v), sc) = \lceil asize\_sc(0, pfx((\lambda x.x \neq v), sc)) \rceil_{algn(snd(v))}$$

Thus, the essential part of the invariant is:

$$\begin{aligned}
&\exists cl, sc.\ CmpList\_ref(\mathsf{strCmp}_t(\mathsf{cnt}_v(\mathsf{hd})), \tau) = cl \wedge \\
&\qquad CmpList\_cont(\mathsf{strCmp}_t(\mathsf{cnt}_v(\mathsf{hd})), \tau) = Str(sc) \wedge \\
&\mathsf{cmps} \neq Null \longrightarrow \exists j.\ \mathsf{cmps} = cl_j \wedge \mathsf{asz} + \mathsf{dsp} = asize\_cs(0, (sc_0, \ldots, sc_{j-1})) \wedge \\
&\forall x \in_* (cl_0, \ldots, cl_{j-1}).\ \mathsf{dsp}_v(\mathsf{cnt}_v(x)) = displ(0, sc, fst(sc_j))
\end{aligned}$$

The proof that the invariant is maintained is similar to the proof given for variable displacement in Section 8.3.

When heap function dsp_v is updated for some value $\mathsf{cnt}_v(x)$ such that x belongs to some reference list cl, we need to show not only that displacement for all other components stays unchanged (based on $distinct(map(\mathsf{cnt}_v, cl))$), but also that values in the other component lists have not been rewritten. That proof exploits the disjointness of any variable list (global or local) and structure component lists of the type table (stated in abstraction function $Typetable$).

8.3 Variable Displacement Computation

The displacement computation for variables is performed after the type parameters are computed, i.e. it starts working with a data structure described by the $FTypetable$ abstraction function.

Displacement computation is done by the procedure $vars\_allocation$ the source code of which is presented in Figure 8.3. It takes a pointer to a variable list and an initial offset (size of a frame header) as parameters and returns the allocated size of the given symbol table. Since the computation for an element depends on values of displacement and allocation size of the previous one, which are kept in the variables displ and asz respectively, we initialize these with the header size and 0. To describe the result of this procedure execution we introduce a wrapping abstraction function.

Definition 8.3.1 Let σ be a state, p be a reference, t point to the type table, $vd \in Ref \rightarrow \mathbb{N}$ be a heap function, and $ofs \in \mathbb{N}$ be a number. Then the abstraction function

$$\begin{aligned}
&FVarList(p, t, ofs, \sigma, dl, vl) \equiv \\
&VarList(p, t, \sigma, dl, vl) \wedge \\
&\forall i < |dl|.\ \mathsf{dsp}_v(\mathsf{cnt}_v(dl_i)) = displ(ofs, snd(vl_i), vl)
\end{aligned}$$

defines p as a pointer to a variable list with computed displacement for its variables.

```
vars_allocation ( hd , start | res_nat) =
displ := start ;
asz := 0 ;
WHILE hd $\neq$ Null DO
    tmp := hd $\rightarrow$ cnt $\rightarrow$ ty;
    displ := CALL min_gt_div(displ + asz, tmp $\rightarrow$ align);
    hd $\rightarrow$ cnt $\rightarrow$ dsp := displ ;
    asz := tmp $\rightarrow$ asize ;
    hd := hd $\rightarrow$ nxt
OD ;
res_nat := displ + asz
```

$hd : varT\_list*;\ \ start, displ, asz, res\_nat : nat;\ \ tmp : typeT*$

Figure 8.3: Implementation of the procedure computing variable displacement

In the proof of the procedure correctness we use an auxiliary lemma which follows from the construction of a variable list. It states that for any pointer from such a list the corresponding type field (ty_v) points to a filled type entry (if the allocated size and alignment were computed before):

Lemma 8.3.2

$$\begin{aligned}&FTypetable(t, \sigma, tdl, tt) \wedge VarList(p, t, \sigma, vdl, vl) \wedge i < |vdl|\\ &\Longrightarrow\\ &\mathsf{align}_t(\mathsf{ty}_v(\mathsf{cnt}_v(vdl_i)))) = algn(snd(vl_i)) \wedge\\ &\mathsf{asize}_t(\mathsf{ty}_t(\mathsf{cnt}_v(vdl_i)))) = asize(snd(vl_i))\end{aligned}$$

This lemma can be easily proven since by definition of $VarList$ all pointers from $map(\mathsf{ty}_v \bullet \mathsf{cnt}_t, vdl)$ point to type entries from the $map(cnt_t, tdl)$ list, which are abstracted to the second component of the abstract representation of a variable.

The specification of the procedure is the following:

Theorem 8.3.3 For any program state σ, containing the initialized type table and where variable vars points to a variable list, after execution of procedure $vars\_allocation$, for all elements from this list the field dsp_v contains the value of the displacement for variables they describe:

$$\begin{aligned}&\forall \sigma, t, tdl, tt, vdl, vl.\ \Gamma \vdash\\ &\{\ \sigma \mid FTypetable(t, \sigma, tdl, tt) \wedge VarList(\mathsf{hd}, t, \sigma, vdl, vl) \wedge\\ &\qquad \forall x \in_* map(\mathsf{cnt}_t, tdl).\ \mathsf{id}_t(x) = STR \longrightarrow\\ &\qquad\qquad\qquad \{CompList\_ref(\mathsf{strCmp}_t(x), \sigma)\} \cap \{vdl\} = \varnothing\ \}\\ &\mathsf{res\_nat} := \mathsf{CALL}\ \ \mathsf{vars\_allocation(hd,\ start)}\\ &\{\sigma' \mid FVarList(\mathsf{hd}^\sigma, t, \mathsf{start}^\sigma, \sigma', vdl, vl) \wedge \mathsf{res\_nat} = asize_{ST}(\mathsf{start}^\sigma, vl) \wedge\\ &\qquad FTypetable(t, \sigma', tdl, tt) \wedge\ \forall x \notin_* map(\mathsf{cnt}_v, vdl).\ \mathsf{dsp}_v(x) = \mathsf{dsp}_v^\sigma(x)\}\end{aligned}$$

To be sure that computing variable displacement does not change the values for component displacement in the type table, that share the same heap function dsp_v, the precondition contains the disjointness condition that must be shown when calling this procedure. In the postcondition we have not only the statement that the displacement is computed, but also that the heap function dsp_v is changed only for this particular list (so we do not affect any other variable lists) and we do not destroy the type table. Of course, the latter does not necessarily need to be in the postconditions, since it can be shown from the last conjuncts from both the pre- and postconditions, but it is more efficient to include it here and do not show it every time when the procedure is invoked.

Proof: The loop invariant is formulated according to the same principle as in Theorem 8.2.1. Let τ be any state during the loop execution. The invariant $I(\tau)$ of the loop is the following:

$$\begin{array}{ll}
 & P(\tau) \wedge \\
(I_1) & (\mathsf{hd} \neq \mathit{Null} \longrightarrow \exists i.\ \mathsf{hd} = \mathit{vdl}_i \wedge \\
 & \quad \lceil \mathsf{displ} + \mathsf{asz} \rceil_{\mathsf{align}_t(\mathsf{ty}_v(\mathsf{cnt}_v(\mathsf{hd})))} = \mathit{displ}(\mathsf{start}^\sigma, \mathit{vl}, \mathit{fst}(\mathit{vl}_i)) \wedge \\
 & \quad \forall x \in_* (\mathit{vdl}_0, \ldots, \mathit{vdl}_{i-1}).\ \mathsf{dsp}_v(\mathsf{cnt}_v(x)) = \mathit{displ}(\mathsf{start}^\sigma, \mathit{vl}, \mathit{fst}(\mathit{vl}_i)) \wedge \\
(I_2) & (\mathsf{hd} = \mathit{Null} \longrightarrow (\forall x \in_* \mathit{vdl}.\ \mathsf{dsp}_v(\mathsf{cnt}_v(x)) = \mathit{displ}(\mathsf{start}^\sigma, \mathit{vl}, \mathit{fst}(\mathit{vl}_i))) \wedge \\
 & \quad \mathsf{displ} + \mathsf{asz} = \mathit{asize}_{ST}(\mathsf{start}^\sigma, \mathit{vl})\ ) \wedge \\
(I_3) & \forall x \notin_* \mathit{map}(\mathsf{cnt}_t, \mathit{vdl}).\ \mathsf{dsp}_v(x) = \mathsf{dsp}_v^\sigma(x)
\end{array}$$

Goal 1. The precondition must imply the invariant in the state before entering the loop. Formally,

$$P(\sigma) \Longrightarrow I(\sigma[\mathsf{displ} := \mathsf{start}, \mathsf{asz} := 0])$$

since before we enter the loop variables hd and displ are set to their initial values.

Preconditions P included in I are directly implied. By definition of $VarList$ we have that hd points to a non-empty list of references vdl. Thus, by Lemma 7.1.2 we conclude $\mathsf{hd} = \mathit{vdl}_0$ and $\mathsf{hd} \neq \mathit{Null}$. Thus, to finish the goal we only need to show part I_1 (I_3 is trivial). We instantiate the invariant with $i = 0$ and to complete the goal we need to show that the values of variables displ and asz before the first loop execution satisfy the invariant:

$$\lceil \mathsf{start} + 0 \rceil_{\mathsf{align}_t(\mathsf{ty}_v(\mathsf{cnt}_v(\mathsf{hd})))} = \mathit{displ}(\mathsf{start}, \mathit{vl}, \mathit{fst}(\mathit{vl}_0))$$

Expanding the definition of *displ* (Definition 6.1.6) with the second case (since vl_0 is the head of vl) we have to prove the equality: $\mathsf{align}_t(\mathsf{ty}_v(\mathsf{cnt}_v(\mathsf{hd}))) = \mathit{algn}(\mathit{snd}(\mathit{vl}_0))$, which follows with Lemma 8.3.2.

Goal 2. Since procedure *min_gt_div* is invoked, we need to show that its second parameter is greater than zero. By definition of $VarList$ and Var the type pointer of the current variable belongs to $\mathit{map}(\mathsf{cnt}_t, \mathit{dl})$ and hence, its conversion to the intermediate form belongs to tt, therefore further conversion to $\mathcal{T}$ is valid according to Lemma 7.4.15 and its alignment is positive.

According to the procedure implementation we update: i) hd with the pointer stored in its next field; ii) heap function dsp_v and variable displ with the same value, which is the result of procedure *min_gt_div*; iii) and variable *asz* with the allocated size of the current variable. Thus, we have the following goal to show:

$$\begin{aligned}
&I(\tau) \wedge \mathsf{hd} \neq Null \Longrightarrow \\
&I(\tau[\mathsf{hd} := \mathsf{nxt}_v(\mathsf{hd}), \\
&\quad \mathsf{dsp}_v := \mathsf{dsp}_v[\mathsf{cnt}_v(\mathsf{hd}) := \lceil \mathsf{displ} + \mathsf{asz} \rceil_{\mathsf{align}_t(\mathsf{ty}_v(\mathsf{cnt}_v(\mathsf{hd})))}], \\
&\quad \mathsf{displ} := \lceil \mathsf{displ} + \mathsf{asz} \rceil_{\mathsf{align}_t(\mathsf{ty}_v(\mathsf{cnt}_v(\mathsf{hd})))}, \\
&\quad \mathsf{asz} := \mathsf{asize}_t(\mathsf{ty}_v(\mathsf{cnt}_v(\mathsf{hd})))])
\end{aligned}$$

The principle of the case distinction is the same as in Goal 2 of Theorem 8.2.1. Let us consider the proof of the case presented by part I_1 of the invariant (i.e. $\mathsf{hd} \neq last(vdl)$) first. From the assumption we have that there exists i satisfying the invariant; again we instantiate the existential quantifier in the claim with $i+1$.

To show that the displacement is computed correctly we need to prove the following statement (displ and asz are replaced with the new values):

$$\begin{aligned}
&\lceil \lceil \mathsf{displ} + \mathsf{asz} \rceil_{align_i} + asize_i \rceil_{align_{i+1}} = displ(\mathsf{start}^\sigma, vl, fst(vl_{i+1})), \\
&\text{where } align_i = \mathsf{align}_t(\mathsf{vtype}_v(\mathsf{var}_v(\mathsf{hd}))), \\
&\qquad align_{i+1} = \mathsf{align}_t(\mathsf{ty}_v(\mathsf{cnt}_v(\mathsf{nxt}_v(\mathsf{hd})))), \\
&\qquad asize_i = \mathsf{asize}_t(\mathsf{ty}_v(\mathsf{cnt}_v(\mathsf{hd})))
\end{aligned}$$

From $I(\tau)$ we have that $\lceil \mathsf{displ} + \mathsf{asz} \rceil_{align_i}$ is equal to the displacement of i-th variable: $displ(\mathsf{start}^\sigma, vl, fst(vl_i))$. Applying this observation to the claim above we have:

$$\lceil displ(\mathsf{start}^\sigma, vl, fst(vl_i)) + asize_i \rceil_{align_{i+1}} = displ(\mathsf{start}^\sigma, vl, fst(vl_{i+1}))$$

It is true according to Lemma 6.1.7, since by definition of $VarList$ we have $unique(vl)$, which satisfies the lemma's assumption.

The rest of I_1 follows from $I(\tau)$ and disjointness of elements in the variable list analogously to the before described theorems.

Since the value of heap function dsp_v is only changed for $\mathsf{cnt}_v(vdl_i)$ which obviously belongs to vdl, the last part of invariant (I_3) also holds.

Goal 3. $I(\sigma') \wedge \mathsf{hd} = Null \Longrightarrow Q(\sigma')$

From I_3 and disjointness of the type table component lists with the proceeding variable list we can conclude that the type table has not changed during execution of the procedure.

The rest follows directly with the invariant and definition of $FVarList$. □

8.4 Expression Code Generation

The global scheme of the implementation for the expression code generation is presented in Figure 8.4, where the components are:

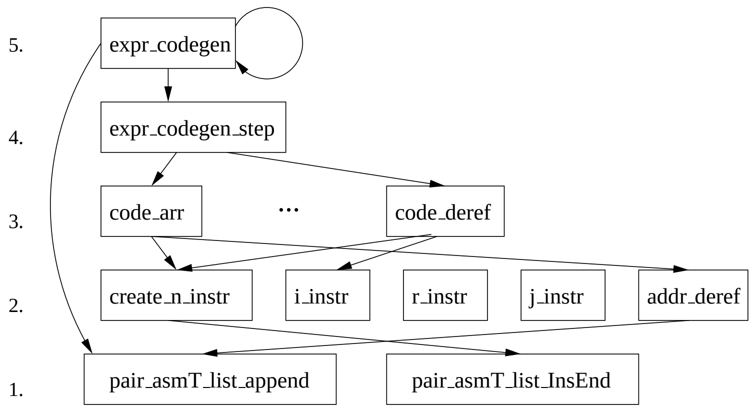


Figure 8.4: Expression code generation implementation

1. Procedures on lists: append and insert (implemented in the procedures *pair_asmT_list_append* and *pair_asmT_list_InsEnd*), used to construct instruction lists from existing lists and single assembler instructions.

2. Auxiliary procedures: generating a single instruction, address dereferencing procedure etc.

3. Procedures generating code templates for every type of expression (type of nodes of the expression syntax tree) without connecting the code for subexpressions. These use procedures from the first group.

4. Combining procedures from 3. into a procedure *expr_codegen_step* providing "one step" of code generation, i.e. code for one node of an expression syntax tree.

5. The recursive procedure *expr_codegen* which takes a pointer to an expression syntax tree as argument and generates the assembler code for the whole expression.

From the implementation scheme it is clear, that the result of procedure *expr_codegen* should be equivalent to function $code_{\mathcal{E}}$. Single procedures generating code for one node of an expression syntax tree depending on the node type also have an obvious counter part in the specification. However, the procedure *expr_codegen_step*, which combines the procedures for different expression types together, does not have a counter part in the specification.

Moreover, the recursion scheme is differently constructed in the specification and the implementation. In the former we define cases for every constructor of inductive type $\mathcal{E}$ and apply the recursive function invocation to subexpressions. In the latter we do case distinction according to the number of subtrees of the C0 data structure (none, one or two) invoking the procedure recursively for the subtrees. Recall, that data for constant representation are also differently organized. Thus, we have a different structure for procedure *expr_codegen* and function $code_{\mathcal{E}}$, what obviously makes proving the equivalence more difficult.

The natural question appearing here is: why do we not write the implementation in such a way, that it would be as close to the specification as possible. It is obvious, that having the implementation written in that way the equivalence would be much easier to show. However, to repeat the structure of the code generation in the way it is done in the specification would be highly ineffective, since in the implementation we intend to reuse the existing code as much as possible to keep the size of the final program small. In the specification, though, we have to use the same pattern for the register distribution dealing with different expression types again and again. It is inevitable, since we define the code generation function recursively for each constructor of $\mathcal{E}$ type.

The second point is that we want to show, that verification of the existing real application, in the form as it is written, is possible. Of course, at some points, where changing of the implementation has minor influence to the structure of the code and its effectiveness and simultaneously makes the verification sufficiently easier, changing the implementation is reasonable.

In the following sections we show some examples of verification of expression code generation according the diagram presented in Figure 8.4.

8.4.1 Append and Insert for Pairs

As we mentioned in Section 7.10 we keep pointers to both the first and the last element of the list to have a possibility for the fast append operation and inserting a new element to the end of a list of assembler instructions.

In this section we present a specification of procedures implementing these operations. We provide the procedures as templates, i.e. they can be reused for any structure type organizing double linked list. Let it include fields nxt_x and prv_x, where x can be parameterized.

Insert The procedure *pair_T_InsEnd* takes a variable x of a structure type T with a pair of pointers and a length field (e.g. *asmT_pair*), i.e. pointers to the first and last elements and the length of a list, and a pointer p to an element to be inserted as parameters. The output is variable res of the same type as x. The results of the program execution are given by the following theorem.

Theorem 8.4.1 Let references $\mathsf{x\_head}, \mathsf{x\_last} \in \mathit{Ref}$ organize a reference list dl through heap functions $\mathsf{nxt}_x, \mathsf{prv}_x \in \mathit{Ref} \to \mathit{Ref}$ and $\mathsf{x\_length}$ be equal to its length in state σ. If pointer p does not belong to dl then after the procedure is executed the pointers $\mathsf{res\_head}$ and $\mathsf{res\_last}$ point to the list where p is inserted at the end of dl:

$$\begin{aligned}
&\forall \sigma, dl.\ \Gamma \vdash \{\sigma \mid \mathit{dList}(\mathsf{x\_head}, \mathsf{nxt}_x, \mathsf{prv}_x, \mathsf{x\_last}, \mathit{dl}) \wedge \mathsf{x\_length} = |\mathit{dl}| \wedge \mathsf{p} \notin_* \mathit{dl}\}\\
&\textbf{res} := \mathsf{CALL}\ \ \mathsf{pair\_T\_InsEnd}\ (\ \mathbf{x},\ \mathsf{p})\\
&\{\sigma' \mid (\mathsf{p}^\sigma \neq \mathit{Null} \longrightarrow \mathit{dList}(\mathsf{res\_head}, \mathsf{nxt}_x, \mathsf{prv}_x, \mathsf{res\_last}, \mathit{dl} \circ \mathsf{p}^\sigma) \wedge\\
&\qquad\qquad \mathsf{res\_length} = |\mathit{dl}| + 1 \wedge\\
&\qquad\qquad \forall y \notin (\{\mathit{dl}\} \cup \{\mathsf{p}^\sigma\}).\ \mathsf{nxt}_x(y) = \mathsf{nxt}_x^\sigma(y) \wedge \mathsf{prv}_x(y) = \mathsf{prv}_x^\sigma(y)) \wedge\\
&\qquad (\mathsf{p}^\sigma = \mathit{Null} \longrightarrow \mathit{dList}(\mathsf{res\_head}, \mathsf{nxt}_x, \mathsf{prv}_x, \mathsf{res\_last}, \mathit{dl}) \wedge\\
&\qquad\qquad \mathsf{res\_length} = |\mathit{dl}| \wedge \mathsf{nxt}_x = \mathsf{nxt}_x^\sigma \wedge \mathsf{prv}_x = \mathsf{prv}_x^\sigma)\}
\end{aligned}$$

We also specify the case when nothing is actually changed, since the variables are considered as possibly changed by the system.

The statement about the heap functions nxt_x and prv_x in the first part of the postcondition is stronger than necessary, since the values of the heap functions only change for the last element of dl and elt. However, we usually use such statements to show that a heap function did not change its value for any other list. Therefore, having the assumption about disjointness of lists it is more convenient to have this part in such a form to save the step showing that the last element of one list does not belong to the other. Moreover, the strengthening does not limit the usage of this function.

Append The procedure realizing the append operation takes variables x and y of a pair type and returns the result of the same type, which points to the list, combined from the lists pointed by x and y.

Theorem 8.4.2 Let references $\mathsf{x\_head}, \mathsf{x\_last} \in \mathit{Ref}$ point to a reference list dl and references $\mathsf{y\_head}, \mathsf{y\_last} \in \mathit{Ref}$ point to a reference list dl_1 build by the means of heap functions $\mathsf{nxt}_x, \mathsf{prv}_x \in \mathit{Ref} \rightarrow \mathit{Ref}$. Let $\mathsf{x\_length}$ and $\mathsf{y\_length}$ be equal to lengths of dl and dl_1, respectively. Let dl do not share elements with dl_1 in state σ. Then after execution of procedure $\mathsf{pair\_T\_Append}$ reference $\mathit{res\_head}$ points to the doubly linked list associated with concatenation of dl and dl_1:

$$\begin{aligned}
&\forall \sigma, dl, dl_1.\ \Gamma \vdash \{\sigma \mid \mathit{dList}(\mathsf{x\_head}, \mathsf{nxt}_x, \mathsf{prv}_x, \mathsf{x\_last}, dl) \wedge \mathsf{x\_length} = |dl| \wedge \\
&\qquad\qquad\qquad \mathit{dList}(\mathsf{y\_head}, \mathsf{nxt}_x, \mathsf{prv}_x, \mathsf{y\_last}, dl_1) \wedge \mathsf{y\_length} = |dl_1| \wedge \\
&\qquad\qquad\qquad \{dl\} \cap \{dl_1\} = \varnothing\} \\
&\mathbf{res} := \mathsf{CALL}\ \ \mathsf{pair\_T\_Append}\ (\ \mathbf{x},\ \mathbf{y}\ ) \\
&\{\sigma' | \mathit{dList}(\mathsf{res\_head}, \mathsf{nxt}_x, \mathsf{prv}_x, \mathsf{res\_last}, dl \circ dl_1) \wedge \mathsf{res\_length} = |dl| + |dl_1| \wedge \\
&\qquad \forall y \notin_* dl \circ dl_1.\ \mathsf{nxt}_x(y) = \mathsf{nxt}_x^{\sigma}(y) \wedge \mathsf{prv}_x(y) = \mathsf{prv}_x^{\sigma}(y)\}
\end{aligned}$$

The statement about changing the heap functions is also stronger than necessary: actually only the values for the last element of dl and the first element of dl_1 change. The reasons to the strengthening are the same as for the previous functions.

8.4.2 Empty Instruction List

In order to generate an instruction (or a list of them) we dynamically allocate memory. If we know the number of instructions we will allocate subsequently, it is expedient to generate an empty "container" instruction list of the necessary length (implemented by procedure 8.5) and then assign values to its element invoking procedures for single instructions (see Section 8.4.3). We allocate new elements using a loop and link them to the list. Note that in the real C0 program we allocate the instances of different types on the heap (*asmT* and *asmT_list*), whereas in the model we do not make any difference between them.

According to our model from Section 4.2.1(for the partial correctness of memory allocation) for all functions dealing with memory allocation we have that no null pointer can be allocated and the new pointer is distinct from the ones allocated earlier (Lemmas 4.2.1 and 4.2.2). Hence, we assume enough free memory

```
create_n_instr (n | **res** ) =
p_head := NULL ;
p_last := NULL ;
p_length := 0 ;
WHILE n $\neq$ 0 DO
    cmd := NEW alloc ; tmp := NEW alloc ;
    IF cmd $\neq$ NULL $\wedge$ tmp $\neq$ NULL THEN
        tmp $\rightarrow$ cnt := cmd ;
        **p**:= CALL pair_asmT_list_InsEnd(**p**, tmp) ;
        n := n - 1
    ELSE
        p_head := NULL ; n := 0
FI OD ;
**res** := **p**
```

$p, res : asmT\_pair;\ cmd : asmT*\ ;\ tmp : asmT\_list*\ ;\ n : nat$

Figure 8.5: Creating instruction list container

for running the program. The implementation actually includes the check of returning the null pointer by the new operator (in the case of lack of memory) and processing it (by not creating any list). Proving the partial correctness we do not pay any attention to this part, since it never happens according to our model.

Theorem 8.4.3 Starting in any state σ, after execution of *create_n_instr* we get two new allocated lists of references, which correspond to an instruction list container of length n.

$$\begin{aligned}
&\forall\sigma.\ \Gamma \vdash \{\sigma\}\\
&\textbf{res} := \mathsf{CALL\ create\_n\_instr(n)}\\
&\{\sigma' \mid \{\mathsf{alloc}^{\sigma}\} \subseteq \{\mathsf{alloc}\} \wedge\\
&\exists dl.\ dList(\mathsf{res\_head}, \mathsf{nxt}_a, \mathsf{prv}_a, \mathsf{res\_last}, dl) \wedge \mathsf{res\_length} = |dl| \wedge |dl| = \mathsf{n}^{\sigma} \wedge\\
&\{dl\} \subseteq \{\mathsf{alloc}\} \wedge \{\mathsf{alloc}^{\sigma}\} \cap \{dl\} = \varnothing \wedge\\
&distinct(map(\mathsf{cnt}_a, dl)) \wedge \{dl\} \cap \{map(\mathsf{cnt}_a, dl)\} = \varnothing \wedge\\
&\{map(\mathsf{cnt}_a, dl)\} \subseteq \{\mathsf{alloc}\} \wedge \{\mathsf{alloc}^{\sigma}\} \cap \{map(\mathsf{cnt}_a, dl)\} = \varnothing \wedge\\
&\forall x \in_* \mathsf{alloc}^{\sigma}.\ \mathsf{cnt}_a(x) = \mathsf{cnt}_a^{\sigma}(x) \wedge \mathsf{nxt}_a(x) = \mathsf{nxt}_a^{\sigma}(x) \wedge \mathsf{prv}_a(x) = \mathsf{prv}^{\sigma}(x)\}
\end{aligned}$$

The postcondition of the function must include the information which will be later necessary to show that the relation *AsmProg* holds. So we need to state the relations between all lists of references we work with during execution of this procedure: dl, $map(\mathsf{cnt}_a, dl)$ and alloc (see Figure 8.6). Moreover, by the last statement we state, that we do not change any instruction lists which are placed in $alloc^{\sigma}$, i.e. were allocated before; this is necessary condition for Lemmas 7.10.5, 7.10.4.

Proof: The invariant of the while loop mostly repeats the postcondition with the following changes:

- variables res_* replaced with p_*

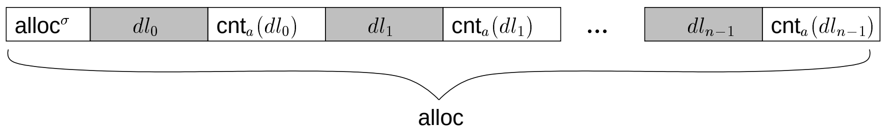


Figure 8.6: Allocation of an empty instruction list

- we add that the value of variable n is decreasing: $\mathsf{n} \leq \mathsf{n}^\sigma$
- variable $\mathsf{p\_length}$ stores value $\mathsf{n}^\sigma - \mathsf{n}$
- The values of heap functions cnt_a, nxt_a, and prv_a stay the same not only for pointers in alloc^σ but also for ones which were allocated during the previous iterations of the loop

Goal 1.

$$P(\sigma) \Longrightarrow I(\sigma[\mathsf{p\_head} := Null; \mathsf{p\_last} := Null; \mathsf{p\_length} := 0])$$

The goal can be easily shown instantiating I with $dl = []$ (we have not created any list elements yet), which turns list relation $dList(Null, \mathsf{nxt}_a, \mathsf{prv}_a, Null, dl)$ and $|dl| = 0$ to valid and satisfies all the stated set relations. Values of the heap function are initial.

Goal 2. The subgoal we need to show has the following scheme:

$$I(\tau) \wedge \mathsf{n} \neq 0 \Longrightarrow P'(\tau_1) \longrightarrow (Q'(\tau_2) \longrightarrow I(\tau_2[\mathsf{n} := \mathsf{n} - 1])),$$

where intermediate state $\tau_1 = \tau[\mathsf{alloc} := [r_2, r_1] \circ \mathsf{alloc}; \mathsf{cnt}_a := \mathsf{cnt}_a[r_1 := r_2]]$ with

$$r_1 = new(\{\mathsf{alloc}^\sigma\})$$
$$r_2 = new(\{\mathsf{alloc}^\sigma\} \cup \{r_1\})$$

P' and Q' represent the pre- and postcondition of procedure $pair\_asmT\_list\_InsEnd$ (Theorem 8.4.1) invoked within the loop. Thus, τ_2 is τ_1 updated according to the called procedure. Preconditions P' are satisfied since from $I(\tau)$ we have that $\mathsf{p\_head}$ points to some reference list dl in the memory, $\mathsf{p\_length} = |dl|$ and the element r_1 to be inserted is not in dl, since $\{dl\} \subseteq \{\mathsf{alloc}^\sigma\}$ and $r_1 \notin_* \mathsf{alloc}^\sigma$ (by Lemma 4.2.2). After we have shown that $P'(\tau_1)$ we have the modified claim:

$$I(\tau) \wedge \mathsf{n} \neq 0 \wedge Q'(\tau_2) \Longrightarrow I(\tau_2[\mathsf{n} := \mathsf{n} - 1]))$$

We instantiate the existential quantifier of the claim with the new reference list $dl \circ [r_1]$, which we have after the current iteration. The list relation $dList(\ldots)$ stated in the claim is shown with postcondition Q'. The relations between sets of pointers (pointers organizing the list and all allocated pointers) stated in the claim can be easily shown. The assumptions are taken from $I(\tau)$, postconditions

Q', and the distinctness of references r_1 and r_2 from $\{\mathsf{alloc}^\sigma\}$ and hence, from lists dl and $map(\mathsf{cnt}_a, dl)$. For example

$$\{dl \circ [r_1]\} \cap \{map(\mathsf{cnt}_a, dl \circ [r_1])\} = \varnothing$$

is equivalent to

$$\begin{array}{lr} \{dl\} \cap \{map(\mathsf{cnt}_a, dl)\} = \varnothing \wedge & (\ I(\tau)\ ) \\ r_2 \neq r_1 \wedge & (\ r_2 \notin \{\mathsf{alloc}^\sigma\} \cup \{r_1\}) \\ r_2 \notin \{dl\} \wedge & (\ r_2 \notin \{\mathsf{alloc}^\sigma\}) \\ r_1 \notin \{map(\mathsf{cnt}_a, dl)\} & (\ r_1 \notin \{\mathsf{alloc}^\sigma\}\ ) \end{array}$$

Goal 3. This is straight forward (by the definitions of I and Q).

$$I(\tau) \wedge \mathsf{n} = 0 \Longrightarrow Q(\tau[\mathsf{res\_*} := \mathsf{tmp\_*}])$$

The postconditions of this procedure provide enough information to show that all the relations between the involved reference lists stated in the abstraction function *AsmProg* hold.

8.4.3 Single Instruction

It is clear that we need to have certain information about types used in a program when we prove total correctness, where we have to show the absence of interrupts. For example, to show that array boundary violation has not happened, we need to know the number of elements in the array. Recall, that information about the sizes of array types included in the program gets lost during the translation of programs to Hoare Logic representation.

We use the small procedure whose code is depicted in Figure 8.7 to demonstrate where such a kind of information can be needed proving the partial correctness.

This procedure takes an "empty" data structure for an instruction (e.g. an element of the container created by *create_n_instr*) and "fills" it with a content forming an I-type instruction.

```
i_instr (tmp, ird, irs1, iimm, c1, c2, c3, c4 | res_int) =
tmp → id := 0 ;
tmp → opc [0] := c1 ;
tmp → opc [1] := c2 ;
tmp → opc [2] := c3 ;
tmp → opc [3] := c4 ;
tmp → rd := ird ;
tmp → rs1 := irs1 ;
tmp → imm := iimm ;
res_int := 0
```

Figure 8.7: Procedure filling an instruction of I-type

As lists in Isabelle/HOL are inductively constructed and there is no subtyping (i.e. we cannot declare a type of lists with n elements) we need to have an additional assumption about the length of the array opc, which is part of the type information in C0 and lost in the state space representation for the verification environment.

Theorem 8.4.4 The procedure *i_instr* is specified by:

$$\begin{gathered}\forall\sigma.\{\sigma \mid \forall x.|\mathsf{opc}_a(x)| = 4\}\\ \mathsf{res\_int}{:=}\ \mathsf{CALL\ i\_instr\ (tmp,\ ird,\ irs1,\ iimm,\ c1,\ c2,\ c3,\ c4)}\\ \{\sigma' \mid \mathsf{id}_a(p) = 0 \wedge \mathsf{opc}_a(p) = (\mathsf{c1}^\sigma, \mathsf{c2}^\sigma, \mathsf{c3}^\sigma, \mathsf{c4}^\sigma) \wedge\\ \mathsf{rd}_a(p) = \mathsf{ird}^\sigma \wedge \mathsf{rs1}_a(p) = \mathsf{irs1}^\sigma \wedge \mathsf{imm}_a(p) = \mathsf{iimm}^\sigma \wedge\\ (\forall x \neq p, h_a.\ h_a^\sigma(x) = h_a(x)) \wedge \forall x.|\mathsf{opc}_a(x)| = 4\},\\ \text{where } p = \mathsf{tmp}^\sigma\end{gathered}$$

Without having the statement about the array length in the precondition, we only could argue about the update of the first four elements of list $\mathsf{opc}_a(\mathsf{tmp}^\sigma)$, otherwise the proof becomes trivial. Hence, all procedures creating lists of instructions are needed to be provided with this information in preconditions. Moreover, they need to imply it in the postconditions as well in order to transfer this property when having sequential creation of several code pieces. To prove this postcondition is no burden.

There are similar procedures for the other types of instructions.

8.4.4 Short Instruction List

There are several procedures that create some predefined templates with the size of several assembler instructions (e.g. address dereference, comparison of unsigned integers or code generation for type casting). Then a procedure realizing this template consists of: i) creating an empty instruction list of the necessary length and ii) filling the content of the instruction data structure using procedures for single instructions of different types (see Section 8.4.3).

In Figure 8.8 the approximate scheme of a procedure creating a short template is depicted. Procedure name $i[r,j]$*_instr* means that either *i_instr*, or *r_instr*, or *j_instr* is called here. k is the number of instructions in it. Parameters of procedures forming a single instruction opc_i, rd_i, $\mathsf{rs1}_i$, $\mathsf{rs2}_i$, sa_i, imm_i can be constants or input parameters of the template procedure and must code an abstract program prg.

The common specification scheme looks then as follows:

$\forall\sigma.\ \Gamma \vdash \{\sigma \mid P(\sigma)\}$

$\mathbf{res}{:=}\ \mathsf{CALL\ template\_short(...)}$

$\{\sigma' \mid \exists dl.\ AsmProg(\mathsf{res\_head}, \mathsf{res\_last}, \mathsf{res\_length}, \sigma', dl, prg) \wedge Aux(\mathsf{alloc}^\sigma, dl)\}$

Predicate P defines properties of the input (if necessary), e.g. $\forall x.|\mathsf{opc}_a(x)| = 4$ etc. The postcondition must state the creation of the new program piece in the memory disjoint from the previous ones and keeping any instructions created before unchanged (through heap functions equality). Thus, with $Aux(\mathsf{alloc}^\sigma, dl)$ we denote the following statement:

$$\begin{gathered}\mathsf{alloc}^\sigma \subseteq \mathsf{alloc} \wedge \{\mathsf{alloc}^\sigma\} \cap \{dl\} = \varnothing \wedge \{\mathsf{alloc}^\sigma\} \cap \{map(\mathsf{cnt}_a, dl)\} = \varnothing \wedge\\ (\forall x \in_* \mathsf{alloc}^\sigma, h_a.\ h_a(x) = h_a^\sigma(x)) \wedge (\forall x.|\mathsf{opc}_a| = 4)\end{gathered}$$

```
template_short (...| **res**) =
**p** := CALL create_n_instr($k$);
tmp := p_head ;
void := CALL i[r,j]_instr(tmp→cnt, rd$_0$, rs1$_0$, rs2$_0$, sa$_0$, imm$_0$, opc$_0$) ;
tmp := tmp→nxt;
...
void := CALL i[r,j]_instr(tmp→cnt, rd$_{k-1}$, rs1$_{k-1}$, rs2$_{k-1}$, sa$_{k-1}$, imm$_{k-1}$, opc$_{k-1}$)
;
**res** := **p**
```

$p, res : asmT\_pair;\ tmp : asmT\_list*;\ void : int$

Figure 8.8: Procedure realizing a short template of assembler instructions

```
**add_code_deref** (d, s, **p** | **res**) =
**tmp**:= CALL create_n_instr(1);
IF s = 1 THEN
    void := CALL i_instr(tmp_head→cnt, d, d, 0, 'l', 'b', 'u', ' ') ;
ELSE IF s = 2 THEN
        void := CALL i_instr(tmp_head→cnt, d, d, 0, 'l', 'h', 'u', ' ') ;
    ELSE
        void := CALL i_instr(tmp_head→cnt, d, d, 0, 'l', 'w', 'u', ' ') ;
FI FI;
**p**:= CALL pair_asmT_list_InsEnd(**p**, tmp_head) ;
**res**:= **p**
```

$p, res, temp : asmT\_pair;\ void : int$

Figure 8.9: Address dereferencing procedure

The proof is based on the following observations. After *create_n_instr* is invoked we get a empty container for the instruction of the necessary length which provides all the properties of the *AsmProg* abstraction function except for instruction coding. Then after each invocation of procedure $i[r,j]\_instr$ we get an intermediate state σ_i , where $\mathsf{tmp} = dl_i$ and the relation $ref2instr(\mathsf{cnt}_a(dl_i), \sigma_i, prg_i)$ holds. Moreover, from the disjointness of elements of $map(\mathsf{cnt}_a, dl)$ and postcondition $\forall x \neq \mathsf{tmp}^{\sigma_{i-1}}, h_a.\ h^{\sigma}_a(x) = h_a(x)$, that we get from the specification of the singe instruction procedure, we can conclude that for all instructions created in steps $0 < m < i$ the relation $ref2instr(\mathsf{cnt}_a(dl_m), \sigma_i, prg_m)$ also holds. After the last instruction procedure is called we get $i = k$ and the last observation implies the instruction coding part in *AsmProg* in the postconditions.

8.4.5 Address Dereferencing Procedure Example

In this section we give an example of generating a small template, which differs slightly from the scheme described above, and show some of the procedures specified before in combination.

The procedure *add_code_deref* implements the function drf (Definition 6.2.1) and attaches the generated code to any instruction list, which is given by procedure parameter p (see Figure 8.9).

Theorem 8.4.5 Let σ be a program state, where variable p of type *asmT* points to a list of instructions not located in $al \in Ref^*$. Variable d contains the number of the destination and register for address dereferencing. Variable s contains the width of the access. Then

$$\begin{aligned}\forall \sigma, dl, prg, al. \Gamma \vdash \{\sigma \mid\ & AsmProg(\mathsf{p\_head}, \mathsf{p\_last}, \mathsf{p\_length}, \sigma, dl, prg) \wedge \\ & \{al\} \subseteq \{\mathsf{alloc}\} \wedge \{al\} \cap \{dl\} = \varnothing \wedge \\ & \{al\} \cap \{map(\mathsf{cnt}_a, dl)\} = \varnothing \wedge \forall x.\ |\mathsf{opc}_a(x)| = 4\} \\ \mathbf{res}{:=}\ & \mathsf{CALL\ add\_code\_deref(d,\ s,\ \mathbf{p})} \\ \{\sigma' \mid\ & \exists dl_1.\ AsmProg(\mathsf{res\_head}, \mathsf{res\_last}, \mathsf{res\_length}, \sigma', dl_1, prg \circ drf(\mathsf{d}^\sigma, \mathsf{s}^\sigma)) \wedge \\ & Aux(al, dl_1)\}\end{aligned}$$

Proof: Since the procedure does not include any loops we proceed consecutively.

1. First we need to show that the preconditions of *create_n_instr* are satisfied. Since it has trivial preconditions, it can be applied to any state. As the result of its execution we get some intermediate state σ_1 with properties described by the postconditions stated in Theorem 8.4.3 (we denote them by $Q_1(\sigma_1)$): σ_1 contains a "container" list with one element, based on some reference list ql, which belongs to $\mathsf{alloc}^{\sigma_1}$ and does not belong to alloc^{σ}.

2. Then the proof is split into three cases according to the *IF* statements. Let us consider the first one, namely $\mathsf{s}^\sigma = 1$. After invoking of *i_instr* we get another intermediate state σ_2 satisfying the postconditions stated by Theorem 8.4.4(denoted by Q_2).

3. Finally we invoke the insert operation on the initial (dl) list and reference tmp_head, so we need to show that the preconditions of *pair_asmT_list_InsEnd* hold for state σ_2.

The list relation $dList(\mathsf{p\_head}^{\sigma_2}, \mathsf{nxt}_a^{\sigma_2}, \mathsf{prv}_a^{\sigma_2}, \mathsf{p\_last}^{\sigma_2}, dl)$ is valid with the following sequence of observations:

Precondition $AsmProg(\mathsf{p\_head}^\sigma, \mathsf{p\_last}^\sigma, \mathsf{p\_length}^\sigma, \sigma, dl, prg)$ implies by definition $dList(\mathsf{p\_head}^\sigma, \mathsf{nxt}_a^\sigma, \mathsf{prv}_a^\sigma, \mathsf{p\_last}^\sigma, dl)$ and $\mathsf{p\_length}^\sigma = |dl|$. From postcondition $Q_1(\sigma_1)$ we have

$$\forall x \in_* \mathsf{alloc}^\sigma.\ \mathsf{cnt}_a^{\sigma_1}(x) = \mathsf{cnt}_a^\sigma(x) \wedge \mathsf{nxt}_a^{\sigma_1}(x) = \mathsf{nxt}_a^\sigma(x) \wedge \mathsf{prv}_a^{\sigma_1}(x) = \mathsf{prv}_a^\sigma(x). \quad (8.7)$$

From (8.7) and $\{dl\} \subseteq \{\mathsf{alloc}^\sigma\}$ included in *AsmProg* we conclude that $\forall x \in_* dl.\ \mathsf{nxt}_a^{\sigma_1}(x) = \mathsf{nxt}_a^\sigma(x) \wedge \mathsf{prv}_a^{\sigma_1}(x) = \mathsf{prv}_a^\sigma(x)$. Thus, as variable **p** stays the same in σ_1, by Lemma 7.1.7 we get $dList(\mathsf{p\_head}^{\sigma_1}, \mathsf{nxt}_a^{\sigma_1}, \mathsf{prv}_a^{\sigma_1}, \mathsf{p\_last}^{\sigma_1}, dl)$.

Since the values of fields nxt and prv are the same in σ_1 and σ_2 (not changed by singe instruction procedures), relation $dList(\mathsf{p\_head}^{\sigma_2}, \mathsf{nxt}_a^{\sigma_2}, \mathsf{prv}_a^{\sigma_2}, \mathsf{p\_last}^{\sigma_2}, dl)$ also holds.

Let us show the last precondition of the inserting procedure. From postcondition $Q_1(\sigma_1)$ we have that there exists some reference list dl', such that relation $dList(\mathsf{tmp\_head}^{\sigma_1}, \mathsf{nxt}_a^{\sigma_1}, \mathsf{prv}_a^{\sigma_1}, \mathsf{tmp\_last}^{\sigma_1}, dl')$ holds, and moreover, this list is disjoint with references allocated earlier: $\{dl'\} \cap \{\mathsf{alloc}^{\sigma}\} = \varnothing$. Since we have $\mathsf{tmp\_head}^{\sigma_1} = dl'_0$ (by Lemma 7.1.2), then we conclude $\mathsf{tmp\_head}^{\sigma_1} \notin_* \mathsf{alloc}^{\sigma}$. That obviously implies that $\mathsf{tmp\_head}^{\sigma_2} \notin_* dl$ (recall $\{dl\} \subseteq \{\mathsf{alloc}^{\sigma}\}$).

4. Thus, after the inserting procedure is called, we have some state σ_3 with properties stated by Theorem 8.4.1. Since $\mathsf{tmp\_head}^{\sigma_2} \neq Null$ (can be shown from $|dl'| = 1$), we have $dList(\mathsf{p\_head}^{\sigma_3}, \mathsf{nxt}_a^{\sigma_3}, \mathsf{prv}_a^{\sigma_3}, \mathsf{p\_last}^{\sigma_2}, dl \circ dl')$.

We instantiate the existence quantifier in $AsmProg$ predicate included in the claim with $dl \circ dl'$.

The part of the abstraction function $AsmProg$

$$\forall i < |dl \circ dl'|.\ ref2instr(\mathsf{cnt}_a^{\sigma_3}((dl \circ dl')_i), \sigma_3, prg \circ drf(\mathsf{d}^{\sigma}, \mathsf{s}^{\sigma}))$$

follows with i) $AsmProg$ in the theorem preconditions since heap functions h_a have not changed for any reference in dl; ii) postcondition $Q_2(\sigma_2)$ which implies $ref2instr(\mathsf{cnt}_a^{\sigma_2}(\mathsf{tmp}^{\sigma_2}), \sigma_2, drf(\mathsf{d}^{\sigma}, \mathsf{s}^{\sigma})_0)$; iii) only nxt and prv is changed in σ_3.

The rest of the claim (inner sets relations from $AsmProg$ and $Diff$) can be shown by arguing on sets having inclusion

$$al \subseteq \mathsf{alloc}^{\sigma} \subseteq \mathsf{alloc}^{\sigma_1} = \ldots = \mathsf{alloc}^{\sigma_3}$$

and disjointness of sets, heap function updates etc. stated by postconditions $Q_1 - Q_3$ (similar to a case considered in the proof of Theorem 8.4.3). That is of no mathematical interest but pure mechanical work similar to the case considered in the proof of Theorem 8.4.3. □

8.4.6 Long Instruction List

To generate code for some expressions we use fixed templates with a large number of instructions (e.g. emulation of multiplication and division instructions, which are not implemented in the hardware). Filling in the container list by consecutive invocations of the large number of single instruction procedure in such a case is ineffective not only with respect to execution (we lose the time and memory space creating new procedure stacks) but also to verification of such procedures. In this case we need to prove the same things again and again, e.g. showing that each procedure invocation will not destroy any existing instruction in the memory of the program.

We can enhance the verification effort by dividing such a procedure into two parts:

- initialize an array arr of type $asmT[k]$, where k is a number of instructions in the template, at position i with values needed to be written in the i-th element in the list of instructions

- in a loop for all $0 \leq i < k$ do an assignment of $\mathsf{arr}[i]$ to the content of a list element

```
template_long (...| **res**) =
**arr** :=
{ { $\mathrm{id}_0$, $\mathrm{opc}_0$, $\mathrm{rd}_0$, $\mathrm{rs1}_0$, $\mathrm{rs2}_0$, $\mathrm{sa}_0$, $\mathrm{imm}_0$}, ...
        { $\mathrm{id}_{k-1}$, $\mathrm{opc}_{k-1}$, $\mathrm{rd}_{k-1}$, $\mathrm{rs1}_{k-1}$, $\mathrm{rs2}_{k-1}$, $\mathrm{sa}_{k-1}$, $\mathrm{imm}_{k-1}$}}
**p** := CALL create_n_instr($k$);
tmp := p_head ; n := 0 ;
WHILE n < $k$ DO
tmp→cnt → id := arr_id[n] ;
tmp→cnt → opc := arr_opc[n] ;
tmp→cnt → rd := arr_rd[n] ;
tmp→cnt → rs1 := arr_rs1[n] ;
tmp→cnt → rs2 := arr_rs2[n] ;
tmp→cnt → sa := arr_sa[n] ;
tmp→cnt → imm := arr_imm[n] ;
tmp := tmp → nxt ;
n := n - 1
OD;
**res** := **p**;
```

$arr : asmT[k];\ p, res : asmT\_pair;\ n : nat$

Figure 8.10: Long template scheme

A procedure scheme is given in Figure 8.10. In the case of a long template it actually does not matter how many lines it contains. The main goal is to show that the values used to initialize the array actually correspond to the instruction list they should code. Thus, the essential part of the loop invariant is the following:

$$\begin{aligned}
&\exists dl.\ dList(\mathsf{p\_head}, \mathsf{nxt}_a, \mathsf{prv}_a, \mathsf{p\_last}, dl) \wedge |dl| = k \wedge \\
&\mathsf{n} < k \longrightarrow \exists i.\ \mathsf{tmp} = dl_i \wedge \\
&\qquad \forall i < \mathsf{n}.\ ref2instr(\mathsf{cnt}_a(\mathsf{tmp}), \tau, prg_i) \wedge \mathsf{id}_a(\mathsf{cnt}_a(\mathsf{tmp})) = type(prg_i)
\end{aligned}$$

Analogously to specifications presented before the invariant includes relations on pointer sets, states that values in the heap functions that existed before the procedure execution are unchanged etc. Found once, the invariant can be used for every template of this kind.

One can notice that this approach can also be applied to short templates. However, it is inexpedient to apply that to the procedures we mentioned as examples of short templates because in the most cases they include conditionals. Building-in the corresponding case-splitting in the invariant takes more effort than arguing about consecutive invocations of filling procedures for several instructions.

8.4.7 One Step of Expression Code Generation

The procedures generating code for different kinds of expressions are written similar to their abstract counter parts and proven correct with approaches we have presented in the previous sections.

After we have verified procedures for every kind of expression or expression group (e.g. we have the separate procedure for all type of arithmetical and logical expression except for multiplication and division), we need to combine them together (by means of the procedure *expr_codegen_step*) and show the equivalence to some function from the specification. The structure of *expr_codegen_step* is nested conditionals on the expression identifier and the call of the corresponding code generation procedure for every case. As we mentioned at the beginning of Section 8.4, this procedure does not have an counter part in the specification, which we could use to specify it and describe the result of its execution.

To solve the problem we artificially add to the initial specification given in [1] a wrapper function $code\_step_{\mathcal{E}}$, which combines functions for different expression types generating code for one node (in the same way as it is done in the implementation). Such a function is not necessary in the specification of the compiler, but after we needed to introduce it to show the equivalence. The function parameters are the same as for $code_{\mathcal{E}}$ added by $d_1, d_2 \in \mathbb{N}$, which are numbers of the register where the results of evaluation of subexpression are stored.

$$
\begin{aligned}
&code\_step_{\mathcal{E}}(gst, lst, r, d, d_1, d_2, e) = \\
&\quad\begin{cases}
code_{\mathcal{C}}(d, v) & \text{if } e = Lit(v) \\
\dots & \dots \\
code_{op_u}(d, d_1, op) & \text{if } e = UnOp(e_1, op), \\
\dots & \dots \\
code_{stracc}(d, d_1, r, t_1, cn) & \text{if } e = StrAcc(e_1, cn)
\end{cases} \\
&\text{where } t_1 = type_{\mathcal{E}}(p, lst, e_1)
\end{aligned}
$$

Then the procedure specification looks as follows:

Theorem 8.4.6 Let in state σ global variables t, gv, lv, p point to the type table, to symbol tables, and to some expression, respectively. Then after execution of procedure *expr_codegen_step* the variable res points to the assembler code generated for the root node of the expression syntax tree.

$$
\begin{aligned}
&\forall \sigma, tdl, tt, pt, gdl, gst, ldl, lst, tr, ex.\ \Gamma \vdash \\
&\{\sigma \mid FTypetable(\mathsf{t}, \sigma, tdl, tt) \wedge \\
&\qquad FVarList(\mathsf{gv}, \mathsf{t}, \sigma, gdl, gst) \wedge FVarList(\mathsf{lv}, \mathsf{t}, \sigma, ldl, lst) \wedge \\
&\qquad Expr(\mathsf{p}, \mathsf{t}, \mathsf{gv}, \mathsf{lv}, \sigma, tr, ex) \wedge inj(ref2nm) \wedge \forall x.\ |\mathsf{opc}_a(x)| = 4\} \\
&\mathsf{res}{:=}\ \mathsf{CALL\ expr\_codegen\_step(p,\ d,\ l,\ r,\ flag)} \\
&\{\sigma' \mid \exists dl.\ AsmProg(\mathsf{res\_head}, \mathsf{res\_last}, \mathsf{res\_length}, \sigma', dl, prg) \wedge Aux(\mathsf{alloc}^{\sigma}, dl)\} \\
&\text{where } prg = code\_step_{\mathcal{E}}((type\_env(tt), gst, pt), \mathsf{d}^{\sigma}, \mathsf{l}^{\sigma}, \mathsf{r}^{\sigma}, \mathsf{flag}^{\sigma}, ex)
\end{aligned}
$$

The proof of the theorem combines specifications for every procedure included in *expr_codegen_step*.

8.4.8 Main Theorem

Free Register List We organize the numbers of free registers we use for expression evaluation by a list data structure whose content is a register number. In the part of the implementation which is presented in Figure 8.11 this list is referenced by pointer *free*. The heap functions generated for the free register list data structure are marked by $_r$.

Procedure Description The procedure *expr_codegen* takes as parameters a pointer p to the expression syntax tree we generate the code for, a pointer d to the destination register, and a flag – right/left expression flag. Figure 8.11 presents the most difficult case of procedure *expr_codegen*, the implementation of all other cases can be derived from it. The case is a non-lazy binary expression, so we need to choose the subexpression to be evaluated first.

We proceed as in the specification: choose the largest subtree and evaluate the corresponding expression first, then evaluate the remaining one, evaluate the root node, and concatenate all code pieces.

The meanings of local variables are: fi - initial value of rigth/left flag for subexpressions, sz, sz1 - sizes of the left and right subexpressions, free - global pointer to the list of free registers (i.e. an implicit parameter). e1, hd1, f1 - parameters for the first recursive evaluations (see procedure parameters description), e2, hd2, f2 - for the second one.

Let us present the most important details of the specification for the used auxiliary procedures, which we have not mentioned before.

Tree size If $p \in Ref$ points to a binary tree, then procedure *Tree_size* computes its size.

$$\forall \sigma, tr.\ \Gamma \vdash \{\sigma \mid Tree(p, l, r, tr)\}\ \mathsf{sz} := \mathsf{CALL\ Tree\_size(p)}\ \{\sigma' \mid \mathsf{sz} = |tr|\}$$

Delete list element Procedure *dList_delete* deletes the element referenced by elt from the list referenced by p (if it is located there).

$$\begin{aligned}&\forall \sigma, dl.\ \Gamma \vdash \{\sigma \mid \exists lst.\ dList(\mathsf{p}, \mathsf{nxt}, \mathsf{prv}, lst, dl)\ \}\\&\mathsf{q} := \mathsf{CALL\ dList\_delete(p,\ elt)}\\&\{\sigma' \mid (\exists i.\ \mathsf{elt}^{\sigma} = dl_i \longrightarrow\\&\qquad \exists lst.\ dList(\mathsf{q}, \mathsf{nxt}, \mathsf{prv}, lst, (dl_0, \ldots, dl_{i-1}, dl_{i+1}, last(dl)) \wedge \ldots) \wedge\\&\qquad (\mathsf{elt}^{\sigma} \notin \{dl\} \longrightarrow dList(q, \mathsf{nxt}, \mathsf{prv}, lst, dl) \wedge \ldots)\}\end{aligned}$$

Insert element at the beginning of the list The procedure *dList_InsHead* inserts the element referenced by elt to a list referenced by p (if it is not already in the list).

$$\begin{aligned}&\forall \sigma, dl.\ \Gamma \vdash \{\sigma \mid \exists lst.\ dList(p, \mathsf{nxt}, \mathsf{prv}, lst, dl) \wedge \mathsf{elt} \notin dl\ \}\\&\mathsf{q} := \mathsf{CALL\ dList\_InsHead(p,\ elt)}\\&\{\sigma' \mid \exists lst.\ dList(\mathsf{q}, \mathsf{nxt}, \mathsf{prv}, lst, (\mathsf{elt}^{\sigma}, dl)) \wedge \ldots\}\end{aligned}$$

The postconditions we have omitted in the list procedure specifications describe references for which the heap functions do not change.

The specification of *expr_codegen* mostly repeats the specification of the procedure *expr_codegen_step*. We need to add that we have a list of available registers

```
expr_codegen (p, d, flag | **res**) =
σ   IF id = 26 ∨ id = 27 THEN
... (base case - variable access or constant)
    ELSE IF id < 2 THEN
... (lazy binary operation)
    ELSE IF id < 17 ∨ id = 26 THEN
σ0     IF id = 26 THEN fi = False ELSE fi = True ;
σ1
       sz := CALL Tree_size(p → lt) ;
       sz1 := CALL Tree_size(p → rt) ;
σ2
       hd1 := free ;
       free := CALL dList_delete(free, free) ;
σ3      hd2 := free ;
    free := CALL dList_InsHead(free, d) ;
σ4
       IF sz1 < sz THEN
         e1 := p → lt ; f1 := fi; e2 := p → rt ; f2 := True ;
         d1 := hd1→cnt; d2 := hd2→cnt
       ELSE
         e1 := p → rt ; f1 := True; e2 := p → lt ; f2 := fi;
         d1 := hd2→cnt; d2 := hd1→cnt
       FI ;
σ5
       x := CALL expr_codegen (e1, hd1, f1) ;
σ6      free := CALL dList_delete(free, free) ;

       free := CALL dList_delete(free, free) ;
       free := CALL dList_InsHead(free, d) ;
ω      y := CALL expr_codegen (e2, hd2, f2) ;
ω1      free := CALL dList_delete(free, free) ;

       tmp := CALL pair_asmT_list_Append(x, y);
       z := CALL expr_codegen_step (p, d→cnt, d1, d2, flag) ;
ν      tmp := CALL pair_asmT_list_Append(tmp, z);

       free := CALL dList_InsHead(free, hd1) ;
       free := CALL dList_InsHead(free, hd2) ;
...
res := tmp;
```

$x, y, z, tmp, res : asmT\_pair;\ id, sz, sz1 : nat\ d, hd1, hd2 : regT\_list*;$
$flag, fi, f1, f2 : bool;\ p, e1, e2 : exprT * hd$

Figure 8.11: Implementation of one of the expression evaluation cases

and moreover, that they are enough to evaluate the given expression. The last assumption together with the validity of expression guarantee that the result of the code generation function is defined.

Theorem 8.4.7 Procedure *expr_codegen* is equivalent to function $code_{\mathcal{E}}$.

$$
\begin{array}{l}
\forall \sigma, tdl, tt, pt, gdl, gst, ldl, lst, tr, ex, fr.\ \Gamma \vdash \\
\{\sigma \mid FTypetable(\mathsf{t}, \sigma, tdl, tt) \wedge \\
\quad FVarList(\mathsf{gv}, \mathsf{t}, \sigma, gdl, gst) \wedge FVarList(\mathsf{lv}, \mathsf{t}, \sigma, ldl, lst) \wedge \\
\quad Expr(\mathsf{e}, \mathsf{t}, \mathsf{gv}, \mathsf{lv}, \sigma, tr, ex) \wedge \\
\quad inj(ref2nm) \wedge \forall x.\ |\mathsf{opc}_a(x)| = 4 \wedge \\
\quad \exists lst.\ dList(\mathsf{free}, \mathsf{nxt}_r, \mathsf{prv}_r, lst, fr) \wedge \\
\quad \mathsf{d} \notin_* fr \wedge enough(gst, lst, \mathsf{flag}, ex, |fr|)\} \\
\mathbf{res}{:=}\ \mathsf{CALL\ expr\_codegen(e,\ d,\ flag)} \\
\{\sigma' \mid \exists dl.\ AsmProg(\mathsf{res\_head}, \mathsf{res\_last}, \mathsf{res\_length}, \sigma', dl, prg) \wedge \\
\quad Aux(\mathsf{alloc}^{\sigma}, adl) \wedge \exists lst.\ dList(\mathsf{free}, \mathsf{nxt}_r, \mathsf{prv}_r, lst, fr)\},
\end{array}
$$

$$
\begin{array}{l}
\text{where } p = (type\_env(tt), gst, pt), \\
\qquad prg = code_{\mathcal{E}}(p, lst, \mathsf{flag}^{\sigma}, \mathsf{cnt}_r(\mathsf{d}^{\sigma}), map(\mathsf{cnt}_r, fr), ex)
\end{array}
$$

Proof: We present a rough sketch for the case presented in Figure 8.11.

Technical Note Notice, that for a conditional which is followed by other statements

$$\{\sigma \mid P(\sigma)\}\ (IF\ e\ THEN\ s_1\ ELSE\ s_2); stmts\ \{\sigma' \mid Q(\sigma')\},$$

the VCG provides a goal of the following form:

$$(P(\sigma) \wedge e(\sigma) \longrightarrow Q(\sigma_1)) \wedge (P(\sigma) \wedge \neg e(\sigma) \longrightarrow Q(\sigma_2)),$$

where σ_1 and σ_2 are σ updated according to execution of $s_1; stmts$ and $s_2; stmts$ respectively. Thus, if *stmts* includes the same pattern n times more, we will have 2^n goals to prove. In most cases they differ slightly and we need to prove similar things again and again. There is a possibility to specify an intermediate state:

$$\{\sigma \mid P(\sigma)\}\ (IF\ e\ THEN\ s_1\ ELSE\ s_2)\ \{\tau \mid P'(\tau)\}\ stmts\ \{\sigma' \mid Q(\sigma')\},$$

so that the new proof goals will be

$$
\begin{array}{l}
1.(P(\sigma) \wedge e(\sigma) \longrightarrow P'(\tau_1)) \wedge (P(\sigma) \wedge \neg e(\sigma) \longrightarrow P'(\tau_2)) \\
2.P'(\tau) \longrightarrow Q(\sigma')
\end{array}
$$

where τ_1 and τ_2 are σ after execution of s_1 and s_2 respectively, and σ' is τ updated by *stmt*.

1. We denote preconditions of the theorem by P, so the starting state σ_0 of the case equals to:

$$P(\sigma_0) \wedge \mathsf{id}_e(\mathsf{e}) \neq 27 \wedge \mathsf{id}_e(\mathsf{e}) \neq 28 \wedge \mathsf{id}_e(\mathsf{e}) > 1 \wedge (\mathsf{id}_e(\mathsf{e}\ ) < 17 \vee \mathsf{id}_e(\mathsf{e}) = 26)$$

The only heap functions changed during the execution of this part of code are all functions h_a, nxt_r, and prv_r. We omit the state mark for variables that do not change during the execution.

Following the approach we have presented above we specify an intermediate state σ_1 after the first conditional. It almost copies σ_0 with exception in variable fi, so precondition P_1 is the following:

$$P_0(\sigma_1) \wedge \mathsf{fi} = (if\ \mathsf{id}_e(\mathsf{e}) = 26\ then\ False\ else\ True).$$

Thus, we transfer the external IF from the program implementation to the internal logical if.

2. After execution of both *Tree_size* procedures we have some state σ_2. Additionally to properties stated by P_1 the σ_2 state satisfies the condition $\mathsf{sz} = |lt(tr)| \wedge \mathsf{sz1} = |rt(tr)|$.

3. As the following step we fix the register where the result of the first evaluation will be stored $\mathsf{hd1} = \mathsf{free}^{\sigma_2} = \mathsf{free}^{\sigma}$. From the assumption $enough(lst, \mathsf{flag}, ex, |fr|)$ we can conclude that $|fr| \geq 2$ (in details presented below), so $fr \neq []$ and according to Lemma 7.1.2 we have $\mathsf{hd1} = fr_0 = hd(fr)$. The preconditions of *dList_delete* are obviously satisfied by assumption $dList(\mathsf{free}^{\sigma}, \mathsf{nxt}_r^{\sigma}, \mathsf{prv}_r^{\sigma}, lst, fr)$ (we have not changed any state components involved in the relation, so it still holds in σ_2). After the execution we have state σ_3 where variable free points to the tail of fr, i.e. $dList(\mathsf{free}^{\sigma_3}, \mathsf{nxt}_r^{\sigma_3}, \mathsf{prv}_r^{\sigma_3}, lst, tl(fr))$.

4. We set a register to store the result of the next evaluated subexpression. $\mathsf{hd2} = \mathsf{free}^{\sigma_3}$. Since $|fr| \geq 2 \longrightarrow |tl(fr)| \geq 1$ we can show that $\mathsf{free}^{\sigma_3} = hd(tl(fr))$. Since the destination register is needed to be used for evaluation of subexpressions we add it in the free register list. The specification of *dList_regT_InsHead* is instantiated with σ_3 and $tl(fr)$. The first part of the precondition holds since free^{σ_3} points to the concrete version of $tl(fr)$, the second one follows with the initial precondition $\mathsf{d} \notin_* fr$ that implies $\mathsf{d} \notin_* tl(fr)$. Therefore, the result of the call is state σ_4, where $dList(\mathsf{free}^{\sigma_4}, \mathsf{nxt}_r^{\sigma_4}, \mathsf{prv}_r^{\sigma_4}, lst, (\mathsf{d}, tl(fr))\ )$.

5. After this step we again have a situation similar to the one described in paragraph 1. So we specify the new state (σ_5) after the second conditional, which satisfies all properties stated for σ_4 and moreover for variables $e1, f1, ...$ changed inside the conditional we have e.g.

$$\mathsf{e1} = (if\ \mathsf{sz1} < \mathsf{sz}\ then\ \mathsf{lt}_e(\mathsf{e})\ else\ \mathsf{rt}_e(\mathsf{e}))$$

and so on.

6. In the next step we recursively invoke the procedure for the subexpression referenced by $\mathsf{e1}$. We need to show that preconditions of the recursive call in state σ_5 are implied. All quantifiers are instantiated with the same values except for the last three, which we replace with $t_1 = (\mathit{if}\ \mathsf{sz1} < \mathsf{sz}\ \mathit{then}\ \mathit{lt}(\mathit{tr})\ \mathit{else}\ \mathit{rt}(\mathit{tr}))$, $\mathit{ref2ex}(e1, \sigma)$, and $(\mathsf{d}, \mathit{tl}(\mathit{fr}))$, respectively. The relations for the type table and both symbol tables are still valid in σ_5 (we do not assign values to any of related heap functions). Expression relation

$$\mathit{Expr}(\mathsf{e1}, \mathsf{t}, \mathsf{gv}, \mathsf{lv}, \sigma_5, t_1, \mathit{ref2ex}(e1, \sigma))$$

holds with Lemma 7.6.4. The assumptions of this lemma are true since $ex = \mathit{ref2ex}(\mathsf{p}, \sigma_5) = \mathit{ref2ex}(\mathsf{p}, \sigma)$, and pointers $\mathsf{lt}_e(\mathsf{e})$, $\mathsf{rt}_e(\mathsf{e})$ are not equal to Null (by definition of Expr for the cases of binary operations and array access). The free register list relation is inherited from σ_4. Precondition $\mathsf{d1} \notin_* (\mathsf{d}, \mathit{tl}(\mathit{fr}))$ is equivalent to

$$\begin{aligned}&\mathsf{hd1} \neq \mathsf{d} \wedge \mathsf{hd1} \notin_* \mathit{tl}(\mathit{fr}) =\\ &\mathit{hd}(\mathit{fr}) \neq \mathsf{d} \wedge \mathit{hd}(\mathit{fr}) \notin_* \mathit{tl}(\mathit{fr}) =\\ &\mathit{hd}(\mathit{fr}) \neq \mathsf{d} \wedge \mathit{distinct}(\mathit{fr})\end{aligned}$$

where the first conjunct follows from assumption $\mathsf{d} \notin_* \mathit{fr}$ and the second one by Lemma 7.1.5 applied to assumption $\mathit{dList}(\mathsf{free}^{\sigma}, \mathsf{nxt}_r^{\sigma}, \mathsf{prv}_r^{\sigma}, \mathit{lst}, \mathit{fr})$.

The last precondition to show, i.e. $\mathit{enough}(\mathit{lst}, \mathsf{f1}, \mathit{ref2ex}(e1, \sigma_5), |(\mathsf{d}, \mathit{tl}(\mathit{fr}))|)$ is valid by the following observations. Let us consider it for a case $\mathsf{lt}_e(\mathsf{e}) = 26$, where $\mathsf{sz1} < \mathsf{sz}$. Expression $\mathit{ref2ex}(p, \sigma_5)$ is equal to $\mathit{ref2ex}(p, \sigma)$, since changes of the state during the execution are irrelevant to this conversion.

$$\begin{aligned}&\mathit{enough}(\mathit{lst}, \mathsf{flag}, ex, |\mathit{fr}|) =\\ &\mathit{enough}(\mathit{lst}, \mathsf{flag}, \mathit{ArrAcc}(\mathit{ref2ex}(\mathsf{lt}_e(\mathsf{e}), \sigma), \mathit{ref2ex}(\mathsf{rt}_e(\mathsf{e}), \sigma)), |\mathit{fr}|) =\\ &\mathit{enough}(\mathit{lst}, \mathsf{flag}, \mathit{ArrAcc}(\mathit{ref2ex}(\mathsf{e2}, \sigma), \mathit{ref2ex}(\mathsf{e1}, \sigma)), |\mathit{fr}|) =\\ &|\mathit{fr}| \geq 2 \wedge \mathit{enough}(\mathit{lst}, \mathit{True}, \mathit{ref2ex}(\mathsf{e1}), \sigma)), |\mathit{fr}|) \wedge\\ &\mathit{enough}(\mathit{lst}, \mathit{False}, \mathit{ref2ex}(\mathsf{e2}), \sigma)), |\mathit{fr} - 1|)\end{aligned}$$

Since $\mathsf{f1} = \mathit{True}$ and $|(\mathsf{d}, \mathit{tl}(\mathit{fr}))| = 1 + (|\mathit{fr}| - 1) = |\mathit{fr}|$, the claim is shown.

According to the postconditions of *expr_codegen* we get state σ_6 with the generated piece of code based on some reference list dl :

$$\mathit{AsmProg}(\mathsf{x\_head}^{\sigma_6}, \mathsf{x\_last}^{\sigma_6}, \mathsf{x\_length}^{\sigma_6}, \sigma_6, \mathit{dl}, \mathit{prg}),$$

where $\mathit{prg} = \mathit{code}_{\mathcal{E}}(\mathit{lst}, \mathsf{f1}, \mathsf{hd1}, \mathit{map}(\mathsf{cnt}_r, ((\mathsf{d}, \mathit{tl}(\mathit{fr})))), \mathit{ref2ex}(\mathsf{e1}, \sigma)$. Moreover, we have the same register list $\mathit{dList}(\mathsf{free}^{\sigma_6}, \mathsf{nxt}_r^{\sigma_6}, \mathsf{prv}_r^{\sigma_6}, \mathit{lst}, (\mathsf{d}, \mathit{tl}(\mathit{fr}))\ )$, which is the necessary precondition for the following procedures working with the list.

7. The following argumentation can be done analogously to the first case, as the result of it before the second recursive call we have some state ω with $\mathit{dList}(\mathsf{free}^{\omega}, \mathsf{nxt}_r^{\omega}, \mathsf{prv}_r^{\omega}, \mathit{lst}, (\mathsf{d}, \mathit{tl}(\mathit{tl}(\mathit{fr})))\ )$.

We instantiate the specification for the second call with t_2, which is done oppositely to t_1 defined above, $\mathit{ref2ex}(\mathsf{e2}, \sigma)$, and $(\mathsf{d}, \mathit{tl}(\mathit{tl}(\mathit{fr})))$. The implication of the precondition can also be shown analogously to the first case. Thus, after the second recursive call we have some state w_1 where structure y points to the second piece of the expression evaluation code.

$$AsmProg(\mathsf{y\_head}, \mathsf{y\_last}, \mathsf{y\_length}, \omega_1, dl, prg),$$

where $prg = code_{\mathcal{E}}(lst, \mathsf{f2}, \mathsf{hd2}, map(\mathsf{cnt}_r, ((\mathsf{d}, tl(tl(fr))))), ref2ex(\mathsf{e2}, \sigma)$.

The append result for these code pieces can be shown using Lemma 7.10.5 and information provided by Aux statement in the postconditions of the recursive calls.

8. To get the result of code generation for the root node we take the specification of procedure *expr_codegen_step* (Theorem 8.4.6) with resulting program code in some state ν, where variable z points to a piece of assembler code equivalent to

$$code\_step_{\mathcal{E}}(\mathsf{cnt}_r(\mathsf{d}), \mathsf{d1}, \mathsf{d2}, \mathsf{flag}, ex).$$

Our goal is to show that the program we have after the last append, i.e.

$$\begin{aligned}&code_{\mathcal{E}}(lst, \mathsf{f1}, \mathsf{cnt}_r(\mathsf{hd1}), map(\mathsf{cnt}_r, ((\mathsf{d}, tl(fr)))), ref2ex(\mathsf{e1}, \sigma) \circ\\ &code_{\mathcal{E}}(lst, \mathsf{f2}, \mathsf{cnt}_r(\mathsf{hd2}), map(\mathsf{cnt}_r, ((\mathsf{d}, tl(tl(fr))))), ref2ex(\mathsf{e2}, \sigma) \circ\\ &code\_step_{\mathcal{E}}(\mathsf{cnt}_r(\mathsf{d}), \mathsf{d1}, \mathsf{d2}, \mathsf{flag}, ex)\end{aligned}$$

is equal to $code_{\mathcal{E}}(lst, \mathsf{flag}, \mathsf{cnt}_r(\mathsf{d}), map(\mathsf{cnt}_r, fr), ex)$. We prove it by case distinction. Let us consider $\mathsf{id}_e(\mathsf{e}) = 26$ and $\mathsf{sz1} < \mathsf{sz}$. By definition of $Expr$ we have that $ex = ArrAcc(ref2ex(\mathsf{lt}_e(\mathsf{e}), \sigma), ref2ex(\mathsf{rt}_e(\mathsf{e}), \sigma))$. From $Tree$ relation included in $Expr$ for subexpressions and Lemma 7.6.9 we can conclude $\mathsf{sz1} < \mathsf{sz} \longrightarrow size(ref2ex(\mathsf{rt}_e(\mathsf{e}), \sigma)) < size(ref2ex(\mathsf{lt}_e(\mathsf{e}), \sigma))$. So by definition of $code_{\mathcal{E}}$ we have

$$\begin{aligned}&code_{\mathcal{E}}(lst, False, d_1, (\mathsf{cnt}_r(\mathsf{d}), fr_{tl}), ref2ex(\mathsf{lt}_e(\mathsf{e}), \sigma)) \circ\\ &code_{\mathcal{E}}(lst, True, hd(fr_{tl}), (\mathsf{cnt}_a(\mathsf{d}^{\sigma}), tl(fr_{tl})), ref2ex(\mathsf{rt}_e(\mathsf{e}), \sigma)) \circ\\ &code_{arracc}(d, d_1, hd(fr_{tl}), \mathsf{flag}^{\sigma}, t),\\ &\text{where } d_1 = hd(map(\mathsf{cnt}_r, fr)), fr_{tl} = tl(map(\mathsf{cnt}_r, fr)),\\ &\qquad t = type_{\mathcal{E}}(p.tenv, gst, lst, ref2ex(\mathsf{lt}_e(\mathsf{e}), \sigma))\end{aligned}$$

Expanding the definition of $code\_step_{\mathcal{E}}$ for the array case and showing that for the considered case the equalities presented below hold, we have finished this part of the claim.

$$\begin{gathered}\mathsf{f1} = \mathsf{fi} = False, \mathsf{f2} = True\\ \mathsf{cnt}_r(\mathsf{hd1}) = \mathsf{cnt}_r(hd(fr)) = hd(map(\mathsf{cnt}_r, fr)) = d_1\\ \mathsf{cnt}_r(\mathsf{hd2}) = \mathsf{cnt}_r(hd(tl(fr))) = hd(tl(map(\mathsf{cnt}_r, fr)))\\ map(\mathsf{cnt}_r, (\mathsf{d}, tl(fr))) = (\mathsf{cnt}_r(\mathsf{d}), map(\mathsf{cnt}_r, tl(fr))) = (\mathsf{cnt}_r(\mathsf{d}), fr_{tl})\end{gathered}$$

Moreover, using the observations on $\mathsf{hd1}$, $\mathsf{hd2}$ made in 3., 4. we easily show that the last two procedure calls lead to the claimed relation on free register list since $(\mathsf{hd1}, (\mathsf{hd2}, tl(tl(fr)))) = fr$.

The rest of the postcondition is shown analogously to the example given before, arguing on sets.

8.5 Statement Code Generation

Procedure $stmt\_codegen(p, offset)$ implements code generation for statements. It takes two parameters: a pointer to a statement data structure and an offset accumulator, which is equal to an offset of the code piece generated for this statement in the code generated for the procedure body where this statement is located (the value is equivalent to function *rba* from the specification). The second parameter is used to compute the absolute offset of jump instructions in the code for the procedure where the call statement is located.

Code generation for statements is implemented in a similar way as for expressions. The procedure works on the syntax tree, it is structured as case distinction on the statement identifier and is called recursively for the subtrees in the case of the recursively constructed statements (i.e. sequence, conditional, loop). Such a structure is obviously similar to the specification. Since the structural form of the input is also similar, we do not need here any tricks to "fit in" the specification.

The code for non-recursive statements and code for the root nodes of different types is generated by auxiliary procedures, which are called from the main one. The basic approach verifying these is the same as for expressions, e.g. templates for new or return statements, or consecutive execution of function calls for assignment, where the specification of already verified *expr_codegen* is used.

Despite the structural similarity between the implementation and specification we cannot use the function $code_{\mathcal{S}}$ to specify *stmt_codegen*, since in the specification the two passes are combined into one (with the help of function *csize*) and the output of this pass is the final version of the generated code, whereas in the implementation we set the values of the jump immediate constants when generating code for procedure call temporarily to zero.

To solve this problem we introduce an auxiliary function $code_{\mathcal{S}_0}$, which is the same as $code_{\mathcal{S}}$ except for procedure call statements, where it sets jump immediate constant to zero as it is done in the implementation. Now we can use $code_{\mathcal{S}_0}$ to specify the resulting assembler code of the first pass.

However, such a modification is not enough, since after the execution of procedure *stmt_codegen* we have another result, which cannot be specified by $code_{\mathcal{S}_0}$, namely the list of pointers to jumps generated for procedure calls, which needed to be filled during the second pass. The data type used to keep this list is presented in the following section.

Thus, we need to state and prove two theorems: the first pass produces the code that is abstracted to $code_{\mathcal{P}_0}$ (analogous to $code_{\mathcal{S}_0}$ but for an entire program); the second pass applied to the result of the first one is finished with the code equivalent to $code_{\mathcal{P}}$. This means we need to create the formal description of the second pass as some function F and prove the equality $F(code_{\mathcal{P}_0}(\ldots), \ldots) = code_{\mathcal{P}}(\ldots)$, which is one of the crucial points in the proof of the correctness of the implementation, to show the correctness of the second pass.

8.5.1 Link List

The list of pointers to jump instructions, which we call *links*, is build on the data type *linkT* presented in Figure 8.12.

```
linkT = struct{      call : funcT*        //pointer to the called function
                     orig : funcT*        //pointer to the current function
                     cmd : asmT*          //pointer to a jump instruction
                     offs : nat }         //jump instruction offset

list_linkT = struct{ nxt : list_linkT*
                     prv : list_linkT*
                     cnt : linkT* }
```

Figure 8.12: C0 data structure keeping auxiliary information for the second pass of the compiler

Thus, we keep:

- pointers to the procedure, which is called
- to the origin procedure, where the call statement is located
- a pointer to the jump instruction to be filled
- offset of the jump instruction in the code generated for the body of the procedure where the call is located

One of the last parameters is, of course, redundant, but having them both we do not lose any time neither for computation of the offset (if we had only the pointer) nor for searching the place in the code to set the jump constant (if we knew only the offset). We denote the corresponding heap functions by index $_{\iota}$.

8.5.2 Procedure Call Code Generation

Let us concentrate on the code generation for procedure call in mode details, since it close, but is not completely equivalent to the specification (see 6.4, **Function Call**). The layout of the compiled code completely corresponds to Figure 6.6 with only difference that the immediate constant in the jump instruction is set to zero.

We perform the following steps:

1. Generate code for the expression where the result returned by the function call will be stored
2. Compute allocation sizes for both frames (for calling and called procedures)
3. Generate parameter passing code
4. Generate code updating the header of the new frame in the stack (including the jump to the called procedure). The jump distance is set to zero in the first pass.
5. Create a new element for the link list and set the fields of the link to the following values: i) *call* to the value of field *cf* of the processed statement (see Figure 7.8); ii) *orig* to the value of the global variable pointing to the currently processed procedure iii) *cmd* to the pointer to the jump instruction

generated in 4.; iv) $offs$ to value $offset$ (second parameter of procedure *stmt_codegen*, i.e. length of the code before the call statement) added with the lengths of the code pieces generated in 1., 3., and 4. minus two (recall that the jump instruction is the second from the end)

6. Add the newly created link at the beginning of the existing link list

Steps 1.-4. are implemented completely equivalent to the specification (see Section 6.4 (paragraph **Function Call**), Figure 6.6), except for the value of the jump constant.

8.5.3 Abstract Link Type and The Link Type Abstraction Function

In order to specify the link list we need some abstract data type for it and the abstraction function setting the relation between both.

Definition 8.5.1 We define a link $l \in link$ be a tuple

$$l = (orig, call, offs),$$

where components are the same as in the implementation except for the pointer to the instruction, which is not needed in the abstract model:
$l.orig \in (nm_{\mathcal{P}} \times \mathcal{P})$ - current procedure declaration
$l.call \in (nm_{\mathcal{P}} \times \mathcal{P})$ - procedure declaration of the called procedure
$l.offs \in \mathbb{N}$ - instruction offset

Now we can establish the relation between the link data structure and the abstract link type based on the abstraction function approach we presented in Chapter 7.

Definition 8.5.2 Let σ be a state, $p \in Ref$ a pointer to a link data structure, and $fdl \in Ref^*$ be a reference list which is the base for the procedure table. Then link l is the corresponding abstract link if the following predicate returns true:

$$\begin{aligned} Link(p, \sigma, fdl, l) \equiv\ & Func(\mathsf{call}_l(p), \mathsf{t}, \mathsf{gv}, \sigma, fdl, snd(l.call)) \wedge \mathsf{call}_l(p) \in_* fdl \wedge \\ & String(\mathsf{name}_f(\mathsf{call}_l(p)), \sigma, fst(l.call)) \wedge \\ & Func(\mathsf{orig}_l(p), \mathsf{t}, \mathsf{gv}, \sigma, fdl, snd(l.orig)) \wedge \mathsf{orig}_l(p) \in_* fdl \wedge \\ & String(\mathsf{name}_f(\mathsf{orig}_l(p)), \sigma, fst(l.curr)) \wedge \\ & l.offs = \mathsf{offs}_l(p) \end{aligned}$$

Components t and gv of the state are pointers to the type table and the list of global variables, respectively. Since the link list is created during the program execution, we add to the abstraction function the explicit information on the changes of the relevant heap functions. Thus, all the pointers of the container list (lists dl and $map(\mathsf{cnt}_l, dl)$) are in the $alloc$ component of a state. This information is used in the proof (analogously to assembler instructions creation) to show that allocation of a new link does not destroy the old link list.

Definition 8.5.3 Let σ be a state, $p \in \mathit{Ref}$ is a pointer to a link list data structure, and $fdl \in \mathit{Ref}^*$ be a reference list which is the procedure table based on. Then the list is based on reference list dl and $ll \in \mathit{link}^*$ is the corresponding abstract link list if the following predicate holds:

$$\begin{aligned}
&\mathit{LinkList}(p, \sigma, fdl, dl, ll) \equiv \\
&\exists l.\ \mathit{dList}(p, \mathsf{nxt}_l, \mathsf{prv}_l, l, dl) \wedge |dl| = |lnk| \wedge \mathit{distinct}(\mathit{map}(\mathsf{cnt}_l, dl)) \wedge \\
&\{dl\} \cap \{\mathit{map}(\mathsf{cnt}_l, dl)\} = \varnothing \wedge \{dl\} \subseteq \{\mathsf{alloc}\} \wedge \\
&\{\mathit{map}(\mathsf{cnt}_l, dl)\} \in \{\mathsf{alloc}\} \wedge \forall i < |dl|.\ \mathit{Link}(dl_i, \sigma, fdl, ll_i)
\end{aligned}$$

8.5.4 Specification Extension

To describe how a link list for some statement or for an entire procedure list is created and used during the second pass we define the following functions:

Definition 8.5.4 Let p be an abstract program, $f \in (nm_{\mathcal{P}} \times \mathcal{P})$ be a procedure table entry, where statement $s \in \mathcal{S}$ is located. Let $o \in \mathbb{N}$ be an offset value which is equal to the number of instruction where the code generated for s starts inside the code generated for body of f. Then the link list for statement s in program p is produced as below:

$$\begin{aligned}
&\mathit{links}_{\mathcal{S}}(f, o, s) = \\
&\begin{cases}
(f, (pn, \mathit{map\_of}(p.pt, pn)), o + \mathit{csize}_{\mathcal{S}}(f, s) - 2) & \text{if } s = \mathit{Call}(vn, pn, el, id) \wedge \\
 & \quad pn \in_* \mathit{map}(fst, p.pt) \\
\mathit{links}_{\mathcal{S}}(f, o_1, s_2) \circ \mathit{links}_{\mathcal{S}}(f, o, s_1) & \text{if } s = \mathit{Comp}(s_1, s_2) \\
\mathit{links}_{\mathcal{S}}(f, o_3, s_2) \circ \mathit{links}_{\mathcal{S}}(f, o_2, s_1) & \text{if } s = \mathit{Ifte}(e, s_1, s_2, id) \\
\mathit{links}_{\mathcal{S}}(f, o_2, s_1) & \text{if } s = \mathit{Loop}(e, s_1, id) \\
[] & \text{otherwise}
\end{cases} \\
&\text{where } o_1 = o + \mathit{csize}_{\mathcal{S}}(f, s_1), \\
&\qquad\quad o_2 = o + \mathit{csize}_{\mathcal{E}}(snd(f).loc, \mathit{True}, e) + 2, \\
&\qquad\quad o_3 = o_2 + \mathit{csize}_{\mathcal{S}}(f, s_1) + 2
\end{aligned}$$

So, we recursively create links for all procedure call substatements of statement s, adding each new link at the head of the old list.

The distances stored as the third component of every link is accumulative and equals to the position of the jump to be filled in the code for the currently processed procedure. This is illustrated in Figure 8.13. For the case (d) according to the definition presented above we generate one link (called l in the figure).

Collecting links over the whole program is done by appending links from the whole procedure table. Starting offset for links collected in every function body is 0.

$$\mathit{links}_{PT}(pl) = \begin{cases} [] & \text{if } pl = [] \\ \mathit{links}_{PT}(xs) \circ \mathit{links}_{\mathcal{S}}(x, 0, snd(x).body) & \text{if } pl = (x, xs) \end{cases}$$

The last link is placed at the head of the list (as in the implementation).

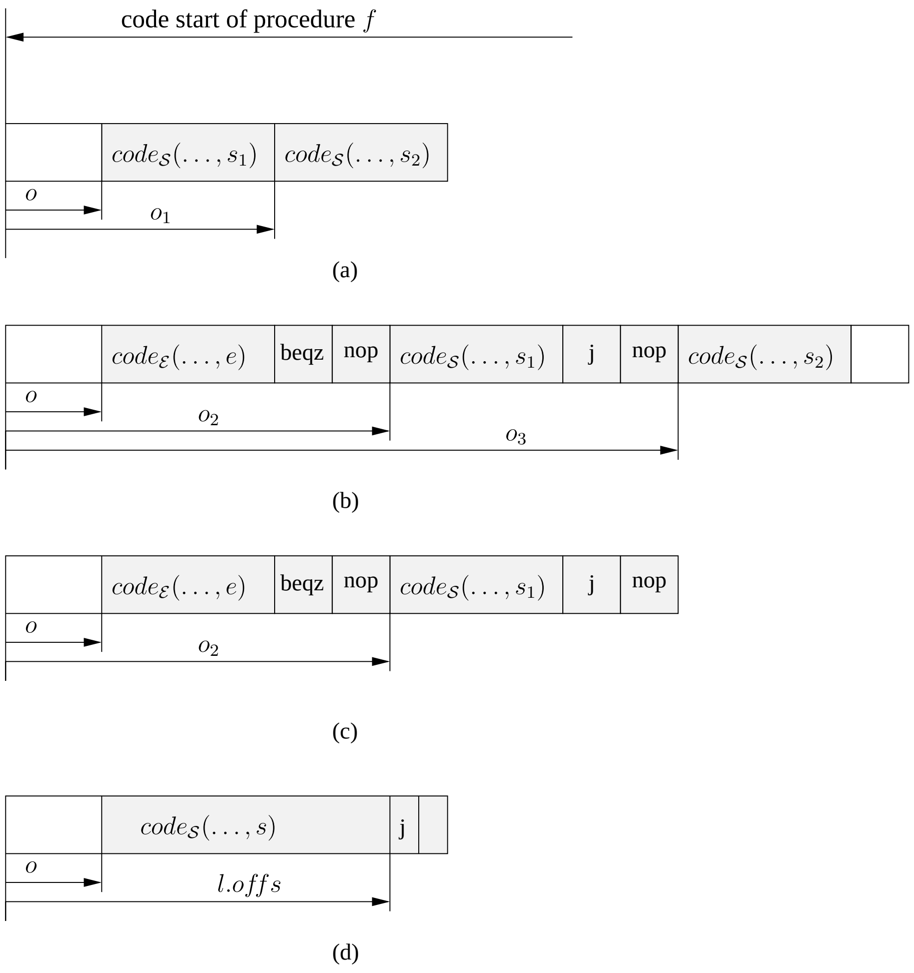

code start of procedure f
$code_{\mathcal{S}}(\ldots,s_1)$
$code_{\mathcal{S}}(\ldots,s_2)$
o
o_1
(a)
$code_{\mathcal{E}}(\ldots,e)$
beqz
nop
$code_{\mathcal{S}}(\ldots,s_1)$
j
nop
$code_{\mathcal{S}}(\ldots,s_2)$
o
o_2
o_3
(b)
$code_{\mathcal{E}}(\ldots,e)$
beqz
nop
$code_{\mathcal{S}}(\ldots,s_1)$
j
nop
o
o_2
(c)
$code_{\mathcal{S}}(\ldots,s)$
j
o
$l.offs$
(d)

Figure 8.13: Offsets of substatements of statement s used in the definition of $links_{\mathcal{S}}$: (a) sequential composition $s = Comp(s_1, s_2)$; (b) conditional $s = Ifte(e, s_1, s_2, id)$; (c) loop $s = Loop(e, s_1, id)$; (d) function call $s = Call(vn, pn, el, id)$

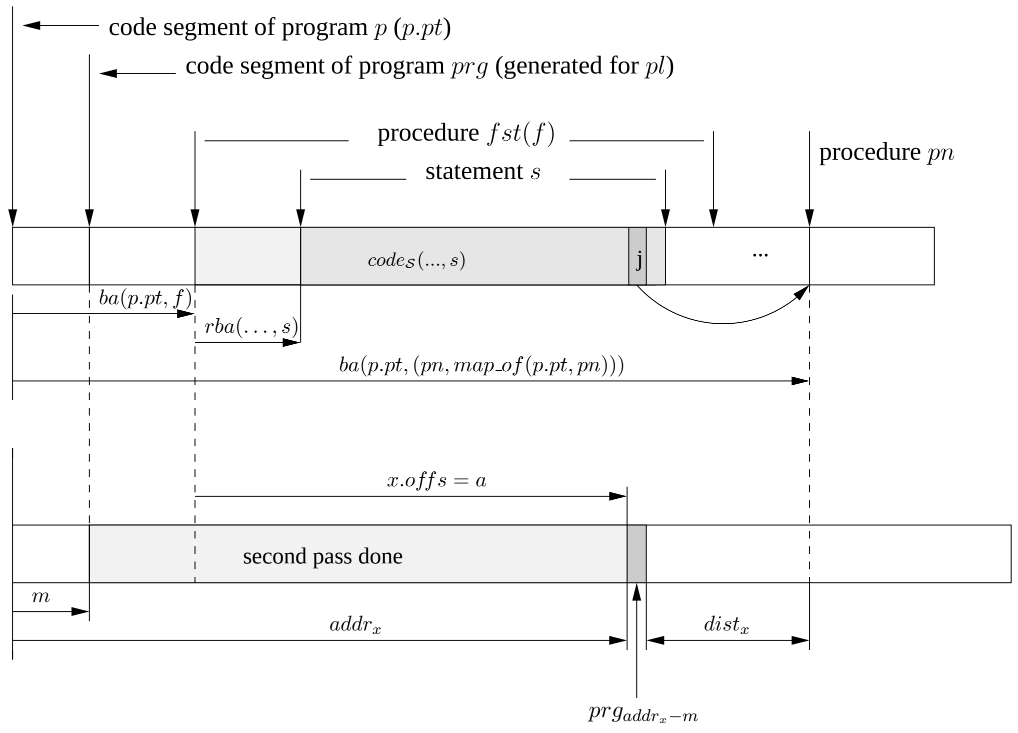


Figure 8.14: Second pass: filling the jump distance

Definition 8.5.5 Let $prg \in Instr^*$ be some piece of assembler code generated during the first pass, and $pl \in (nm_{\mathcal{P}} \times \mathcal{P})^*$ be a procedure list (in general, only a part of the program procedure table $p.pt$). Then the following procedure sets jump constants in prg according to the link list ll.

$$2pass(prg, m, pl, ll) = \begin{cases} prg & \text{if } ll = [] \\ 2pass(prg', m, pl, xs) & \text{if } ll = (x, xs) \end{cases},$$

$$\text{where } prg' = prg[addr_x - m := jal(4 \cdot dist_x)],$$
$$addr_x = ba(pl, x.orig) + x.offs,$$
$$dist_x = ba(p.pt, x.call) - ba(p.pt, x.orig) - x.offs - 1$$

Thus, we do no changes if we have an empty link list. We compute the point in the code where the new jump constant needs to be written and its value based on the currently proceeding link.

Intuitively, pl is some suffix of the program procedure $p.pt$. Parameter $m \in \mathbb{N}$ compensates an offset of prg within the code generated for pl. Therefore, we compute the address of the jump according to pl and the jump value according to the whole procedure table $p.pt$.

Figure 8.14 illustrates how the jump distance is computed from the information stored as link x. Function $cf = x.call$ is the shortcut for $(pn, map(p.pt, pn))$. Parameter m is used to correctly compute the jump location in part prg of the entire code for program p.

8.5.5 Combining Passes Correctness Proof

The main theorem we need to show is the following. Setting jumps collected for a program in the code generated for it during the first pass produces the code which is abstracted to the result of function $code_{PT}$.

Theorem 8.5.6

$$valid_{PT}(p, p.pt) \Longrightarrow 2pass(code_{PT_0}(p.pt), 0, p.pt, links_{PT}(p.pt)) = code_{PT}(p.pt)$$

where for all procedure tables pt the predicate $valid_{PT}(p, pt)$ is a shortcut for $valid_{PT}(p.tenv, p.gst, pt)$.

We start from auxiliary lemmas. Notations $addr_x$, $dist_x$ are used as it stated in Definition 8.5.5.

Lemma 8.5.7 Setting jumps in part prg_1 of code $prg_1 \circ prg_2$ does not affect prg_2.

$$\begin{aligned}&\forall prg_1, l \in_* ll.\ addr_l - m < |prg_1| \Longrightarrow\\ &2pass(prg_1 \circ prg_2, m, pl, ll) = 2pass(prg_1, m, pl, ll) \circ prg_2\end{aligned}$$

Proof: By induction on ll.

Induction base $ll = []$ is trivial by definition of $2pass$.

For induction step $ll = (x, xs)$ expanding $2pass$ we get

$$\begin{aligned}&2pass((prg_1 \circ prg_2)[addr_x := ...], m, pl, xs) =\\ &2pass(prg_1[addr_x := ...], m, pl, xs) \circ prg_2\end{aligned}$$

From Lemma 1.1 and assumption for $addr_x$ we can show $(prg_1 \circ prg_2)[addr_x := ...] = prg_1[addr_x := ...] \circ prg_2$. Therefore, instantiating the induction hypothesis with program $prg_1[addr_x := ...]$ we get the claim, since $|prg_1[addr_x := ...]| = |prg_1|$. $\square$

Lemma 8.5.8 Setting jumps in part prg_2 of code $prg_1 \circ prg_2$ do not affect prg_1.

$$\begin{aligned}&\forall prg_2, l \in_* ll.\ addr_l - m \geq |prg_1| \Longrightarrow\\ &2pass(prg_1 \circ prg_2, m, pl, ll) = prg_1 \circ 2pass(prg_2, o, pl, ll)\end{aligned}$$

Proof: By induction on ll.

Induction base $ll = []$ is trivial.

Induction step $ll = (x, xs)$ is similar to the previous lemma. By Lemma 1.1 and assumption for $l = x$ we get $(prg_1 \circ prg_2)[addr_x := ...] = prg_1 \circ prg_2[addr_x := ...]$. Induction hypothesis for program $prg_2[addr_x := ...]$ shows the claim. $\square$

Lemma 8.5.9 The second pass for a program can be decomposed as follows. The idea of this lemma is illustrated in Figure 8.15.

$$\begin{aligned}&(\forall l \in_* l_1.\ addr_l - m < |prg_1|) \wedge (\forall l \in_* l_2.\ addr_l - m \geq |prg_1|) \Longrightarrow\\ &2pass(prg_1 \circ prg_2, m, pl, l_2 \circ l_1) =\\ &2pass(prg_1, m, pl, l_1) \circ 2pass(prg_2, m + |prg_1|, pl, l_2)\end{aligned}$$

Proof: Combining Lemmas 8.5.7, 8.5.8.

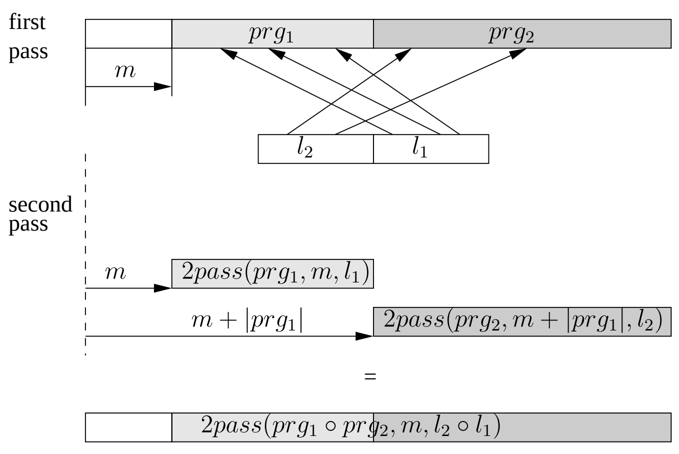


Figure 8.15: Second pass decomposition

Lemma 8.5.10 The links generated for some statement s have the following properties:

$$\forall o.\ valid_{\mathcal{S}}(p, snd(f).loc, s) \Longrightarrow$$
$$\forall l \in_* links_{\mathcal{S}}(f, o, s).\ l.orig = f \wedge l.offs - o < csize(f, s) \wedge l.offs > o,$$

Proof: By structural induction on s.

In the cases where we do not produce any links the proof is trivial.

Let $s = Comp(s_1, s_2)$. By definition of $links_{\mathcal{S}}$ any link l' can be either in $links_{\mathcal{S}}(f, o, s_1)$ or in $links_{\mathcal{S}}(f, o + csize(f, s_1), s_2)$. Let us consider the latter case. We have $valid_{\mathcal{S}}(p, snd(f).loc, s) \longrightarrow valid_{\mathcal{S}}(p, snd(f).loc, s_2)$. Hence by the induction hypothesis (for offset equal to $o + csize(f, s_1)$) we have

$$\forall l \in_* links_{\mathcal{S}}(f, o + csize(f, s_1), s_2).$$
$$l.orig = f \wedge l.offs - (o + csize(f, s_1)) < csize(f, s_2) \wedge$$
$$l.offs > o + csize(f, s_1).$$

Instantiating it with l', we have $l'.orig = f$ by assumption. The rest can be shown by arithmetics considering the assumptions and $csize(f, s) = csize(f, s_1) + csize(f, s_2)$.

The second case and the cases for conditional and loop are done analogously.

Case $s = Call(vn, pn, el, id)$ follows directly from the definition of $links_{\mathcal{S}}$. $\square$

Based on this lemma we can show by induction:

Lemma 8.5.11 $valid_{PT}(p, pl) \Longrightarrow \forall l \in_* links_{PT}(pl).\ l.orig \in_* pl$

Lemma 8.5.12 Filling jump constants for the code and link list generated for statement s placed within procedure f is correct.

$$\forall o.\ pl \neq [] \wedge valid_{PT}(p, pl) \wedge f \in_* pl \wedge$$
$$sub\_stmt(snd(f).body, s) \wedge 4 \cdot o = rba(snd(f).body, hd(s2l(s))) \Longrightarrow$$
$$2pass(code_{\mathcal{S}_0}(f, s), ba(pl, f) + o, links_{\mathcal{S}}(f, o, s)) = code_{\mathcal{S}}(f, s)$$

Proof: By structural induction on s.

Cases for s equal to *Skip*, *Alloc*, or *Ass* are trivial.

Sequential composition $s = Comp(s_1, s_2)$ Expanding $linkss_{\mathcal{S}}$ and $code_{\mathcal{S}_0}$ for that case we have to show:

$$2pass(c_1 \circ c_2, ba(pl, f) + o, l_2 \circ l_1) = code_{\mathcal{S}}(f, s)$$

where $c_1 = code_{\mathcal{S}_0}(f, s_1)$, $c_2 = code_{\mathcal{S}_0}(f, s_2)$, $l_1 = linkss_{\mathcal{S}}(f, o, s_1)$, and $l_2 = linkss_{\mathcal{S}}(f, o + csize(x, s_1), s_2)$.

We use Lemma 8.5.9 provided that its assumptions hold:

1. $\forall l \in_* l_1.\ addr_l - (ba(pl, f) + o) < |c_1|$

From $valid_{PT}(p, pl)$ we have that $valid_{\mathcal{S}}(p, snd(f).loc, snd(f).body)$ holds and hence $valid_{\mathcal{S}}(p, snd(f).loc, s_1)$ holds as well. By Lemma 8.5.10 for all l from l_1 value $l.orig$ is equal to x and $l.offs - o < csize(f, s)$. That finishes the goal since $addr_l - (ba(pl, f) + o)$ is equal to $l.offs - o$ and $|c_1| = |code_{\mathcal{S}}(f, s_1)| = csize_{\mathcal{S}}(f, s_1)$, where the last equality is implied by Lemma 6.4.2 for s_1.

2. $\forall l \in_* l_2.|c_1| \leq addr_l - (ba(pl, f) + o)$

Analogously to the previous case. We have $l.orig = f$ and $l.off > (ba(pl, f) + o) + csize_{\mathcal{S}}(f, s_1)$ (Lemma 8.5.11) and finish the claim with $|c_1| = csize_{\mathcal{S}}(f, s_1)$.

Thus, it remains to show

$$\begin{aligned} &2pass(p, c_1, ba(pl, f) + o, pl, l_1) \circ 2pass(p, c_2, ba(pl, f) + o + |c_1|, pl, l_2) = \\ &code_{\mathcal{S}}(f, s_1) \circ code_{\mathcal{S}}(f, s_2). \end{aligned} \tag{8.8}$$

Let $s2l(s_1) = [] \wedge s2l(s_2) = []$, then we can show, that $l_1 = c_1 = l_2 = c_2 = []$. Moreover, $code_{\mathcal{S}}(f, s_1) = code_{\mathcal{S}}(f, s_2) = []$. This case is trivial by definition of $pass2$.

Let $s2l(s_1) \neq [] \wedge s2l(s_2) \neq []$ (both remaining cases can be derived from it). Assumptions of the induction hypothesis for s_1 instantiated with offset o hold since: i) $sub\_stmt(snd(f).body, s) \longrightarrow sub\_stmt(snd(f).body, s_1)$; ii) we can show $s2l(s_1) \neq [] \longrightarrow hd(s2l(s)) = hd(s2l(s_1))$, so $o \cdot 4 = hd(s2l(s_1))$ is implied by the lemma assumption; iii) all other assumptions are the same. So, the left parts of the appends in (8.8) are equal with the induction hypothesis.

We clearly need to instantiate the induction hypothesis for s_2 with offset $o + |c_1|$. By Lemma 6.4.4 for $x = snd(f).body$ (necessary condition $distinct_{\mathcal{S}}(f)$ follows from $valid_{PT}(p, pl)$) and recalling that $|c_1| = csize(x, s_1)$ the assumption $4 \cdot (o + |c_1|) = rba(snd(x).body, s_2)$ is valid. The case is finished.

Cases $Ifte$**,** $Loop$ The proof is done by the same approach but more involved, since $code_{\mathcal{S}}(f, s)$ concatenates not only the code generated for substatements (cs_1, cs_2), but also the code for the branching expression ce and the connecting instructions ci_1, ci_2 (see $code_{\mathcal{S}}$) as well. For example, for $s = Ifte(e, s_1, s_2, id)$ we additionally use Lemmas 8.5.8, 8.5.10, and 6.2.2 to show that

$$\begin{aligned} &2pass(ce_1 \circ ci_1 \circ cs_1 \circ ci_2 \circ cs_2, ba(pl, f) + o, \ldots) = \\ &ce_1 \circ ci_1 \circ 2pass(cs_1 \circ ci_2 \circ cs_2, ba(pl, f) + o + |ce_1| + 2, \ldots) = \\ &ce_1 \circ ci_1 \circ 2pass(cs_1, ba(pl, f) + o + |ce_1| + 2, \ldots) \circ ci_2 \circ \\ &\ 2pass(cs_2, ba(pl, f) + o + |ce_1| + |cs_1| + 4, \ldots) \end{aligned}$$

The additional offsets are needed to be taken into account when instantiating the induction hypothesis. The correctness of the relations between rba for direct substatements can be shown by lemmas similar to Lemma 6.4.4.

Call statement $s = Call(vn, pn, el, id)$ We have only one link l. So, updating $code_{\mathcal{S}_0}(f, s)$ at position $addr_l - (ba(pl, x) + o) = l.offs - o = csize(x, s) - 2$ with $jal_x(4 \cdot dist_l)$ we need to show that $dist_l$ is equal to the constant $dist$, which is on the same position in $code_{\mathcal{S}}(x, s)$. By definition of function $2pass$ and considering the value of $l.offs$ we have

$$dist_l = ba(p.pt, map\_of(p.pt, pn)) - ba(p.pt, x) - (csize_{\mathcal{S}}(x, s) - 2 + o)$$

Since by assumption $4 \cdot o$ is equal to $rba(x, s)$ and

$$\begin{aligned}&csize_{\mathcal{S}}(x, s) = \\ &\quad csize_{\mathcal{E}}(lst, False, VarAcc(vn)) + csize_{par\_pass}(lst, el) + |init\_frame(...)|,\end{aligned}$$

the equality to $dist$ is obvious. □

Lemma 8.5.13 If jumps to be set are placed in the code for the tail of a procedure environment, the application of $2pass$ can be decomposed as below.

$$\begin{aligned}&\forall prg.\ (\forall l \in_* ll.\ l.orig \in_* xs) \wedge unique((x, xs)) \Longrightarrow \\ &2pass(prg, m, (x, xs), ll) = 2pass(prg, m - csize(x, snd(x).body), xs, ll)\end{aligned}$$

Proof: By induction on ll. Induction base is trivial, since prg is not updated. For induction step $ll = (y, ys)$ after expanding the definition for $2pass$ we get

$$2pass(prg', m, (x, xs), ys) = 2pass(prg'', m - csize(x, snd(x).body), xs, ys),$$

where prg' is equal to prg updated at position $ba((x, xs), y.orig) + y.offs - o$ and prg'' is prg, which is updated at $ba(xs, y.orig) + y.offs - (m - csize(x, snd(x).body))$. Since from the assumptions $y.orig \in_* xs$ and considering $unique((x, xs))$, we have $y.orig \neq x$. Thus, $ba((x, xs), y.orig) = ba(xs, y.orig) + csize(x, snd(x).body)$ what implies $prg' = prg''$. Using the induction hypothesis for prg' (its preconditions are obviously implied by the assumptions) we finish the proof. □

The main theorem follows from the lemma we prove below by replacing pl with procedure environment $p.pt$.

Lemma 8.5.14

$$\begin{aligned}&valid_{PT}(p, p.pt) \wedge valid_{PT}(p, pl) \Longrightarrow \\ &2pass(code_{PT_0}(pl), 0, pl, links_{PT}(pl)) = code_{PT}(pl)\end{aligned}$$

Proof: by induction on pl.

Induction base $pl = []$ is obviously true, since by definition $links_{PT}([]) = []$ and $code_{PT_0}([]) = [] = code_{PT}([])$

Induction step $pl = (x, xs)$ Expanding the definitions of $links_{PT}$ and $code_{PT_0}$ we get the following goal to show.

$$\begin{aligned}
&2pass(code_x \circ code_{PT_0}(xs), 0, pl, links_{PT}(xs) \circ links_x) \\
&= \qquad\qquad (8.9)\\
&code_{PT}(pl), \\
&\text{where } code_x = code_{\mathcal{S}_0}(x, snd(x).body) \text{ and } links_x = links_{\mathcal{S}}(x, 0, snd(x).body)
\end{aligned}$$

We use Lemma 8.5.9 provided that its assumptions hold:
1. $\forall l \in_* links_x.\ ba((x, xs), l.orig) + l.offs < |code_x|$

Since from $valid_{PT}(p, pl)$ we have $valid_{\mathcal{S}}(p, snd(x).loc, snd(x).body)$, then we have by Lemma 8.5.10 for all l from $links_x$ that value $l.orig$ is equal to x and hence, base address $ba((x, xs), l.orig) = 0$. By the same lemma we have that $l.offs < csize_{\mathcal{S}}(x, snd(x).body)$, so we finish the goal concluding with Lemma 6.4.2 that $|code_x| = csize_{\mathcal{S}}(x, snd(x).body)$.
2. $\forall l \in_* links_{PT}(xs).\ |code_x| \leq ba((x, xs), l.orig) + l.offs$

By Lemma 8.5.11 we have $l.orig \in_* xs$, so from $valid_{PT}(p, pl)$ we conclude $unique(pl)$ and hence $l.orig \neq x$. By definition of ba we get

$$ba((x, xs), l.orig) = csize(x, snd(x).body) + ba(xs, l.orig)$$

Analogously to the previous case we can show $|code_x| = csize_{\mathcal{S}}(x, snd(x).body)$. That finishes the claim.

As the result we get the left side of (8.9) transformed to:

$$2pass(code_x, 0, (x, xs), links_x) \circ 2pass(code_{PT_0}(xs), |code_x|, (x, xs), links_{PT}(xs)) \tag{8.10}$$

By Lemma 8.5.12 for $pl = (x, xs)$, $f = x$, $s = snd(x).body$; and offset $o = 0$ we get the first operand of the concatenation in (8.10) to be equal to $code_{\mathcal{S}}(x, snd(x).body)$ (since $0 = ba((x, xs), x)$). The preconditions of the lemma follow from the assumptions, Lemma 6.4.3, and property $\forall s.\ sub\_stmt(s, s)$.

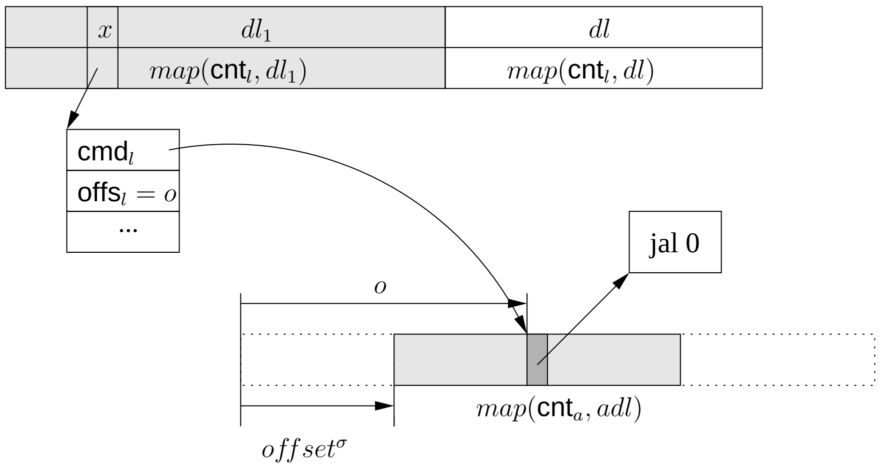
By the induction hypothesis we have

$$2pass(code_{PT_0}(xs), 0, xs), links_{PT}(xs)) = code_{PT}(xs).$$

To apply it to the second concatenation operand in (8.10) we use Lemma 8.5.13, where its precondition can be derived from assumption $valid_{PT}(p, pl)$ and from Lemma 8.5.11. □

8.5.6 Code Generation Procedure Specification

The essential part of the specification is presented below. Some variables that have not been mentioned before: i) prog - global pointer to the program structure; ii) func - global pointer to the currently proceeding procedure (is needed for linking); iii) link - global pointer to the link list.



x
dl_1
dl
$map(\mathsf{cnt}_l, dl_1)$
$map(\mathsf{cnt}_l, dl)$
cmd_l
$\mathsf{offs}_l = o$
...
jal 0
o
$map(\mathsf{cnt}_a, adl)$
$offset^\sigma$

Figure 8.16: Connection between a link structure and position of the corresponding jump

Theorem 8.5.15 Procedure *stmt_codegen* is equivalent to function $code_{\mathcal{S}_0}$ and satisfies the following specification.

$$\begin{aligned}
&\forall \sigma, tdl, tt, fdl, pt, gdl, gst, ldl, lst, tr, s, dl, ll, fr.\ \Gamma \vdash\\
&\{\sigma \mid FTypetable(\mathsf{t}, \sigma, tdl, tt) \wedge FFuncList(\mathsf{pt(prog)}, \mathsf{t}, \mathsf{gv}, \sigma, fdl, pt) \wedge\\
&\qquad FGVarList(\mathsf{gv}, \mathsf{t}, \sigma, gdl, gst) \wedge\\
&\qquad \mathsf{func} \in_* map(\mathsf{cnt}_f, fdl) \wedge \mathsf{loc}_f(\mathsf{func}) = \mathsf{lv} \wedge FLVarList(\mathsf{lv}, \mathsf{t}, \sigma, ldl, lst) \wedge\\
&\qquad CStmt(\mathsf{p}, \mathsf{t}, \mathsf{gv}, \mathsf{lv}, \sigma, map(\mathsf{cnt}_f, fdl), tr, s) \wedge\\
&\qquad LinkList(\mathsf{link}, \sigma, map(\mathsf{cnt}_f, fdl), dl, ll) \wedge\\
&\qquad \exists lst.\ dList(\mathsf{free}, \mathsf{nxt}_r, \mathsf{prv}_r, lst, fr) \wedge (Prop)\}\\
&\textbf{res} := \mathsf{CALL\ stmt\_codegen(p,\ offset)}\\
&\{\sigma' \mid \exists adl.\ AsmProg(\mathsf{res\_head}, \mathsf{res\_last}, \mathsf{res\_length}, \sigma', adl, code_{\mathcal{S}_0}(f, s)) \wedge\\
&(\exists dl_1.\ LinkList(\mathsf{link}, \sigma', map(\mathsf{cnt}_f, fdl), dl_1 \circ dl, link_{\mathcal{S}}(f, \mathsf{offset}^\sigma, s) \circ ll) \wedge\\
&\forall x \in_* dl_1.\ \mathsf{cmd}_l(\mathsf{cnt}_l(x)) = map(\mathsf{cnt}_a, adl)_{\mathsf{offs}_l(\mathsf{cnt}_l(x)) - \mathsf{offset}^\sigma})\\
&\exists lst.\ dList(\mathsf{free}, \mathsf{nxt}_r, \mathsf{prv}_r, lst, fr) \wedge Post\},
\end{aligned}$$

$$\begin{aligned}
\text{where } f &= Func\_cont(\mathsf{cnt}_f(\mathsf{func}), \mathsf{t}, \mathsf{gv}, \sigma, map(\mathsf{cnt}_f, fdl)),\\
p &= (type\_env(tt), gst, pt)
\end{aligned}$$

Thus, in addition to the generated code we specify the link list extended by links for the statement. Moreover, since field *cmd* of structure *linkT* is not described by relation *LinkList* we need to specify it additionally. Thus, in every link in *dl* pointed by x we have $\mathsf{x} \to \mathsf{cnt} \to \mathsf{cmd}$ is a pointer to an instruction in the instruction list implementation at position $\mathsf{x} \to \mathsf{cnt} \to \mathsf{offs}$, see Figure 8.16.

The assumptions *Prop* include other necessary properties of the input, such as the length of the opcode field of assembler instructions data, injectivity of the $ref2nm$ function etc.

The postconditions *Post* include the information of the heap functions that are changed during the execution of this procedure.

The proof approach is analogous to Theorem 8.4.7. Let us shortly consider its main points.

Code generation We proceed similarly to expression correctness proof. For every non-inductive statement data (differentiating on identifier) we show that it is abstracted to some constructor $S(\ldots)$. We use specifications for procedures generating code cs for that type, and show the conclusion chain $code_{\mathcal{S}}(f, s) = code_{\mathcal{S}}(f, S(\ldots)) = cs$.

For inductive statements we use the specification for recursive calls (shown that pointers to left and right subtrees can be abstracted to substatements beforehand with Lemma 7.7.3) and use Lemma 7.10.5 to show that concatenation of the program pieces leads to the necessary result.

Links Since non-inductive statements do not produce any links (excluding the call statement), i.e. formally $link_{\mathcal{S}}(f, \mathsf{offset}^{\sigma}, s) = []$ we instantiate the existence quantifier in the corresponding part of the claim with $dl_1 = []$ and the link list relation follows from the preconditions.

Setting the link structure for the call statement in the implementation is fully equivalent to $link_{\mathcal{S}}$. So instantiating dl_1 with the newly produced pointer $[r]$, we need to show that *LinkList* relation holds. The tail of the list is unchanged (from the precondition on *LinkList* and the fact that creating a new link does not destroy others). To prove $Link(\mathsf{cnt}_l(r), ...)$ we need to show that both pointers func and $\mathsf{cf}_s(\mathsf{p})$ (pointer to the called function) can be abstracted to procedure declarations stated in the new abstract link. This can be easily done based on relation *FFuncList* and the fact that the pointers belong to $\mathsf{cnt}_f(fdl)$ (from precondition $\mathsf{func} \in_* map(\mathsf{cnt}_f, fdl)$ and *Stmt* relation for the call, respectively).

Considering implementation we have that $\mathsf{offs}(\mathsf{cnt}_l(r)) = \mathsf{offset}^{\sigma} + \mathsf{res\_length} - 2$ and $\mathsf{cmd}(\mathsf{cnt}_l(r)) = \mathsf{cnt}_a(\mathsf{prv}_a(\mathsf{res\_last}))$. Since $\mathsf{res\_length} = |adl|$, $\mathsf{res\_last} = adl_{|adl|-1}$ (by relation *AsmProg*), and $\mathsf{prv}_a(\mathsf{res\_last}) = adl_{|adl|-2}$, the value of reference $\mathsf{cmd}(\mathsf{cnt}_l(r))$ satisfies the postcondition.

For inductive statements we use the induction hypothesis (the specification of the call for the pointers to the left and right subtrees). Let us consider the simplest case. For $\mathsf{id}_s(\mathsf{p}) = 2$ we have the following implementation:

```
x:= CALL stmt_codegen(p → lt, offset);
y:= CALL stmt_codegen(p → rt, offset + x_length);
tmp := CALL pair_asmT_list_append(x, y) ;
...;
res := tmp
```

From *Stmt* we get $s = Comp(s_1, s_2)$ and instantiate the specifications for the first recursive calls with $lt(tr)$ for tr, s_1 for s; all other parameter stay the same. The preconditions of the recursive call follow from the theorems' preconditions and Lemma 7.7.3. As the part of the postconditions for the resulting state σ_1 we get

```
compiler(p | **res** )=
prog := p
sizes := [a, b, c, d];
free := CALL set_reg_list();
**x** := CALL first_pass(prog);
WHILE links $\neq$ $Null$ DO
    l:= links $\rightarrow$ cnt ;
    n := (int(l $\rightarrow$ call $\rightarrow$ of) - int(l $\rightarrow$ orig $\rightarrow$ of + int(l $\rightarrow$ offs)) - 1) * 4 ;
    l$\rightarrow$cmd$\rightarrow$imm := n
    links := links $\rightarrow$ nxt; OD;
...
```

$res, x : asmT\_pair;\ p, prog : progT*;$
$sizes : nat[4];\ free : regT\_list*;\ l : linkT*;\ n : int$

Figure 8.17: Main compiling procedure

that there exists some reference list dl_1 such that the following relation holds:

$$LinkList(\mathsf{link}, \sigma_1, map(\mathsf{cnt}_f, fdl), dl_1 \circ dl, link_{\mathcal{S}}(f, \mathsf{offset}^\sigma, s_1) \circ ll)$$

The specification of the second recursive call is clearly instantiated with $rt(tr)$ for tr, s_2 for s. Moreover, we instantiate $dl_1 \circ dl$ for dl and $link_{\mathcal{S}}(f, \mathsf{offset}^\sigma, s_1) \circ ll$ for ll. The linked list relation in the new state σ_2 will be the following:

$$LinkList(\mathsf{link}, \sigma_2, map(\mathsf{cnt}_f, fdl), dl_1' \circ (dl_1 \circ dl), new\_ll)$$
$$\text{where } new\_ll = link_{\mathcal{S}}(f, \mathsf{offset}^\sigma + \mathsf{x\_length}^{\sigma_1}, s_2) \circ (link_{\mathcal{S}}(f, \mathsf{offset}^\sigma, s_1) \circ ll)$$

From relation $AsmProg$ in the postconditions of the first recursive call we get $\mathsf{x\_length}^{\sigma_1} = |code_{\mathcal{S}_0}(f, s_1)|$. By Lemma 7.8.3 we can show the validity of s_1 and hence $|code_{\mathcal{S}_0}(f, s_1)| = csize(f, s_1)$ by Lemma 6.4.2. Then $link_{\mathcal{S}}(f, \mathsf{offset}^\sigma + \mathsf{x\_length}^{\sigma_1}, s_2) \circ (link_{\mathcal{S}}(f, \mathsf{offset}^\sigma, s_1)$ is obviously equal to $(link_{\mathcal{S}}(f, \mathsf{offset}^\sigma, s)$ (by definition) and instantiating the existence quantifier for the $LinkList$ relation with $dl_1' \circ dl_1$ we finish this part of the proof.

Free registers The free register list relation is implied from the specification of procedure *expr_codegen* whenever it is used. Recall, that one of the postconditions for procedure *expr_codegen* (see Theorem 8.4.7) is that variable free points to the same list of free registers as before a call of *expr_codegen*.

8.6 Main Compiling Procedure

The main procedure includes the call of the first pass procedure for the whole program and the following linking using information from the link list (Figure 8.17).

Theorem 8.6.1

$$\begin{array}{l}\forall \sigma, tenv, pt, gst.\ \Gamma \vdash \{CProg(\mathsf{p}, \sigma, tenv, pt, gst)\}\\ \mathsf{\textbf{res}} := \mathsf{CALL\ compiler(p)}\\ \{\sigma' \mid \exists adl.\ AsmProg(\mathsf{res\_head}, \mathsf{res\_last}, \mathsf{res\_length}, \sigma', adl, code_{PT}(pt))\}\\ \text{where } p = (tenv, pt, gst)\end{array}$$

Proof:

1. After execution of the first three statements we have the preconditions necessary for the type table initialization ($\forall i < 4.\ \mathsf{sizes}_i = w(i2ty(i))$) and the free register list available.

2. Since during the first pass we call *stmt_code* in the loop through the reference list fdl which the procedure table is based on, then the postconditions of $first\_pass$ (and hence, the precondition for *links*) are similar to the ones stated by Theorem 8.5.15 replacing the functions for statements with the corresponding ones for a procedure list.

Proving the invariant includes nothing surprising, the intermediate result of code generation (pointed by variable tmp) after the i-th iteration is the following:

$$AsmProg(\mathsf{tmp\_head}, \mathsf{tmp\_last}, \mathsf{tmp\_length}, \tau_i, adl, code_{PT}(pt_0, \ldots pt_{i-1}))$$

Proving the maintenance of the invariant ($\tau \rightarrow \tau'$) to compensate the different recursion direction and to use the invariant at the previous step we need the additional lemma:

$$code_{PT}(pt_1 \circ pt_2) = code_{PT}(pt_1) \circ code_{PT}(pt_2),$$

which is straightforward by induction on pt_1. Thus,

$$\begin{array}{l} code_{PT}(pt_0, \ldots pt_i) = code_{PT}(pt_0, \ldots pt_{i-1}) \circ code_{PT}([pt_i]) = \\ code_{PT}(pt_0, \ldots pt_{i-1}) \circ code_{\mathcal{S}}(pt_i, snd(pt_i).body) \end{array}$$

The last appended operand is produced during loop iteration i and is appended to the existing piece of code (provided by the invariant at the previous step).

3. We emphasize some properties we have in state σ' after the first pass. During the first pass we have set the offset field for every procedure:

$$\forall i < |fdl|.\ \mathsf{of}_f(\mathsf{cnt}_f(fdl_i)) = ba(pt, pt_i) \tag{8.11}$$

The specification for the field *cmd* for all references from the linked list (based on the reference list dl) is the following:

$$\forall x \in_* map(\mathsf{cnt}_l, dl).\ \mathsf{cmd}_l(x) = map(\mathsf{cnt}_a, adl)_{\mathsf{of}_f(\mathsf{orig}_l(x)) + \mathsf{offs}_l(x)} \tag{8.12}$$

The latter holds after the appending of the code pieces generated for each procedure. This statement is illustrated in Figure 8.18, where the position of one jump instruction is shown with respect to different code pieces.

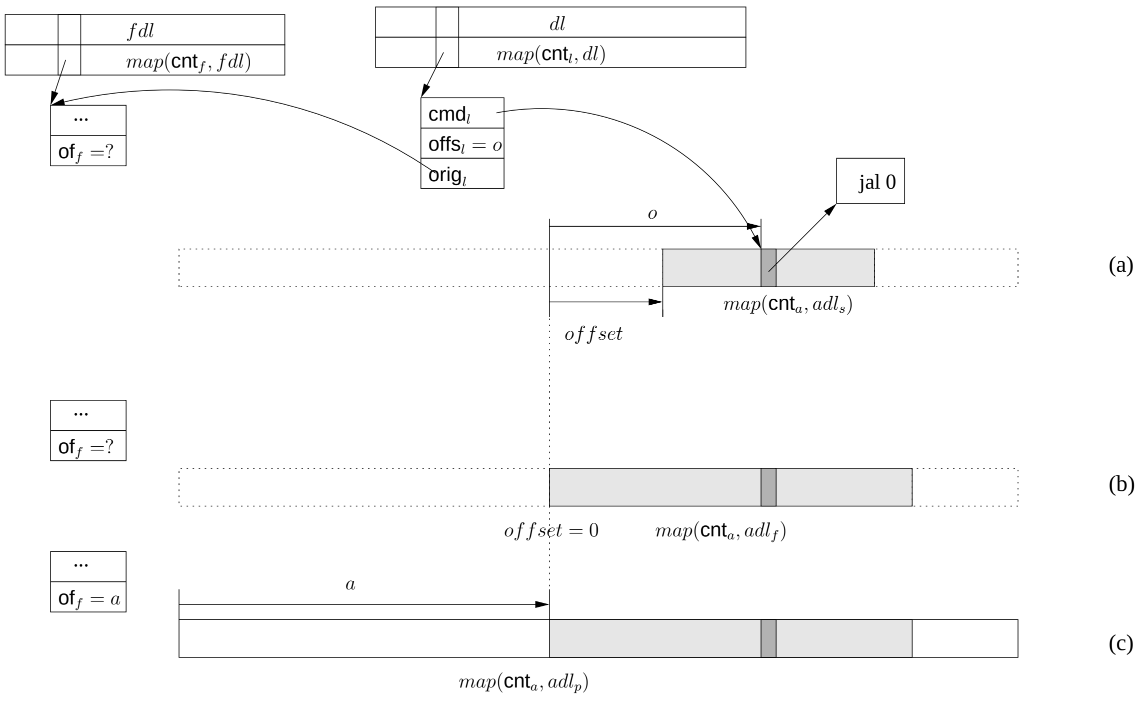


Figure 8.18: Connection between a link and position of the corresponding jump according to: (a) a statement code inside a procedure body code; (b) a procedure body code; (c) the program code

For some substatement s of a procedure body $snd(f).body$, the position of the jump is expressed through parameter $offset$ of the call and the offset kept in the corresponding link instance (see 8.18, (a)). In this case the jump position is computed according to the instruction list based on reference list adl_s, which is generated for s.

When we generate a code for a procedure body, parameter $offset$ of the generating function is set to zero. The jump position is defined by the corresponding link according to the generated reference list adl_f(see 8.18, (b)).

After the connection of code pieces for every procedure into the single instruction list based on adl_p, to define the position of the jump we need to take into account the offset of adl_f inside adl_p (see 8.18, (c)).

4. The next step is to verify the while loop of the program that actually performs the second pass. The part of the invariant we are interested in is:

$$\begin{aligned}
&(\mathsf{link} \neq Null \longrightarrow \exists i.\ \mathsf{link} = dl_i \wedge \\
&AsmProg(\mathsf{x\_head}, \mathsf{x\_last}, \mathsf{x\_length}, \tau, adl, prg)) \wedge \\
&(\mathsf{link} = Null \longrightarrow \\
&AsmProg(\mathsf{x\_head}, \mathsf{x\_last}, \mathsf{x\_length}, \tau, adl, pass2(code_{PT_0}(pt), 0, pt, ll))) \\
&\text{where } ll = links_{PT}(pt),\ prg = pass2(code_{PT_0}(pt), 0, pt, (ll_0, \ldots, ll_{i-1}))
\end{aligned}$$

The invariant together with the condition $\mathsf{link} = Null$, i.e. when loop is finished, obviously imply the claim using Theorem 8.5.6.

The proof that the invariant is maintained iterating the loop is not completely obvious, i.e.

$$\begin{aligned}
&I(\tau) \wedge \mathsf{link} \neq Null \longrightarrow I(\tau[\mathsf{imm}_a := \mathsf{imm}_a[l := nv], \mathsf{link} := \mathsf{nxt}_l(\mathsf{link})]),\\
&\text{where } l = \mathsf{cmd}_l(\mathsf{cnt}_l(\mathsf{link})),\\
&nv = (int(\mathsf{of}_f(\mathsf{call}_l(\mathsf{cnt}_l(\mathsf{link}))) -\\
&\qquad\qquad int(\mathsf{of}_f(\mathsf{orig}_l(\mathsf{cnt}_l(\mathsf{link}))) + \mathsf{offs}_l(\mathsf{cnt}_l(\mathsf{link})))\\
&-1) \cdot 4)
\end{aligned}$$

Let us consider the case when $\mathsf{link} \neq last(dl)$ which implies $\mathsf{nxt}_l(\mathsf{link}) = dl_{i+1}$ and not $Null$. We need to show that the structure variable x (decomposed in its fields in the invariant) now points to the concrete version of program $prg' = pass2(code_{PT_0}(pt), 0, pt, (ll_0, \ldots, ll_i)))$. After expanding abstraction function $AsmProg$ the only thing we need to show (the rest is independent from changes of imm_a) is that for all $j < |adl|$:

$$ref2instr(\mathsf{cnt}_a(adl_j), \tau, prg_j) \longrightarrow ref2instr(\mathsf{cnt}_a(adl_j), \tau', prg'_j)$$

Using lemma

$$\forall c.\ pass2(c, o, pl, ll \circ [l]) = pass2(c, o, pl, ll)[addr_l := jal(4 \cdot dist_l)], \tag{8.13}$$

that can be easily shown by induction on ll, we can decompose program prg' to $prg[addr_{ll_i} := jal(4 \cdot dist_{ll_i})]$.

From relation $LinkList$ considering the i-th element we have

$$ll_i.offs = \mathsf{offs}_l(\mathsf{cnt}_l(ldl_i)) \tag{8.14}$$

and that references $po = \mathsf{orig}_l(\mathsf{cnt}_l(\mathsf{link}))$ and $pc = \mathsf{call}_l(\mathsf{cnt}_l(\mathsf{link}))$ belong to reference list $\mathsf{cnt}_f(fdl)$. The last observation means that there exist two numbers $k, m < |fdl|$ such that $po = \mathsf{cnt}_f(fdl_k)$ and $pc = \mathsf{cnt}_f(fdl_m)$. From the same relation we have these pointers equivalent with abstract $ll_i.orig$ and $ll_i.call$. Considering $FuncList$ and $Func$ relations we also have these pointers equivalent to pt_k and pt_m abstract procedure entries. From the uniqueness of strings (Lemma 7.3.2) and the uniqueness property for procedure data structure we conclude that $ll_i.orig = pt_k$ and $ll_i.call = pt_m$ hold.

Using properties stated in (8.11, 8.12) we get $\mathsf{of}_f(po) = ba(pt, pt_k)$ and $\mathsf{of}_f(pc) = ba(pt, pt_m)$. Moreover, the address where the current link points is equal to:

$$\mathsf{cmd}_l(\mathsf{cnt}_l(ldl_i)) = map(\mathsf{cnt}, adl)_{ba(pt,pt_k)+\mathsf{offs}_l(\mathsf{cnt}_l(ldl_i))} \tag{8.15}$$

Taking into account (8.14) we have that the link points to $map(\mathsf{cnt}, adl)_{addr_{ll_i}}$.

Based on (8.13), for all $j \neq addr_{ll_i}$ we have $prg'_j = prg_j$. Our goal is to show that the latter observation on the specification will be supported by the similar behaviour in the implementation. Thus, we need to show that changing the heap function imm_a for the address in the program, which is computed based

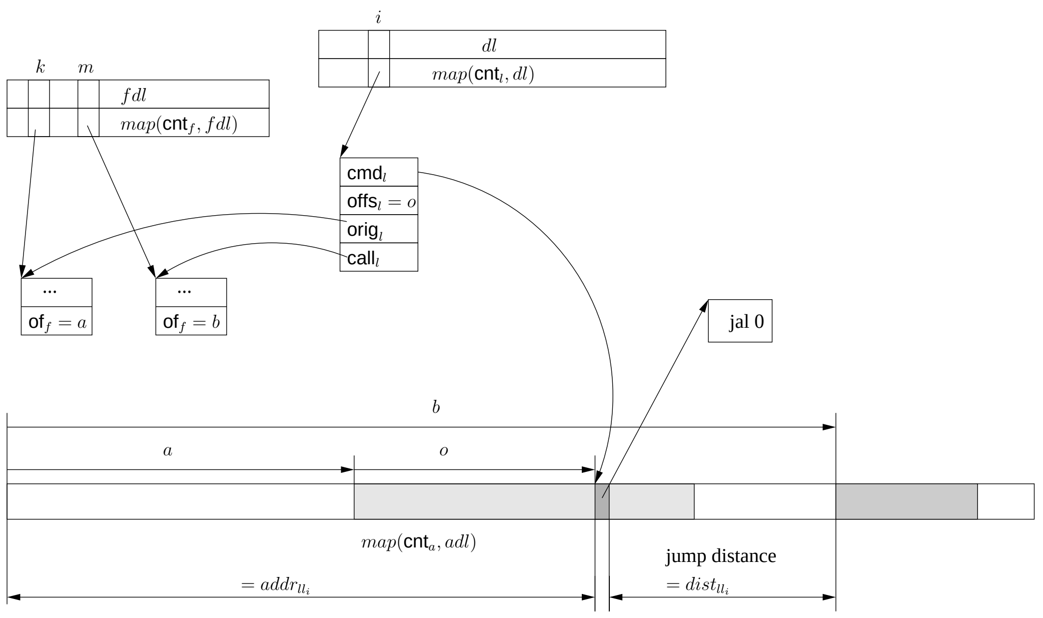


Figure 8.19: Computation of a jump distance in the implementation

on the current link, does not have any effect on the concrete implementation at position j as well. Formally, $\mathsf{cnt}_a(adl)_j \neq \mathsf{cnt}_a(adl)_{addr_{ll_i}}$ since we have shown that $\mathsf{cmd}_l(\mathsf{cnt}_l(ldl_i))$ points to position $addr_{ll_i}$. From the distinctness of $map(\mathsf{cnt}_a, adl)$ we have $\forall i, j < |adl|.\ i \neq j \longrightarrow \mathsf{cnt}_a(adl_i) \neq \mathsf{cnt}_a(adl_j)$, so it is enough to show $addr_{ll_i} < |adl|$.

Obviously, $ll_i.orig = pt_k \longrightarrow ll_i \in_* links_{\mathcal{S}}(pt_k, 0, snd(pt_k).body)$ and with Lemma 8.5.10 we have $ll_i.offs < csize_{\mathcal{S}}(pt_k, snd(pt_k).body)$. For all $i < |pt|$ we can show by induction that $ba(pt, pt_i) + csize_{\mathcal{S}}(pt_i, snd(pt_i).body) < csize_{PT}(pt)$. Moreover, since $|pt| = |fdl|$ and $csize_{PT}(pt) = csize_{PT_0}(pt) = |adl|$ we have finished the case.

For $j = addr_{ll_i}$ the only thing to show is that at the position, where we update imm_a, the instruction opcode is exactly jal, as we only rewrite the immediate constant in the implementation and the whole instruction in the specification. Figure 8.19 illustrates the computation of the jump distance in the implementation and the structures involved.

Using the following lemma about links collected from the procedure table (whose proof requires a similar lemma for $links_{\mathcal{S}}$ in order to be proven)

$$\forall l \in_* links_{PT}(pt).\ \exists imm.\ code_{PT_0}(pt)_{addr_l} = jal(imm)$$

we get that there exists some imm such that $ref2instr(\mathsf{cnt}_a(adl_j), \sigma', jal(imm))$ holds (recall that σ' is the state after the first pass). As we know that the heap function opc_a was not changed since then, we have $\mathsf{opc}_a(\mathsf{cnt}_a(adl_j)) = jal$ in state τ. With the observation presented above we obviously have

$$nv = int(\mathsf{of}_f(\mathsf{call}_l(\mathsf{cnt}_l(\mathsf{link}))) - int(\mathsf{of}_f(\mathsf{orig}_l(\mathsf{cnt}_l(\mathsf{link}))) + \mathsf{offs}_l(\mathsf{cnt}_l(\mathsf{link}))) - 1) = dist_{ll_i}$$

and hence, $ref2instr(\mathsf{cnt}_a(adl_j), \tau', jal(4 \cdot dist_{ll_i}))$ holds. $\square$

Chapter 9

Conclusion

9.1 Summary

The work described in this thesis has resulted in some nontrivial results. We considered the small-step semantics and the Hoare logic as a practical means for verification of sequential imperative programs.

An important result we obtained by having proved correctness of an example program in the frame of the C0 small-step semantics. We showed the ability to do such a proof. In addition to the showed program correctness we have given (very rudimentary) evidence that the semantics performs as intended. Moreover, the verified function operates with *new* operator and pointer data structures, that is usually avoided in verification examples for semantics with deep embedded memory models.

The first intention to use small-step semantics as an only means for program verification has revealed itself as infeasible for real-sized programs. The main reason for that is an involved memory model and the deep embedded expressions. Thus, the proof of results of expression evaluation in such a memory model is very time consuming. Despite this the small-step semantics is still an appropriate means if we want to argue about small programs, that we are not able (by some reason) to handle in the Hoare Logic.

We have implemented a non-optimizing compiler for the C0-language into VAMP assembler grounding on the formal description of the compilation rules in Isabelle/HOL. The part of the compiler implementing these rules is written in C0 as well. The compiler is integrated as a back-end into a tool running on an existing operating system.

The most significant result of this work is mechanized verification of the compiler back-end implementation mentioned above. This was performed in the verification environment based on Hoare Logic [3]. The flat memory model with independent variable updates and shallow embedding of expressions/types used in the environment, eliminates some problems which we faced during the verification of the example program using the C0 small-step semantics. Since the proof of the example presented in the first part of the thesis would cost to an experienced user not more than one hour being done inside the Hoare Logic verification environment, the perspective for proving real-sized programs is very promising.

This was totally confirmed by the successful verification of the compiler implementation we have mentioned above. Thus, the compiler we have verified is the largest mechanically verified application written in an imperative language (with comparable features), we are aware of.

The implementation is about 1500 lines of source code organized in 60 procedures. Its verification took about 10 months, 900 lemmas, and 25000 proof steps. One of the crucial technical tasks during the verification was to show the equivalence between the implementation in an imperative language and the specification, which is written, in fact, in a functional ML-like language. Thus, e.g. the equivalence of while-constructs to recursive calls had to be shown. Taking into account the author's lack of the experience with the Isabelle/HOL theorem prover (which the environment is built on) and the verification environment at the beginning of the work, the timing can be improved even more by an experienced scientist.

9.2 Work Building on this Thesis

Work building on the basis of this thesis has several directions concerning implementation as well as verification, which extend the functionality of the compiler. The topics we list have been covered during the time the thesis was written or are ongoing work.

Inline Assembler One of the modifications is the extension of the compiler for the support of inline assembler. This is clearly indispensable for writing system-level software (e.g. operating system, drivers).

The changes in the compiler implementation are minimal: i) an additional data structure for the incoming assembler statement and ii) one more if-branch when generating code for statements considering an assembler block node in the syntax tree.

The verification side will need the following changes:

- create the abstraction function to the assembler instruction type I
- embed it into abstraction function $Stmt$ as a separate case
- specify and verify the code generation procedure for the assembler statement, which in most cases just copies the instructions. For the load/store variable macros, we also need to correctly generate an immediate constant.
- extend the proof for the general statement procedure. Since it is a separate if-branch, it will not influence any other kinds of statements.

Thus, the changes are completely irrelevant to most parts of the existing proof. This work has been carried out by myself.

Total Correctness Since the presented work covers only the partial correctness of the implementation, the natural extension is to prove the total correctness accompanied by guards. Guards are additional conditions included into the Hoare rules to model runtime faults. A command is executed if and only if the guard tests hold. Guards are included to test fault possibilities during the expression evaluation: in the branching conditions, assignments, function calls. The verification environment supports automatic inclusion of the following guard categories into the verification goal: check for null pointer dereferencing; over-/underflow check for types, that are modelled as infinite in Isabelle/HOL; array boundary violation.

Thus, two sorts of additional subgoals need to be shown for every procedure. The first one is the monotonicity of some measure for every recursive procedure or a procedure which includes loops. The measure is a function modelling the number of remaining recursion steps, which obviously needs to decrease during the execution. Measure functions for the compiler implementation will be typically connected to the length of a list or to the depth of a tree to be traversed from the current state. The second part of subgoals to be proved concerns showing that guards are not violated during the execution.

The proof of the total correctness can be supported by the tools developed in the frame of the Verisoft project. A software model checker called ACSAR [63] and termination analysis ([64,65]) embedded into the verification environment [66] allow to discharge guards (arithmetic overflows, array bound checks, null pointer testing) automatically. The guards that could not be discharged are to prove interactively.

Usually, all the information needed to prove those of them, which depend on the structure of the used data, is already included in their abstraction functions. However, there can be situations when we need to extend an abstraction function to show that some pointers are not null. For instance, some defined value of $p \to f$ does not imply that $p \neq Null$. Proving partial correctness in some cases it is enough to have information about $p \to f$. Thus, information that p is not a null pointer could be missing.

The restrictions on array sizes and ranges of variable values are also needed to be included in preconditions. Again, in our implementation such an extension will not always be necessary. For example, to test against array boundaries for opc_a field we do not need include more information in the precondition. We already have the necessary assumption that $\forall x.\ |\mathsf{opc}_a(x)| = 4$ in the preconditions.

The proof extension to total correctness has been finished by myself and was not very time consuming.

Binary Output Since the output format of the compiler implementation is a list of assembler instruction, the compiler implementation can be extended with a procedure which codes assembler in the corresponding binary output. Since the translation between assembler instructions and the format of VAMP instruction words is well established, the implementation of such a procedure is straightforward. The proof of the correctness for such a translation (namely, the translation algorithm modelled in Isabelle/HOL) is not trivial being performed in Isabelle/HOL. This is the case because of technical difficulties working with Isabelle-specific realization of bit vectors. The proof, that the implementation satisfies the specification is rather simple. Both subtask were carried out by Alexandra Tsyban as a part of Verisoft.

Garbage Collector As it was mentioned before, C0 does not include a special statement for dynamic memory deallocation. One of the following extensions to the current work is implementation and verification of a garbage collector for our compiler. There are two main parts in this work: i) implementation of a garbage collector as a single program, ii) incorporation of the garbage collector into the existing framework.

For the first task any garbage collection algorithm can be chosen, our intention is to have a *copying garbage collector* [67]. A simplest variant of a copying garbage collector works in two memory spaces of equal size. Only one of the spaces is used at a time. When the allocation in the currently active space is no more possible, the garbage collector traverses all of the reachable cells in the active space and copies them into the inactive space. Then the spaces are switched. The traversal of reachable cells can be done using a depth-first search, which takes time proportional to the number of reachable cells. The most important property of the copying garbage collection is that its runtime is independent of the amount of garbage (since it never visits garbage cells) and it depends only on the number and size of reachable cells.

The second task requires creation of the interface between the compiler and the garbage collector to provide the necessary information about pointer locations and program types for the allocation/deallocation of the memory. Moreover, we need to incorporate the source code of the garbage collector into the code of the compiled program and modify the `new` operator of the compiler to call the garbage collector. The garbage collector verification is a work in progress.

Appendix A

Summary of the VAMP Instruction Set

The VAMP instruction set is summarized based on [55] with slight changes (mostly concerning opcode names). We omit some instructions that are not used in the compiler output.

A.1 Bitvectors

First we introduce some notations for bit vectors.

For set of boolean values $\mathbb{B} = \{0, 1\}$ we denote by $a \in \mathbb{B}^n$ a bitvector a of length n, which is any sequence of boolean values.

A bit at position j in bitvector a is denoted by $a[j]$, the subvector including bits from position k to position j (we assume $k < j$) is denoted as $a[j : k]$. By x^n we denote a bit string consisting of n copies of x, e.g. $0^4 = 0000$.

Concatenation of bitvectors is denoted without any additional symbols, e.g. $a[0]^4a[20 : 12]$ denotes the bitvecor, where the 4 most significant bits are copies of bit $a[0]$ concatenated with subvector $a[20 : 12]$.

The natural number represented by binary string $a \in \mathbb{B}^n$ is computed as:

$$\langle a \rangle = \Sigma_{i=0}^{n-1} a_i \cdot 2^i$$

A.2 Instruction Set

VAMP uses 32 general purpose registers; *GPR* denotes register file, s.t. $GPR \in \{0, \ldots, 31\} \to \mathbb{B}^{32}\}$. Register $GPR[0]$ always keeps zero value.

Delayed Branch Suppose instruction I_i fetched in cycle T is an any control operation (i.e. any jump or branch instruction). We say that branch is taken if by results of the control instruction we need to perform a jump,i.e. branch is taken if I_i is a jump instruction or if it is *beqz* and value in the corresponding $RS1$ register is equal to zero. If branch is taken the next value of program counter depends on I_i itself and cannot be computed before cycle $T + 1$. So, the first instruction of the code piece where the jump was done can only be fetched at cycle $T + 2$.

There are different concepts that allow to overcome this problems and provide correct computation of the program counter.

VAMP implements the delayed PC concept, i.e. computation of the program counter is delayed by one instruction. The VAMP configuration d has two program counters: the program counter $d.pc$ and the delayed program counter $d.dpc$. The delayed program counter is used to fetch instructions from the memory.

$$d'.dpc = d.pc$$
$$I(d) = d.m[d.dpc : d.dpc + 3],$$

where d' denotes one computation step of the VAMP, $I(d)$ is the fetched instruction at d, and $d.m$ stays for the memory component of the VAMP machine. Thus, instruction I_{i+1} (called the delayed slot), which is placed after any control instruction, will be executed before the execution of the code where the jump was done will be started. Thus it does not matter, whether the branch was taken or not I_{i+1} will be always executed after I_i. There is a restriction that the delay slot cannot be any control instruction. Obviously, the most safe way for writing program is filling delay slots by *nop* instructions. However, clever usage of delay slot when programming increases performance. For example, the delay slot can be used to store the return address somewhere when performing a jump-and-link instruction. For construction and proof simplicity the presented C0 compiler fills delayed slots by *nop* operations.

Instruction Formats VAMP has several instruction formats, we are especially interested in three presented in Figure A.1. The I-type (immediate) format specifies an instruction with two registers and a 16-bit immediate operand. The J-type(jump) format specifies only a 26-bit immediate operand and no register operand; it is used for control instructions. The R-type(register) format provides three general purpose registers, an additional opcode (*Function* field), and a 5-bit constant for a shift amount.

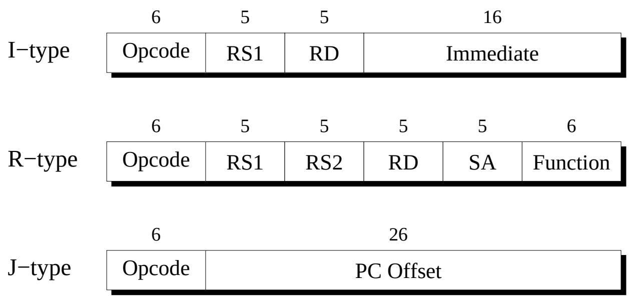


Figure A.1: Instruction Formats of the VAMP

Let us shortly present the notations used in the tables below.

- *IR* stands for an instruction word
- RD stands for a destination register
- RS1 and RS2 are source registers

- SA specifies an immediate shift amount
- *imm* stands for an immediate constant

Load and store instruction transfer data between *GPR* and memory (denoted by M). In the table below *pa* stands for the effective address, which is computed according to the following equation:

$$pa = \langle GPR[RS1]\rangle + \langle sxt(IR[15:0])\rangle,$$

where $\langle a\rangle$ denotes conversion of a bit string a to a natural number, $sxt(a)$ denotes a sign-extended version of a, i.e.

$$sxt(IR[15:0]) = IR[15]^{16}IR[15:0]$$

Thus, Tables A.1, A.2, A.3 give an overview of available instructions of I-type, R-type, and J-type, respectively.

$IR[31:26]$	Mnem.	d	Effect
Memory operations			
100000	*lb*	1	$RD = sext(M[pa+d-1:pa])$
100001	*lh*	2	$RD = sext(M[pa+d-1:pa])$
100011	*lw*	4	$RD = M[pa+d-1:pa]$
100100	*lbu*	1	$RD = 0^{24}M[pa+d-1:pa]$
100101	*lhu*	2	$RD = 0^{16}M[pa+d-1:pa]$
101000	*sb*	1	$M[pa+d-1:pa] = RD[7:0]$
101001	*sh*	2	$M[pa+d-1:pa] = RD[15:0]$
101011	*sw*	4	$M[pa+d-1:pa] = RD$
Arithmetic, logical operation			
001000	*addi*		$RD = RS1 + imm$
001010	*subi*		$RD = RS1 - imm$
001100	*andi*		$RD = RS1 \wedge imm$
001101	*ori*		$RD = RS1 \vee imm$
001110	*xori*		$RD = RS1 \oplus imm$
001111	*lhgi*		$RD = imm0^{16}$
Test and set operations			
011001	*sgri*		$RD = 0^{31}(RS1 > imm)$
011010	*seqi*		$RD = 0^{31}(RS1 = imm)$
011011	*sgei*		$RD = 0^{31}(RS1 \geq imm)$
011100	*slsi*		$RD = 0^{31}(RS1 < imm)$
011101	*snei*		$RD = 0^{31}(RS1 \neq imm)$
011110	*slei*		$RD = 0^{31}(RS1 \leq imm)$
Control operation			
000100	*beqz*		$PC' = PC' + 4 + (RS1 = 0 ? imm : 0)$
000101	*bnez*		$PC' = PC' + 4 + (RS1 \neq 0 ? imm : 0)$
000110	*jr*		$PC' = RS1$
000111	*jalr*		$R31 = PC' + 4; PC' = RS1$

Table A.1: I-type Instruction Layout

$IR[5:0]$	Mnem.	Effect
Shift operations		
000000	*slli*	$RD = RS1 \ll SA$
000001	*slai*	$RD = RS1 \ll SA$ (arith.)
000010	*srli*	$RD = RS1 \gg SA$
000011	*srai*	$RD = RS1 \gg SA$ (arith.)
000100	*sll*	$RD = RS1 \ll RS2[4:0]$
000101	*sla*	$RD = RS1 \ll RS2[4:0]$ (arith.)
000110	*srl*	$RD = RS1 \gg RS2[4:0]$
000111	*sra*	$RD = RS1 \gg RS2[4:0]$ (arith.)
Arithmetic and logical operations		
100000	*add*	$RD = RS1 + RS2$
100001	*addu*	$RD = RS1 + RS2$ (no overfl.)
100010	*sub*	$RD = RS1 - RS2$
100011	*subu*	$RD = RS1 - RS2$ (no overfl.)
100100	*and*	$RD = RS1 \wedge RS2$
100101	*or*	$RD = RS1 \vee RS2$
100110	*xor*	$RD = RS1 \oplus RS2$
100111	*lhg*	$RD = RS2[15:0]0^{16}$
Test and set operations		
101001	*sgr*	$RD = 0^{31}(RS1 > RS2)$
101010	*seq*	$RD = 0^{31}(RS1 = RS2)$
101011	*sge*	$RD = 0^{31}(RS1 \geq RS2)$
101100	*sls*	$RD = 0^{31}(RS1 < RS2)$
101101	*sne*	$RD = 0^{31}(RS1 \neq RS2)$
101110	*sle*	$RD = 0^{31}(RS1 \leq RS2)$

Table A.2: R-type Instruction Layout
Note that $IR[31:26] = 0^6$ holds for all instructions in this table and that we identify a boolean value of $true$ with 1 and $false$ with 0.

$IR[31:26]$	Mnem.	Effect
000010	*j*	$PC' = PC' + 4 + imm$
000011	*jal*	$GPR[31] = PC' + 4$; $PC' = PC' + 4 + imm$

Table A.3: J-type Instruction Layout

Appendix B

Code generation templates

B.1 Expression Code Generation

The code generation for binary operators:

$$code_{op_b}(d, d_2, d_1, t_2, t_1, op) =$$

$$\begin{cases} [seq(d, d_1, d_2)] & \text{if } op = equal \\ [sne(d, d_1, d_2)] & \text{if } op = not\_equal \\ [sls(d, d_1, d_2)] & \text{if } op = less \wedge t_1 \neq UsgnT \\ unsigned\_code\_comp(d, d_1, d_2, False) & \text{if } op = less \wedge t_1 = UsgnT \\ [sle(d, d_1, d_2)] & \text{if } op = lessequal \wedge t_1 \neq UsgnT \\ unsigned\_code\_comp(d, d_1, d_2, True) & \text{if } op = lessequal \wedge t_1 = UsgnT \\ [sgr(d, d_1, d_2)] & \text{if } op = greater \wedge t_1 \neq UsgnT \\ unsigned\_code\_comp(d, d_1, d_2, False) & \text{if } op = greater \wedge t_1 = UsgnT \\ [sge(d, d_1, d_2)] & \text{if } op = greaterequal \wedge t_1 \neq UsgnT \\ unsigned\_code\_comp(d, d_1, d_2, True) & \text{if } op = lessequal \wedge t_1 = UsgnT \\ [sll(d, d_1, d_2)] & \text{if } op = shiftleft \\ [srl(d, d_1, d_2)] & \text{if } op = shiftright \\ [add(d, d_1, d_2)] & \text{if } op = plus \\ [sub(d, d_1, d_2)] & \text{if } op = minus \\ mult\_template(d, d_1, d_2) & \text{if } op = times \\ div\_template() & \text{if } op = divide \end{cases}$$

The code generation for the unsigned comparison (parameter *eq* defines the test: either less or less-or-equal):

$$\begin{aligned} & unsigned\_code\_comp(d, d_1, d_2, eq) \equiv \\ & [\begin{cases} sle(d, d_1, d_2) & \text{if } eq \\ sls(d, d_1, d_2) & \text{if } \neg eq \end{cases}, \\ & xor(R_1, d_1, d_2), \\ & sls(R_1, R_1, R_0), \\ & xor(d, d, R_1)] \end{aligned}$$

Software emulation of multiplication ($mult\_template(d, d_1, d_2)$):

$$\begin{array}{ccc}
[beqz(d_1, 84), & beqz(R_1, 12), & add(d, d, d_1), \\
xor(d, d, d), & nop, & j(-24), \\
sls(R_1, d_1, 0), & xori(R_2, R_2, 1), & slli(d_1, d_1, 1), \\
beqz(R_1, 12), & sub(d_2, 0, d_2), & beqz(R_2, 8), \\
xor(R_2, R_2, R_2), & beqz(d_2, 24), & nop, \\
ori(R_2, R_2, 1), & andi(R_1, d_2, 1), & sub(d, 0, d)] \\
sub(d_1, 0, d_1), & beqz(R_1, 8), & \\
sls(R_1, d_2, 0), & srli(d_2, d_2, 1), &
\end{array}$$

Software emulation of division ($div\_template()$):

$$\begin{array}{ccc}
[beqz(d_2, -4), & sls(R_1, d_2, R0), & srli(d_2, d_2, 1), \\
xor(d, d, d), & bnez(R_1, 36), & sls(R_1, d_2, R_3), \\
sls(R_1, d_1, d_2), & nop, & bnez(R_1, 36), \\
xor(R_2, d_1, d_2), & slli(d_2, d_2, 1), & nop, \\
sls(R_2, R_2, R0), & sle(R_1, d_2, d_1), & slli(d, d, 1), \\
xor(R_1, R_1, R_2), & xor(R_2, d_1, d_2), & sle(R_1, d_2, d_1), \\
bnez(R_1, 124), & sls(R_2, R_2, R0), & beqz(R_1, 12), \\
addi(R_3, d_2, 0), & xor(R_1, R_1, R_2), & nop, \\
sls(R_1, d_2, R0), & bnez(R_1, -36), & sub(d_1, d_1 d_2), \\
beqz(R_1, 12), & nop, & addi(d, d, 1), \\
nop, & srli(d_2, d_2, 1), & j(-44), \\
j(104), & sub(d_1, d_1, d_2), & nop] \\
addi(d, d, 1), & ori(d, d, 1), &
\end{array}$$

The code generation for unary operators:

$$code_{un\_op}(d, d_1, op) = \begin{cases}
[xori(d, d_1, 1)] & \text{if } op = log\_not \\
[xori(d, d_1, -1)] & \text{if } op = bw\_neg \\
[sub(d, R_0, d_1)] & \text{if } op = un\_minus \\
[addi(d, d_1, 0)] & \text{if } op = to\_int \\
[addi(d, d_1, 0)] & \text{if } op = to\_unsigned \\
[slli(d, d_1, 24), srai(d, d, 24)] & \text{if } op = to\_char
\end{cases}$$

B.2 Statement Code Generation

In this section we present some templates used in the code generation for statements.

The code generation template for memory allocation ($Alloc$ statement) $alloc\_template(d, a)$:

$$\begin{array}{ccc}
[sls(R_2, HR, R0), & add(R_3, R0, HR), & j(-36), \\
bnez(R_2, 16), & slri(R_1, R_1, 1), & nop \\
addi(R_1, R0, a), & sub(R_3, R_3, R_1), & subi(R_3, R_3, a/4), \\
slei(R_2, HR, a), & beqz(R_3, 44), & sub(HR, HR, R_3), \\
bnez(R_2, 92), & slri(R_1, R_1, 1), & j(8), \\
nop, & slsi(R_2, R_1, a/4), & slli(HR, HR, 2), \\
slri(HR, HR, 2), & bnez(R_2, 24), & addi(HR, R0, a), \\
slri(R_1, R_1, 2), & sle(R_2, R_1, R_3), & sw(HR, d, 0) \\
slli(R_1, R_1, 1), & beqz(R_2, 8), & addi(HR, HR, s)] \\
sle(R_2, R_1, HR), & nop, & \\
bnez(R_2, -12), & sub(R_3, R_3, 1), &
\end{array}$$

The template which fills up the new frame header ($Call$ statement)$init\_frame(d, lba)$:

$$\begin{array}{l}
[add(R_2, R_0, LR), \\
add(LR, LR, R_1), \\
sw(d, LR, 4), \\
sw(R_2, LR, 8)]\circ \\
code_{\mathcal{C}}(R_1, Usgn(lba))\circ \\
[sw(R_1, LR, 12), \\
jal(0), \\
Isw(R31, LR, 0)]
\end{array}$$

The restore template, which pops the previous frame of the stack ($Return$ statement):

$$\begin{array}{l}
[lw(R_3, LR, 0), \\
lw(LR, LR, 8), \\
jr(R_3), \\
nop]
\end{array}$$

Appendix C

Lemmata Correspondence

Here we point for some of the lemmata and definitions used in the thesis, where they can be found in the Isabelle/HOL theories. The list is of cause is not complete, but it gives an idea where to find the necessary information. The full list is to large to place it here.

The correspondence sometimes is not very precise, so the lemma we have presented in the thesis can slightly differ from the source in order to better fit in the paper-and-pencil description. However, it can be easily derived from the source variant. By capital letters D and L we denote the definition (if it is given) and the lemma numbers. The last column contains the name of the theory where the definition or lemma is given. If a location is marked with (I/H), then the definition/lemma is included in the standard distribution of Isabelle/HOL.

Number	Name	Theory
Basics		
D 1.4.14	*takeWhile*	`List` (I/H)
D 1.4.15	*drop_with*	`verification/libisa/MoreList`
L 1.4.16	*rev_drop_with_takeWhile_eq*	`verification/libisa/MoreList`
L 1.1	*list_update_append*	`List` (I/H)
ε	*SOME*	`Hilbert_Choice(I/H)`
L 1.4.21	*some_equality*	`Hilbert_Choice(I/H)`

Number	Name	Theory
C0 Semantics		
D 2.1.1	*Ty*	`c0syntax/Type`
D 2.1.2	*valid_type*	`semantics_isa/t_spec`
D 2.1.3	*sizeof_type*	`semantics_isa/sizeof`
D 2.1	*prim*	`c0syntax/Value`
D 2.1.7	*expr*	`c0syntax/Expr`
D 2.1.9	*typeof_expr*	`semantics_isa/typeof_expr`
$valid_{\mathcal{E}}$	*valid_expr*	`semantics_isa/valid_expr`
D 2.1.10	*stmt*	`c0syntax/Stmt`
$valid_{\mathcal{S}}$	*valid_stmt*	`semantics_isa/valid_stmt`
D 2.1.11	*s2l*	`c0syntax/Stmt`
D 2.1.12	*procT*	`semantics_isa/proctable`
D 2.1	*valid_functions*	`semantics_isa/valid_conf`
D 2.1	*valid_proctables*	`semantics_isa/valid_conf`
$distinct_{\mathcal{S}}$	*stmts_distinct*	`semantics_isa/stmt_structure`
$distinct_{\mathcal{P}}$	*stmts_distinct_pt*	`semantics_isa/program_structure`

Number	Name	Theory
Abstract Compiling Function		
		`compiler/...`
D 6.1.1	*close_dvd*	`.../ceiling`
L 6.1.2	*ceiling_close_dvd*	`.../ceiling`
D 6.1.3	*algn_type*	`.../asize`
L 6.1.4	*algn_type_not_zero*	`.../asize`
D 6.1.5	*asize_type*	`.../asize`
D 6.1.6	*displ_var*	`.../displ`
L 6.1.7	*displ_var_Suc*	`.../displ_lemmas`
D 6.2.1	*addr_deref_code*	`.../codegen_expr`
$code_{\mathcal{E}}$	*codegen_expr*	`.../codegen_expr`
$csize_{\mathcal{E}}$	*codesize_expr*	`.../codesize_expr`
L 6.2.2	*codesize_expr_correct*	`.../codesize_expr_lemmas`
D 6.1	*codegen_functionlist*	`.../codegen_main`

Number	Name	Theory
Data Abstractions: *dList*		
		`datastructures/dList/HeapdList`
D 7.1.1	*dList*	
L 7.1.2	*dList_Null_head*	
	dList_ptr1_is_head	
L 7.1.3	*dList_Suc_nth_is_next*	
L 7.1.4	*dList_next_last_is_Null*	
L 7.1.5	*dList_distinct*	
L 7.1.6	*dList_unique*	
L 7.1.7	*dList_heaps_eq1*	

Number	Name	Theory
Data Abstractions		
		`datastructures/String/HeapString`
D 7.3.1	*String*	
L 7.3.2	*String_list_and string_are_unique*	
		`c0compiler/Partial/...`
String_cont	*String_cont*	`.../stmt_heap`
L 7.3.3	*String_String_cont*	`.../function_heap_lemmas`
		`datastructures/Tree/...`
D 7.1.8	*'a tree*	`.../Tree`
D 7.1.10	*left_tree*, *right_tree*	`.../Tree`
D 7.1.11	*set_of*	`.../Tree`
D 7.1	*Tree*	`.../HeapTree`
D 7.1.15	*Tree_unique*	`.../HeapTree`
		`c0compiler/Partial/...`
D 7.4.1	*Compnt*	`.../struct_component_heap`
D 7.4.2	*CompList*	`.../struct_component_heap`
D 7.4.3	*convert_type*	`.../typetable_heap`
D 7.4.6	*convert_ty*	`.../convert_type_into_ty`
D 7.4.7	*type_distinct*	`.../typetable_heap`
D 7.4.9	*correct_typetable*	`.../typetable_heap`
D 7.4.4	*name2type*	`.../convert_type_into_ty`
D 7.4.10	*Typetable*	`.../typetable_heap`
L 7.4.11	*name2type_result1*	`.../convert_type_into_ty`
L 7.4.12	*convert_ty_∗*	`.../convert_type_into_ty`
L 7.4.13	*convert_ty_arr*	`.../convert_type_into_ty`
L 7.4.14	*convert_ty_str*	`.../convert_type_into_ty`
L 7.4.15	*Typetable_valid_types*	`.../valid_typetable`
D 7.5.1	*Var*	`.../variable_heap`
L 7.5.3	*VarList_valid_symboltable*	`.../variable_heap`
D 7.6.2	*convert_to_node convert_expr*	`.../expr_heap`
D 7.6.3	*Expr*	`.../expr_heap`
L 7.6.4	*Expr_implies_left_expr*	`.../expr_heap_lemmas`
D 7.6.5	*node_type*	`.../expr_heap`
L 7.6.6	*type_env_map_of_Ptr Convert_expr_Deref*	`.../expr_heap_lemmas`
L 7.6.7	*Expr_typeof_expr*	`.../expr_heap_lemmas`
L 7.6.8	*Expr_in_valid_exprs*	`.../expr_heap_lemmas`
L 7.6.9	*Expr_size_tree_size*	`.../expr_heap_lemmas`
D 7.7.1	*convert_to_s_node convert_stmt*	`.../stmt_heap`
D 7.7.2	*Stmt*	`.../stmt_heap`
L 7.7.3	*Stmt_impl_left_Stmt*	`.../stmt_heap_lemmas`
D 7.8.1	*Function*	`.../function_heap`
L 7.8.3	*Stmt_in_valid_stmts*	`.../function_heap_lemmas`
D 7.9.1	*CompilableProgram*	`.../first_pass`

Number	Name	Theory
Data Abstractions		
		`c0compiler/Partial/...`
D 7.10.2	*is_trans_asm*	`.../asm_prog_heap`
D 7.10.3	*AsmProgPair*	`.../asm_prog_heap`
L 7.10.4	*AsmProgPair_next_2*	`.../asm_heap`
L 7.10.5	*AsmProgPair_append_progs_2*	`.../asm_heap`
L 8.2.1	*comute_align$_s$pec*	`.../align_computation`
L 8.2.2	*algn_struct_components_rev*	`.../asize_displ_lemmata`
L 8.2.3	*min_gt_div_spec*	`../ceiling_computation`
D 8.2.4	*FilledTypetable*	`.../convert_type_into_ty`
L 8.2.5	*compute_asize_displ_spec*	`.../asize_computation`
L 8.3.3	*vars_allocation_spec*	`.../variable_` `displacement_computation`
		`datastructures/Pair/...`
L 8.4.1	*pair_insert_end_spec*	`.../pair_operations`
L 8.4.2	*pair_append_spec*	`.../pair_operations`
L 8.4.3	*create_n_instr_spec*	`.../expr_eval_help`
L 8.4.4	*i_type_instr_spec*	`.../expr_eval_help`
L 8.4.5	*add_deref_addr_spec*	`.../expr_eval_short`
L 8.4.6	*expr_eval_step_spec*	`.../expr_codegen_step`
L 8.4.7	*expr_eval_code_spec*	`.../expr_codegen`
D 8.5.1	*link*	`.../link`
D 8.5.2	*Link*	`.../link_heap`
D 8.5.4	*collect_links_in_stmt*	`.../link`
D 8.5.5	*set_links*	`.../link`
L 8.5.7	*link_append_links_in_first_aux*	`.../link`
L 8.5.8	*link_append_links_in_second_aux*	`.../link`
L 8.5.9	*set_links_append_aux*	`.../link`
L 8.5.10	*collect_links_curr_func*	`.../link`
L 8.5.12	*set_links_in_stmt_aux*	`.../link`
L 8.5.6	*set_links_aux_collect_links_in_prog*	`.../link`
L 8.5.15	*code_gen_statements*	`.../stmt_code_gen`
L 8.6.1	*compile_spec*	`.../compile`

Index

Bibliography

[1] Dirk Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, Computer Science Department, 2007. Draft.

[2] The Verisoft Consortium. The Verisoft project. http://www.verisoft.de/, 2003.

[3] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2005.

[4] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[5] Sven Beyer. *Putting It All Together: Formal Verification of the VAMP*. PhD thesis, Saarland University, Computer Science Department, March 2005.

[6] S.Owre, N.Shankar, and J.M.Rushby. PVS: A prototype verification system. In *CADE 11*, volume 607 of *LNAI*, pages 748–752. Springer, 1992.

[7] Glynn Winskel. *The Formal Semantics of Programming Languages*, volume II: Compiling. The MIT Press, 1997.

[8] L.M. Chirica and D.F.Martin. Toward compiler implementation correctness proofs. In *ACM Transactions on Programming Languages and Systems*, volume 8(2), 1986.

[9] 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, pages 122–136, 1998.

[10] W.R. Bevier, W.A. Hunt, Jr., J. S. Moore, and W.D. Young. An approach to systems verification. 5(4):411–428, December 1989.

[11] 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, pages 161–172. Springer, 2003.

[12] C. A. R. Hoare and N.Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2:335–355, 1973.

[13] Michael Norrish. *C Formalised in HOL*. PhD thesis, University of Cambridge, Computer Laboratory, December 1998.

[14] Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In Gerwin Klein, editor, *Proceedings of the NICTA workshop on OS Verification 2004*, pages 99–121. National ICT Australia, 2004.

[15] William D. Young. Verified compilation in micro-gypsy. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 10 – 26, 1989.

[16] Matthew Wilding. A mechanically verified application for a mechanically verified environment. In *Computer Aided Verification*, pages 268–279, 1993.

[17] R.S.Boyer and J.S.Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[18] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[19] Yuan You. *Automated Proofs of Object Code for a Widely Used Microprocessor*. PhD thesis, University of Texas at Austin, 1992.

[20] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[21] A. Faivre J.-M. Meynadier P. Behm, P. Benoit. Meteor: A successful application of b in a large project. In *FM'99 - Formal Methods: World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999*, volume 1 of *Lecture Notes in Computer Science*, pages 369–387. Springer, 1999.

[22] A. Amelot F. Badeau. Using b as a high level programming language in an industrial project: Roissy val. In *ZB 2005: Formal Specification and Development in Z and B*, Lecture Notes in Computer Science, pages 334–354. Springer, 2005.

[23] R. Chapman A. Pryor S. King, J. Hammond. Is proof more cost-effective than testing? In *Software Engineering, IEEE Transactions on Software Engineering*, volume 26, pages 675–686, 2000.

[24] F.Mehta and T.Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *CADE'03*, volume 2741, pages 121–135. Springer, 2003.

[25] Veronika Ortner. Verification of BDD algorithms. Master's thesis, Technische Universitaet Muenchen, 2004.

[26] Harvey Tuch and Gerwin Klein. A unified memory model for pointers. In Geoff Sutcliffe and Andrei Voronkov, editors, *12th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR-12)*, volume 3835 of *Lecture Notes in Computer Science*, pages 474–488, Jamaica, December 2005.

[27] P.W. OHearn, J.C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL*, 2001.

[28] H. Yang. *Local reasoning for stateful programs.* PhD thesis, 2001.

[29] L. Birkedal, N. Torp-Smith, and J.C. Reynolds. Local reasoning about a copying garbage collector. In *Proceedings of POPL*, 2004.

[30] Tjark Weber. Towards mechanized program verification with separation logic. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic – 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 2004, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 250–264. Springer, September 2004.

[31] J. McCarthy and J. Painher. Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science*, volume Proceedings of Symposia in Applied Mathematics, pages 33 – 41. American Mathematical Society, 1967.

[32] M. Strecker. Formal verification of a java compiler in isabelle. In *Proceedings on Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77. Springer Verlag, 2002.

[33] Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298:583–626, 2003.

[34] J. Strother Moore. Piton: A verified assembly level language. Technical Report 22, Comp. Logic Inc. Austin, Texas, 1988.

[35] Dino P. Oliva and Mitchell Wand. A verified compiler for pure prescheme. Technical Report NU-CCS-92-5, 1992.

[36] Sabine Glesner and Jan Olaf Blech. Logische und softwaretechnische herausforderungen bei der verifikation optimierender compiler. In *Proceedings der Tagung Software Engineering 2005.* Lecture Notes in Informatics (LNI), 2005.

[37] Lawrence Paulson. A semantics directed compiler generator. In *Ninth Symposium on Principles of Programming Languages*, pages 224 – 233. ACM Press, 1982.

[38] Martin Raskovsky. Generating a real compiler from a denotational semantics. Department of Computer Science, University of Essex, 1981.

[39] C.Gomard and N.Jones. Partial evaluator for the untyped lambda-calculus. In *Journal of Functional Programming*, volume 1(1).

[40] Jens Palsberg. An automatically generated and provably correct compiler for a subset of ada. In *Proceedings of the fourth IEEE International Conference on Computer languages, San-Francisco, CA*, 1992.

[41] Jens Palsberg. *Provably Correct Compiler Generation.* PhD thesis, Aarhus University, 1992.

[42] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Symposium on Principles Of Programming Languages (POPL), Charleston, USA*, page 4254. ACM Press, 2006.

[43] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *Formal Methods 2006*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[44] A.Pnueli, M.Siegel, and E.Singerman. Translation validation. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, Lisbon, Portugal, April 1998*, volume 1384, pages 155 – 166. Springer.

[45] Martin Rinard and Darko Marinov. Credible compilation. In *In Proceedings of the Run-Time Result Verication Workshop*, 1999.

[46] M. Rinard and D. Marinov. Credible compilation with pointers. In *In Proceedings of the FLoC Workshop on Run-Time Result Verfication*, 1999.

[47] Amir Pnueli, Ofer Shtrichman, and Michael Siegel. The code validation tool CVT: Automatic verification of a compilation process. *International Journal on Software Tools for Technology Transfer*, 2(2):192–201, 1998.

[48] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. Voc: A translation validator for optimizing compilers. *International Workshop on Compilers Optimization Meets Compiler Verification (COCV'02).In ENTCS, Elsevier Science*, 65, 2002.

[49] George C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000.

[50] W.Goerigk, T.Gaul, and W.Zimmerman. Correct programs without proof? on checker-based program verification. In *Tool Support for System Specification and Verification, ATOOLS98, Malente, Germany.* Springer Series Advances in Computing Science, 1998.

[51] W.Zimmermann and T.Gaul. On the construction of correct compiler backends: An ASM approach. In *Journal of Universal Computer Science*, volume 3(5), pages 504 – 567, 1997.

[52] R. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence 7*, pages 23–50, 1972.

[53] R. Bornat. Proving pointer programs in hoare logic. *Mathematics of Program Construction (MPC 2000)*, 1837 of Lecture Notes in Computer Science:102–126, 2000.

[54] J.Hennessy and D.Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, INC., San Mateo, CA, 2nd edition, 1996.

[55] S. M. Mueller and W. J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[56] D. Leinenbach, W.Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005), 5-9 September 2005, Koblenz, Germany*, 2005.

[57] J. Loeckx, K. Mehlhorn, and R. Wilhelm. *Grundlagen der Programmiersprachen*. Teubner Verlag, 1986.

[58] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume II: Compiling. Prentice-Hall, 1973.

[59] Richard Bornat. *Understanding and Writing Compilers*. Macmillan, 1979.

[60] V.G. Nguiekom. *Verifikation von doppelt Verketteten Listen auf Pointerebene*. Dimplomarbeit, University of Saarland, Computer Science Department, Germany, May 2005.

[61] H. Prediger. *Formal Verification of a C-Library for Strings*. Diploma Thesis, University of Saarland, Computer Science Department, Germany, July 2005.

[62] Artem Starostin. *Formal Verification of a C-Library for Strings*. Diploma Thesis, University of Saarland, Computer Science Department, Germany, August 2005.

[63] Acsar (automatic checker of safety properties based on abstraction refinement). http://www.mpi-sb.mpg.de/∼seghir/ACSAR/ACSAR-webpage.html.

[64] B.Cook, A. Podelski, and A.Rybalchenko. Abstraction refinement for termination. In Chris Hankin and Igor Siveroni, editors, *Static Analysis: 12th International Symposium*, volume 3672. SAS 2005, London, UK, September 7-9, 2005.

[65] A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In POPL05, editor, *Proceedings of the 32nd ACM SIGPLANSIGACT symposium on Principles of programming languages*, pages 132 – 144. ACM Press, 2005.

[66] N. Schirmer M. Daum, S. Maus and M. N. Seghir. Integration of a software model checker into isabelle. In G. Sutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, Lect. Notes in Art. Int. Springer.

[67] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.