FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Scalable Translation Validation

Tools, Techniques and Framework

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Dipl.-Ing. Roland Lezuo
Registration Number 0227059

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr. Andreas Krall

The dissertation has been reviewed by:

(Ao.Univ.Prof. Dipl.-Ing. Dr. Andreas Krall)

(Prof. Dr. rer. nat. habil. Wolf Zimmermann)

Wien, 20.03.2014

(Dipl.-Ing. Roland Lezuo)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Roland Lezuo
Burggasse 35/1, 1070 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

For Maya

Acknowledgments

First I want to thank my wife for her encouraging support and understanding during this time of ups and downs and my children for cheering me up as only children can.

Many thanks to my advisor, Andreas Krall, for letting me pursue the topic with such a high degree of freedom and personal responsibility. The trust he put into me was always motivating for me. I also want to thank Wolf Zimmermann for the enlightening discussion on the topics of programming language semantics and translation validation. Without his support this work would not have been possible. And last, but not least, I want to thank Laura Kovács for her invaluable support regarding theorem provers.

I also want to mention all my colleagues at the Complang group in Vienna which always made going to the office a joy. Especially I want to emphasize Gergö Barany for his precise comments on paper drafts and fruitful discussion on CASM and Ioan Dragan for his collaboration regarding vanHelsing. Special thanks go to Dominik Inführ and Philipp Paulweber for their commitment to provide solid implementations of my prototypes. And very special thanks to my sister Cornelia for her heroic proof reading.

Funding: This work is partially supported by the Austrian Research Promotion Agency (FFG) under contract 827485, Correct Compilers for Correct Application Specific Processors and Catena DSP GmbH.

Abstract

Today embedded computer systems are often used in safety-critical applications. A malfunction in such a system (e.g. X-by-wire) often has severe effects, even life-threatening consequences. Lots of effort is put into certification to assure the correct and safe behavior of safety-critical applications. Due to the high complexity of modern technical systems, a high-level programming language like C is commonly used to implement their software.

This makes the compiler a critical component in the certification of safety-critical systems. Even if the source code is fully certified and error-free an erroneous compiler could introduce unintended behavior and hence the certification would be in vain. This is one motivation of research in compiler correctness, a discipline which develops methods to show that the compiler behaves correctly. One approach, namely translation validation, formally proves that a single, specific run of the compiler was error-free.

This thesis contributes a framework which allows to apply translation validation from the source code down to its binary representation. The CASM language, based on the formal method of Abstract State Machines (ASM), has been developed as part of this thesis to specify the semantics of the source language and machine code. Using the novel technique of direct symbolic execution a first-order logic representation is created as the foundation for the formal proofs. To exploit the common structure found in problems originating from translation validation problems a specialized prover called vanHelsing has been implemented as part of this thesis. Its visual proof debugger enables non-domain experts to analyze failing proofs and pinpoint the causing, erroneous translation.

The detailed evaluation shows that CASM is by far the best performing ASM implementation. It is efficient enough to synthesize Instruction Set Simulation and Compiled Simulation tools. The vanHelsing prover performs much better than other state of the art provers on problems stemming from translation validation. These efficient tools and the high degree of parallelism in our translation validation framework enable fast validations. The implemented prototypes for instruction selection, register allocation and VLIW scheduling demonstrate that validation of real-world applications like *bzip2* is possible within a few dozen minutes.

Kurzfassung

Viele der heute verwendeten eingebetteten Computersysteme übernehmen sicherheitskritische (*safety-critical*) Aufgaben. Die Fehlfunktion eines sicherheitskritischen Systems führt per Definition zu großen Schäden, unter ungünstigen Umständen auch zur Gefahr für Leib und Leben. Teure Zertifizierungsverfahren werden durchlaufen um das korrekte und sichere Verhalten solcher Systeme sicherzustellen. Aufgrund der allgemein hohen Komplexität moderner technischer Systeme wird die Software, auch von sicherheitskritischen Anwendungen, oft in Hochsprachen wie C implementiert.

Dadurch wird der Übersetzer (*compiler*) dieser Sprache zertifizierungsrelevant. Selbst wenn der zugrunde liegende Quellcode der Software bewiesenermaßen fehlerfrei ist kann ein einziger Übersetzungsfehler ein verändertes Verhalten in der Ausführung bewirken. Dieser würde jedoch die komplette Zertifizierung obsolet machen, eine Motivationen für Forschung im Gebiet der Übersetzerkorrektheit (*compiler correctness*), einer Disziplin welche Techniken und Methoden erforscht um mit Hilfe formaler Verfahren die Korrektheit von Übersetzern sicherzustellen. Ein methodisches Vorgehen, die sogenannte *Translation Validation*, prüft dabei a posteriori die semantische Äquivalenz des Quellcodes mit dem übersetzten Programm.

Diese Dissertation beschreibt einen strukturellen Ansatz, welcher es ermöglicht, alle Schritte einer Übersetzung mit der *Translation Validation* Methode zu verifizieren. Um eine präzise Beschreibung der Semantik des Quellcodes und der ausführenden Maschine zu erstellen wurde, basierend auf der Theorie der *Abstract State Machine (ASM)*, eine geeignete Sprache (CASM) spezifiziert und implementiert. Durch die innovative Technik der direkten symbolischen Ausführung von ASM kann die Semantikspezifikation konkreter Programme in Prädikatenlogik erster Stufe dargestellt werden. Diese Darstellung bildet die Grundlage für den formalen Beweis der Übersetzerkorrektheit. Die sich ergebenden Beweisverpflichtungen weisen eine gemeinsame, problembezogene Struktur auf. Der im Zuge dieser Arbeit entwickelte vanHelsing Beweiser ist in Hinblick auf diese Struktur optimiert. Die Möglichkeit nicht bewiesene Probleme grafisch zu untersuchen bietet, auch ungeübten Anwendern von Theorembeweisern, ein Werkzeug um die jeweilige Ursache in der Problemdomäne zu identifizieren.

In der ausführlichen empirischen Untersuchung wird gezeigt, dass die CASM Sprache wesentlich schnellere Programmausführung ermöglicht als andere ASM Implementierungen. Die Geschwindigkeit ist hoch genug um sowohl Befehlssatz Simulatoren (*Instruction Set Simulator*) als auch übersetzende Simulation (*compiled simulation*) aus den CASM Spezifikationen der Maschine zu erzeugen. Der vanHelsing Beweiser ist, für Probleme hinsichtlich derer er optimiert wurde, wesentlich schneller als andere Theorembeweiser. Erst diese effizienten Implemen-

tierungen ermöglichen dass die Laufzeiten, der im Zuge dieser Arbeiten erstellten Prototypen für *Translation Validation* (Codeerzeugung, Registerzuteilung und Befehlsanordnung für VLIW Maschinen), selbst für realistisch große Programme wie z.B.: *bzip2*, jeweils nur wenige Minuten betragen.

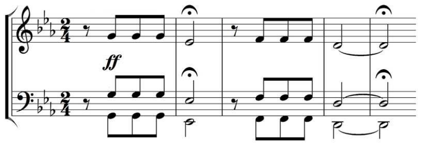
List of used Acronyms

ABI	Application Binary Interface
ADL	Architecture Description Language
ALU	Arithmetic Logic Unit
ASM	Abstract State Machine
AST	Abstract Syntax Tree
ATP	Automated Theorem Proving
BV	Bit Vector
CFA	Control Flow Association
CFG	Control Flow Graph
cLIR	causal LIR
CS	Compiled Simulation
DFT	Data Flow Tree
DSL	Domain Specific Language
EBNF	Extended Backus-Naur Form
EMF	Eclipse Modeling Framework
FFT	Fast Fourier Transformation
FU	Functional Unit (part of CPU dedicated to specific operation)
FUF	FU Field (part of internal state of a FU)
FV	Field Value (decoded field of an instruction)
ILP	Instruction Level Parallelism

IR	Intermediate Representation
ISS	Instruction Set Simulation
LASM	Linked Assembly Module
LIR	Low-level IR
LOC	Lines of Code
MIR	Mid-End IR
mMIR	matcher MIR
SIMD	Single Instruction Multiple Data
sLIR	scheduled LIR
SMT	Satisfiability Modulo Theories
STS	State Transition System
SSA	Static Single Assignment
VLIW	Very Long Instruction Word
XML	eXtensible Markup Language

Contents

1 Introduction



The number of embedded systems used in everyday life has increased significantly in the last decade and will increase further. With the increasing prevalence, more and more systems are used in safety-critical applications. According to Wikipedia a malfunction of a safety-critical system may result in death or serious injury to people, or loss of severe damage to equipment or environmental harm. To manage the complexity, software used to operate critical systems is often written in a high-level programming language, like C. There are industry-wide standards on the usage of C in such systems, e.g. MISRA C:2004 [1], a *guideline to the use of the C language in critical systems*.

To assure the correctness of safety-critical systems a significant effort is put into their verification. A good point in case is the seL4 micro-kernel [40], a kernel for security-critical applications with high reliability demands. It has been shown that this approximately 10.000 Lines of Code (LOC) C program is consistent with its specification models and free of a large class of common bugs (including null pointer access, alignment constraints, termination, processing unchecked user data). The manual labor put in (and therefore the costs of) such a verification are very high, i.e. many person years of work. The result of the verification is a trusted base of C code.

But the guaranteed properties of verified source code do not imply that those properties hold in the embedded system. The source code is compiled to the target hardware and executed by a real micro-processor, and both, the compiler and the hardware, may be erroneous. Hardware verification is a well studied problem and today's designs are at least partially verified [37]. Although there exist hardware bugs (e.g. the famous Intel FDIV bug [2]) they are much less problematic than software bugs in today's systems.

[1] `http://www.misra.org.uk/`

[2] `http://en.wikipedia.org/wiki/Pentium_FDIV_bug`

All it takes to invalidate the verification results is a single compilation error resulting in a behavioral difference between source code and binary. Such an change in behavior invalidates the preconditions made to verify the source code, and thous the results don't hold for the binary. A recent study by Yang et al. [68] reports on a large scale random testing of 11 C compilers including GCC, LLVM and CompCert [43]. They found silently erroneous compiled code in every single compiler, including the CompCert compiler, which is the only major commercial available (in large parts) verified C compiler. The conclusion is that compiler errors are probably more common than anticipated and can not be ignored in safety-critical systems.

Compiler verification is the field of research which deals with methods and techniques to show that a compiler behaves as specified. A verified compiler is a compiler for which has been shown that it is error-free, i.e. each program translated with a verified compiler is error-free. A disadvantage of this approach is that the smallest change in the compiler triggers a full re-verification. The other major approach is translation validation, which does not show that the compiler is correct, but that a specific, single compilation is correct. The compiler itself may contain errors, but they do not matter as long as those errors do not manifest themselves (i.e. changing the behavior of the binary). One advantage of this approach is that the compiler can be developed using common software engineering techniques, the disadvantage is that the validation has to be performed for each compilation.

To apply translation validation the *semantics* of the source and target languages must be known *concisely (formally)*. The proof itself relies on formal methods and tools. Automated Theorem Proving (ATP) is an established field of research and a number of mature theorem provers is available. The method presented in this thesis makes use of that knowledge by creating a problem formulation suitable for ATP tools.

Contribution

This thesis contributes to the field of compiler verification by developing scalable, fully automated methods which allow to verify the whole compilation process (from source to binary). A translation validation framework is proposed which splits the validation task along compiler passes. This allows each pass to be validated in isolation (*local correctness*), keeping the validators simple. The validators further split the validation task along basic blocks of the program. This gives a very high degree of parallelism which can be exploited to achieve fast validation even for very large programs. The framework allows to extend those *local* proofs to the notion of a *global* correctness.

We developed a variant of Abstract State Machine (ASM), called CASM, to formally specify the semantics of the involved languages. The novel technique of *direct symbolic execution* is used to create first-order logic representations of the CASM specifications, which allows the use of theorem provers.

Another contribution of this work is the vanHelsing theorem prover. This prover is specialized for the type of problems stemming from translation validation. Due to the specialization on this specific class of problems it also delivers excellent performance. A distinct feature is its support for graphical debugging failing proofs. This allows to analyze problems in proofs without expert knowledge in theorem proving. Even a novice user is able pinpoint the causing issue in the *problem domain*.

We also feel that verification in an industrial context should not be an isolated task. Therefore we developed tools to reuse the semantic models to synthesize instruction set simulators and perform compiled simulation. We argue that a single rigorous (formal) specification of the hardware is sufficient for verification, and synthesization of Instruction Set Simulation (ISS) and Compiled Simulation (CS) tools.

Layout of the Thesis

The remainder of this thesis is structured as follows: Chapter 2 discusses related work in the field of compiler verification, ASM and first-order theorem provers. Chapter 3 introduces the CASM language developed as part of this thesis. It defines the novel method of *direct symbolic execution* and describes the optimizing compiler. In chapter 4 the semantic aspects of compiler Intermediate Representations (IRs) are presented. A method to specify the semantics for a retargetable compiler for application specific processors is given. The technical aspects of creating IR dumps as CASM programs are presented. Chapter 5 introduces the proof techniques used in this work. The simulation proof technique and the common semantic vocabulary are a core concept of this thesis.

After the technical foundations have been laid chapter 6 presents our translation validation framework and describes the verification from C source code down to machine code. The definition of the term *correct compilation* is given. Front-end and mid-end are discussed, while the focus of this thesis is on the compiler back-end. In chapter 7 the validator tools for specific back-end passes developed in this work are presented. This chapter is the technical core of this thesis. Chapter 8 describes the vanHelsing prover, a fully-mechanized first-order theorem prover developed as part of this thesis. The main motivation to implement a custom prover is its ability to provide graphical debugging aids for failing proofs. vanHelsing is specialized on a specific class of proofs which commonly occur in the context of translation validation. This chapter describes the proof class and argues why a specific prover performs significantly better than more general theorem provers. Chapter 9 describes the implementation details of ISS and CS based on the CASM models. Chapter 10 reports on the performance of the methods developed in this work. We present benchmark data of selected validator prototypes, the vanHelsing prover and our ISS and CS implementations. In chapter 11 possible extensions to the tools and open issues which were not addressed by this work are discussed, and chapter 12 finally concludes this thesis.

2 Related Work

pic unrelated

2.1 Compiler Verification

Compiler verification is a very old idea with its roots in the 1960s. One idea is to prove that a compiler will only generate correct code. In this work we call this a verified compiler *in the strict sense*. The main disadvantage of this approach is that any change in the compiler triggers a complete re-verification. A verified compiler is therefore a piece of software set in stone.

Nonetheless Leroy's CompCert [43], the most important commercial available verified compiler, is a verified compiler *in the strict sense*. Large parts of CompCert are specified in Coq [9], an interactive proving tool which allows to extract executable code out of a specification. The compiler front-end (parser, type-checker and simplifier) and the assembler output module are not verified, though. With Yang et al. [68] recently finding serious bugs in the simplifier a complete source-to-binary verification seems to be necessary. The CompCert approach also fully trusts the assembler (creating the binary representation of the assembler language) and linker. Our approach covers parsing and linking.

The second important approach to compiler verification is translation validation. It has been suggested by Pnueli [57] and only verifies that a specific compilation is correct. The idea is similar to program checking [10]. An external observer (the validator) determines the correctness of an computation by inspection of the input and the calculated output. After a source program has been translated by an unverified compiler a validator tries to prove the target program to be a correct compilation. The proofs are performed by simulating source and target in a common semantic framework. The motivation of translation validation is that the validator may be significantly easier to write (and verify) than the compiler itself. In addition the software development process of the compiler is unconstrained (as changes don't trigger expensive re-verification).

Zimmermann and Gaul showed that this approach can be applied to realistic compilers (Verifix project [70]). They suggested ASM to build the common semantic framework, an idea also used in this work. Tree pattern matching rules are partially checked when generating the Verifix back-end, it is therefore (partly) verified *in the strict sense*. It is also entangled with register allocation. Our approach separates these passes and fully validates tree pattern matching at compile time.

Zuck et al. present VOC, a translation validation framework and tool for an optimizing compiler. They focus on the compiler IR and optimizations performed on it but do not cover machine code. Otherwise their notation of correctness is very similar to ours. Using a common semantic framework to describe source and target program they rely on *simulation* proofs to show that the target is a *refinement* of the source. Our approach is more general as we allow the source and target program to be in different IR languages. The validation tool operates on functions and considers whole paths through the function which may lead to scalability issues for large functions. Our approach operates on basic blocks which are on average significant smaller. They also discuss structure changing loop transformations which are quite hard to validate. In contrast to our work they do not consider pointers and aliasing at all.

Leviatan [44] presents a translation validation tool for software pipelining. They derive a large number (depending on the number of parallel stages) of correctness conditions to be verified. Pointers and aliasing is also not handled by this approach.

In [53] Necula describes a translation validation tool for an unmodified version of the GCC compiler. His approach operates on GCC's IR directly which is dumped before and after each optimization pass has been performed. Using a small set of heuristics enables identification of the applied transformations in the majority of the cases (i.e. there are false negatives). As in our approach symbolic execution of basic blocks combined with tracking of liveness information yields a good scalability. The main limitations of this approach are that it is unclear how to extend it to machine code as it operates on GCC IR directly.

More work on compiler verification can be found in Maulik's bibliography [20].

Differentiation

Figure 2.1 compares the discussed approaches with respect to various features. A ++ means very well suited, + means well suited, - means not so well and - - finally is not well suited. An important aspect is whether the approach can handle the whole compilation, that is from source down to binary (*source-to-binary*). Providing witness information means that the compiler must be changed which is an undesirable property for industrial application. By *scalability* we mean whether the method can cope with large programs. Modern industrial compilers transform programs in multiple (probably repeated) passes. Splitting the validation into passes is therefore important (*multi-pass compilers*). The effort to create a validator for an additional pass is an important aspect for industrial acceptance of translation validation. And finally the size of the trusted code base matters. The larger it is the more effort (and monetary resources) must be put into its certification.

	CompCert	Necula	VOC	Verifix	this work
source-to-binary	-	- -	- -	++	++
compiler-provided witness	++	+	-	-	–
scalability	++	++	-	-	++
multi-pass compilers	+	++	+	-	++
effort for additional pass	- -	+	+	-	++
size of trusted code-base	-	++	++	+	+

Figure 2.1: Compiler verification feature matrix

2.2 Abstract State Machines

ASM was introduced by Gurevich (originally named evolving algebras) in the Lipari Guide [31]. The original motivation was to bridge the gap created by the computational model of Turing machines. A coding-free technique to describe algorithms on a *natural abstraction level* was sought.

The ideas of ASMs were further developed by Gurevich and others at Microsoft Research resulting in a powerful specification language called AsmL [33]. AsmL is designed to be simple, precise, executable, testable, inter operable, integrated, scalable and analyzable. The language is statically typed, supports object oriented features, has call-by-value semantics and supports exceptions. An efficient compiler for .NET has been developed and the language has been fully integrated into the .NET framework and the Microsoft development environment [7]. The tool environment comprehends *parameter generation* for providing method calls with parameter sets, *finite state machine* generation from an ASM, *sequence generation* for deriving test sequences and *run-time verification* for testing if an implementation performs conforming to the model. The tool environment around AsmL is the most advanced currently available.

One of the most performance critical issue in ASMs is the problem of partial updates. Gurevich and Tillmann discussed the problem in detail and showed how concurrent data modifications can be implemented efficiently [34]. Similar problems occur in version control systems on software merging [52]. Techniques which work only on the delta (the differences) of the data sets inspire optimizations on efficient update implementation in CASM.

Castillo describes the ASM Workbench in [17]. Similar to CASM he added a type system to his language. The ASM Workbench is implemented in ML[1] in an extensible way. Castillo describes an interpreter and a plugin for a model checker, which allows to translate certain restricted classes of ASMs to models for the SMV[2] model checker.

Schmid describes compiling ASM to C++ [60]. The compiler uses the ASM Workbench language as input. He proposes a double buffering technique avoiding implementing update sets at all. This approach is limited to parallel execution semantics only, though.

Schmid also introduced AsmGofer in [61]. AsmGofer is an interpreter for an ASM based language. It is written in the Gofer[3] language (a subset of Haskell) and covers most of the

[1] http://en.wikipedia.org/wiki/Standard_ML

[2] http://www.cs.cmu.edu/~modelcheck/smv.html

[3] http://web.cecs.pdx.edu/~mpj/goferarc/index.html

features described in the Lipari guide. The author notes however that the implementation is aimed at prototype modeling and too slow for performance critical applications.

Anlauff introduces XASM, a component based ASM language compiled to C [4]. The novel feature of XASM is the introduction of a component model, allowing implementation of reusable components. XASM supports *functions* implemented in C using the *extern* keyword. CASM does not feature modularization, but can be extended using C code as well. XASM was used as the core of the gem-mex system, a graphical language for ASMs.

Gargantini et al. report on ASMETA [28]. Part of the development is a compiler translating ASM models into Eclipse Modeling Framework (EMF). The focus of this ASM implementation are high level models and design space exploration.

Farahbod designed CoreASM, an extensible ASM execution engine [23]. The CoreASM project is actively maintained and has a large user base. The CASM language is inspired by the CoreASM language, but over time they have diverged significantly.

Teich, Kutter and Weper describe a method to extract an ASM based instruction set description from a hardware description language [66]. This description is then used to automatically generate C code for a cycle accurate simulator of the processor. The Gem-Mex tool used provides support for implementing a parser which is used to read in assembler files for the simulator. Bit-true arithmetic functions are implemented in C and linked to the generated code. The feasibility of the approach is demonstrated by simulating very short programs on an ARM processor. No comparison to conventional simulators and no performance data are presented however.

Differentiation of CASM

The available ASM tools are not implemented with efficient execution of ASM in mind. One reason is that ASM are often used to create high level models and explore the design space. Concrete implementations are written by hand and verified against the ASM model, often using model checkers. CASM differs from these approaches as it aims to be executed efficiently. Our models are not just specifications, but concrete and efficient implementations are synthesized. For that purpose we have developed an optimizing compiler. CASM is to the best of our knowledge the only ASM implementation offering symbolic execution.

2.3 First-Order Theorem Provers

Most state-of-the art theorem provers are based on the superposition calculus [6, 54]. Those provers try to perform proofs by *refutation*, i.e. by deriving an contradiction. A common classification of provers is whether they use an OTTER [50] style saturation algorithm or DISCOUNT [5] style. The difference is in the treatment of generated clauses. As this set constantly grows the prover may need to remove *inferred* clauses at some point in time. DISCOUNT based prover therefore never utilize clauses from this set for inference or simplification. Schulz's E [62] prover is a modern, fast implementation based on DISCOUNT.

Vampire [59, 41] on the other hand implements both variants. Vampire is among the fastest theorem prover and has won the first-order section of the CASC [64] competition many years in a row now.

Satisfiability Modulo Theories (SMT) is another major branch in ATP. SMT is a generalization of the boolean SAT problem. Certain predicates in a SMT problem are interpreted using additional theories (hence the name SMT). The solvers for the theories (e.g. bit vector arithmetic) need to feed back results into the generic SAT solving part. A popular implementation is Microsoft's Z3 [21] prover. It is available under a shared-source license for many platforms.

Manna et al.'s STeP prover [49] has a rich graphical user interface allowing the user to guide the proof system. Counter examples can be derived automatically and debugging of problems is possible. STeP is an interactive tool, though, and its primarily intended for temporal specifications.

Differentiation of vanHelsing

The proving tool developed in this thesis is solely *unification* based. In contrast to superposition based provers, it is not searching for a refutation of the problem but performing unification until a fix-point is reached. The conjecture must then be provable with the found unifications. This simplicity allows a very efficient implementation based on a graph data structure. The graph data structure can be visualized and allows graphical introspection and debugging of problems. We are not aware of a fully-mechanized prover which offers graphical proof introspection.

In contrast to SMT proving no background theories are implemented. Uninterpreted predicates are axiomatized using first-order formulae. This allows more flexibility, but may negatively influence performance.

3 CASM - Efficient Abstract State Machines

restrain from equiring whether the name comes from the letters, the
pillars, the leather, the place, or the mode of behavior
Puck, The Sandman by Neil Gaiman

This chapter introduces the CASM language, its tools and focuses on the features distinguishing CASM from other ASM implementations. The novel technique of direct symbolic execution of ASM is formally defined and the implementation in the CASM interpreter is described. Further an efficient compilation scheme and an optimizing compiler are presented. During this thesis python prototypes of the CASM interpreter (including symbolic execution) and the compiler have been developed. The knowledge gained by the prototypes influenced the language design. A more efficient implementation of the interpreter using the C language was developed by Dominik Inführ in his bachelor thesis [36] under supervision by the author. The optimizing compiler was implemented by Philipp Paulweber in his master thesis [56] under supervision by the author. Interpreter and compiler are implemented as a single binary sharing the front-end (parser, type annotation, AST).

3.1 CASM - An Implementation of ASM

For a formal definition of CASM we refer to Gurevich's Lipari guide [31], Börger and Schmid's introduction of sequential execution [11] and Farahbod's CoreASM handbook [22]. More details on the CASM language can be found in [45]. An Extended Backus-Naur Form (EBNF) grammar can be found in appendix B. A novel feature of the interpreter is its capability to *symbolically execute* [39, 18] ASM models.

3.1.1 Types

The CASM language is statically typed and offers *Int*, *sub-range Int*, *Boolean*, and *String* as atomic types. Compound types are *List* and *Tuple*. There are no implicit type conversions performed by CASM. If desired the programmer can convert types using the (range checking) built-in functions: *Int2Boolean, Boolean2Int, Int2Enum, Enum2Int*.

3.1.2 State

The central notion of ASM is the *state*. It is described using a set of *functions*. Each *function* has an arity. Let $\overline{a}$ be an vector of arity *n* and *f* and *n-ary function*, then $(\overline{a})$ is called a *location*. In ASM *functions* are mathematical objects and are therefore *defined* over their whole *range*. CASM functions are typed and programs are checked statically. A (finite) program only uses a *finite* subset of values (of the state). The special value *undef* is assigned to *locations* which have never been defined by the CASM program. *Undef* is a *continuation* of the underlying *function* which assures a mathematical sound model. This gives a sound semantics to programs reading undefined *locations*, in contrast to C's behavior which e.g. is undefined if a program reads undefined memory.

ASM *rules* (statements) do not change the *state* directly, but create *updates*. An *update* is a tuple $(f(\overline{a}), v)$ which describes that the *location* $f(\overline{a})$ was changed to value v. ASM rules are executed in parallel, the updates produced by rules are joined to an *update set*. An update set which contains more than one update to the same *location* is called *inconsistent*. Inconsistent update sets are a run-time error in CASM.

Each CASM program has a *top-level rule*. This distinct rule is executed repeatedly, until the program is terminated explicitly. Whenever the *top-level* rule *concludes* (function return) the update set is applied to the state in an *atomic* operation. Hence ASM programs have *transactional* semantics.

3.1.3 Rules

The following list briefly summarizes the most important rules implemented in CASM. Börger and Schmid [11] is an excellent reference and we use the notational conventions introduced there. All rules except the *call* rule have exactly the semantics given there (and we omit it here). R_i is a rule and $t_j : \mathit{Type}$ is a term of the specific type. We omit the type specification for some terms if it is not needed to capture the semantics, but keep in mind that all terms are typed in CASM.

- **Skip Rule**: This rule does nothing and returns an empty update set.

  $$\mathbf{skip}$$

- **Update Rule**: Creates an update for a *n-ary* function f, assigning v to $f(\overline{a})$.

  $$f(a_0, a_1, \ldots, a_n){:=}v$$

- **Block Rule**: This is the most basic rule defining parallel composition of enclosed rules.

  $$\{\ R_1 R_2 \ldots R_n\ \}$$

- **Sequential Block Rule**: All enclosed rules are composed using sequential execution semantics.

$$\{|\ R_1 R_2 \ldots R_n\ |\}$$

- **Conditional Rule**: The basic ASM conditional rule. The conditional expression must be of boolean type and the else-branch is optional.

$$\textbf{if } t : \textit{Boolean} \textbf{ then } R_1 \textbf{ else } R_2$$

- **Case Rule**: An optional default case label is provided which will be executed if none of the given cases match the value of the conditional expression t. The types of all t_i must be equal to the type of t.

$$\begin{array}{l} \textbf{case } t : \textit{Enum}, \textit{Int}, \textit{String} \textbf{ of} \\ \qquad t_0 : R_0 \\ \qquad \vdots \\ \qquad t_n : R_n \\ \qquad \textbf{default} : R \\ \textbf{endcase} \end{array}$$

- **Forall Rule**: The forall rule evaluates the rule of the body composing the resulting update sets in parallel. Rule R will be evaluated for each element of the set described by t, binding the element's value to i. t may be an *Enum* type in which case each element of the enumeration is used as value or a *List* (with obvious semantics).

$$\textbf{forall } i \textbf{ in } t : \textit{Enum}, \textit{List} \textbf{ do } R$$

- **Iterate Rule**: Turbo ASM's iterate rule iteratively evaluates R using the intermediate state of the previous iteration (sequential composition) until R's update set is empty.

$$\textbf{iterate } R$$

- **Let Rule**: This rule adds a variable v to the environment. v is assigned the value t and R is evaluated in this environment. CASM performs type inference so the type of v can be skipped in most cases.

$$\textbf{let } v = t \textbf{ in } R$$

- **Call Rule**: Basic ASM call semantics are defined as call-by-name which can be implemented by a so called thunk [8], but this mechanism is not very efficient. CASM therefore has a modified semantics of the *call* rule. All arguments are evaluated before being passed as arguments (one could simulate this with *let* rules in the basic ASM definition). This makes call-by-name equivalent to call-by-value which is what CASM actually implements. The other change is that a *call* rule is evaluated in a new (empty) environment.

This effectively reduces the scope of variables introduced by *let* and *forall* rules (and the scope of rule arguments). Dynamically scoped variables cannot be compiled efficiently and cannot be typed statically. For all arguments a_i of the rule R the type of a_i must match the type of the term t_i. ($\mathfrak{A}$ is a state, ζ the environment, $\zeta\frac{x}{u}$ creates a new environment ζ' which equals ζ except that $\zeta'(x) = u$. $\zeta\frac{x_0}{u_0}\frac{x_1}{u_0}$ is the obvious composition of the new environment operator and $\zeta_\emptyset$ is the empty environment.)

$$[\![\textbf{call } R(t_0, \ldots, t_n)]\!]^{\mathfrak{A}}_{\zeta} = [\![R]\!]^{\mathfrak{A}}_{\zeta_\emptyset \frac{a_0}{v_0} \cdots \frac{a_n}{v_n}} \text{ where } v_i = [\![t_i]\!]^{\mathfrak{A}}_{\zeta}$$

- **Indirect Call Rule**: CASM supports indirect invocation of rules. This mechanism uses a slightly different syntax where r is an expression returning a reference to a rule. The semantics is otherwise identical to the *call* rule.

$$\textbf{call } (r : \mathit{RuleRef})(t_0, \ldots, t_n)$$

Further implemented rules are *print*, *debuginfo* (in accordance to [22]) and *assert*. The *debuginfo* facilities support named *channels* and CASM tools accept a list of active channels. Only the output produced by active channels is actually printed. The *assert* rule is in accordance to the *assert* statement of the C language.

3.1.4 Expressions

The common boolean operations *and, or, xor* and *not* are implemented for *Boolean* values. *Int* operations are $+$,$-$,$*$,$/$ and modulo ($\%$). Comparisons operators are $<=$, $>=$, $! =$ and equality ($=$).

A standard library is provided by CASM providing the following operations: *cons, app* for list construction, *peek, tail* to extract the list head (and tail) and *hex* converts an *Int* to a *String* with its hexadecimal representation.

3.2 Direct symbolic execution of ASM

This section introduces symbolic execution of ASM and describes how it is implemented in the CASM interpreter. We first define the formal foundations of symbolic execution in the context of ASM. A way to represent symbolic trace as first-order logic predicates is given and the implementation is described.

3.2.1 Definition of a Symbolic ASM

In this section we introduce symbolic execution and present an extension of Gurevich's basic ASM definition [31]. We tried our best to extend the basic ASM in a most natural way and in the spirit of the original definition. Although we output *symbolic traces* as first-order logical predicates, the definition of a *symbolic ASM* is generic and output formats (e.g. for model checkers) could be generated as well. The remainder of this section discusses various design decisions and gives definitions S1-S5 of a *symbolic ASM*.

Symbolic Execution

Symbolic execution is a technique where input values for a program may be so called *symbols* representing any possible concrete value. When a symbolic value appears as argument to an operation the result of the operation becomes a *symbolic expression*. Assume an addition $y = x+1$ and let x be a *symbol*, y then becomes the *symbolic expression* $x+1$. Symbolic expressions itself may be used as arguments for operations, e.g. $z = y*2$. The value of z would then become the *symbolic expression* $(x + 1) * 2$. By construction *symbolic expressions* are expressions consisting of operations applied to *input symbols*. There may be multiple input symbols for a program and there must be a way to distinguish them.

Things get difficult (but interesting) when a symbolic expression appears as the conditional in a control flow statement. The exact program continuation can't be determined and execution is forked to consider all paths. Instead of a single trace a tree of possible traces is generated. The continuation chosen on a fork point implies a condition for the (symbolic) conditional (i.e. conditional evaluated to true or false). The sum of all those conditions is called the *path condition*.

A system performing symbolic execution can without loss of generality create a *fresh* (never used before) symbol for each *symbolic expression* and only operate on symbols. We assume such a system for the remainder of this paper and use *symbolic expressions*, *symbol* and *symbolic value* synonymous.

The Symbolic Universe

All *basic* ASM [31] contain the special null-ary function *undef* to deal with partial functions. One could allow symbols to represent that value, but we think it is in the spirit of the basic ASM definition that symbols cannot represent the value *undef*. We define symbols to be a distinct sort (although compatible to other sorts), therefore:

A *symbolic* ASM is a basic ASM with addition of an universe *Symbol*. (S1)

All symbolic values are elements of this universe. Following the rule that *undef* is not part of any universe we state:

$$s \in \mathit{Symbol} \implies s \neq \mathit{undef}. \qquad \text{(S2)}$$

A symbolic value is an unknown but concrete value, whereas *undef* is used to express the value of an undefined location. In that sense the definition is natural. Finally it is important to note that symbols can be uniquely identified (e.g. by numbering them), or in other words

The equality operator $(=)$ is defined for all $s_1, s_2 \in \mathit{Symbol}$. (S3)

Summarizing S1-S3 a *symbolic ASM* contains at least 2 universes (*Boolean* and *Symbol*). All symbols are unique and part of the *Symbol* universe, they cannot represent the value *undef*.

Symbolic Functions

To perform symbolic execution a way to provide input symbols to the ASM programs is needed. Similar to *undef* being a continuation of partially defined functions, we define a symbolic continuation of partially defined symbolic functions. Partially defined *symbolic* functions are continued using pairwise distinct *symbols*. Obviously this definition implies an infinite number of symbols for infinite domains. This allows to model systems with an unknown or unbound number of *input symbols*. In section 3.2.3 we present a technique to efficiently handle infinite domains.

In ASM different types of functions (e.g. *static* for read-only) are possible. We add a new function type – *symbolic* – which can be combined with the existing types. Only *symbolic* functions are continued with pairwise distinct symbols.

$$\text{Functions tagged } \textit{symbolic} \implies \text{associated locations to be } \textit{symbolic}. \tag{S4}$$

$$\text{Each } \textit{symbolic} \text{ location contains a unique symbolic value.} \tag{S5}$$

3.2.2 Mapping Symbolic Traces to First Order Logic

While building *symbolic expressions* is well understood, presenting them for further processing is an open issue. There is no single, clearly superior solution to the problem. Various authors proposed different solutions, all suited for their special needs. Boyer's [12] SELECT tool allows the user to symbolically execute a program under his interactive control to support manual proving of properties. Khurshid et al. [38] generate output for model checkers supporting non-deterministic choice. Coen et al. [19] present symbolic expressions in the path description language (PDL) output and process them using the SAVE tool.

We wanted a human readable format also suitable for automated processing and proving tools. First-order theorem-proving is a mature branch of automated theorem proving with a number of commercial and free provers available (e.g. Isabelle [55], SPASS [67] or Otter [51]) and is well suited for our needs. The TPTP language proposed by Sutcliffe et al. [65] is a text based format understood by a wide range of automated theorem provers. We therefore decided to generate *symbolic traces* as first-order logic formulas in TPTP format. Each trace created describes exactly one path the program takes while being executed symbolically.

There is a semantic gap between a bunch of logic formulas and a trace describing a (symbolical) program execution. The later has a notion of time whilst a set of logical formulas is not time-aware. The remainder of this section describes how to map *symbolic expressions* and the changed *state* produced by each computation step to first-order formulas.

Mapping of State and a Notion of Time

The basic idea is to map the value v of a location $f(\overline{a})$ to a predicate $f(\overline{a}, v)$. Locations can change their value over time, therefore they cannot be mapped to a logical predicate directly. One needs to add a notion of time. We add a logical time-stamp as an additional (first) argument to functions.

For a basic ASM, only consisting of synchronous parallel updates, a notion of time is easy to give. Gurevich argues that each step of the computation corresponds to a tick of the logical clock [32]. Assume the ASM (infinitely) executing $\mathbf{rule} = x := x + 1$. The trace of predicates resulting of this program would be the (infinite) set $\{x(t, x_i + t) : t = 0, 1, \dots\}$ where x_i is the initial value of the function x.

A critical question is how to measure (logical) time in presence of the *sequential block* rule. Consider a parallel block containing a sequential block, i.e.

$$
\begin{array}{l}
\{ \\
\ \ \{| \; R_1 \; R_2 \; |\} \\
\ \ R_3 \\
\}
\end{array}
$$

Clearly the time-stamp for rule $R_1 < R_2$, as R_1 is executed before R_2. On the other hand both of them are executed in parallel to R_3. So $R_3 = R_1 \wedge R_3 = R_2$ also is an arguable requirement for the time-stamps. Obviously these requirements are contradicting. Following the definition of hidden internal computation steps (Fruja and Stärk [26]) we simply do not emit predicates for sequential block internal state changes at all. As a consequence, the model must make all state transitions an application wants to reason about non-hidden. We think this is reasonable.

After each computation step of the (symbolic) ASM, predicates for all symbolic locations are written to the *symbolic trace*. The logical time-stamp of each of the predicates equals the number of computation steps the ASM performed so far. Due to definition S5 there is an infinite number of symbolic locations if there is at least one symbolic function with an infinite domain. Obviously only a *relevant* subset of symbolic locations can be written to the symbolic trace. For now we vaguely define the *relevant set* as the set of symbolic locations the application using the symbolic trace is interested in. In section 3.2.3 we present a solution to this problem.

The initial state is assigned logical time 1. The final state is (additionally) labeled with logical time 0. Thus the time-stamps of the initial and final states are always known which is comfortable for many proofs.

Mapping of Symbolic Expressions - DFT

At the beginning of program execution only *input symbols* exists. During program execution more complex *symbolic expressions* are built (by applying operators to symbols). Internally we use fresh symbols to abbreviate complex symbolic expressions. A simple inductive argument shows that each fresh symbol implicitly represents a tree of of *input symbols* and operators.

We therefore map each application of a n-ary ASM operator to a n+1-ary predicate. The additional (last) argument is a fresh symbol (the result of the operation), the name of the predicate is the name of the operator. Again assume the program calculating $y := x + 1$ and $z = 2 * y$ yielding the symbolic expression $2 * (x + 1)$ for z and let x be the (input) symbol *sym1*. The addition will be mapped to the predicate *add(sym1, 1, sym2)* where *sym2* is a fresh symbol. This can be read as: it is true that the result of the addition of *sym1* and 1 is named *sym2*. Next the multiplication will be mapped to *mul(2, sym2, sym3)* with a fresh symbol *sym3*.

One may wonder about expressions modifying intermediate state inside of a *sequential block* rule. Although the intermediate state is not visible in the *symbolic trace* it may be updated and read by expressions. The above argument however showed that *symbolic expressions* are expressed solely by means of *input symbols* and operator application. An update may store a symbolic expression to an intermediate location, if it should be read again it is the symbolic expression itself, not the state, that matters.

Actually the state is completely transparent for expression evaluation. Consider for example the following ASM program fragment: $\{|\ x := x + 1;\ x := x * 2\ |\}$, further assume x to contain the symbolic value $sym0$. The addition operator creates a fresh symbol $sym1$ representing $sym0 + 1$ and updates x. The multiplication operator creates a fresh symbol $sym2$ representing $sym1 * 2$. This will be mapped to the two predicates $add(sym0, 1, sym1)$ and $mul(sym1, 2, sym2)$, correctly representing the resulting *symbolic expression* but without any reference to the (intermediate) state x.

In our proof applications the input symbols are mapped to program variables and registers. The *symbolic expressions* directly correspond to the concept of Data Flow Tree (DFT) which is implicitly described by the predicates.

3.2.3 Implementation of Symbolic Execution in CASM

This section describes the implementation of symbolic execution in the CASM interpreter.

Lazy Initialization of Symbolic Functions

As indicated in section 3.2.2 one needs to identify a *relevant subset* of symbolic locations to be written to the *symbolic trace* after each computation step of the ASM. The CASM implementation uses a technique called *lazy initialization* (Khurshid et al. [38]). A fresh symbol is created for a *symbolic location* when it is accessed for the very first time. Therefore a CASM symbolic trace contains the (finite) set of all symbols ever accessed during program execution. Unless the application wants to reason about locations *not* affected by the CASM program it is justified to assume that this set forms a super-set of the *relevant* set.

Two problems need to be considered implementing lazy initialization of symbolic functions. A symbolic location $f(\overline{a})$ may first be accessed in computation step $n > 1$, leading to creation of a fresh symbolic value s_k. All symbols accessed in a symbolic function are considered to be *input symbols*. Therefore predicates for all previous computation steps have to be emitted as well. This allows to reason about symbolic value in the preceding steps (including the initial one).

The second problem is due to the tree structure of currently active intermediate states induced by *sequential block* rules. Assume two rules R_x and R_y both accessing the same uninitialized *symbolic location* $f(\overline{a})$ and the following context:

```
{
  {| $R_1\ R_x$ |}
  {| $R_2\ R_y$ |}
}
```

The intermediate states used to evaluate R_x and R_y are different, nonetheless the implementation of symbolic execution has to assure that both rules read the same *symbolic value* for $f(\overline{a})$.

Trace Output Format

TPTP is a human-readable text based format in essence consisting of *annotated formulas* with the following general form: `language(name, role, formula)`. The language specifies the type of the formula, we exclusively use first-order forms with the language specifier `fof`. While *name* is an otherwise ignored arbitrary identifier, *role* specifies the user semantics of the formula (e.g. `axiom, hypothesis, conjecture`). An example of a formula is `fof(id, hypothesis, sym3 = 5)` stating that the constant term *sym3* equals to 5.

An Example

Consider the CASM program (fragment) given in listing 3.1 which swaps and increases values of *foo* and *bar* with *bar* being symbolic. Listing 3.2 shows the symbolic trace resulting from a single evaluation of rule r.

```
 1  function (symbolic) foo : -> Int
 2    initially {3}
 3  function (symbolic) bar : -> Int
 4
 5  rule r = {
 6    {|
 7      foo := bar
 8      foo := foo + 1
 9    |}
10    bar := foo + 1
11  }
```

Listing 3.1: Swap and increment

```
fof(0, hypothesis, bar(1, sym2)).          1
fof(1, hypothesis, add(sym2, 1, sym3)).    2
fof(2, hypothesis, foo(2, sym3)).          3
fof(3, hypothesis, bar(2, 4)).             4
                                           5
```

Listing 3.2: Symbolic trace in TPTP

Line 7 in the program corresponds to line 1 in the trace. The location *bar* is used by the update rule (assignment) which triggers creation of the symbol *sym2*. This symbol is temporarily stored at location *foo*, but this is a hidden intermediate state not visible in the trace. Line 8 (of the program) triggers the output of the *add* predicate describing the addition which creates a *symbolic expression* named *sym3* (line 2 in of the trace). Finally *sym3* is assigned to location *foo*, which can be seen in line 3 of the trace. In line 4 of the trace one sees that location *bar* contains the concrete value 4 (*foo* was initially 3 and the sequential block is executed in parallel to this update, so foo is still 3 here) after the first computation step (logical time is now 2).

Semantic Annotation

For some proofs it is useful to know when exactly a certain *symbolic location* was accessed for the very first time. Assume this happens at logical time t. As the symbolic value existed since the initial state (and will exist till the final state unless updated), the trace contains predicates for all this logical times. The trace alone is therefore not sufficient to determine time t. The predicates emitted for times less than t are therefore annotated by appending the TPTP comment `%SYMBOLIC` at the end of the line. Predicates emitted when a symbolic location is accessed the very first time are annotated with the comment `%CREATE`. Although the annotations for predicates corresponding to the update of a location (`%UPDATE`) are redundant and could be extracted from the trace, they are added to ease the programming of validators.

Symbolic Control Flow

In the CASM language we implemented symbolic execution for *if-then-else* and *case* rules. When the conditional expression of the rule is a symbolic value, all continuations are possible. That is *if-branch* taken or (optional) *else-branch* taken for *if-then-else* and each of the *cases* (including an optional *default* case) taken for the *case* rule. Program execution needs to be forked and continued for each of the possible continuations. Each continuation writes its trace to a separate trace file, so the application can reason about different paths taken by the program. Which continuation has been chosen induces a constraint for the symbol presenting the conditional expression. Those constraints are memorized in a *path condition* store and further control flow decisions on the symbolic values are evaluating the store. This eliminates the creation of contradicting traces.

The path condition is crucial for proving program properties. Listing 3.3 illustrates the importance. The code assures that no division by zero can occur, but this can only be proven utilizing the path condition in listing 3.5.

Listing 3.3: Path Condition

```
if x = 0 then skip
         else foo := 12 / x
```

```
1 fof(id0, hypothesis, x(0, sym2)).
2 fof(’if’, hypothesis, sym2=0).
```

Listing 3.4: *if-branch*

```
fof(id0, hypothesis, x(0, sym2)).              1
fof(’else’, hypothesis, sym2!=0).              2
                                               3
fof(id1, hypothesis,                           4
         divide(12, sym2, sym3)).              5
fof(id2, hypothesis, foo(0, sym4)).            6
fof(id3, hypothesis, foo(1, sym3)).            7
```

Listing 3.5: *else-branch*

Line 2 in both listings (3.4 and 3.5) shows the emitted constraint. A theorem prover could now be used to prove that listing 3.3 is division-by-zero free (by proving both traces to be division-by-zero free).

3.3 Efficient Compilation of CASM

The CASM language is designed with efficient compilation in mind (static type system, call-by-value for rule invocations). As part of this work a prototype implementation of a CASM compiler has been developed in python. Knowledge gained from this implementation has been used to refine the language design. Performance critical issues of the run-time system have been identified and an optimizing compiler has been designed. Theses ideas have been implemented by Philipp Paulweber in his master thesis [56]. Details on the analysis framework and run-time implementation are published in an article by Lezuo, Paulweber and Krall [48].

We will give a brief explanation of the main ideas behind the CASM compiler here. The run-time is based on the assumption that the *state* of the compiled programs is larger than the *update sets*.

3.3.1 Dynamic Memory Allocation

Only *functions* and updates need to be allocated dynamically. Due to the transactional semantics of ASM languages the life-span of an update is exactly one *step* of the machine. A pre-allocated memory pool is used to store updates until a *step* is made and all updates are committed to the function storage. This pool can simply be reused in subsequent steps (dump-allocation). The run-time therefore has virtually no memory management overheads.

3.3.2 Storage for CASM Functions

To properly implement *functions*, set operations are necessary. All *locations* which are not explicitly defined otherwise have the special value *undef* (demanding an *is-element-of* set operation). A distinct hash-map (with linear probing) is used as storage for each *function*. The function arguments are concatenated to form the key. Each slot of the map has two special properties, *undef* and *branded*. The *undef* property is set if the *location* has the special value *undef*. An update may set a previously defined *location* to *undef*, so such *locations* need to be tracked explicitly. A slot is *branded* when its corresponding *location* is accessed for the first time. (*Branding* allows to use other default values than *undef*, CASM supports this feature). The run-time uses the slot's address, which must be guaranteed to be stable, as a unique identifier.

After each *step* of the machine the hash-map can safely be enlarged, if the load factor should have become too large. In the rare case that during a single *step* the hash-map would overflow, additional memory is allocated. In-between the next machine *step* the hash-map is resized and the overflow memory gets merged.

If a sub range integer type is used for the domain of a CASM function, an array is used as *function* storage instead of a hash-map (for reasonable sizes of the domain). An additional byte is needed to keep track of the special value *undef*.

3.3.3 Updates and Pseudo States

Due to the interleaving of parallel and sequential execution semantics the state used to evaluate a statement and the state affected by its updates are in general not equal [26]. Listing 3.6 illustrates the problem. $stmt_1$ and the sequential blocks containing $stmt_2$ and $stmt_4$ are in a parallel block. Therefore they are evaluated under the same state S_0, their updates however are applied to different states. While updates produced by $stmt_1$ are applied to S_0, updates produced by $stmt_2$ are used to create a *temporary* state S_1. The sequential composition with $stmt_3$ may modify updates produced by $stmt_2$ and only the resulting update set will be applied to S_0. The same situation is with $stmt_4$ and $stmt_5$. As e.g. $stmt_4$ may contain a nested parallel block a tree-like structure of states is created. The nesting of update sets is very similar to nested transactions in software transactional memory (STM) [2]. The major difference is that an STM transaction aborts when reading an object for which a commit is pending while in ASM read access can never fail. Multiple updates to the same *location* in a parallel context is a run-time error (*inconsistent update*) in CASM.

Our assumption is that the number of updated *locations* (in a single ASM step) is much smaller than the whole state of the program. We therefore do not duplicate the state but keep

```
{
    $stmt_1$
    { | $stmt_2$
        $stmt_3$
    | }
    { | $stmt_4$
        $stmt_5$
    | }
}
```

Listing 3.6: Interleaving PAR/SEQ

track of all updates produced so far in a data structure called *update set*. When looking up a *location* the run-time has to query the *update set* for updates affecting the current state (due to sequential execution semantics).

We use the notation of *pseudo state* to keep track of updates affecting the current state. The *pseudo state* is a counter which is increased (at run-time) when a block with *different* execution semantics is entered. When a block is left (and control-flow returns into a block with *different* execution semantics) the *update set* is merged into the *update set* of the surrounding block. This is a serialization of the (partial) parallel execution semantics of ASM. Initially the system starts in parallel execution state, so *pseudo state* 0 denotes a block with parallel execution semantics. When entering a block with sequential semantics the *pseudo state* will be increased by 1. By construction this counter is odd when executing a block with sequential execution semantics and even when in parallel mode.

The *update set* is implemented as a hash-map. The keys are 64 bit values, the lower 16 bits are the *pseudo state* of the block the update originates from, the remaining bits are the lower bits of the slot used to store the *location*. (This limits the number of nested states to 65536 and the keys may collide for slots whose addresses only differ in the uppermost 16 bits. A key collision triggers an erroneous program abort, but no wrong behavior. We never hit any of the limits in our applications.)

Additionally the slots in the *update set* are forming a linked list with the latest update being the head. This property is used when merging update sets. Figure 3.1 shows the update set data structure.

3.3.4 Lookup and Update

A lookup for a specific *location* first needs to query the *functions* storage to acquire the address of the slot. This address and the current *pseudo state* are used to query the *update set* for any updates to this location which may be visible in the current state. By construction of the *update set* the corresponding keys can be efficiently calculated using the current key. The sequential states are all odd numbered *pseudo states* with a number that is lower than the current one. The complexity of this operation is linear in the number of active *pseudo states* (dynamic nesting depth of parallel and sequential blocks).

An update also needs to query the functions storage to acquire the address of the slot corresponding to the location. This address and the current *pseudo state* form the key for the *update*

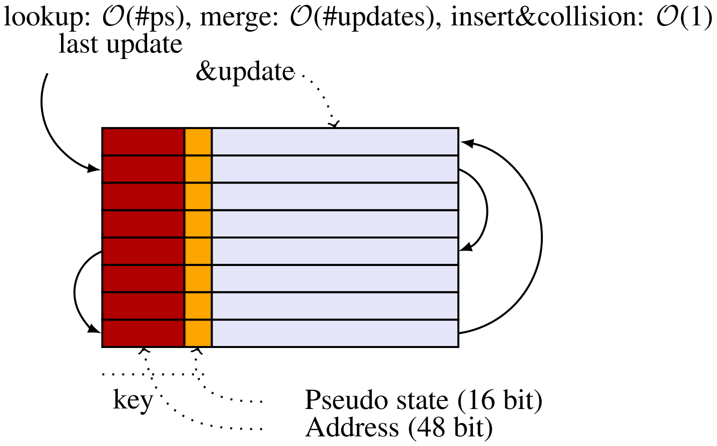


Figure 3.1: CASM Update Set

set. If the slot in the *update set* already contains a value, the further behavior depends on the current *pseudo state*. In sequential execution mode (odd pseudo state) the value will be overwritten, in parallel mode an *inconsistent update* error is triggered. The complexity of the *inconsistency* check is constant.

3.3.5 Merging of Update Sets

When leaving a block (with *different* execution semantics) the list property of the *update set* is exploited to efficiently merge all updates into the surrounding *update set*. The list is traversed backwards until the first update not belonging to the current update set is found (encoded in the lower 16 bits of the key). All updates are removed from the *update set* and re-inserted with the *pseudo state* part of their key reduced by one. Merging of *update sets* produced by sequential blocks may trigger inconsistent update errors as they are re-inserted into an *update set* with parallel execution semantics. The complexity of merging is linear in the size of the *update set* to be merged.

3.3.6 Optimizations

The hash-maps used to implement the *update set* and *functions* are obviously very expensive in terms of performance. In this section we describe two optimizations called *lookup elimination* and *update elimination* that aim to reduce the number of hash-map operations. The first observation is that lookups from a parallel execution context will always retrieve the same value (for same *locations*). In such situations only the first lookup needs to query the function storage and the *update set* to retrieve the value. The second observation is that, in sequential execution context, updates and lookup behave like local variables in the language C.

The idea is to introduce so called *local locations*. That is a rule-local storage which will be used by optimized lookup and update code. Once fetched, the *local location* can be used by

```
{

  if X(3) = 3 then
    skip
  if X(3) = 4 then
    skip
}
```

```
{
  local X_3 = X(3) in
    if X_3 = 3 then
      skip
    if X_3 = 4 then
      skip
}
```

Table 3.1: Redundant Lookup and its Elimination

```

{|
  X(4) := foo
  if X(4) > 0 then
    skip
|}
```

```
local L_1 = foo in
  {|
    X(4) := L_1
    if L_1 > 0 then
      skip
  |}
```

Table 3.2: Preceded Lookup and its Elimination

```
{|
  X(5) := foo
  X(5) := bar
|}
```

```
{|

  X(5) := bar
|}
```

Table 3.3: Redundant Update and its Elimination

subsequent lookups without the overheads of a hash-map. Table 3.1 illustrates the basic idea (*local* is not a valid CASM keyword).

Another pattern which allows the elimination of a lookup arises from updates (to the same location) preceding the lookup in a sequential context. In this case the value to be retrieved is known already and can be propagated instead of performing an expensive lookup. We call this pattern a *preceded lookup*, for an example see table 3.2.

Update elimination tries to reduce the number of updates stored in the *update set*. If a specific *location* is updated multiple times in a sequential context, only the last update will be committed to the state. All preceding updates can safely be omitted. See table 3.3 for an example.

Paulweber [56] has developed an analysis framework and implemented the described optimizations as part of this master thesis. In section 10.1 we report on the effectiveness of these optimizations.

4 Semantics and Compilers

מְנֵא מְנֵא תְקֵל וּפַרְסִן

This chapter describes how CASM is used in our retargetable research compiler. A method to give concise semantics to a configurable instruction set based on a Architecture Description Language (ADL) is described. The technical aspects of creating CASM representations of the matcher MIR (mMIR) and Low-level IR (LIR) languages are discussed. The ADL contains highly redundant specifications for ISS and CS tools, which motivate using CASM as an unified model.

4.1 ADL for Retargetable Compilers

Our industrial partner offers a retargetable toolchain for application specific processors. The instruction set of such a processor is created by the custom demands of its software application. This includes removal of unused instructions, choosing an optimal instruction encoding and providing instructions tailored for the application and customization of the data paths. A rich eXtensible Markup Language (XML) based ADL has been developed to cover all the demands (Farfeleder et al. [25]).

So called *micro instructions* describe the hardware building blocks for the Arithmetic Logic Unit (ALU) Each *instruction* implemented by a specific processor is built from such *micro instructions*. The set of implemented instructions define the data paths and layout of the resulting ALU. Listing 4.1 shows the *micro instruction* describing a hardware *add* operation. The dataflow of this instruction accepts 2 operands (lines 2 and 3) and defines 3 values (lines 4 to 6). The bit width is not determined yet, but the result has the same width as the first operand (line 4). The semantics of this *micro instruction* is specified by the *asm_semantic* tag.

In listing 4.2 the specification of a full machine instruction (addition) is shown. Lines 2-4 specify 3 variants with different assembler mnemonics. The *expr* attribute of the mnemonic nodes specify the value of the instruction's flags which are then used to conditionally invoke *micro instructions*. In this example the *addition* instruction is available as *half word* addition and *saturated* addition. The *when* attribute specifies the pipeline stage the *micro operation* is executed at. A normal addition variant would therefore execute the micro instruction

```
 1 <minstr id="ADD">
 2   <in name="op1"/>
 3   <in name="op2"/>
 4   <out name="sum" width="%in(op1)"/>
 5   <out name="overflow" width="1"/>
 6   <out name="overflow_sign" width="1"/>
 7   <asm_semantic>
 8     %fuvar(%out(sum)) :=
 9       BVadd_result(%width(%in(op1)), %fuvar(%in(op1)), %fuvar(%in(op2)))
10     %fuvar(%out(overflow)) :=
11       BVadd_overflow(%width(%in(op1)), %fuvar(%in(op1)), %fuvar(%in(op2)))
12     %fuvar(%out(overflow_sign)) :=
13       BVadd_overflowsign(%width(%in(op1)), %fuvar(%in(op1)), %fuvar(%in(op2)))
14   </asm_semantic>
15 </minstr>
```

Listing 4.1: Micro Instruction for addition

READ_REGISTER at time *CMP_READ_OPERAND_1* and store that as *op_left_* (line 12) to later copy the value to *op_left* (as *h==0*, line 16). Line 18 - 25 read the second operand and the remaining markup handles performing the hardware *add* operation (line 28) and storing the result in the register set (line 38).

To specify the *semantics* of the micro-processor the *semantics* of each instruction (and all its variants) must be defined. Listing 4.1 showed that the semantics is attached to *micro instructions*. Lines 8-13 show CASM markup where concrete input and output values still need to be replaced by a simple macro substitution. All % prefixed strings are preprocessed and replaced by strings. E.g. %in(op1) is replaced by the input operand of that name (*op_left*) in this case (%out(overflow) by *ov*).

A tool called *casm_gen* is used to create a complete CASM specification for a specific processor out of the ADL. Each instruction is mapped to a single CASM rule implementing all variants thereof. Listing 4.3 shows the general structure of an instruction's CASM model.

```
rule ADDITION(bndladdr:Int, addr:Int, stage:PipelineStages,
              phase:PipelineCycles) =
  let h  = decode_field(addr, FV_h)  in
  let r  = decode_field(addr, FV_r)  in
  let sa = decode_field(addr, FV_sa) in
  {
    if h=0 and r=0 and sa=0 then
    {
      {|
        if stage=EX1 and phase=begin then { /* ... */ }
        if stage=EX1 and phase=end then   { /* ... */ }
      |}
      /* for more stages */
    }
    /* more variants */
  }
```

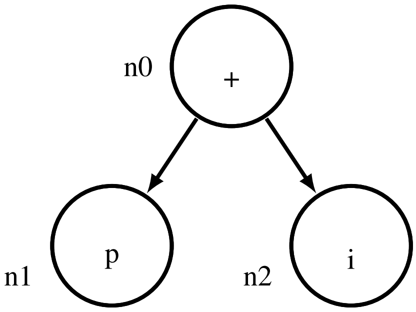
Listing 4.3: CASM markup of an instruction

```
<instr id="ADDITION" group="CMP" xml:base="../instructions.xml">
  <mnemonic expr="sa==0␣and␣h==0">add</mnemonic>
  <mnemonic expr="sa==1␣and␣h==0">add.s</mnemonic>
  <mnemonic expr="sa==0␣and␣h==1">add.h</mnemonic>

  <op field="a"/>
  <op field="b"/>
  <op field="c"/>
  <op_type field="r" operands="op1,op2,op3">TYPE_COMBINATION_ALU_SAME_3</op_type>

  <invoke when="CMP_READ_OPERAND_1">
    op_left_ = READ_REGISTER(%op(1),1)</invoke>
  <invoke when="CMP_READ_OPERAND_1" expr="h==1">
    op_left = SIGN_EXTEND1(op_left_)</invoke>
  <invoke when="CMP_READ_OPERAND_1" expr="h==0">
    op_left = op_left_</invoke>

  <invoke when="CMP_READ_OPERAND_2">
    reg_right_ = READ_REGISTER(%op(2),2)</invoke>
  <invoke when="CMP_READ_OPERAND_2" expr="h==1">
    reg_right = SIGN_EXTEND1(reg_right_)</invoke>
  <invoke when="CMP_READ_OPERAND_2" expr="h==0">
    reg_right = reg_right_</invoke>
  <invoke when="CMP_READ_OPERAND_2">
    op_right = reg_right</invoke>

  <invoke when="CMP_WRITE_OPERAND_3" expr="h==0">
    [sum, ov, os] = ADD(op_left, op_right)</invoke>
  <invoke when="CMP_WRITE_OPERAND_3" expr="h==1">
    sum = ADD_HALVE(op_left, op_right)</invoke>
  <invoke when="CMP_WRITE_OPERAND_3" expr="h==1">
    [ov, os] = [0, 0]</invoke>
  <invoke when="CMP_WRITE_OPERAND_3" expr="sa==0">
    result = sum</invoke>
  <invoke when="CMP_WRITE_OPERAND_3" expr="sa==1">
    result = SATURATE(sum, ov, os)</invoke>
  <invoke when="CMP_WRITE_OPERAND_3">
    WRITE_REGISTER(%op(3), result, ov, os, 0)</invoke>
</instr>
```

Listing 4.2: Addition Instruction

4.2 Compiler Overview

Our research compiler uses a number of IRs. The front-end is based on GCC and therefore uses GIMPLE [63]. GIMPLE is then translated to a tree-ish Mid-End IR (MIR) language. Mid-end optimizations are implemented as MIR to MIR transformations. Before the program is lowered to a sequential machine like LIR certain MIR operations are replaced. This restricted language is called mMIR and is used to perform instruction selection. LIR has two main variants, code which is not scheduled yet (causal LIR (cLIR)) and fully scheduled code (scheduled LIR (sLIR)).



```
POINTER(n0):=true,
WIDTH(n0):=20

POINTER(n1):=true,
WIDTH(n1):=20

INTEGER(n2):=true,
WIDTH(n2):=16,
SIGN(n2):=true
```

Figure 4.1: mMIR example

4.3 Semantics of Compiler IR

For translation validation the compiler must dump the IR before (*pre*) and after (*post*) each pass. In this section the modeling of the IRs and creation of the dump is described.

4.3.1 mMIR

MIR code is a tree-ish language which needs to be translated into a sequential (CASM) notation. The nodes are either *instruction* nodes (that are operations, e.g. addition, multiplication) or *operand* nodes (e.g. constant values, variables, results created by instruction nodes).

Figure 4.1 shows an example of a mMIR tree (addition of a signed offset i to a pointer *p*). Each node has a unique label (*n0, n1, n2*) and is of a specific type. The top-level node is an addition, while its leaves are *variables* (mMIR is in Static Single Assignment (SSA) form). mMIR nodes are polymorph and the exact type of the operation is determined by its operands and result. The CASM functions *POINTER, INTEGER, WIDTH* and *SIGN* are used to describe the exact types of *variables*. Evaluating trees is implemented by updating the *VALUE* function which associates values to nodes. The result of this addition is stored in *VALUE(n0)*.

Table 4.1 shows the most important CASM functions used to model mMIR. The *VALUE* function contains the (dynamically) calculated values associated to nodes. *WIDTH, SIGN, SIMDTYPE, SIMDFACTOR, INTEGER, POINTER* are used to describe the types of *variables*. mMIR operates on Bit Vectors (BVs) of arbitrary size. *WIDTH* determines the size of the BV. *INTEGER* and *POINTER* are used to determine whether a value is used arithmetically or as a pointer. *SIGN* is only used to *INTEGER* types and specifies whether an integer is signed or unsigned. *SIMDTYPE* is also only valid for *INTEGER* types and if set to true the underlying bit vector is interpreted as a container holding multiple data points. *SIMDFACTOR* finally specifies the number of data points (of equal size) in an Single Instruction Multiple Data (SIMD) container. mMIR operates on a linear memory which has the same properties as the target machine's memory (endianess, word size). Control flow is modeled using a dedicated call stack (*MIRHWSTACK*). mMIR also has support for hardware based loops and has explicit support through *MIRHWLOOP*. Loops are denoted by their start node, end node and number of iter-

function	signature	comment
VALUE	Node → Int	the value of a node
WIDTH	Node → Int	width of value in bits
SIGN	Node → Boolean	signed or unsigned value
SIMDTYPE	Node → Boolean	value is a SIMD container
SIMDFACTOR	Node → Int	number of SIMD words
INTEGER	Node → Boolean	value is a number
POINTER	Node → Boolean	value is a pointer
MIRMEMORY	Int → Int	the system memory
MIRHWSTACK	→ List (Node)	MIR call stack
MIRHWLOOP	→ List (Tuple(Node,Node,Int))	hardware loop
MIRPC	→ Node	currently executed MIR node
NEXTMIRPC	Node → Node	static successor MIR node

Table 4.1: mMIR State Representation

ations. A node's (statically) determined successor node is modeled using the CASM function *NEXTMIRPC* and *MIRPC* addresses the node currently executed.

The tree is traversed in a depth-first fashion, processing all child nodes before the parent is processed. Each node is assumed to have a unique identifier. MIR is polymorph in the sense that the exact semantics of an *instruction* may depend on the type of its operands and type of the result (i.e. conversions). For *operand* nodes all type information is emitted in hash-maps (their CASM equivalents), e.g. *WIDTH, SIGN*, with the unique node identifier serving as key. *Instruction* nodes are replaced using a macro expansion.

Listing 4.4 shows the template used when replacing an *ADDITION* node. The *%in* macro is replaced by the numbered child of the node, the result is the last child (by convention). *BVadd_result* is the CASM operation corresponding to the semantic operation *add_result*. Four cases are handled i) both operands have the same width (line 2) ii) first operand is wider and second signed (sign-extend second operand, line 6) iii) first operand is wider and second unsigned (zero-extend second operand, line 10) and iv) second operand is wider (extract relevant bits from second operand, line 15).

Our reference implementation is limited to handle mMIR programs, which are basically single expressions (see section 7.2). A full execution semantics would need to model the current environment (mapping variable name to value) and the activation records as well.

4.3.2 cLIR and sLIR

cLIR and sLIR consist of sequentially executed instructions closely resembling the target machine. The difference between the two is that sLIR instructions are forming bundles and are executed in a pipeline. cLIR instructions are scalar and not pipelined (they can be seen as a functional model of the same sLIR instruction). In this work cLIR instruction models are created from sLIR instruction models by consecutively executing all pipeline stages. Thus during

```
if TYPE(%in(1)) = TYPE(%in(2)) or WIDTH(%in(1)) = WIDTH(%in(2)) then
  VALUE(%in(3)) := BVadd_result(WIDTH(%in(1)), VALUE(%in(1)), VALUE(%in(2)))
else
  if WIDTH(%in(1)) > WIDTH(%in(2)) then {
    if SIGN(%in(2)) then
      VALUE(%in(3)) := BVadd_result(WIDTH(%in(1)), VALUE(%in(1)),
                  BVse(WIDTH(%in(2)),WIDTH(%in(1)),VALUE(%in(2))))
    else {
      // ! SIGN(%in(2))
      VALUE(%in(3)) := BVadd_result(WIDTH(%in(1)), VALUE(%in(1)),
                  BVze(WIDTH(%in(2)),WIDTH(%in(1)),VALUE(%in(2))))
    }
  } else  {
    // ! WIDTH(%in(1)) > WIDTH(%in(2))
    VALUE(%in(3)) := BVadd_result(WIDTH(%in(1)), VALUE(%in(1)),
                  BVex( 0, WIDTH(%in(1)) -1,VALUE(%in(2))))
  }
```

Listing 4.4: ADDITION MIR macro template

execution at most a single cLIR instruction is in the pipeline, preventing subsequent cLIR instructions to interact with each other.

Table 4.2 shows the most important functions used to model the state. *PC* is the program counter used to address the currently executed machine instruction. The target machines are supposed to have a hardware loop unit which state is modeled by the *HWLOOP* function. Each active hardware loop consists of the first and last instruction of the loop as well as the number of iterations (still to be done). *MEM* and *HWSTACK* model the machines memory and hardware call stacks. All functions starting with *REG_* are used to model the register file of the target machine, in this case data and address registers, flags and so called modify registers. The target machine used in this example offers auto post and pre increments, circular buffer addressing and a special addressing mode for Fast Fourier Transformation (FFT). *PMEM* models the program memory (instruction stream) while *PARG* holds the decoded Field Values (FVs) for each instruction. Internal state of each Functional Unit (FU) is modeled by the *FUstate* function. Each component of the internal state is called a FU Field (FUF). The target hardware features predicated execution which may disable certain FUs. This is modeled using the *PE_unit_disabled* function which holds the disabled status for each pipeline stages and FU. *pipeline* holds a list of instructions (addresses) for each pipeline stage of the CPU. For cLIR the size of the list is at most one, but for sLIR (which models Very Long Instruction Word (VLIW) bundles) larger values are valid. The branching unit may trigger (partial) flushes of the pipeline which is modeled by the functions *inval_pipeline*.

Technically cLIR models are simulated by sLIR models. cLIR is semantically equivalent to sLIR without considering (potentially conflicting) pipeline resources. By consecutive execution of each pipeline step of a sLIR model in an otherwise empty pipeline model the semantic effects of the cLIR model are simulated.

CASM function	function signature	comment
PC	→ Int	current program counter
HWLOOP	→ List(Tuple(Int,Int,Int))	hardware loop (start, end, iterations)
MEM	Int → Int	machine data memory
HWSTACK	→ List(Int)	machine call stack
FUstate	FU * FUF → Int	internal state of FU
PE_unit_disabled	PS * FU → Boolean	predicated execution
pipeline	PS → List(Int)	CPU pipeline
inval_pipeline_AL	→ Boolean	interaction pipeline, branch unit
inval_pipeline_EX1	→ Boolean	interaction pipeline, branch unit
REG_GuardBits	Int → Int	DSP register guard bits
REG_DataRegister	Int → Int	data registers
REG_AddressRegister	Int → Int	address registers
REG_ModifyRegister	Int → Int	auto modification for address registers
REG_Flag	Int → Int	separate flags for each data register
PMEM	Int → RuleRef	program memory
PARG	Int * FV → Int	decoded instruction fields

Table 4.2: cLIR State Representation

On Predicated Execution

The used research architecture features predicated execution. The actual instructions operate on *bundles*, hence they are only available in sLIR code. For cLIR code wrapper instructions have been implemented which are aware of the condition and condition registers. During VLIW scheduling (see section 7.4) actual sLIR instructions are created and added to predicated bundles.

Predicated instructions create *forks* during symbolic execution (caused by symbolic evaluation of the conditions). The traces can easily be matched by inspection of the *path condition*, though. No special treatment is therefore needed to handle predicated execution in our framework.

To validate passes which create predicates (i.e. if-conversion, see section 7.3), it may be necessary to create a trace where the result of the conditions is known a-priori (i.e. whether condition evaluates to true or false). To support such validators we have added *override* fields to the LIR instructions, i.e. *BRANCH_CONDITION_OVERRIDE* and *PE_CONDITION_OVERRIDE*. The models for conditional branching check those fields first and can be forced act as if the condition were true of false. If no override is given they (symbolically) evaluate the condition and (if needed) create forks. Our current validators do not use this feature, though.

4.4 A unified Machine Model

The ADL of our research compiler contains two (highly redundant) *semantic* specifications for each *micro-instruction*. Two types of simulators are synthesized out of the ADL, an cycle-accurate ISS and a tool to perform CS. Each of the tools have a different run-time system but both need *semantics* specifications of the instructions. Interaction with the simulator run-time is explicitly encoded in those specifications. Each simulator uses (preprocessed) snippets of C code which invokes its run-time functions. Figure 4.2 shows the simulator specifications for the addition. Without going into the details here the semantics of the addition is given by a C implementation. In the ISS specification (listing 4.5) the $-prefixed strings are replaced by macro substitution (similar to %-prefixed strings in listing 4.1). The CS specification (listing 4.6) uses $-prefixed and %-prefixed strings to be replaced. There are two different specifications in use, because ISS operates on structured models of the hardware (i.e. the execution unit) while CS performs partial evaluation on level of architecture features and operates on temporary variables [24].

Our CASM specification 4.1 is similar to the ISS specification, the *%fuvar* hides the structured access to the models of the underlying functional units of the micro-processor. A compiler which allows CASM programs to be executed efficiently could enable the reuse of the CASM specification to synthesize an ISS. An optimizing compiler which is able to perform partial evaluation could enable to synthesize the CS specification out of the CASM models as well. We have developed ISS and CS synthesization as part of this work which is covered in chapter 9. Therefore the two highly redundant simulator specifications can be replaced by our CASM specification, which also acts as the *formal* foundation used for translation validation.

```
<idata id="xsim_impl">
exec_unit_-&gt;result_ = exec_unit_
  -&gt;operand1_ +
  exec_unit_-&gt;operand2_;

if (width_$2_ &lt; width_$1_)
{
  if (exec_unit_-&gt;operand2_
      &amp; (1llu &lt;&lt;
            (width_$2_ - 1)))
  exec_unit_-&gt;result_ +=
    (uint64)-1 &lt;&lt; width_$2_;
}

if (((exec_unit_-&gt;operand1_ &amp;
      exec_unit_-&gt;operand2_ &amp;
     ~exec_unit_-&gt;result_) &gt;&gt;
      (width_$1_ - 1)) &amp; 0x1)
{
  exec_unit_-&gt;
    check_negative_overflow_ = true;
}
else if (((~exec_unit_-&gt;operand1_
        &amp; ~exec_unit_-&gt;operand2_
        &amp; exec_unit_-&gt;result_)
        &gt;&gt; (width_$1_ - 1))
        &amp; 0x1)
{
  exec_unit_-&gt;
    check_positive_overflow_ = true;
}
</idata>
```

Listing 4.5: ISS Specification

```
<idata id="csim_impl">
%tmp(result) = %tmp($1) + %tmp($2);
\#if (%width($2) &lt; %width($1))
if (%tmp($2) &amp;
    TWO_POWER_N(%width($2) - 1))
%tmp(result) += (uintmax_t)-1
  &lt;&lt; %width($2);
\#endif
%tmp(overflow) = (((%tmp($1)
  &amp; %tmp($2)
  &amp; ~%tmp(result))
  | (~%tmp($1)
    &amp; ~%tmp($2)
    &amp; %tmp(result)))
    &gt;&gt; (%width(result) - 1))
  &amp; 0x1;
if (%tmp(overflow))
{
  if (!(%tmp(result)
      &gt;&gt; (%width(result)-1)
      &amp; 0x1))
  {
    %tmp(overflow_sign) = 1;
  }
}
</idata>
```

Listing 4.6: CS Specification

Figure 4.2: Simulator Specification Languages

5 Proof Techniques

PROOF, n. Evidence having a shade more of plausibility than of unlikelihood.
The Devil's Dictionary

In this chapter the proving techniques used to implement validations tools are described.

5.1 Program Checking

Program Checking is a very old technique first described by Blum and Kannan [10]. A checker is a program which is able to verify if the output of another program is correct on a given input. At first glance the checker may look like a re-implementation of the original program but there are many problems which are much easier to prove than to solve. In a compiler a lot of heuristics are applied to optimize non-functional requirements, i.e. performance and code size. These heuristics often need to consider various contradicting optimization goals (low register pressure, high Instruction Level Parallelism (ILP)) and are therefore difficult and error-prone to implement. The result of the heuristic transformations may have correctness properties which are easy to check, though. In such cases a program checker is a viable option to implement a validator resulting in a small trusted code base.

5.2 Simulation Proofs

In this section we describe the flavor of simulation proofs used in this work. We assume two State Transition Systems (STSs), p and $C(p)$. The original problem is to show that $C(p)$ (in target language TL) is a *correct compilation* of the source program p (in language SL). Semantics of p and $C(p)$ is given by corresponding CASM models as described in chapter 4. This reduces the problem to show that the CASM models are in a relation which is compatible to *correct compilation* ($\precapprox$). In this section we define *correct compilation* as *equivalence*. This is adequate to explain the technique but too restrictive in practice. We will refine the *correct compilation* relation in section 6.1. Symbolic execution (see section 3.2) results in a first-order representation of p and $C(p)$. Figure 5.1 shows the overview.

$$\begin{tikzcd}
p \in SL \arrow[r, "\text{compile}"] \arrow[d, dashed, "\text{SL semantics (ASM)}"'] & C(p) \in TL \arrow[d, dashed, "\text{TL semantics (ASM)}"] \\
\llbracket p \rrbracket_{SL} \arrow[r, dash, "\precapprox"] \arrow[d, "\text{symbolic execution}"'] & \llbracket C(p) \rrbracket_{TL} \arrow[d, "\text{symbolic execution}"] \\
\text{first-order formulae} \arrow[r, dash, "\text{Theorem Prover}"] & \text{first-order formulae}
\end{tikzcd}$$

Figure 5.1: Generic Simulation Proof

To prove the STSs $\llbracket p \rrbracket_{SL}$ and $\llbracket C(p) \rrbracket_{TL}$ *equivalent* it can be shown that both STSs induce the same state transition for all initial states. Because of the *symbolic execution technique* the initial state is a *symbolic* state. Showing that the *symbolic* state transition of the STSs on an *arbitrary* initial state results in an equivalent *final* state is therefore enough.

Our proof technique *assumes* the same *symbolic initial* state for p and $C(p)$. Technically prefix *pre* is assigned to all *CASM functions* used by p and *post* to the ones used by $C(p)$. The initial state has time-stamp 1, therefore the first-order formula 5.1 is sufficient to assume equal initial state for the STSs p and $C(p)$. $function$ is the set of CASM functions (i.e. their predicate representation resulted from symbolic execution).

$$\forall f \in \text{function} : \mathit{pref}(1, \overline{a}, v_0) \wedge \mathit{postf}(1, \overline{a}, v_1) \implies v_0 = v_1 \tag{5.1}$$

The similar formula 5.2 is the first-order formulation for an identical final state. This is the *conjecture* which remains to be shown by a theorem prover.

$$\forall f \in \text{function} : \mathit{pref}(0, \overline{a}, v_0) \wedge \mathit{postf}(0, \overline{a}, v_1) \implies v_0 = v_1 \tag{5.2}$$

5.2.1 Common Semantic Vocabulary

To prove the conjecture the implicitly encoded data-flow (resulted from symbolic execution) must be formulated using a common *semantic* vocabulary. The chosen abstraction level has a great impact on the performance of the validation system. The lowest level would be bit-level formulation. Each operation is defined by its effects on single bits. The advantage of such a formulation is that all calculations leading to bitwise equal results are recognized by the validation system without additional information. The disadvantage is the permanent rediscovering of basic algebraic laws (i.e. the impacts on performance).

A high level of abstraction is a word-wise formulation, the approach chosen in this work. The disadvantage of this approach is that semantically equal but syntactical different formulations must be axiomatized. The main advantage is the gain in performance. Our experience has shown that real-world compilers only use a limited set of equalities which have to be axiomatized (a complete list can be found in appendix A). The common *semantic* vocabulary is also quite small (43 operations) as the listing in appendix A shows. Two examples are given in the following excerpt. Each item is the name of a predicate, in braces the arguments and a short description of its semantic meaning.

- *and (w, o1, o2, res)* logical and operation on *w*-bit wide values *o1* and *o2*. The result is called *res*.

- *mir_if (w, c, ai, ae, res) res* contains the value *ai* if the *w*-bit wide value *c* does not equal zero, *ae* otherwise. This predicate is used to model conditional control flow in mMIR.

- . . . (the full list is in appendix A)

The set of operations is designed according to the mMIR language of the compiler and is therefore architecture independent. Each operation is implemented in CASM, prefixed with *BV* (i.e. *BVand* corresponds to the semantic operation *and*).

IR conditional branches

The only surprising operation is the *mir_if* operation which is used to encapsulate conditional branches of the mMIR language. In essence the problem is that control flow can not be easily described using symbolic execution. A motivating example is given in figure 5.2. Assume a translation with identical source and target language. Let listing 5.1 be the source program and listing 5.4 the translation. Obviously the translation is erroneous (x and y have been swapped in the condition). The validator tool is confronted with the task to find corresponding traces in the set of *pre* and *post* traces. But the traces resulting from symbolic execution can not be distinguished by inspection of the final (and initial state), though. Lines 7-9 in listings 5.2, 5.3, 5.5 and 5.6.

A translation validation tool would need to inspect the whole trace and identify symbols which forked the control flow. This could be done easily, but identifying corresponding symbols in *pre* and *post* traces (if there is more than 1 fork) seems to be a hard problem. Notably the compiler may transform the expressions leading to forking symbols. Instead of tackling this problem we use the semantic *mir_if* operation. This allows the formulation of a conditional branch without an *if-then-else* statement (in CASM) and bind the conditional expression to the program counter. Figure 5.3 shows a CASM fragment (listing 5.7) and its symbolic trace (listing 5.8). The crucial difference in the trace is that the program counter *PC* is now a symbol (line 10). Validation tools now do not need to inspect the whole trace, conditional expressions are part of the final state.

In our concrete setting we use *mir_if* to model mMIR and conditional branches modeled with *if-then-else* in LIR. LIR code therefore may create multiple traces when executed symbolically, but its source program (mMIR) creates exactly one trace. The validation tool therefore has no problem to identify corresponding pairs of traces (as the correspondence is one to many). The *path condition* of each LIR trace makes the value of forking (LIR) symbols concrete. The theorem prover must be able to derive a concrete value (from the path condition) for *mir_if's*

```
function (symbolic) x : -> Int
function (symbolic) y : -> Int
function (symbolic) PC : -> Int

...

let c = BVand(1,x, BVnot(1,y)) in
        if Int2Boolean(c) then
                PC := 23
        else
                PC := 42
```

Listing 5.1: $x \wedge \neg y$

```
function (symbolic) x : -> Int
function (symbolic) y : -> Int
function (symbolic) PC : -> Int

...

let c = BVand(1,y, BVnot(1,x)) in
        if Int2Boolean(c) then
                PC := 23
        else
                PC := 42
```

Listing 5.4: $y \wedge \neg x$

```
fof(0,hypotesis,x(1,sym2)).
fof(1,hypothesis,y(1,sym3)).
fof(2,hypothesis,not(sym3,sym5)).
fof(3,hypothesis,and(sym2,sym5,sym7)).
fof('if',hypothesis,sym7=1).
fof(4,hypothesis,stPC(1,sym8)).
fof(final0,hypothesis,x(0,sym2)).
fof(final1,hypothesis,y(0,sym3)).
fof(final2,hypothesis,PC(0,23)).
```

Listing 5.2: Trace If

```
fof(0,hypothesis,y(1,sym2)).
fof(1,hypothesis,x(1,sym3)).
fof(2,hypothesis,not(sym3,sym5)).
fof(3,hypothesis,and(sym2,sym5,sym7)).
fof('if',hypothesis,sym7=1).
fof(4,hypothesis,PC(1,sym8)).
fof(final0,hypothesis,x(0,sym3)).
fof(final1,hypothesis,y(0,sym2)).
fof(final2,hypothesis,PC(0,23)).
```

Listing 5.5: Trace If

```
fof(0,hypothesis,x(1,sym2)).
fof(1,hypothesis,y(1,sym3)).
fof(2,hypothesis,not(sym3,sym5)).
fof(3,hypothesis,and(sym2,sym5,sym7)).
fof('else',hypothesis,sym7=0).
fof(4,hypothesis,PC(1,sym8)).
fof(final0,hypothesis,x(0,sym2)).
fof(final1,hypothesis,y(0,sym3)).
fof(final2,hypothesis,PC(0,42)).
```

Listing 5.3: Trace Else

```
fof(0,hypothesis,y(1,sym2)).
fof(1,hypothesis,x(1,sym3)).
fof(2,hypothesis,not(sym3,sym5)).
fof(3,hypothesis,and(sym2,sym5,sym7)).
fof('else',hypothesis,sym7=0).
fof(4,hypothesis,PC(1,sym8)).
fof(final0,hypothesis,x(0,sym3)).
fof(final1,hypothesis,y(0,sym2)).
fof(final2,hypothesis,PC(0,42)).
```

Listing 5.6: Trace Else

Figure 5.2: Control Flow and Symbolic Execution w/o *mir_if*

```
function (symbolic) x : -> Int
function (symbolic) y : -> Int
function (symbolic) PC : -> Int

...

let c = BVand(x, BVnot(1, y)) in
        PC := mir_if(c, 23, 42)
```

Listing 5.7: $x \wedge \neg y$

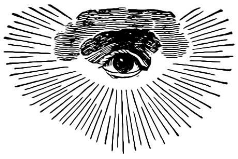
```
fof(0,hypothesis,x(1,sym2)).
fof(1,hypothesis,y(1,sym3)).
fof(2,hypothesis,not(sym3,sym5)).
fof(3,hypothesis,and(sym2,sym5,sym7)).
fof(4,hypothesis,mir_if(sym7,23,42,
                        sym9)).
fof(5,hypothesis,PC(1,sym10)).
fof(final0,hypothesis,x(0,sym2)).
fof(final1,hypothesis,y(0,sym3)).
fof(final2,hypothesis,PC(0,sym9)).
```

Listing 5.8: Trace

Figure 5.3: Control Flow and Symbolic Execution with *mir_if*

conditional for the proof to succeed. This is done by providing a proper axiomization of the common semantic vocabulary.

6 The Big Picture - Chain of Trust



This chapter presents our translation validation framework (see figure 6). The compilation is performed in consecutive passes. Each pass is understood as a free-standing compilation ac-

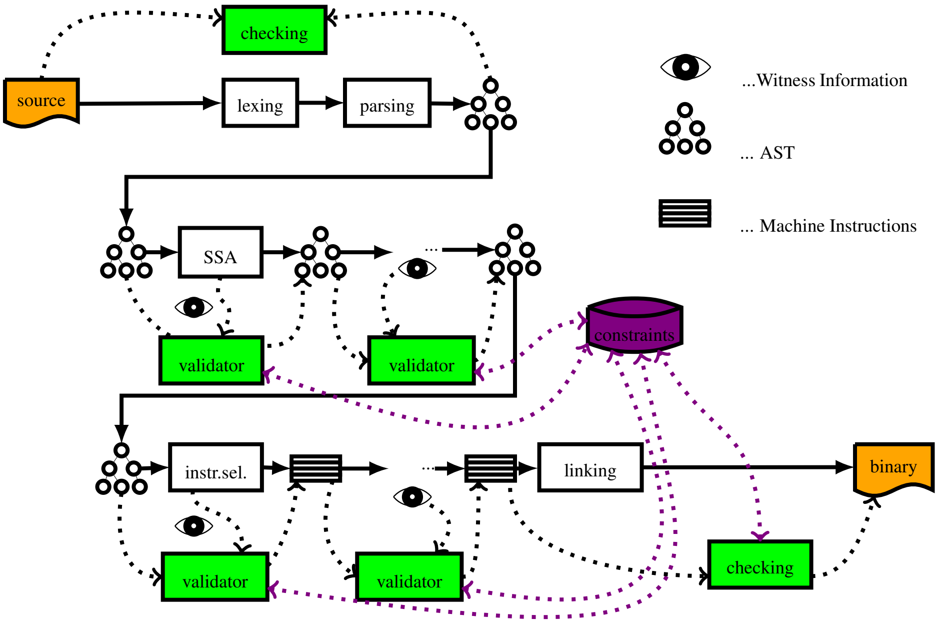


Figure 6.1: The Big Picture - Chain of Trust

cepting an input (*pre*) and producing an output (*post*). The compiler dumps witness information for each pass describing the performed transformations. A validation tool checks the (local) correctness of each of these passes, based on *pre*, *post* and the witness information.

The validation framework assures that *pre* of any single pass equals *post* of the preceding one. The correctness of the whole compilation can the be argued by following the chain *backwards*. Starting with a given binary it must be assured that linking was correct. If it has been verified that linking was indeed correct, the problem is reduced to the question whether the *pre-IR* (i.e. the input to the linker) is correct. Which is assured by a validator for the pass preceding linking (and assuring that *post* of this preceding past equal *pre* of the linker pass). Eventually the lexing and parsing pass is validated, and at this stage the validation framework is able to associate the source code to the binary which ultimately answers the question whether the given binary is a correct compilation of the given source code. It is so if all passes have been translated correctly. This sequence of (locally) correct passes is called the *chain of trust*.

Some aspects of the transformation can not be locally validated, though. A prime example is the correct translation of function calls. In most stages of the compilation functions are *named* entities. Only in the linking stage *addresses* become known. The branch instructions at all call sites must be patched (correctly).

During the (local) validation *constraints* are collected (e.g. a specific branch instruction is a call to a specific named function). *Constraints* are maintained by the validation framework and are used in later validators (e.g. the linking validator needs to verify that all call sites have been patched). These *constraints* are essential to achieve (global) correctness of the whole compilation.

6.1 Definition of Correct Compilation

Real world languages like C have properties which make the definition of a correct compiler quite hard. There are 54 unspecified, 190 undefined, 112 implementation-defined and 15 locale-specific behaviors listed in Annex J of the C99 (TC3 draft) specification [1].

The implementation-defined behavior includes such elementary things like the number of bits in a byte. Even the order of evaluating function arguments (they are expressions) is unspecified behavior. As they may have side-effects and even call further functions the order of evaluation has impacts on the semantics of a C program.

As a result of this language properties a C program can only be compiled correctly with respect to a specific compiler. A specific conforming compiler however will provide definitions for all implementation-defined behavior. Theoretically it may (randomly) choose among different conforming implementations of *implementation-defined behavior*. For such a compiler a correct translation must be one of the (many) possible conforming behaviors. In the remainder of this work we assume a concrete (deterministic) implementation of the C language (i.e. the validation tools know the exact behavior of the compiler).

6.1.1 Resource Limitations

In contrast to a C program a concrete machine has resource limitations (i.e. program size and available memory). An infinite number of C programs violate the constraints of any concrete machine (i.e. all programs larger than the machines memory). A translation is still considered correct if it fails solely due to resource limitations [30].

6.1.2 Observable Behavior

A correct compilation of a C program can be characterized as *preserving the observable behavior*. Assuming a concrete implementation of the C language a precise semantics is associated with any C program not invoking *undefined* behavior. The question of what a correct translation is boils down to the question of what exactly *observable behavior* is. In this work we assume typical embedded C programs, executed in a hosted environment though. The *external observer* therefore is

- The hosting operating system (i.e. its system call interface)
- Memory mapped I/O regions (for programs directly accessing hardware)

The *observable behavior* of a program is any interaction of the program with the *system call interface* and with *memory mapped I/O*. A translation of a C program is called correct if it results in the exactly same sequence of *system calls* (with exactly same arguments) and operations to memory mapped I/O regions. This definition has the advantage of being as generic as possible, but due to its global scope hard to prove in concrete situations. On the other hand the definition can easily be tightened by adding more *external observable* events (i.e. function entry and exit, basic block entry and exit) which allows to give local correctness conditions for many transformations.

6.1.3 Undefined Behavior

While *undefined* behavior of a C program (i.e. accessing arrays out of bound) is a major problem in real-world applications (so called buffer overflows have security impacts) it barely affects the correctness of its translation. By definition the behavior of such a program is undefined, hence *any behavior* of the compiled program is conforming to the C standard (and therefore a correct compilation).

6.1.4 Function Invocation

One common tightening is the addition of function invocations to the list of *observable behavior*. This disallows arbitrary inlining optimizations by the compiler. The proof system otherwise would always need to investigate whether deviating behavior could be explained by inlining (or even worse, partial inlining). Function invocations act as a black box, locally having the same semantic effect if invoked with the same arguments.

6.2 Front-end

The front-end of a compiler includes lexical, syntactic and semantic analysis of a program's source representation. After these steps an intermediate representation is available often in form of a decorated Abstract Syntax Tree (AST). Only after the AST has been created it is possible to reason about semantics of the program (a stream of lexer token has no semantics). A verifying compiler needs to rule out the possibility of errors in the front-end part as well. Semantic correctness can not be checked, though, so program checking is a viable option. If the original program representation can be *reconstructed* using the resulting AST as input its construction was correct.

A C compiler front-end includes a preprocessing step performing text substitutions resulting in a valid C compilation unit. Traditional C compilers also compute compile-time constants and remove parentheses, braces and semicolons before building the AST. To reconstruct a program's representation all those transformations need to be reversible stored in the AST. LLVM's AST implementation keeps track of this information utilizing the *clang::SourceRange* class, the current implementation is incomplete though [1].

6.3 Mid-end - Verification of Analyses

A program's AST is subject to analyses and transformations. While certain transformations have a clear semantics which can be used to verify their correctness (e.g. substitution of the AST describing the arithmetic expression *1 * a* by just *a*), other don't. Consider dead function elimination, and its semantics. The key insight is that this transformation does *not* change a program semantics, *iff* the analysis results have been correct. A proof justifying the removal of a function would need to make sure that it is indeed dead, thereby in a way performing the analysis. We therefore propose a method to correctly perform analysis in the first place.

In a separate project a Domain Specific Language (DSL) called Hydra has been developed which allows to describe analyses on a very high abstraction level. The Hydra specification can be used in two different modes. In checking mode it takes as input an AST, the name of an analysis and the analysis results computed by the compiler (witness information). Hydra performs the specified analysis on the AST and checks if the compiler results are consistent with its own results. This allows the compiler to produce weaker results (or no results at all when e.g. skipping a pass).

In generating mode a specification of the AST and an analysis is used to synthesize source code implementing the analysis. This allows the hydra specification to be used directly in the compiler. Please note that the compiler can always use a different implementation (which may be more resource efficient) as long as the results are always weaker than the results of the version used in checking mode.

A more detailed description of Hydra is out-of-scope for this work. In the remainder of this work we assume that correct analysis results are available throughout the compiler.

[1] http://clang.llvm.org/docs/IntroductionToTheClangAST.html

6.4 Back-end - Verification of Transformations

In contrast to the front-end most transformations performed in the back-end can be verified semantically. Each back-end pass accepts an input language (pre) and produces an output language (post). The languages *pre* and *post* have each defined semantics which allows to prove whether the *observable behavior* is preserved or not.

For each pass the compiler produces 3 artifacts i) input IR ii) output IR iii) witness information. The translation validation tool of each pass tests whether the given output IR is consistent to the given input IR under the (pass-specific) transformations described by the witness information. If the output IR is not consistent a compiler error is reported. Please note that the exact type of error is not important, i.e. it is irrelevant if *just* the witness information is corrupt or a *real error* occurred.

Additionally each pass may produce *constraints* which can only be checked in later stages of the compilation. One example is spilling registers to local stack slots. To be correct all spill locations must be pairwise non-overlapping. As the exact stack locations may only become available in a later pass validating these constraints must be deferred.

6.5 Multiple Iterated Passes

Modern optimizing compilers commonly perform certain transformations until a fix-point is reached. For example register allocation and VLIW scheduling are tightly coupled [29]. Common implementations come up with an initial schedule and try to allocate registers afterwards. In case the register pressure is to high (spill code) a less tight schedule is tried until a satisfactory result is found.

A compiler could implement this in an iterative fashion, i.e. it could have regions of code for which a schedule and register allocation was found and other for which no solution has been found yet. Each of these intermediate steps could be dumped and processed by the validation tool. This would complicate implementation of the validator, as it would need to distinguish such regions and either validate the transformation or skip non-transformed regions. For each iteration it would either re-validate previously transformed regions (expensive in terms of CPU time) or cache validation results.

We assume a compiler which only dumps its IR for the *final* solution and suppressed the IR dumps for unsuccessful attempts. The compiler needs to try the transformations on intermediate IR which are fully rolled back if any transformation fails. It therefore needs to build up internal data structures to keep track of information regarding future transformation attempts (and can not store them in its IR directly).

7 Correctness of Selected Back-end Transformations

Das also war des Pudels Kern!
Faust

In this chapter validators for selected passes are presented. For each pass the IRs it operates on, needed witness information, the criterion for local correctness and possible constraints are given. The *observable behavior* of all back-end passes is tightened to include *function invocations* (see section 6.1). Technically, basic blocks are ended on function *calls* (even if the function is known to return in all cases).

A formal description of each validator could be given, but is of limited applicability for compilers others than our research compiler. The validators need to be adapted to the concrete implementation of the passes for each specific compiler. Passes also depend on the semantics of the source (and IR) languages, e.g. in Java a stack overflow triggers an exception at run-time. Validators for passes modifying the stack-size (e.g. register allocation due to spilling) must then also validate that overflow detection is updated accordingly. In a language like C such behavior is undefined and needs not be validated at all.

Figure 7.1 gives an overview and also shows the IR each pass operates on. MIR is the compiler's tree-ish internal representation used in the mid-end. Before instruction selection is performed MIR is rewritten to mMIR, which is a variant of MIR. Instruction selection is the first back-end pass producing an IR closely resembling machine code. The IR is now organized in blocks of sequentially executed instructions operating on an infinite number of registers. Its name cLIR comes from the fact that pipeline effects are not considered yet, thus *causal*. After scheduling the VLIW bundles have been constructed and pipeline resources are considered. The IR is now called sLIR. Only after linking the final binary representation is constructed, called Linked Assembly Module (LASM).

7.1 Prolog and Epilog Insertion

This pass transforms function invocations into branching instructions supported by the hardware. For each function prolog and epilog code is inserted to implement its activation record and

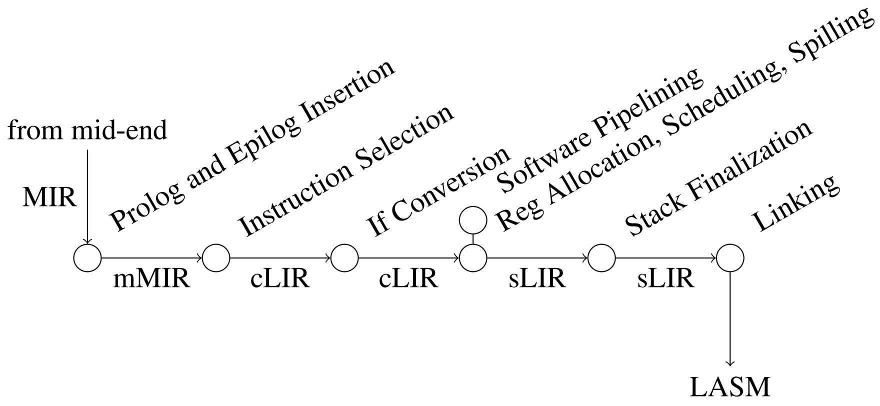


Figure 7.1: Covered Back-end Passes

preserve callee saved registers. Function arguments are mapped to argument registers and stack slots according to the machine Application Binary Interface (ABI).

In MIR function arguments in the callee are represented by *PARAMETER* nodes (listing 7.1). This pass replaces all *PARAMETER* nodes by i) register copies (into a named register) if the argument has been passed by register ii) a stack-relative load if the argument was passed on stack (see listing 7.2). A MIR *RETURNVAL* node is converted into a *RETURN* node and an instruction copying the value to be returned into the result register is inserted. The call site is also rewritten by this pass (see listings 7.3 and 7.4). All arguments to the function are either copied to hardware registers or stack slots (determined by the calling conventions). A potential return value is copied from the hardware register to its named register. All calls are converted to calls without any arguments.

```
Function foo


param A
param B

/* implementation of function */

returnval C



EndFunction
```

Listing 7.1: MIR function w/o epilog

```
Function foo
/* setup stack frame */
/* stored callee saved registers */
MOVE_REGISTER hwRegArg0 symRegVarA
MOVE_REGISTER hwRegArg1 symRegVarB

/* implementation of function */

MOVE_REGISTER symRegVarC hwRegRet
/* restore calle saved registers */
/* remove stack frame */
EndFunction
```

Listing 7.2: LIR function with epilog

```
...


res <- call func arg0 arg1

...
```

Listing 7.3: MIR call site

```
...
MOVE_IMMEDIATE arg0 hwRegArg0
MOVE_IMMEDIATE arg1 hwRegArg1
BRANCH_SUBROUTINE func
MOVE_REGISTER hwRegRet res
...
```

Listing 7.4: LIR call site

7.1.1 Constraints

- All named registers holding function arguments (copies of *PARAMETERs*) are live-in.
- The named register holding the return value (if any) is live-out.
- The liveness constraints are used during register allocation (see section 7.6).
- Stack locations used by arguments, validated as part of stack finalization (see section 7.7).
- The initial Control Flow Association (CFA) (see section 7.2.1) is build here as MIR *calls* are translated into machine branching instructions.

7.1.2 The Validator

A validator tool for this pass must check that i) all function prologs have been inserted correctly ii) all function epilogues have been inserted correctly iii) all function arguments (parameters) have been rewritten to copy instructions iv) return values have been copied to the return register v) each call site has been rewritten correctly and vi) no other changes have been made. All the checks require the validator to exactly know the used calling-conventions. The copying of arguments to registers and stack slots create *constraints* which affect later validation steps. The validator can be implemented using simple program checking techniques (see section 5.1).

7.2 Instruction Selection

This pass is one of the most important back-end transformations as the tree-ish mMIR gets lowered to the machine-like cLIR. While mMIR still operates on (named) variables cLIR uses an infinite number of (named) registers. The validator tool must verify that the selected instructions are semantically equivalent to their mMIR counterparts and that the mapping of (symbolic) variables to (symbolic) registers is consistent.

As a preparatory step the MIR nodes are partitioned into trees with respect to dependencies (true, anti, output) of registers and memory. Control flow modifying MIR instructions always start a new tree as well. By construction the partitioned trees match either a side-effect free expression, a function call, a conditional branch or an expression with a single side-effect at the root node (e.g. pre/post increment load and store instructions). The validator needs to check i) the partition is complete ii) no additional instructions were added and iii) the partition is correct relative to the dependencies. This can easily be implemented by program checking techniques (see section 5.1).

The challenging part of instruction selection is whether the selected sequence of machine instructions is a correct translation of the MIR tree. In this work we implemented a full semantic validation of the selected instructions utilizing the simulation proof technique (5.2). Because the IR is changed during this transformation a mapping of the state (mMIR → cLIR) needs to be defined.

7.2.1 Mapping State Representation: mMIR → cLIR

To apply the simulation proof technique (5.2) the cLIR state (see section 4.3.1) needs to be mapped to the mMIR state (see section 4.3.2). As only *observable behavior* is of interest this mapping needs only to be defined for *observable program state*. For the instruction selection the definition of *observable behavior* is tightened to include program state before and after each matched mMIR tree.

Most LIR functions are not *observable* (although they indirectly affect *observable behavior*). E.g. *FUstate* which holds intermediate results is not directly *observable* in the states before and after mMIR trees and is therefore not part of the mapping. The same argument holds for *REG_Flag, REG_ModifyRegister* and *REG_GuardBits*. *pipeline* and *inval_pipeline_** are not used by cLIR models, *PE_unit_disabled* can only be observed indirectly. *PARG* only contains decoded FV and is not part of the mapping either. This only leaves a few functions for which a mapping must be defined. *MEM* and *MIRMEMORY* is a simple one-to-one mapping (by design of mMIR memory).

MIRHWLOOP and *HWLOOP*, *MIRHWSTACK* and *HWSTACK* as well as *MIRPC* and *PC* are one-to-one mappings with the delicate difference of different addresses: mMIR operates on unique node identifiers and cLIR instructions have addresses. The final value of those addresses is only known after linking (see section 7.8), though. Unique instruction identifiers are therefore used in cLIR as well.

A data-structure called CFA is used to map mMIR node identifiers to cLIR instruction identifiers. Passes modifying LIR instructions must update the CFA accordingly. The CFA is handled like any other constraint (see chapter 6).

The most important mapping is mMIR's *VALUE* function which will be mapped to either *REG_DataRegister* or *REG_AddressRegister*. Which of mMIR's variables is mapped to which register number is part of the compiler's *register allocation mapping*. Both (mMIR and cLIR) are in SSA form, the mapping is therefore one-to-one.

7.2.2 Semantic Validation

By construction of the mMIR trees, side-effects (memory and control flow) can only occur at the root node. As a result only the last cLIR instruction of the selected sequence may be a branch instruction. Because mMIR conditional branches are modeled using the semantic *mir_if* operations (see section 5.2.1) the corresponding traces of *pre* and *post* are trivial to find (there is only one mMIR trace). The simulation proof technique (see section 5.2) is then applied to show that each trace of a cLIR instruction sequence is a correct translation of its corresponding mMIR trace.

7.2.3 Plausible Register Allocation Mapping

By proving each mMIR tree in isolation only local correctness is shown. The register allocation mapping (mapping of named variables to named registers) must be identical for each mMIR tree of a specific C function and must fulfill certain correctness properties. The *register allocation mapping* must not map different mMIR variables to the same cLIR register as the validator has no notion of variable lifetimes. When named and hardware registers share the same namespace mMIR variables must not be assigned to hardware registers. If global register allocation is performed the same global variable must be assigned to the same cLIR register consistently. Hence a validator must check the witness information (i.e. register allocation mapping) for plausibility. These properties can easily be checked by using the program checking technique (see section 5.1).

7.2.4 Proof Preparation

Our research architecture can combine its registers to larger units (i.e. two 16 bit registers can be combined to a single 32 bit register). The model of the register file describes the effects on the atomic units only (i.e. a 32 bit register write is modeled as two 16 bit writes). Symbolic traces and their first-order representation are therefore totally unaware of combined registers. The register allocation mapping (and other parts of the research compiler) on the other hand make use of the combined register notation. To close this gap we inject knowledge of combined registers into the problem files before they are handed to the prover. For each occurrence of a combined register the formulae

$$\begin{gathered}\forall T, X : \text{32\_bit\_reg}(T, \text{idx}, X) \implies \text{16\_bit\_reg}(T, \text{reg}_l(\text{idx}), xl_{\text{idx}}) \wedge \\ \text{16\_bit\_reg}(T, \text{reg}_h(\text{idx}), xh_{\text{idx}}) \wedge ex(0, 15, X, xl_{\text{idx}}) \wedge ex(16, 31, X, xh_{\text{idx}})\end{gathered}$$

and

$$\begin{gathered}\forall XL, XH, X1, X : \text{16\_bit\_reg}(T, \text{reg}_l(\text{idx}), XL) \wedge \text{16\_bit\_reg}(T, \text{reg}_h(\text{idx}), XH) \wedge \\ sb(16, 31, 0, XH, X1) \wedge sb(0, 15, X1, XL, X) \implies \text{32\_bit\_reg}(T, \text{idx}, X)\end{gathered}$$

where *idx* is the number of the combined register, xl_{idx} and xh_{idx} are fresh logical variables, are inserted into the problem file. They describe the fact that xl_{idx} and xh_{idx} can be extracted from (*ex* operation) respectively combined to (*sb* operations) the value of the combined register (X). reg_l as well as reg_h are register file specific functions mapping the index number of a combined register to the indices of its registers it is built from. The validator thus has (at least some) knowledge of the concrete architecture.

7.2.5 Side Effects

For each *observable* side-effect (i.e. memory access) on one side (*pre* or *post*) the equivalent effect must be observed on the other side as well. Because the memory model is the same for LIR and mMIR, inserting of these clauses into the conjecture is very easy. Assume MIRMEMORY$(0, A, V)$ is observed (logical time-stamp is 0, i.e. final state) then the clause

```
if (x == 0) {
  y = 1;
} else {
  y = 3;
}
```

Listing 7.5: C

```
mov.i 3, R0
b0:
    mov.i 3, R0
    br.cn R0!=0, b3
b1:
    mov.i 1, R0
b3:
```

Listing 7.6: assembler

```
mov.i 3, R0
(R0==0) mov.i 1, R0
```

Listing 7.7: predicated

Figure 7.2: If-Conversion example

MEM$(0, A, V)$ must be part of the conjecture and vice versa. The symbolic trace is scanned using regular expressions to find occurrences of *observed* side-effects and the corresponding function (e.g. *MIRMEMORY* and *MEM*) is substituted. More complex mappings than this simple one-to-one could be realized in this step. Thus the validator must have knowledge about the mapping of (CASM) state functions (their predicate representation) onto their counterparts for translations changing the IR. Because each mMIR tree has at most one side-effect (by construction) the order of side-effects is not important for the correctness of this pass.

7.2.6 Constraints

The register allocation mapping of each function is a global *constraint*. It must be consistent for each mMIR tree of a specific C function.

7.2.7 Validator

The complete validator therefore has to verify that i) the mMIR partition is a complete, non-overlapping cover of the MIR tree ii) without any additional nodes iii) the *register allocation mapping* is plausible (see 7.2.3) iv) each mMIR tree is semantically equivalent to its corresponding cLIR instruction sequence (simulation proofs) and that v) the same side-effects occur (under the state mapping function).

7.3 If Conversion

If-conversion [3] is a technique converting conditional branches into predicated execution. A small increase in code size (adding predicates) removes costs associated with mispredicted branches. But the conversion of control dependencies into data dependencies also enlarges basic blocks. Especially on VLIW architectures the scheduler may therefore better exploit ILP which enables better loop optimizations [69]. Figure 7.2 illustrates the basic idea of if-conversion.

In our research compiler if-conversion is implemented on cLIR. Conditional branches depend on the contents of a register.

The 4 basic patterns shown in figure 7.3 are optimized. Rectangles represent single basic blocks, branches are denoted by arrows. Pattern 1 is the classic if-then-else diamond shape without branches in the then and else block. The predicates added to the instruction in the else

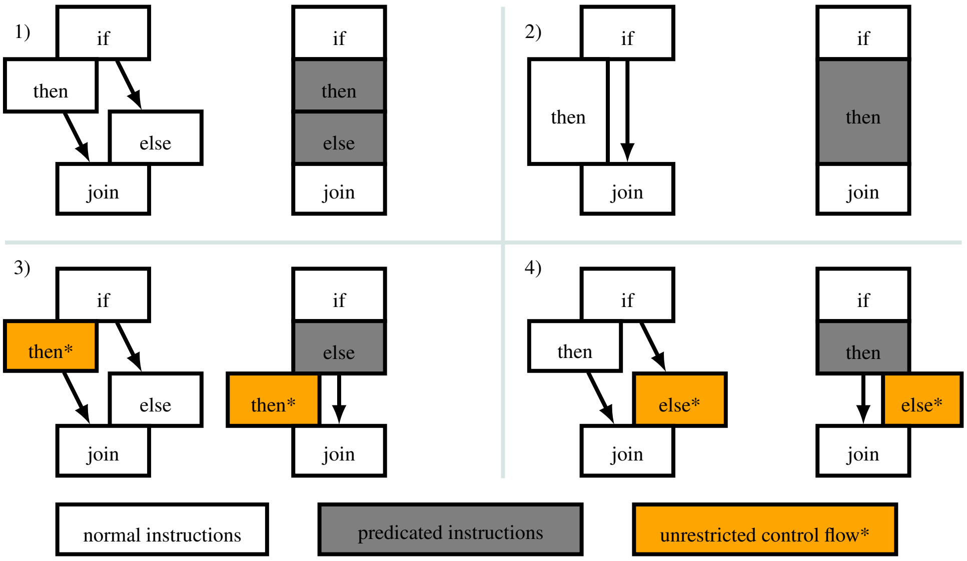


Figure 7.3: Supported If-Conversion Patterns

block are the negation of the then block predicate. Two conditional branches are removed (last instruction of if-block and last instruction of then-block). The result is one large basic block. Pattern 2 has an empty else block. An earlier transformation assures (by inverting the branch condition) that only this form occurs. The conditional branch in the if-block can be removed. Pattern 3 and 4 handle the cases were branch instructions occur in either the then or else block (*unrestricted control flow*). Only the block without branch instruction will then be converted into predicated form. Application of pattern 3 removes a conditional branch in the if-block, but adds a predicated unconditional one to the else-block. The unconditional branch of the then-block is removed. Pattern 4 removes the conditional branch of the if-block and turns the unconditional branch of the then block into a predicated unconditional one.

While the branch condition is evaluated exactly once (at the end of the if-block) predicates are evaluated together with each instruction. The register contents of the evaluated register may be changed, though. Therefore the register used by the conditional branch in the if-block will be copied to a fresh register which will then be used to evaluate the predicates.

7.3.1 Constraints

If the unique identifier of subroutine branch instructions has been changed the CFA must be updated.

7.3.2 Validator

The witness information consists of i) if-block, then-block, join-block ii) optional else-block iii) unique identifier of branching instruction iv) condition on which is branched v) conditional register used by branching instruction vi) register containing the copy of the conditional register and the vii) unique identifier of copy instruction.

The validator therefore has to verify i) if-conversion patterns have been identified correctly (inspection of branching instructions and involved basic blocks) ii) only blocks identified to be part of if-conversion patterns have been modified iii) no other blocks were added or removed iv) a register copy instruction was inserted directly before the branch instruction v) which copies the condition registers vi) into a fresh register not used anywhere else in the whole function vii) predicates were added to the blocks according to the identified pattern (predicate condition and register are known) viii) branching instructions were added, removed or changed according to the pattern. This validator can be implemented by program checking, no simulation proof needs to be performed.

7.4 VLIW Scheduling

This compiler pass is a further essential transformation of the back-end. It combines causal instructions of the sequential stream of cLIR instructions into resource-dependency aware bundles. A new compiler intermediate representation called sLIR is introduced to represent such bundles. The target architecture of our research compiler has a non-interlocking pipeline. This means that resource conflicts are not detected by the hardware and lead to undefined behavior of the program instead of just performance penalties. From a verification point of view VLIW scheduling for non-interlocked micro-processors is a correctness critical transformation.

The VLIW scheduler of the research compiler works on a basic block scope, no instructions are moved between them. Alias analysis is used to reorder load and store operations.

A scheduled block is a correct transformation if it changes the state as the unscheduled block would. The simulation proof technique (see section 5.2) depends on identical initial state, though. Because sLIR has a notion of the pipeline (and cLIR does not) the initial state when entering a sLIR block is not identical with the state when entering the cLIR block. The pipeline state has to be considered as well.

A pipeline has a certain depth and that is the number of preceding bundles which may affect the pipeline state. By extending each basic block to include exactly this number of preceding bundles the pipeline state when entering a basic block can be reconstructed. Depending on the number and size of predecessors (and their predecessors) of a basic block multiple extensions are possible. We call all such extensions the *snake blocks*. Figure 7.4 illustrates the concept. The pipeline depth in the example is 2 bundles. 3 paths can be constructed by extending the basic block with 2 preceding bundles, each of which (may) result in a different pipeline stage when entering the block. If a blocks predecessors are not statically known the first bundles of the block must contain no instructions. Branching to statically unknown labels can not be validated and such code is rejected.

The cLIR instructions used to form the bundles are then used to extend the corresponding cLIR basic block. Assuming an identical state at the beginning of each of the *snake blocks* the extended cLIR and the extended sLIR blocks must now result in identical final states. The properties of symbolic execution ensure that dependency violations introduced by reordering the

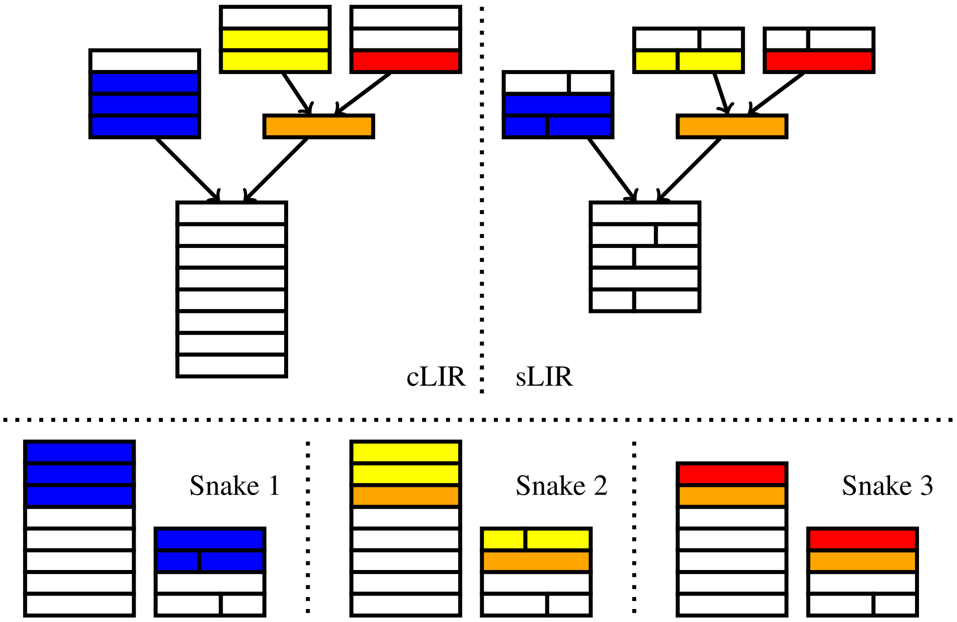


Figure 7.4: Snake Blocks

instructions will change the DFT of the transformed block and verification will fail. Memory dependencies on the other hand are more problematic.

7.4.1 Memory Dependencies and Symbolic Execution

Performing symbolic execution leads to symbolic memory addresses (in the general case). Some transformations reorder instructions which may include memory instructions. Reordering of memory instructions is not safe in general when aliasing is possible (i.e. write before read conflicts). By inspection of the symbolic trace aliased memory addresses can not be recognized, they are simply different symbols. Hence a validator would need to assume all symbolic memory addresses to be aliases, which would prevent any reordering. Clearly such a validator is not desirable.

Symbols representing memory addresses can easily be recognized in the traces (second argument of the *MEM* and *MIRMEMORY* predicates). Those symbols are associated with a *symbolic expression*. As result of the unification performed during the proof a *canonical symbolic expression* becomes available. This *defining expression* (see section 8.4) is identical for symbolic values resulting from the *semantically identical* operations on identical input symbols. Identical symbolic memory addresses are therefore expressed by identical *defining expressions*. The vanHelsing prover (see chapter 8) writes defining expressions to a separate file accompanying each proof. The validator uses this file to identify aliasing memory addresses. Input symbols are mapped to registers (or stack slots), for which the mapping to named program variables is available (by means of constraints, see section 7.2). Alias information of named program variables has been computed by a front-end analysis and is also available as constraints.

If the validator detects that a memory operation has been reordered (i.e. the addresses are not aliased) by the transformation it can validate if the reordering is coherent to the available alias information. The relative order of memory accesses is encoded in the logical time-stamps of the *MEM* and *MIRMEMORY* predicates.

7.4.2 Resource Conflicts

The research target architecture has a pipeline without interlocking. Undetected resource conflicts thus lead to undefined behavior of the hardware. By construction of the CASM models resource conflicts in the pipeline manifest as run-time errors (exploiting CASM's property that multiple updates to the same *location* from a parallel context are *inconsistent*).

7.4.3 The Validator

The validator has to verify that i) the basic block structure of *post* is identical to the one of *pre* (no blocks added or removed) ii) for each basic block all corresponding pairs of *pre* and *post snake blocks* are semantically equivalent iii) all identified reordered memory operations were not aliased.

7.5 Software Pipelining

Software pipelining is a class of optimizations which aim to remove the scheduling barrier imposed by back-edges of loops. The idea is to initiate a number of parallel loop executions in the prolog. The so called steady state then consists of the number of parallel executed loop *iterations*. A epilog is needed to finish loop *iterations* started by the steady state but no finished yet. Iterative modulo scheduling is one effective algorithm which allows to do so [42, 58]. Especially for VLIW architectures a much higher degree of ILP can be exploited by the enlarged loop body (kernel, steady state).

The sLIR to sLIR transformation is sketched in figure 7.5. A hardware loop (green) which consists of a single basic block without branches is transformed. Additional code in the prolog (pro_{post}^{add}) initiates a number of parallel iterations (blue), which reduces the number of loop iterations. The iteration counter has to be adopted (orange). A temporary register may be allocated to calculate the new value (if the number of iterations isn't a compile time constant). Finally code is added to the epilog (epi_{post}^{add}) finishing the started loop iterations (dark blue). Optionally the loop body may be unrolled (by a factor f_u) which also affects the loop iteration counter. This transformation is only performed if the minimum number of transformations is known and unrolling is only performed (in this implementation) if the exact number of iterations is known. The steady state will be executed at least one time (on the target machine hardware loops must be executed at least once).

Our validator prototype uses a combination of program checking, simulation proofs (see section 5.2) and an inductive argument to verify this transformation.

Let $[\![]\!]$ be the semantics of the enclosed expression, $+$ denote sequential execution of sLIR instruction sequences, pro_{post}, bdy_{post}, epi_{post} the prolog, loop body (steady state) respectively the epilog of the *post* transformation program. Let pro_{post}^{add} (blue in figure 7.5) and epi_{post}^{add} (dark

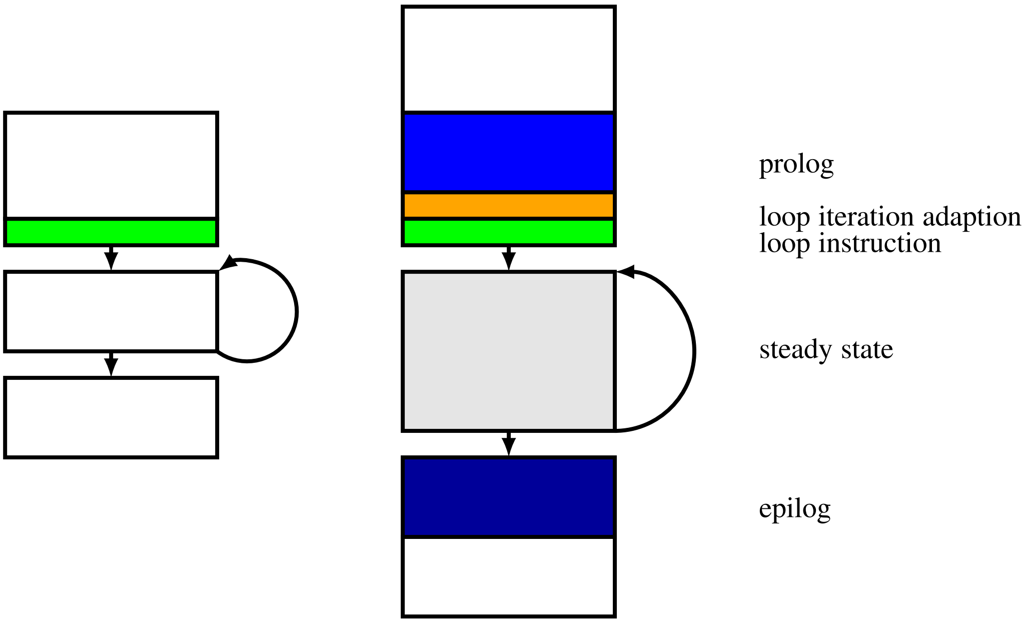


Figure 7.5: Software Pipelining

blue in figure 7.5) be the *additional* code in the *post* prolog and epilog. Let pro_{pre}, bdy_{pre}, epi_{pre} be the prolog, loop body and epilog of *pre*. Finally let $iter_{post}^{prolog}$ be the number of iterations initiated in the prolog code and $iter_{post}^{bdy}$ the number of iterations initiated by each execution of *post* loop body (including unrolling).

Due to the nature of the transformation the states when entering the *pre* and *post* loops are never identical. Without identical states our simulation proof technique can not be applied. The *pre* loop always executes one full iteration, while the *post* loop still has unfinished iterations in-flight. Intermediate results of these in-flight iterations are used in each execution of the *post* loop body. The proof technique is based on the DFT encoded in the symbolic traces and for in-flight intermediate results the DFT is unknown. However, it is known how to calculate the unknown DFT, that is exactly what the software pipelining prolog ($[\![pro_{post}^{add}]\!]$) does. Let the *initial post state* be the (symbolic) state when entering the *post* loop body, and the *synchronized initial post state* be the very same state but assume that pro_{post}^{add} was executed before. (In other words, the operations to compute the current in-flight values are replayed.) Similar the *synchronized final post state* is the state at the end of the *post* loop body, after epi_{post}^{add} was executed.

Now an inductive argument can be made to prove software pipelining using simulation proofs. Equation 7.1 is the base case, for exactly one iteration of the *post* loop body. The *post* prolog, body and epilog are executed exactly once. *Pre* loop body is executed exactly $iter_{post}^{prolog} + iter_{post}^{bdy}$ times, which is the guaranteed minimum loop execution time (a precondition for the transformation to be applied).

$$[\![pro_{post}^{add} + bdy_{post} + epi_{post}^{add}]\!] = [\![\overbrace{bdy_{pre} + \ldots + bdy_{pre}}^{iter_{post}^{prolog}+iter_{post}^{bdy}}]\!] \tag{7.1}$$

To finish the inductive argument it must be shown that an additional execution of the *post* loop body is *semantically* equivalent to an increased number of executions of the *pre* loop body. Now the *synchronized* states come into play as equation 7.2 shows. The *post* loop body is executed twice and hence the *pre* loop body must be executed twice as well (plus the $iter_{post}^{prolog}$ to account for in-flight iterations).

$$[\![\overbrace{pro_{post}^{add} + bdy_{post}}^{\text{synchronized}} + bdy_{post} + \overbrace{epi_{post}^{add}}^{\text{synchronized}}]\!] = [\![\overbrace{bdy_{pre} + \ldots + bdy_{pre}}^{iter_{post}^{prolog}+2iter_{post}^{bdy}}]\!] \tag{7.2}$$

It remains to be shown that pro_{post}^{add} and epi_{post}^{add} have been correctly added to the prolog and epilog blocks. The simulation proof shown in formula 7.3 ensures this.

$$[\![pro_{post} + bdy_{post} + epi_{post}]\!] = [\![pro_{pre} + \overbrace{bdy_{pre} + \ldots + bdy_{pre}}^{iter_{post}^{prolog}+iter_{post}^{bdy}} + epi_{pre}]\!] \tag{7.3}$$

The *post* state may differ by additionally used registers (modulo register renaming), and a different loop counter register. Finally the modification of the loop counter must be verified. This can not be achieved by symbolic evaluation, but the correctness conditions are very simple, so program checking (see section 5.1) is performed.

7.5.1 Validator

The witness information contains i) identifiers for involved basic blocks (prolog, loop body, epilog) ii) additional instructions added to the prolog initiating loop iterations (pro_{post}^{add}) iii) additional instructions added to the epilog finishing in flight loop iterations (epi_{post}^{add}) iv) identifier of loop instruction v) identifiers for loop iteration adaption code (orange in figure 7.5) vi) $iter_{post}^{prolog}$ and $iter_{post}^{bdy}$ vii) and modulo renamed registers.

The validator must verify that i) formulas 7.1, 7.2 and 7.3 hold (by simulation proofs) ii) correct insertion of iteration adaption code (program checking) iii) modification of loop instruction (program checking) iv) additionally used registers (iteration adaption code, register renaming caused by loop unrolling) are not used anywhere else in containing function v) reordered memory operations are coherent with alias information (see section 7.4) vi) and no other changes were made.

7.6 Register Allocation & Spilling

Register allocation and spill code generation is another indispensable pass of a compiler backend. In our research compiler register allocation is a sLIR to sLIR transformation. *Pre* uses *virtual registers*, although some registers are pre-allocated (i.e. function prolog and epilog, see section 7.1) and each register is defined exactly once. *Post* only uses hardware registers and may

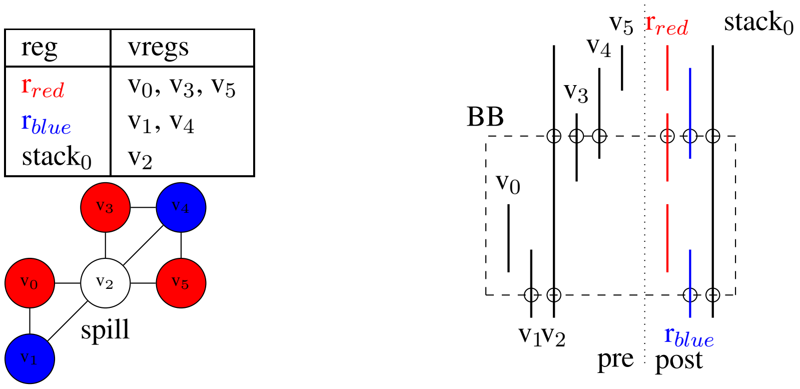


Figure 7.6: Register Allocation and Live-In Live-Out Sets

use additional stack slots (for spilled registers). Each virtual register is mapped to one hardware register for the scope of a whole function (i.e. no live range splitting). Spill code is added to the basic block containing the *use* and *def* (i.e. no spill code motion). Each virtual register of *pre* is therefore mapped to a hardware register of stack slot in *post* (this mapping is called the *register allocation map*). Although the scope of the register allocator is the whole function, a basic block local correctness criterion can mainly be used to prove its correctness.

The *register allocation map* must be proven to be correct, though. Program checking is a viable option here. By inspection of the *pre* trace the *def-use* information for virtual registers can be computed. This allows to create an interference graph. It can easily be checked if the *register allocation map* is a correct coloring of this graph.

The idea is to identify the *live-in* and *live-out* set of each basic block. By inspection of the *pre* and *post* traces the *def-use* set for each basic block can be computed. Using the Control Flow Graph (CFG) (assumed to be available and correct see section 6.3) a simple backwards data-flow analysis is used to calculate the liveness of *pre* and *post* registers. The initial state of a *pre* basic block can be used to compute an equivalent initial state for the *post* basic block utilizing the *register allocation map*. We tighten the *observable behavior* to include the *live-in* and *live-out* sets of each basic block, which still allows a wide range of transformations to be applied by the register allocator. Using the simulation proof technique 5.2 it must be shown that an equivalent initial *live-in* state results in an equivalent final *live-out* state.

Figure 7.6 shows an example with interference graph, *register allocation map* and the *live-in* and *live-out* set of a basic block (for *pre* and *post*). The example architecture has two registers (red and blue). For the translation to be correct the *live-out* sets of *pre* and *post* must be equivalent (considering the *register allocation map*). Also note that the register allocator may map virtual register v_0 (here mapped to register red) without any constraints, as long as the semantics of the block is not changed and the *live-out* set is correct.

All side-effects to the memory observed by *pre* must be performed by *post* without being reordered. *Post* may perform additional memory operations as long as the affected addresses are spilled virtual registers according to the *register allocation map*. It is possible that a spilled virtual register is used by a basic block but not in its *live-in* or *live-out* set (e.g. v_0 in figure 7.6).

```
1 fof(idclir620,hypothesis,stclirMEM(4,symclir368,symclir375)).%CREATE
2
3 symclir368=fadd_result(20,stclirREG_AddressRegister(14),csclirPARG(28,eFV_o))
```

Listing 7.8: Symbolic address (symclir368) & defining expression

7.6.1 Identifying Spill Code

To identify whether an additional memory access in *post* is a (valid) access to a spill location we again rely on the *defining expressions* (see section 8.4). In listing 7.8 line 1 is an example of a clause describing the first reading access (*CREATE* annotation, see section 3.2.3) to a memory cell (function *stclirMEM*). The address is the (symbolic) value *symclir368*, while the value read is the (symbolic) value *symclir375*. Line 3 of the listing shows the *defining expression* of the memory address symclir368. It was calculated by a 20 bit addition of the address register 14 (the stack pointer of the architecture) and the immediate field *eFV_o* of the instruction with unique identifier 28. By consultation of the constraints (see section 6) the validator can check whether instruction 28 references a spill slot (access is okay) or not (report error).

7.6.2 Integrated Rematerialization

Register allocators often perform various other *trival, small, obvious* optimzations on the fly. For a validator these optimizations are of course troublesome, as their correctness must be shown as well. This section describes an enhancement which enables the validation of rematerialization.

LLVM's greedy allocator (the coalescer actually)[1] utilizes *trivial rematerialization*[2] to remove *uses* of a register which are the result of a *register copy*. The *register copy* can be replaced by rematerializing the content of the *source* register. This reduces the lifespan of a register and may reduce register pressure. Loading immediate constants into registers are perfect candidates for rematerialization because load and copy instructions are easily interchanged on many architectures.

Our prototype register allocation validator has support for rematerialization of constant values. After the validator calculated the *live-out* sets a forward data-flow analysis is performed propagating constant register values. When the problem file is generated, clauses are emitted for registers with known constant values (stating the said register contains said constant value). The register allocator may then rematerialize or copy the register, the resulting DFT will be the same.

7.6.3 Constraints

- The register allocation is added to the constraints. For each basic block of a specific function the register allocation must be identical.
- The *liveness* information created by the prolog and epilog insertion validator (see section 7.1) is used.

[1] used version: 3.2

[2] TargetInstrInfo::isTriviallyReMaterializable()

7.6.4 The Validator

The validator must verify that i) the *register allocation map* is a correct coloring of the interference graph (by constructing the graph from the traces) ii) calculate *live-in* and *live-out* sets for each basic block (by inspection of the traces) and optionally constant register values iii) for each pair of *pre* and *post* basic blocks the *live-out* sets are equivalent, given an equivalent *live-in* set (equivalent with respect to the *register allocation map* iv) all memory side-effects of *pre* happen in the same order in *post* (by inspection of the traces) v) all memory side-effects exclusively happening in *post* affect spill memory only.

All spill locations, their *defining expressions* to be exact, are *constrained* to be otherwise unused stack slots. This property will be verified in the stack finalization pass (see section 7.7).

7.7 Stack Finalization

This pass is responsible to allocate concrete stack slots for all stack allocations of a function. Stack allocations are needed for *callee-saved* registers, *stack-passed* arguments, local variables whose address is used and spilled registers. Those previous passes emitted stack slot *constraints*. Our research compiler patches immediate fields of sLIR instructions in this pass.

Earlier passes created stack slot constraints of the generic form: instruction X references stack object S of size S_s at offset S_o. The witness information produced by this pass contains the stack offset o for all stack objects of each function. Our research compiler implements stack slot coalescing, stack objects with disjunctive lifespans may therefore be allocated to the same stack offsets. The same technique is used to reconstruct the interference graph and proves its coloring discussed for register allocation (see section 7.6) can be applied to prove stack slot allocation correct.

7.7.1 The Validator

The validator has to verify that i) stack slot coalescing is correct (by construction of the interference graph from the traces) ii) all stack allocations are non-overlapping iii) the immediate fields of all instructions are patched correctly (collected stack constraints) iv) no other changes are made.

7.8 Linking

The linker finally defines the layout of the binary, including the address of each function and the addresses of global variables. We assume a static linker commonly used for embedded systems which often don't support dynamic linking at all. All instructions referencing functions or global values must be patched (fix-ups, relocations). On some architectures the linker may even inject instructions.

Instructions referencing function addresses have been generated during instruction selection. The name of the called function is known in mMIR code. A constraint is emitted for each function invocation associating the function name with the unique identifier of the cLIR call instruction. Global variables are handled identical.

The linker emits the chosen addresses of functions and global variables as witness information, the validator can be implemented by program checking. There are additional correctness issues to verify, i.e. different objects must not overlap, alignment and others. They all can be verified by program checking, no semantic transformations are performed by a (correct) linker.

7.8.1 The Validator

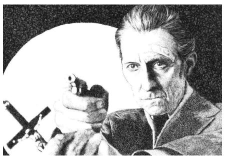
The validator must verify that i) all global objects are assigned to unique, non-overlapping machine addresses ii) considering alignment constraints, validity (i.e. address mapped to actual memory) and properties of the underlying memory (i.e. read-only, execute) iii) all relocations (fix-ups) of instruction fields have been performed correct and consistent iv) no other changes were made.

7.9 Summary

The investigated passes include instruction selection 7.2, register allocation 7.6 and VLIW scheduling 7.4. These are the foundation passes of our research compiler's back-end, without these passes no machine code can be generated at all. The other passes deal with interesting properties of our research architecture, namely predicated execution and optimization of hardware loops.

Each of the passes can be verified using a combination of the simulation proof technique and program checking (see chapter 5). We were able to develop *local correctness criteria* thereby limiting the sizes of the simulation proofs to the size of a basic block. *Global correctness* is achieved by creating constraints (and validating them in later passes). A high degree of parallelism can be exploited when executing the proofs. All basic blocks (many thousands for larger projects) of a program can be validated in parallel. Utilizing a large cluster even very large programs can therefore be validated within minutes (depending on the longest running proofs).

8 vanHelsing: Prover and Debugger



With superposition based provers like Vampire [59, 41] and Eprover [62] there are two problems with our application i) if the *symbolic traces* created from the (not trusted) IR dumps contain a contradiction a *spurious* refutation will be found (interpreted as proof success) and ii) if the translation was erroneous the prover states *satisfiability*, but gives no hint about the nature of the problem. Problem i) can be solved by running the prover on the *symbolic trace* and axiom set without the conjecture, they must be satisfiable. We are not aware of a solution of the second problem, though. A domain expert has to manually check the problem to locate the reason why the proof failed (and locate the compiler bug). This is a tedious and time-consuming task.

Using the same problem formulation SMT produces a model for erroneous translation (they are satisfiable) which helps in identifying the bug in the compiler. For more complex problems finding a model seems to be a very time consuming task (at least for the Z3 prover, see section 10.2). For unsatisfiable problems no further information is given, while superposition based provers create an *evidence*. (The problem can be formulated in a different fashion so a model is found as *evidence*, losing the models for erroneous translations, though.) A missing evidence weakens the strength of the argument that the compilation is proven to be correct (proof can not be retraced independently). The other issue is that if the *translation facts* produced by the (not trusted) compiler are unsatisfiable in itself, the whole problem will be *spuriously* unsatisfiable.

Due to the large size of our problems (often more than 10.000 predicates) our tool should also provide a debug mode to analyze failing proofs. A representation of the encoded DFT as graph helps to catch the interrelation of predicates and values, but also creates the connection to the problem domain. vanHelsing can print a graph representation of the problem utilizing the widely used DOT language [27]. Figure 8.1 shows the initial graph built from the example given in listing 8.2. Function applications are printed as structured rectangular boxes. The first field contains the name of the predicate, the following fields its arguments. By convention the last argument represents the result. All arguments are linked to the referenced value nodes (printed in ellipses). Unified values (that are equal values) are printed as list after the equal sign (there are none).

8.1 Input Language

As input language the vanHelsing prover uses a subset of TPTP v6.0 [65]. This section specifies the supported features and restrictions. A specification of the language in EBNF can be found in appendix C.

As top-level elements TFF (type information for predicates and variables) and FOF (first-order formula) are accepted. TFF formulas are accepted but ignored and vanHelsing assumes integer types for all values. Nonetheless correct type information should be added to achieve compatibility with other provers (i.e. Vampire). All TPTP formulas have the generic form `language(id, role, formula).` with language being `fof` or `tff`. The role of a FOF formula is one of *axiom*, *hypothesis* or *conjecture* (formula to be proven). The subset of accepted FOF formulas is tailored for the problems of our application and all accepted elements are listed below:

- *Values / Variables* – `$true, 1, -4, sym2`
  The values $true and $false encode the boolean constants *true* and *false*. Integer constants represent the corresponding integer value. Boolean and Integer constants are called *well-defined* values (that is: their semantic value is known). All other values (e.g. *sym2* are supposed to be (unknown) integer values. Variables starting with an upper case letter are all-quantor bound free variables used in patterns.

- *Functor Application* – `pred(x,y,z)`
  A *translation fact*, encoded as a predicate. If a functor application may also be part of the conjecture, the corresponding fact must eventually be derived for the proof to succeed.

- *Equality* – `x = y`
  Should *x* and *y* both be *well-defined* (but different) values the problem contains a contradiction. If equality is used in the conjecture the values *x* and *y* must eventually be unified for the proof to succeed.

- *Inequality* – `x != y`
  Should *x* and *y* both be the same value the problems contains a contradiction. If inequality is used in the conjecture the values *x* and *y* must not be unified for the proof to succeed.

- *Implication* – `lhs => rhs`
  If *lhs* (the pattern) evaluates to true *rhs* (the action) will be performed. An action may either be a function application, in that case a new fact will be added to the problem or an equality, which triggers an unification. No unbound free variables must occur in the action. An exemplary usage is the implication `(add(A,B,X) & add(A,B,Y)) => X=Y)`. Implications drive the unification, as their rhs usually contains new equalities.

- *Conjunction* - `formula1 & formula2`
  Informally introduced in above example, conjunction of terms is possible. Important applications of the conjunction is a conjecture consisting of multiple clauses and of course in complex patterns of implications.

- *Equivalence* – `lhs <=> rhs`
  The equivalence pattern will be translated into two implications (`lhs => rhs` and `rhs => lhs`).

8.2 Implementation

In this section we describe the algorithm and data structures used to implement vanHelsing as well as implemented optimizations. The whole problem is represented as graph with 2 basic node types, values and functors. Values are linked to functors using or defining them, and functors are linked to all values they reference. Any path in the graph therefore is an alternating sequence of value and functor nodes.

Each value is exactly stored once in the graph. When unifying two values the node with the fewer edges will be removed from the graph. All its neighbors will be linked to the value it was unified with.

We also remove duplicated functor nodes, that are functors of the same type connected to exactly the same value nodes (in the same order). Initially there are no such nodes by construction. Whenever two value nodes are unified we test their linked functor nodes for duplicates. Due to the structure of our problems a lot of duplicate functor nodes will be produced. The test for duplicated functors can be performed efficiently utilizing the problem graph.

vanHelsing is a command line tool designed for batch processing and implemented in C++. The current implementation uses 64 bit machine integers to represent integer values. Their range is sufficient for our applications.

8.2.1 Unification Algorithm

The idea is to repeatedly match all patterns until the graph does not change any more (problem becomes stable). Only when the problem is stable it is assured that no contradiction has been derived. The fix-point unification algorithm is given in listing 8.1. This is a major difference to superposition based theorem prover which report the first derived contradiction as proof.

When the number of unification reaches 0 we break the main loop (line 5). The patterns get matched one additional time to assure that no contradicting unification was performed during the last round (line 8). Due to the special structure of our problems (the conjecture consists of equalities) it is sufficient to consider the conjecture only here. Either all unifications to prove the conjecture have been derived or we report failure (line 14).

A so-called *proof script* is written into a separate file, containing all performed unification (and which rules induced them). It is also in TPTP format, which allows a vanHelsing derived proof to be validated using other provers.

```
while True:
        nr_uni,contra = patternMatching()
        if contra:
                reportContradiction()
        if nr_uni = 0:
                break

nr_uni,contra = patternMatching()
if contra:
        reportContradiction()

allconj = checkConjectures()
if not allconj:
        reportFailure()

reportSuccess()
```

Listing 8.1: vanHelsing Unification Algorithm

8.2.2 The problem graph

We distinguish 2 types of value nodes. A *well-defined* value node is an integer constant, a string or one of the boolean constants *true* and *false*. They are well-defined in the sense that their semantic value is known.

A hash-map is used to implement efficient lookup of values by their name. This also decouples patterns (which use variable names) from the problem graph. When a variable is referenced by a pattern this hash-map is used to lookup the value node. This allows unification of value nodes without considering the patterns.

8.2.3 Pattern Matching

As complex patterns are constructed using conjunction this is the only interesting case. The first functor pattern is matched against the problem graph by looking up all functors of the name. Each of the functors looked up binds the free variables of the pattern. When matching the next pattern all the looked up functors would need to be tested against the current assignment of free variables. The number of looked up functors can drastically be reduced by using term indexing (described in the next section).

8.2.4 Optimization techniques

We improved the performance of the vanHelsing prover by many magnitudes using the following optimizations.

Dead patterns

A rather obvious optimizations which does not try to match a function application pattern (e.g. `add(A,2,B)`) if there are no terms (here predicated names *add*) it could match. This optimization becomes effective if conjecture patterns inherit this property from their clauses. A generic set of axioms can then be used for each problem as they don't incur to the execution time.

Term Indexing

The unification process is driven by implications. Many of the axioms describe the syntactic equivalence of the data-flow trees. They all have the generic form

$$pred(A, B, X) \wedge pred(A, B, Y) \implies X = Y$$

The first predicate is matched and concrete values are bound to the the free variables *A* and *B*. Matching the second predicate can now be accelerated if *A* or *B* are well-defined (their value is known). vanHelsing stores all functors of a specific type in a hash-map (for the first lookup) and maintains hash-maps for functors with well-defined arguments. The current implementation considered the first 3 arguments. This is the most important single optimization.

Functor freezing

We call a sort of functors frozen if no functor of their name has been modified in the current round. A pattern is called frozen if it matches a functor which is frozen itself. The conjecture pattern inherits its frozen status from its clauses. Initially there are no frozen functors, assuring that each pattern is matched at least once. Frozen patterns may be skipped during the pattern matching phase as they can not produce any new unifications. A functor sort must be unfrozen when a new fact involving this sort is added to the problem.

8.3 Proof Debugger

A distinct feature of the vanHelsing prover is its capability to debug failing proofs. Exploiting the regular structure of our problems it is possible to identify a subgraph relevant to each failing conjecture.

The listing 8.3 shows a failing equality conjecture. Assuming that initially *sym1* equals *sym6* and *sym4* equals *sym8* (that is mapping M_I) it should be shown that *sym5* equals *sym9* (M_F). The calculation resulting in *sym7* exploits the fact that 1 plus 1 equals 2, which is unknown to the prover. Figure 8.2 shows the failing proof graph. The DFT of both values (*sym5* and *sym9*) are constructed. One is colored red, the other yellow. Nodes being part of both DFTs are colored orange. Figure 8.2 shows the resulting graph (unrelated nodes have been removed, i.e. the *unrelated* functor and its values are not printed). Visual inspection quickly reveals that the first different colors begin to appear with *sym2* and *sym7*.

By adding the needed axiom shown in listing 8.4 the proof succeeds. The resulting graph is shown in figure 8.3.

```
fof(id0,hypothesis,add(sym1,1,sym2)).
fof(id1,hypothesis,add(sym2,1,sym3)).
fof(id2,hypothesis,add(sym3,sym4,sym5)).

fof(id3,hypothesis,add(sym6,2,sym7)).
fof(id4,hypothesis,add(sym7,sym8,sym9)).

fof(id5,hypothesis,unrelated(sym10, sym11, sym12)).
```

Listing 8.2: Translation facts with 3 data-flows

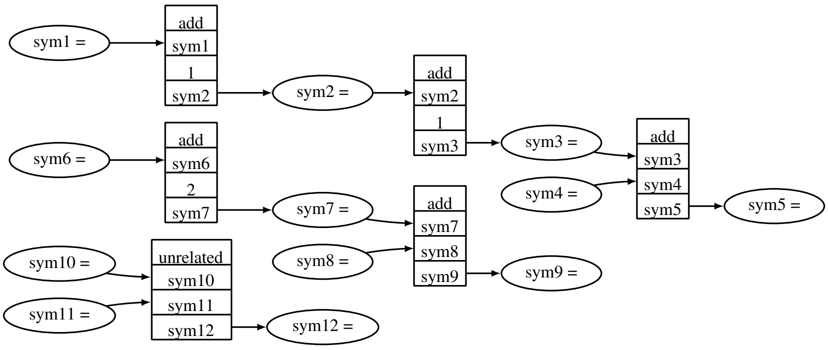


Figure 8.1: Initial Data Flow Trees

```
fof(ax1,axiom,(add(A,B,X)  &
               add(A,B,Y)) => X=Y).
fof(op1,hypothesis, sym1=sym6).
fof(op2,hypothesis, sym4=sym8).

fof(cj1,conjecture, sym5=sym9).
```

Listing 8.3: Missing an axiom

```
fof(ax2,axiom,(add(A,1,B) & add(B,1,C)
                  & add(A,2,D)) => C=D).
```

Listing 8.4: The missing axiom

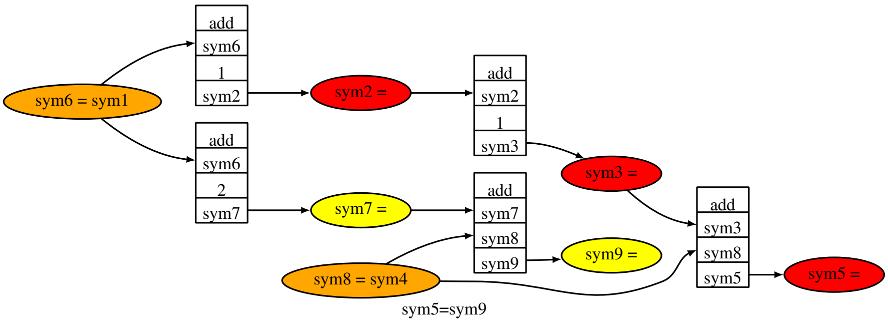


Figure 8.2: A failing proof

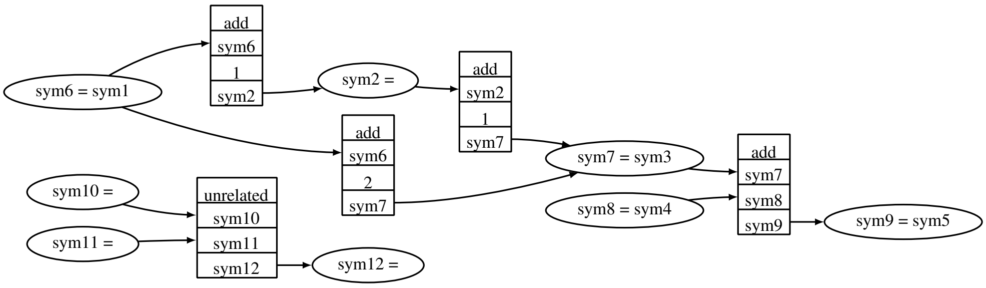


Figure 8.3: A succeeding proof

8.4 Defining Expressions

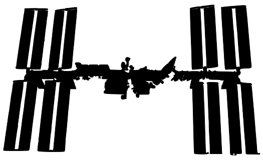
In our problems by convention the last argument of each predicate is the value computed by the function. Therefore for each value an expression leading to its calculation can be computed by a simple recursive algorithm. A function is part of the expression *iff* the value is its last argument.

Listing 8.5 shows the output for the input of listing 8.2 combined with listing 8.3. The *defining expression* is a canonical representation and computed after *initial* equalities (i.e. lines 3 and 4 in listing 8.3 have been resolved). Different symbolic values, calculated by the same expression operating on equal symbolic values therefore have *syntactical* equal *defining expressions*. This property is used in validation tools which need to assure that memory operations are not reordered. The problem is that a memory operation has the form `memory(timestamp, symbolic_address, symbolic_value)`. *Pre* and *post* traces have different symbols for the (semantically) same addresses. By computing the *defining expressions* those different symbols are canonized and corresponding addresses can be identified by string comparison.

```
sym10=sym10
sym9=add(add(sym6,2),sym8)
sym7=add(sym6,2)
sym5=add(add(add(sym6,1),1),sym8)
sym4=sym8
sym3=add(add(sym6,1),1)
sym2=add(sym6,1)
sym8=sym8
sym6=sym6
sym1=sym6
sym12=unrelated(sym10,sym11)
sym11=sym11
```

Listing 8.5: Defining Expressions

9 Instruction Set Simulation & Compiled Simulation



This chapter describes how the *formal* CASM models of a micro-processor can be used to synthesize *efficient* simulators. The empirical evidence (see section 10.1) shows that CASM can serve as the only, *unified* model for micro-processors and thus eliminate redundant special purpose specifications.

9.1 Instruction Set Simulation

In section 4.4 we showed the similarity of the CASM models with the simulator specifications of ISS in our research toolchain. Elimination of those redundant specifications in the ADL was the main motivation to develop the CASM compiler. By compilation of the CASM models the core for an ISS can be synthesized re-using the *semantic* specification of the micro-processor.

An ISS commonly works like an interpreter. It processes a stream of instruction words (of the micro-processor to be simulated) and applies their effect on a representation of the micro-processor state. An ISS alone is of little use, because most programs rely on an environment to operate in. *System Simulators* emulate the needed environment and are widely used. The main interest is the to ability to execute C programs.

It is therefore sufficient to emulate a hosted C environment (to execute operating systems one would need to emulate peripheral hardware). Our implementation is based on newlib [1], which has a well defined system call layer. Other C libraries are similar, though. We have implemented the following elementary C functions: *open, close, read, write, fstat, lseek, isatty, gettimeofday, unlink, lstat, exit*. Additionally a mechanism to pass command line arguments to the simulatee's main function has been implemented (by customization of the *_start* function). A customized linker script overrides the default implementations of the elementary functions with

[1] `http://sourceware.org/newlib/`

```
.globl open
.ent open
open:
        nop
        nop
        nop
        syscall 0
        /* return value in r2,r3 errono in r8 */
        lui     $9,%hi(errno)
        addiu   $9,$9,%lo(errno)
        sw      $8,0($9)
        jr      $ra
.end open
```

Listing 9.1: Elementary C Function Stub

our own when compiling the simulatee. Listing 9.1 shows the (MIPS assembly) implementation of the *open* function. All elementary C functions invoke a special *trap* instruction (i.e. *syscall* on MIPS).

The goal is to map those elementary C functions to corresponding calls of the host operating system. This is not possible if the *syscall* rule would be implemented in CASM. A so called *provider* mechanism is implemented in CASM which allows rules to be implemented in C. The CASM compiler includes header files and links with the *provider* implementation. *Provided* rules must interact with the CASM run-time, i.e. make use of the update set and lookup mechanism.

We have developed implementations for our proprietary DSP micro-processor and for the MIPS architecture. To realize an ISS the common semantic vocabulary must not only be evaluated symbolically, but also concrete (called *BitVector Operations Library*). We have chosen MIPS because well-known benchmark suites like MiBench [35] are available to evaluate the performance of this approach. An unmodified GCC toolchain (CodeSourcery MIPS 2012.03) is used to compile and link programs for use with the simulator. Figure 9.1 gives an overview of the architecture.

The system memory is also provided. This allows the ELF binary loader to efficiently initialize data sections of the program. Two CASM functions are used to read instructions. The opcode is read using the *PMEM: Int ->RuleRef* function, while instruction fields are accessed using the *PARG: Int * FieldValues ->Int*. *FieldValues* is an enumeration of MIPS instruction fields (e.g. RT, RS). The simulator run-time on-demand decodes instructions fetched using those two functions. An evaluation of the synthesized ISS for the MIPS architecture is given in section 10.1.2.

9.2 Instruction Set Simulator Verification

Three specifications of the MIPS instruction set have been developed for this thesis. One is the functional model of the MIPS instruction set. An example is given in listing 9.2. This specification only models the effects of an instruction visible to the programmer, hence functional model. A concrete implementation of a MIPS micro-processor uses a pipeline and the other two spec-

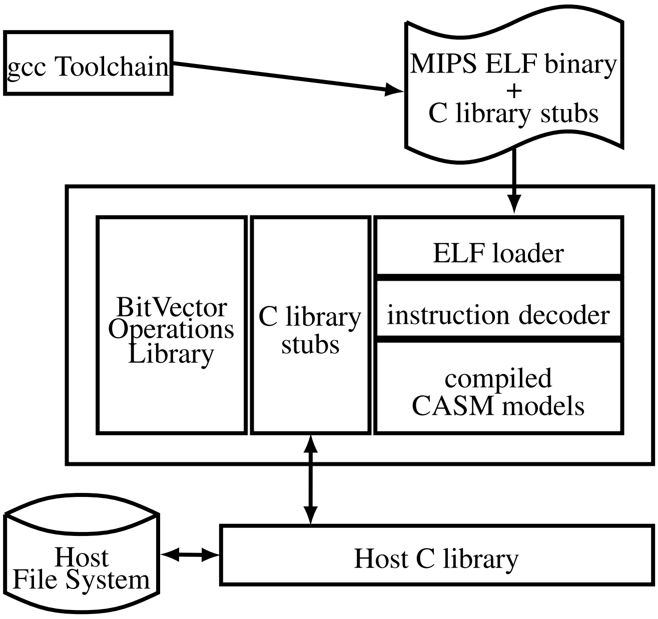


Figure 9.1: Overview of ISS

```
rule write_reg(reg : Int, val : Int) =
  if reg != 0 then
    GPR(reg) := val

rule addiu(addr : Int) =
let rs = PARG(addr, FV_RS) in
let rt = PARG(addr, FV_RT) in
let imm = PARG(addr, FV_IMM) in
  call(write_reg)(rt, BVadd_result(32, GPR(rs),
                  BVSignExtend(imm, 16, 32)))
```

Listing 9.2: ADDIU functional Model

ifications model the effects of the instruction on the pipeline. Listing 9.3 shows the pipelined version of the instruction.

The pipelined instruction models still need a concrete pipeline which executes them. A model implementing operand forwarding and a so called bubbling pipeline have been implemented. A bubbling pipeline dynamically stalls and inserts *nop* if a data hazard is detected.

The functional model can be thought of a specification for a MIPS micro-processor and the forwarding and bubbling model as concrete implementations. Because the models are written in CASM one can immediately profit from the proof techniques and verify the correctness of the implementations. By *symbolic* execution the data-flow of the pipelined models, under a concrete pipeline implementation (i.e. forwarding, bubbling), can be validated against the data-flow of the functional model.

For each MIPS instruction a template is instantiated (by replacing the macro *@INSTRUC-*

```
enum PipelineStages = {ID, EX, MEM, WB}
enum PipelinePhases = {begin, end}

rule addiu(addr:Int, stage:Int, phase:Int) =
{
  if stage = ID and phase = end then
  let rs = PARG(addr, FV_RS) in
  let rt = PARG(addr, FV_RT) in
  let imm = PARG(addr, FV_IMM) in
  {
    call(ID_READ_OP1)(rs)
    IDOP2 := BVSignExtend(imm, 16, 32)
    IDRESREG := rt
  }

  if stage = EX and phase = begin then
    EXRES := BVadd_result(32, EXOP1, EXOP2)

  if stage = WB and phase = begin then
    call(write_reg)(WBRESREG, WBRES)
}
```

Listing 9.3: ADDIU pipelined Model

```
function (symbolic) SYMBOL : Int -> Int
function (symbolic) MEMORY : Int -> Int
function (symbolic) PARG : Int * FieldValues -> Int

rule initR = {|
        BRANCH := undef

        call(@INSTRUCTION@)(0)
        program(self) := undef
|}
```

Listing 9.4: Template for functional Instruction Models

TION@), which is then used to create the data-flows. Listing 9.4 shows the template used to execute functional models and listing 9.5 for pipelined models.

The traces created by symbolic execution of those 2 programs must result in semantically equal expressions for observable state (i.e. register file, memory). This problem is very similar to the translation validation problem and can be expressed by the same techniques.

This does not validate all aspects of the pipeline implementation and instruction models, though. For each pipelined instruction model it has been shown that it correctly implements the semantics, *iff* executed in the pipeline without interference of other instructions. Thus a validation of the pipeline model itself is still needed.

Listing 9.6 shows the CASM program used to perform the pipeline validation (for functional models, pipelined is similar but more complex). A special crafted instruction (*binop*, line 9) is subsequently executed in each pipeline stage (and triggers data hazards). A correct pipeline implementation (either forwarding or bubbling) must generate the same data-flow as the functional model. If a concrete implementation passes this test its operand forwarding is most likely correct. Please note that due the nature of symbolic execution this is not just a simple test.

```
function (symbolic) SYMBOL : Int -> Int
function (symbolic) MEMORY : Int -> Int
function (symbolic) PARG : Int * FieldValues -> Int
function  PMEM : Int -> RuleRef initially {0 -> @@INSTRUCTION@}

rule initR = {|
        call init_pipeline

        PC := 0
        CYCLES := 0
        BRANCH := undef

        call IF_stage
        call execute_pipeline

        call step_pipeline
        call execute_pipeline
        call step_pipeline
        call execute_pipeline
        call step_pipeline
        call execute_pipeline
        call step_pipeline
        program(self) := undef
|}
```

Listing 9.5: Template for pipelined Instruction Models

Any (hidden, non-obvious) control flow branches (e.g. WRITE_REGISTER rule in line 6) are taken (creating multiple traces) and verified. This validation therefore has 100% path coverage, something which is very hard to achieve with ordinary testing.

Although not all aspects of the micro-processor specifications have been validated in this work, it has been demonstrated how easy such a validation can be performed with CASM models. This is a big advantage of *formal* specifications compared to simulator specifications.

9.3 Compiled Simulation

Compiled Simulation [16] is a technique which aims to increase the simulation speed of single applications by partial evaluation performed at compile time. The program to be simulated (simulatee) and a model of the simulated machine are compiled to a single binary. Only those parts of the state which are actually needed for the simulation are computed. In this section we want to investigate the applicability of compiled simulation to CASM models.

Our prototype implementation transforms binary MIPS programs into CASM models on a basic block level. Handling of computed goto and function pointers is currently not implemented, but could easily be done by integrating the CASM based ISS (line 12 in listing 9.9). A python script performs a simple basic block analysis of a MIPS binary and emits a representation using the CASM functions *BASICBLOCK* (mapping of a basic block number to a CASM rule implementing this block), *BASICBLOCK_SA* (original address of first instruction of block), *PARG* (branching instructions have target basic blocks in an additional virtual instruction field) and *PMEM* (branching instructions have been replaced by models using target basic blocks).

```
function (symbolic) SYMBOL : Int -> Int
function (symbolic) GPR : Int -> Int
function PARG : Int * FieldValues -> Int

rule WRITE_REGISTER(reg, val) =
        if reg != 0 then
                GPR(reg) := val

rule binop(addr:Int) =
        call(WRITE_REGISTER)
                (PARG(addr, FV_RD), BVbinop(GPR(PARG(addr, FV_RS)),
                 GPR(PARG(addr, FV_RT))))

rule initR = {|
        PARG(0, FV_RD) := SYMBOL(0)
        PARG(0, FV_RS) := SYMBOL(1)
        PARG(0, FV_RT) := SYMBOL(2)

        PARG(1, FV_RD) := SYMBOL(3)
        PARG(1, FV_RS) := SYMBOL(0)
        PARG(1, FV_RT) := SYMBOL(0)

        PARG(2, FV_RD) := SYMBOL(4)
        PARG(2, FV_RS) := SYMBOL(3)
        PARG(2, FV_RT) := SYMBOL(3)

        call(binop)(0)                  // s0 := s1 binop s2
        call(binop)(1)                  // s3 := s0 binop s0
        call(binop)(2)                  // s4 := s3 binop s3

        program(self) := undef
|}
```

Listing 9.6: Validation of Pipeline Model

Figure 9.2 gives an example for a modified branching instruction (listing 9.7) and the original model (listing 9.8). A branch to an instruction address is signaled by setting the CASM function *BRANCH* while a branch to a basic block address is signaled using the function *BLOCK*.

Our main rule of the simulator is given in listing 9.9. As long as the simulatee did not *trap* and *BLOCK* contains a valid block identifier the rule implementing the referenced basic block will be evaluated. Each such rule is a sequential composition of the instructions forming the basic block.

Because all instructions are captured in a sequential execution context the internal state transition is hidden and can not be observed by the calling rule. As long as the resulting update set is not changed, i.e. the *semantics* of the basic block is preserved, arbitrary optimizations may be performed on the CASM program. Listing 9.10 shows an example of such a block. The successor block is written to *BLOCK* in line 8. The remaining lines of the rule call *bb_call* which loads the given instruction into the pipeline and executes a cycle of the CPU. Note this rule is inlined by the compiler, as well as *bb_execute* and *step_pipeline*.

The optimizer (see section 3.3) then applies the three optimizations redundant lookup, preceded lookup and redundant update elimination. Combined with constant folding and propagation these optimizations cover the partial evaluation done by CS tools. The effect of a preceded

```
rule bne(addr:Int,
         stage:PipelineStages,
         phase : PipelinePhases) =
if stage = ID and phase = end then
let rs = PARG(addr, FV_RS) in
let rt = PARG(addr, FV_RT) in
let offset = PARG(addr, FV_OFF) in
{|
  call (ID_READ_OP1)(rs)
  call (ID_READ_OP2)(rt)
  if BVunequal(32,IDOP1,IDOP2)=1 then
    BRANCH := BVadd_result(32, addr+4,
              BVse(18, 32,
              BVshift_result(18, 0, 0,
                           offset, 2)))
|}
```

Listing 9.7: Original BNE Model

```
rule bb_bne(addr:Int,
            stage:PipelineStages,
            phase : PipelinePhases) =
if stage = ID and phase = end then
let rs = PARG(addr, FV_RS) in
let rt = PARG(addr, FV_RT) in
let offset = PARG(addr, FV_BBOFF) in
{|
  call (ID_READ_OP1)(rs)
  call (ID_READ_OP2)(rt)
  if BVunequal(32,IDOP1,IDOP2)=1 then
    BLOCK := offset



|}
```

Listing 9.8: Modified BNE Model

Figure 9.2: Modifies branch instructions for CS

```
rule run_program dumps (BLOCK, BRANCH, GPR, LO, HI) -> trace  =
{|
  if BLOCK = undef and BRANCH = undef then
  {
    print "program stopped (BLOCK and BRANCH undef)"
    call dump_machine_state
    program(self) := undef
  }
  else if BLOCK = undef then
  {
    print "ERROR, BLOCK is undef, BRANCH=" + hex(BRANCH) +
          " need to enter interpreter -> not implemneted!"
    call dump_machine_state
    program(self) := undef
  }

  debuginfo block "executing " + BLOCK + "@" + hex(BASICBLOCK_SA(BLOCK))
  call (BASICBLOCK(BLOCK))

  if trapped then
  {
    print "program stopped (trapped)"
    call dump_machine_state
    program(self) := undef
  }
|}
```

Listing 9.9: Compiled Simulation Main Rule

```
rule bb_call(r: RuleRef, i: Int) = {|
        pipeline(ID) := [r, i]
        call bb_execute
        call step_pipeline
|}

rule bb_42 = {|
        BLOCK:=38
        call bb_call (@sll, 0x8000132c)
        call bb_call (@sll, 0x80001330)
        call bb_call (@sll, 0x80001334)
        call bb_call (@_syscall, 0x80001338)
        call bb_call (@lui, 0x8000133c)
        call bb_call (@addiu, 0x80001340)
        call bb_call (@bb_jr, 0x80001344)
        call bb_call (@sw, 0x80001348)
|}
```

Listing 9.10: An optimizable Basic Block

lookup optimization e.g. is comparable to map register file access to local variables. Redundant update elimination will prevent the calculations of all unused updates to the flag register file (on micro processors which can implicitly set flags). The combination of redundant and preceding lookup elimination eliminates access to the pipeline function. Benchmark results are presented in section 10.1.2.

A CASM specification can therefore with minimal further tool support be used to perform compiled simulation. Because the optimization is performed on CASM code this approach is fully portable across various architectures.

10 Evaluation

Measure what is measurable, and make measurable what is not so.
Galileo Galilei

The evaluation section of this work is split into three parts Section 10.1 covers performance aspects of the CASM implementation. In section 10.2 we compare the performance of the vanHelsing prover with other well-known tools in the ATP realm. And in section 10.3 we report on the performance of the implemented validator prototypes.

10.1 CASM implementation

Evaluation of the CASM tools is split into three sections. Section 10.1.1 compares the CASM interpreter and baseline compiler to other ASM implementations. In section 10.1.2 we measure the performance of the ISS synthesized from the CASM models. And in section 10.1.2 we finally benchmark the optimization implemented in the CASM compiler. This used benchmarks are an application of the CS approach and therefore also document the ability to synthesize CS tools from CASM models.

10.1.1 CASM and other ASM implementations

In this section we evaluate the quality of the CASM implementation (i.e. interpreter and baseline compiler). For this purpose we compare it to other available implementations of ASM based languages, namely CoreASM and AsmL. CoreASM is an interpreter written in Java while the AsmL language is compiled to .NET code. A small suite of programs each stressing a different implementation detail of ASM languages has been implemented for each language.

The *bubblesort* program (a very naive implementation of the well known sorting algorithm) performs many steps with small update sets. It aims to benchmark the effectiveness of applying update sets to ASM functions. *Fibonacci* uses dynamic programming to calculate the well known numbers. It benchmarks rule invocation (recursive) and has a moderate size of the update set. *Quicksort* (the sorting algorithm) makes heavy use of sequential execution, although the update sets are very small. The *sieve* program is an implementation of Eratosthenes famous prime

		small data sets				
	trivial	sieve	quicksort	gray	fibonacci	bubblesort
CASM	0.0865	0.0857	0.0842	0.0882	0.0854	0.0859
AsmL	0.1292					
CASM-i	0.0048	0.10	0.0212	0.2287	0.0107	0.0466
CoreASM	1.3604	13.82	32.51	57.61	67.24	213.62
		large date sets				
		sieve	quicksort	gray	fibonacci	bubblesort
CASM		0.0822	0.586	0.7702	3.0436	2.5458
AsmL		74.39	3.0628	24.3702	4.1752	5.2748
CASM-i		1.05	35.41	40.83	79.17	95.43
CoreASM						

Table 10.1: Execution times CoreASM, AsmL, CASM

number sieve. This program heavily stresses the implementation of the update set, everything is executed sequentially producing large update sets. The benchmark program *gray* calculates gray codes for a given word length. It is the program with the most output and a mix of sequential execution, rule invocation and numeric operations. *Trivial* is the trivial program, immediately exiting without any operation. It is used to measure start-up overheads of the various implementations

The performance of the various implementations varies a lot. We use small data sets for the interpreters and larger sets for the compilers to have measurable execution times.

For benchmarking we use the CASM compiler (rev. 1a092c) and gcc 4.7.2 (as shipped with Ubuntu 12.10). We do not perform any CASM specific optimizations and disable optimizations of the C compiler (-O0 flag). The CASM interpreter (CASM-i) is the same version as the compiler.

The CoreASM engine version used is 1.5.6-beta using the command line driver Carma 0.7.3 (latest release). We executed CoreASM using Java 1.7 with the 64 bit Server VM (23.7-b01).

Microsoft's AsmL implementation compiles to .NET code and is available under the MSR-LA (free for research) license on `http://asml.codeplex.com/`. We downloaded version 80132 and followed their build instructions using Visual Studio C# 2005 Express Edition.

The benchmarks involving small data sets were executed on a Core i7-Q820 @1.73 GHz with 8 GiB memory under 64 bit Ubuntu 12.10. For the large data sets a dual boot system (Core i7-2600k @3.4 GHz, 8GiB memory) using 64 bit Windows 7 Enterprise SP1 and 64 bit Ubuntu 13.10 was used. We report on the average of 10 runs and started the AsmL binary once before the benchmark to exclude overheads induced by the .NET framework [1].

Our own implementation of a CASM interpreter (CASM-i) is designed to have very low start-up times and is used to execute small programs only. It is used in the compiler verification project. The baseline compiler is a magnitude faster than the interpreter (up to 60 times) which is a good indicator that the baseline compiler performs well.

[1] `http://msdn.microsoft.com/en-us/library/cc656914(v=vs.110).aspx`

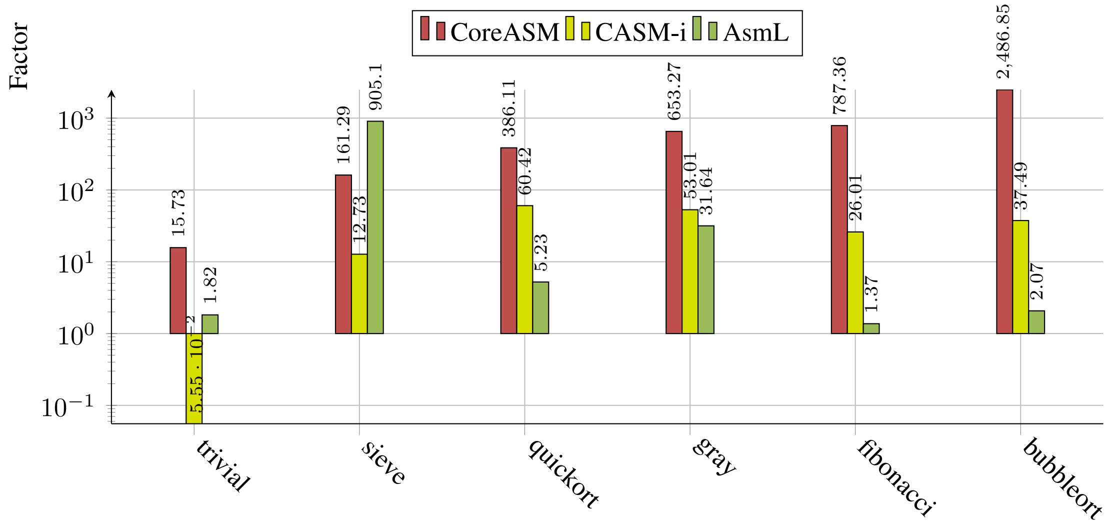


Figure 10.1: CASM relative Performance (Compiler is Baseline, smaller is better, log-scale)

When it comes to performance CoreASM is clearly inferior to the other implementations. Programs compiled by our compiler perform up to 2500 times better and even our interpreter is a magnitude faster. The focus of CoreASM are high level models though.

The AsmL results are varying a lot. For *fibonacci* performance is on par with the CASM compiler (still 35% slower, though). But *fibonacci* is also the benchmark putting the least pressure on ASM specifics. Its mostly recursive function invocation with a comparable small update set. *Bubblesort* is slower by a moderate factor of 2 while *sieve* is slower by a factor of 900. A detailed examination showed that AsmL has quadratic run-time for increased sizes of the sieve. The main difference in the two programs is that *bubblesort* executes a large number of machine steps each with a small update set, while *sieve* exactly executes one step. The update set produced by *sieve* is quite large (the whole array) and a lot of updates need to be merged sequentially. This indicates that AsmL is not optimized for this case and agrees with the observed behavior of *quicksort* (small sequential update sets) and *gray* (moderate sized sequential update sets). Overall the performance of AsmL compiled programs is significantly lower than programs compiled by the CASM compiler.

Figure 10.1 shows the relative performance (with CASM compiler being the baseline) of the 4 implementations, please note the logarithmic scaling of the y axis. Numeric values are found in table 10.1. The CASM baseline compiler is by far the best performing ASM implementation.

10.1.2 Symbolic Execution

We developed a micro-benchmark to stress all important implementation aspects of symbolic execution in the CASM interpreter. The size of the tests is much larger than the size of the

Engine	Creation	Expression	Forking	Start-up
CASM	2.14 s	3.54 s	2.82 s	0.009 s
CoreASM	n/a	n/a	n/a	1.3 s

Table 10.2: Micro Benchmark Results

problems coming from translation validation. Hence we don't expect scalability issues with the implementation of symbolic execution.

This benchmark was executed on a Core i7 @ 1.73 GHz using a 64 bit Ubuntu 12.10 and we report the average value of 5 runs. We compared start-up times to CoreASM 1.5.6-beta using Carma engine 0.7.3. Each of the following tests stresses an implementation detail of the symbolic ASM. Table 10.2 summarizes the results.

- *Creation* of symbols: The lazy initialization of symbols is an important feature of our implementation. Each computation step of the ASM induces one predicate for each symbol in use. The later a symbol is created the more predicates have to be emitted at creation time (as the symbol existed in all previous states as well). To stress the lazy initialization implementation a test program computing 1000 steps is used. In each step a new symbol is created. This results in 1000 symbols and 1002000 predicates to describe the whole state transition (1000 symbols in 1000 states plus initial and final state).

- *Expression* evaluation: Symbols are also created when evaluation expressions containing symbolic values. This test program performs 100000 operations using 100000 different symbols (in a single computation step). This results in 100000 predicates to be written to the symbolic trace.

- *Forking* on symbolic conditions: When control flow branches on a symbolic condition both traces need to be created. This can lead to a problem known as state explosion. An application may be able to handle a certain amount of branching however. A test program consisting of 11 nested if-then-else rules resulting in 2048 traces is executed. For our applications we observed a much smaller number of traces (2-8).

- *Start-up* of interpreter: As the interpreter is intended to execute a large number of short running programs the run-time overhead is of interest as well. A test program executing the trivial CoreASM/CASM program (`rule initR = program(self) := undef` producing one update and then terminating) was executed to measure this start-up overhead.

The empirical data show that the implementation of symbolic execution is efficient even for very large traces. Especially the low start-up overhead (absolute and compared to CoreASM) is very important for the intended application (execute many small CASM programs during translation validation).

	MIPS cycles			seconds		
	BMIPS	MIPS	FMIPS	BMIPS	MIPS	FMIPS
basicmath	550341866	333880341	328575616	2537,37	1323,53	132,62
bf.d	64946652	43557768	43556830	295,12	170,97	18,66
bf.e	64448155	43639002	43638063	292,23	172,02	18,62
crc	802657273	482047058	482020990	3600,91	1967,50	212,58
dijkstra	95287437	48939093	48859858	421,00	190,69	20,01
patricia	144287533	89624367	88147670	655,57	375,17	38,64
qsort	25761496	18442379	18255010	120,51	78,59	10,28
rawcaudio	59614091	34035944	34035911	265,53	138,41	14,82
rawdaudio	15208266	8664016	8663983	68,99	35,83	4,69
rijndael.d	42564700	38204264	38105252	203,91	158,03	16,10
rijndael.e	42567131	38381668	38282652	200,67	186,60	18,30
search	8757478	5776268	5689169	41,81	24,37	3,49
sha	13927633	12093339	12092840	66,36	49,94	5,79
susan	4820668	3420921	3401906	23,06	14,51	2,33

Table 10.3: Performance of FMIPS, MIPS and BMIPS ISSs

CASM synthesized ISS

To evaluate the performance of the synthesized ISSs we have created simulators based on the MIPS models. We compare all three MIPS implementations, that is the functional models (FMIPS), the models implementing a pipeline with operand forwarding (MIPS) and the bubbling pipeline (BMIPS). The most expensive operations in the CASM run-time are the ones involving the *update set*. Because the simulated state of FMIPS is the smallest we expect those models to deliver the best performance. MIPS and BMIPS are expected to have similar performance, their state is similar in size. Execution of MIPS programs using the BMIPS simulator will absolutely perform worse than MIPS. The reason is that pipeline bubbles are added when data hazards occur. The number of executed MIPS cycles is therefore higher than the number of cycles a pipeline with operand forwarding will need.

In table 10.3 we given the number of cycles and seconds each of the ISSs spent on the benchmark programs taken from the MiBench [35] suite. MiBench's small data sets have been used for all but the *search* benchmark. In figure 10.2 the achieved relative performance in MIPS instructions per seconds is presented.

The FMIPS ISS reaches up to 2.47 MHz which is a very good performance. (Brandner [15] reports on approx. 3 MHz peak performance for an interpretative MIPS ISS.) Simulating the whole pipeline (and the resulting update set operations) drastically reduces the performance for the MIPS and BMIPS ISSs. Nonetheless the CASM run-time, i.e. the *branded hash* and *linked list update-set* (see chapter 3) performs much better than the old python prototypes, which only achieved up to 1 MHz (Lezuo and Krall [47]).

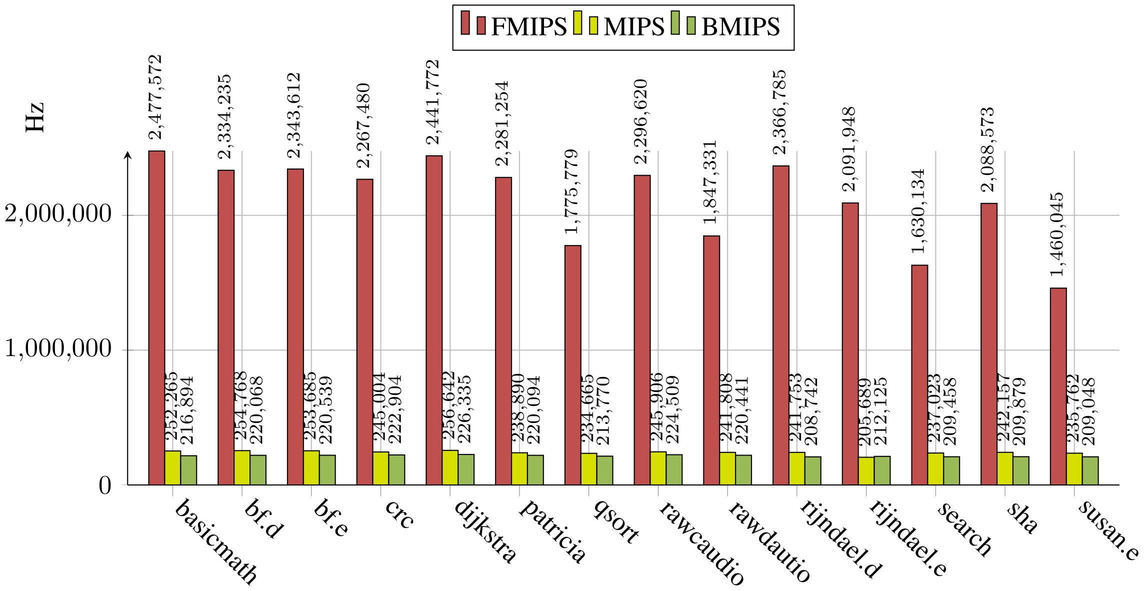


Figure 10.2: Performance of FMIPS, MIPS and BMIPS ISSs

CASM synthesized CS

We have applied our CS approach to a set of MiBench [35] programs using the functional MIPS models. As our optimizations perform aggressive inlining we only want to optimize the kernel of the applications to keep the increase in code size small. Our code generator can instrument the code to collect profiling information measuring the total execution time of each CASM rule (including time spent in invoked rules). Applying a simple heuristic all rules contributing at least 1% to the total run-time have been selected for optimization. Our assumption is that the effects on code size by inlining are small but the achieved effect (in terms of performance gains) large.

Lookup and update elimination work best on large rules, so their impact should be high if the frequently executed rules are large and low for small rules. Figure 10.3 depicts the contribution of a rule's execution time to total program execution time (bars) and their size in LOC (crosses) for the *rijndael* program. (The rule contribution 100% to total program execution time is the top-level rule, the second bar is a dispatching rule, all further bars correspond to basic blocks or instructions.) Note that two of the most contributing rules each have 1000 LOC. We expect to see a high impact of lookup and update elimination for the *rijndael* program. Figure 10.5 on the other hand shows that the *patricia* program has many small rules and the first large rule does not contribute much to total execution time. A high impact can not be expected. Figure 10.4 shows the same diagram for the *dijkstra* program. Medium sized rules with moderate contribution. We expect our optimizations to have an impact for this program.

We use 3 configurations of our compiler in this evaluation. Baseline is without CASM specific optimizations and without optimizations of the C compiler (-O0). The configuration

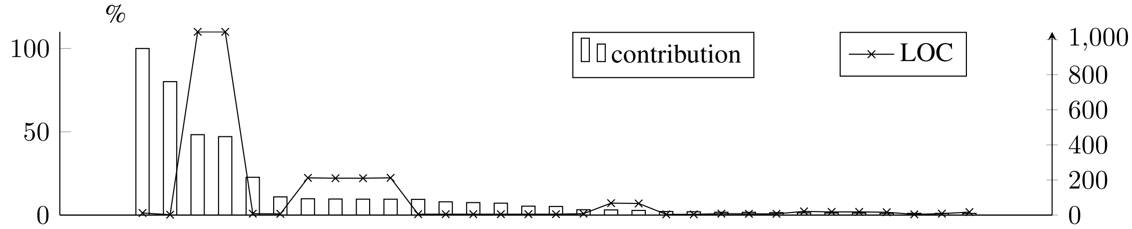


Figure 10.3: Rule Contribution and Size - Rijndael

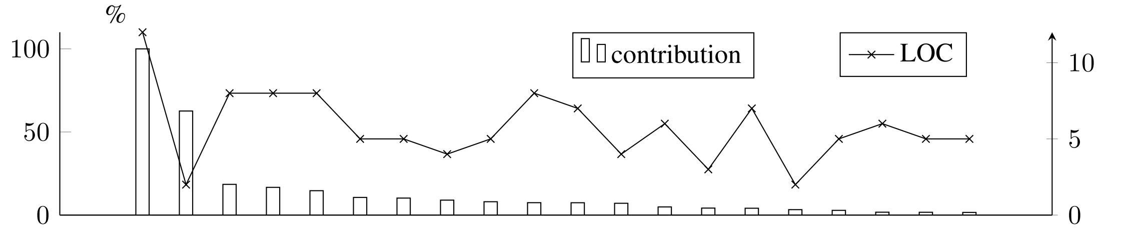


Figure 10.4: Rule Contribution and Size - Dijkstra

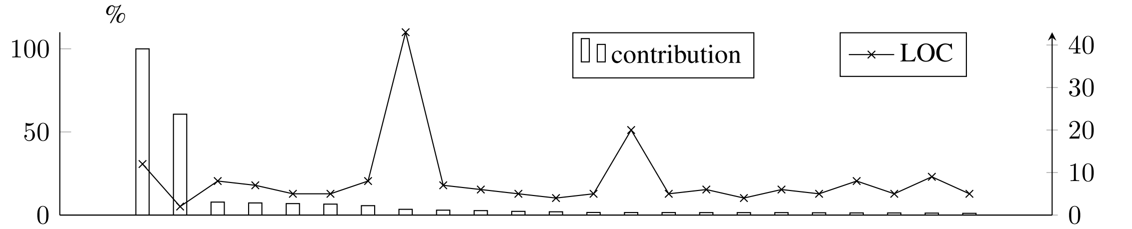


Figure 10.5: Rule Contribution and Size - Patricia

titled *O0* has CASM specific, but no C compiler optimizations (-O0). *O3* has CASM and full C compiler optimizations (-O3). The benchmarks were executed on Xeon E5504 @ 2.00GHz with 8GiB memory (on the Infragrid cluster [2]). We used gcc 4.4.7 on a Red Hat Enterprise Linux Server release 6.4 for compilation. Due to the shared nature of the cluster we report on the best of 10 runs here. MiBench's small data sets have been used for all but the *search* benchmark.

Table 10.4 lists for each benchmark program the total number of rules and the number of rules optimized as well as the total number of optimizations performed. We report on the number of constant propagation (cp), lookup eliminations and update eliminations. As expected we see a large number of optimizations performed in the *rijndael* program. Although *patricia* is doing well in numbers the effects do not materialize due to the disadvantageous distribution of block contribution to the total run-time.

[2] http://hpc.uvt.ro/infrastructure/infragrid/

	rules		optimizations		
	opt	total	cp	lookup	update
basicmath	30	4097	1440	236	22
bf	43	1226	8060	889	451
crc	17	3501	416	56	7
dijkstra	20	5455	494	52	1
patricia	23	5864	761	150	0
qsort	21	5393	720	65	1
rawcaudio	38	3293	656	65	1
rawdaudio	29	3293	656	65	1
rijndael	32	3431	42452	4394	2864
search	28	3239	2086	274	5
sha	26	3291	2840	381	3
susan	29	5337	7570	1192	224

Table 10.4: CASM Optimizations

		w/o opt	full opt
	LOC casm	sec	sec
basicmath	136871	8.16	18.10
bf	48693	3.67	50.06
crc	109625	3.50	4.24
dijkstra	208337	5.78	6.73
patricia	180455	5.60	7.57
qsort	165011	5.08	6.60
rawcaudio	106716	3.23	4.69
rawdaudio	106716	3.30	4.10
rijndael	149435	4.82	218.23
search	122043	3.18	5.62
sha	104539	3.25	8.42
susan	187091	5.46	50.19

Table 10.5: CASM Compiler Statistics (compile time)

In table 10.5 the size of the test programs in LOC and the compilation times with and without optimizations are listed.

In table 10.6 we summarize the output produced by the CASM compiler. Currently a single C file is generated for each rule. We are currently working on merging smaller rules to reduce the number of files. For *rijndael* we see the by far largest increase in code size with moderate 10%. The increase in the size of the binary is approximately 20%.

To assure that the observed behavior is not solely due to optimizations of the C compiler we report on the effects of compiling optimized CASM programs with and without compiler optimizations. Figure 10.6 shows the relative impact of CASM optimizations with and without

		total LOC C		binary MiB	
	C files	w/o opt	full opt	w/o opt	full opt
basicmath	4104	531769	+1357	29	35
bf	1233	179890	+10369	9.1	11
crc	3508	419546	+679	24	29
dijkstra	5462	691294	+866	38	46
patricia	5871	694523	+622	39	48
qsort	5400	641228	+1328	36	44
rawcaudio	3300	402138	+901	23	27
rawdaudio	3300	402138	+946	23	27
rijndael	3438	524481	+49198	26	32
search	3246	419649	+4660	23	28
sha	3298	408628	+5506	23	28
susan	5344	727943	+10029	28	46

Table 10.6: Generated Output Statistics

optimizations by the C compiler. The relative performance is clearly decreased but our optimizations still account for a factor of 2 (*rijndael*) to at least 1% for *patricia*. (On a side note: by using well-known compiled simulation techniques (e.g. [16]) the size of the basic blocks can be enlarged from which our compiler would profit immediately.) In figure 10.7 the overall speedup factors for the applications are shown along with absolute performance data. The speedup is relative from the non-optimized version to the fully optimized one. We are able to achieve factors 6 and above here. For *rijndael* (factor 5.44) more than 50% of this speedup is due to CASM optimizations (the rest is due to the C compiler).

The MHz value relates the total number of simulated MIPS instructions to the absolute runtime of the programs. A solid performance of more than 3 MHz simulation speed is achieved. The numbers also indicate that the *natural* performance of the programs would be approximately 500 kHz. *Search* and *susan* show very low performance here. This is due to the very short execution time of these two programs (5 and 4 seconds). The start-up time of the programs is approximately 1 second (the initial memory state of the MIPS programs (data section) is initialized by a CASM rule producing updates) therefore a significant reduction of simulation speed is expected.

The experimental data show a huge performance increase achieved by the CASM compiler. A speedup of more than factor 6 can be achieved. We showed that the C code generated by the CASM compiler can be very efficiently optimized. Our novel optimizations lookup elimination and update reduction can increase program performance up to 264%. This shows that they are highly effective.

From a pure CS perspective this results are not competitive. Farfeleder [24] reports on simulation speeds of 78 MHz up to 181 MHz for fully cycle-accurate simulation (which our simulation is). By reducing the precision up to 566 MHz are achieved. The main reason that our CS approach does not perform significantly better than synthesized ISS is in the increased size of the *update set*. In the *bf* benchmark the average size of the update set produced by each rule

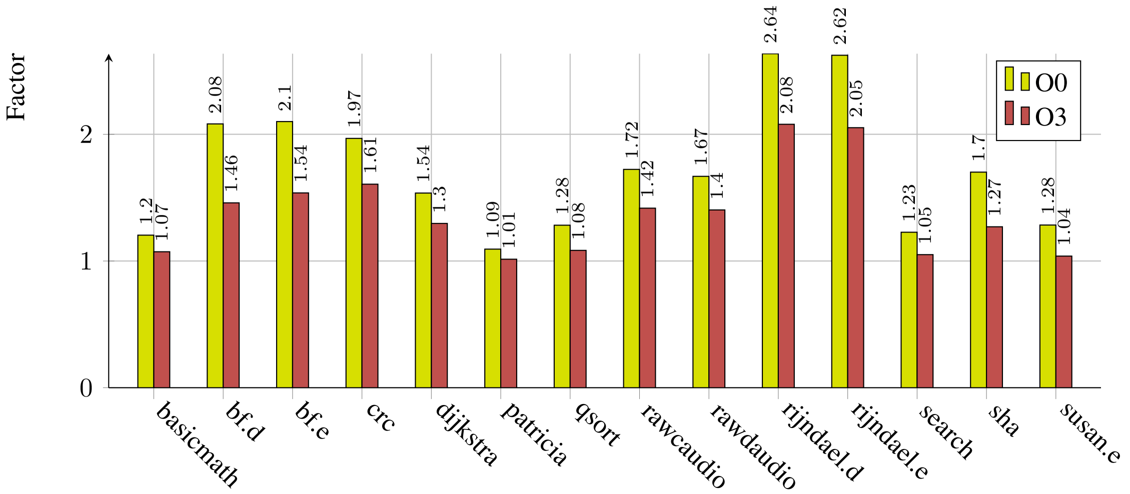


Figure 10.6: Impact of CASM Optimizations

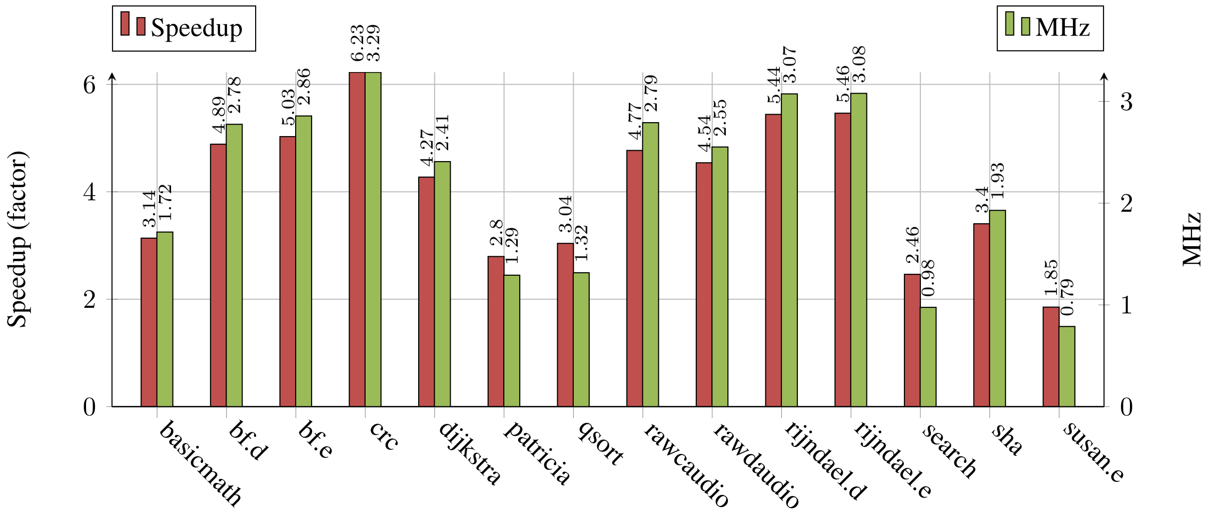


Figure 10.7: Improvements and Total Performance

	Size			Run-time (seconds)			
Set	# files	median	total	vanHelsing	Vampire	E	Z3
isel.succ	1705	25 kiB	49 MiB	**13.76**	24.54	44.31[a]	42.41
regalloc.succ	454	412 kiB	239 MiB	**49.11**	54.79	491.98	55.34
vliw.succ	401	484 kiB	259 MiB	**54.55**	209.13	816.41	233.74
vliw.fail	27	905 kiB	22 MiB	**4.38**	17.54	88.72	81.21[a]
isel.fail	343	29 kiB	12 MiB	**2.97**	7.45	38.82	961.81[a]

[a] see text

Table 10.7: Benchmark Set and Performance

is increased from 3,4 (a single instruction) to 28,9 (a basic block). This increases in size also increases the cost of the *lookup* and *merge* operations and thereby eliminates the performance gained by the optimizations. An optimization which aims to keep the update set as small as possible is discussed in section 11.2. The second cause is that the current implementation of the CASM analysis framework misses many possibilities to optimize redundant updates (can be seen in figure 10.4, often only 0 or 1 update is eliminated). Experiments demonstrate that exploiting the full potential of update redundancy would drastically increase the performance for CS. More details will be published in Paulweber's master thesis [56].

The important aspect however is the fact that the CS tool is synthesized from a *unified* processor specification. The issues negatively affecting performance have been identified and we expect a significant boost in performance from a complete implementation.

10.2 vanHelsing Prover

In this section we compare the performance of our vanHelsing prover compared with Vampire, Eprover and Z3 (on problems coming from translation validation).

We have compiled 5 sets of benchmarks from three different back-end passes of our compiler. The problems within all sets have a common structure, which is different between the sets. Instruction selection (isel) problems are the most complex, because the transformation has the largest impact on the data-flow. Register allocation (regalloc) problems are of modest complexity, depending on the amount of spill code inserted. Without spilling the data-flow does not change at all, but if registers were spilled the changes are intrusive. VLIW scheduling (vliw) problems are most simple. Instructions are reordered, the data-flow will not be changed at all. Normally the problems emitted by our compiler can be proven, i.e. *isel.succ*, *regalloc.succ* and *vliw.succ*. During development we also collected a set of problems which can not be proven (incorrect witness information, missing axiomization), i.e. *isel.fail* and *vliw.fail*. Interestingly we noticed that Z3 does not scale well with respect to performance on the failing problem sets.

The problems emitted by the validators can be directly used by Vampire and vanHelsing. Eprover does not support types, they are removed by a script as a preprocessing step. We used the *tptp2x* program part of TPTP to convert the problems into *smt* format used by Z3.

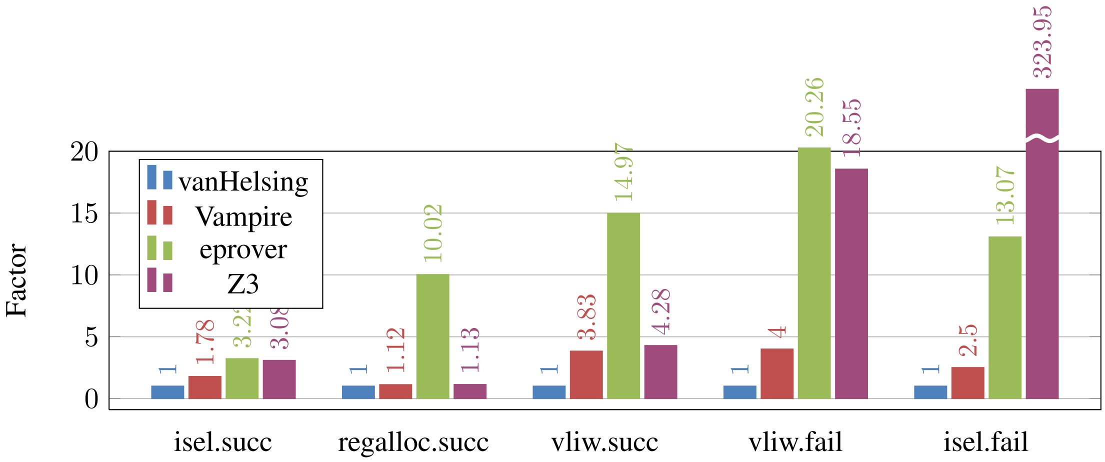


Table 10.8: Relative Performance (Factor), smaller is better

Table 10.7 contains the number of problems in the set, average file size (median) as an indicator of the complexity of each problem, the total size of the set and the time each prover needs to process the whole problem set. We report on the best of 3 runs of vanHelsing (version aa115e4), Vampire (1.8 rev. 1362), Eprover (E 1.8-001 Gopaldhara) and Z3 (4.3.1). The tests were performed on a Core i7 @ 1.73 GHz using a 64 bit Ubuntu 12.10. Eprover has been executed in silent mode and failed to prove one problem of the *isel.succ* set. Vampire is a very fast prover and has won the FOF section of the CASC [64] competition for many years now. vanHelsing always performs better than Vampire (roughly factor 2), which itself always is the second fastest prover. Eprover's performance is generally worse (factor of up to 20). Z3 in general performs similar to Vampire, but has a hard time when the problem has no proof (is *sat*). We used hard timeouts (*-T:3 -t:3*) of 3 seconds (vanHelsing needs less than 3 seconds for all 343 problems in the set), but Z3 failed on all problems of the *vliw.fail* set and only found the solution on 28 problems of *isel.fail*. Increasing the timeout to 240 second results in finding 309 (of 343) models for the *isel.fail* set, but still no solution for any of the problems in *vliw.fail*. Figure 10.8 shows the relative performance for all provers on each of the problem sets, normalized to vanHelsing's total run-time.

The empirical data clearly demonstrate that the simple problem structure produced by our translation validation approach enables the use of a much simpler and therefore more effective specialized prover like vanHelsing. Even the current implementation performs much better than widely used and highly optimized off the shelf provers like Vampire and Z3.

10.3 Translation Validation

In this section we report on the performance of 3 prototype validators which have been implemented as part of this thesis. The tests were performed on a Core i7 @ 1.73 GHz using a 64 bit Ubuntu 12.10 using all 8 available cores to utilize the parallelism. We report the Wall-clock

	preprocessing	execution
MIR CASM	11 s	7 s
LIR CASM	146 s	6 s
vanHelsing		19 s
total	156 s	33 s
validation		244 s

Table 10.9: Instruction Selection Time Spent (8 cores)

time.

10.3.1 Instruction Selection

Our research compiler dumps the mMIR and cLIR as described in sections 4.3.1 and 4.3.2. To ease implementation the compiler also creates a first-order representation of the equalities for the initial state and the conjecture to be proven (using the register allocation mapping, see section 7.2). In a productive version the validator needs to calculate this (as part of the trusted code base).

The dumped CASM fragments are then combined with the machine model (which has been created by the casm_gen tool, see section 4.1). We use the C preprocessor to implement this, which is not very efficient and will be changed in a productive version. A specialized CASM main rule then drives the generation of the symbolic traces.

The research compiler is still in development and the number of applications which can successfully be compiled is therefore limited. We chose the largest well-known supported application as our benchmark program, the AES (*rijndael*) reference implementation, consisting of 3 source files (*rijndael-alg-ref.c, rijndael-api-ref.c* and *rijndael_main.c*).

The compiler matches 1904 mMIR trees in the instruction selection pass. This results in the same number of symbolic mMIR traces (1904), but, due to branches, 2116 cLIR traces. The validator combines them into single problem files and adds the initial equalities, conjecture and the axiom set. Those 2116 problem files are then handed to the vanHelsing prover. To assure the correctness of the proofs our prototype implements an additional validation step. The *proof script* created by vanHelsing is validated against the conjecture (using the Vampire prover), which assures that the proof actually shows the conjecture. vanHelsing's *proof script* is also checked to have no contradiction with the original problem file (again using Vampire). Those two steps (no contradiction unified from original problem file and *proof script* is a witness for the conjecture), validated by a third-party prover, give us a very high confidence in the proofs (and the implementation of the vanHelsing prover). The total execution time of the validator (including the validation step) is 7 minutes and 20 seconds. Table 10.9 breaks this down to the single tasks performed.

The research compiler was not developed with creation of witness information in mind and is very eager to free its internal data structures as quickly as possible. The prototype witness generation code therefore can't access some mapping information and the witness information is wrong in a small number of cases. This results in the proofs to fail. We have investigated those proofs and by manually adding the missing information they can be proven successfully.

It can clearly be seen that most of the time is spent validating the results and in preprocessing the LIR CASM fragments. Validation is not mandatory in a productive implementation and the use of GCC to preprocess the CASM fragments is not optimal. A productive tool would reduce the times spent in preprocessing dramatically. The validation core task (symbolic execution and proving) only needs 33 s (8 cores) for a 841 LOC (excluding header files) C program.

10.3.2 Register allocation

The register allocation prototype was implemented for the MIPS architecture using the LLVM toolchain (version 3.2, basic register allocator). We chose a different compiler for three reasons i) to demonstrate that our approach can be applied to various compilers, ii) to have a more complex register allocator which performs some other optimizations on-the-fly (i.e. rematerialization) and iii) to be able to compile larger programs than it is currently possible with the research compiler.

Dumps of *llc's* IR were obtained using the command-line options *-print-before=virtregmap* and *-print-after=virtregrewriter*. To include the register allocation mapping (*VirtRegMap*) into the dumps a small modification of the method *VirtRegRewriter::runOnMachineFunction* must be made. Those dump files are then processed by a python script which translated each C function (*pre* and *post* transformation) into CASM code and extracts the CFG and function live-in, live-out registers. Each function is then split into their basic blocks and those blocks are combined with CASM models of the MIPS architecture (i.e. instruction set models, register file and memory). The blocks are then symbolically executed to create the trace files.

On the traces a def-use analysis is performed to ultimately calculate liveness (backwards analysis) and constness (fix-point analysis). The current prototype relies on the correctness of the compiler provided CFG and function live-in, live-out registers. However a productive implementation would have a correct CFG available in our proposed framework, and live-in, live-out register would be known due to the prolog and epilog insertion pass (see section 7.1).

For each live-in register a formulae is emitted stating equivalence with the value in the allocated register (or stack slot, if spilled). For all registers which hold a constant value a formulae stating that fact is emitted as well (the compiled may now rematerialize at will). The live-out registers are finally used to create the conjecture for the problem file. A register may be live-out but without a definition in this block. In this case the symbolic trace will contain no information about it and it must be excluded from the conjecture. Memory side-effects are added to the conjecture as well, the details are described in section 7.6.

Axioms are then added and the problem file is passed to the vanHelsing prover. A validation phase utilizing the Vampire prover is performed again (see instruction selection validator in section 10.9). Validation of register allocation of the *bzip2* program, part of SPECINT2000, (including the validation step) needs 15 minutes and 19 seconds. Table 10.10 breaks this down to the single tasks performed.

We can successfully prove all problem files.

The majority of time is consumed by the validation step (again using Vampire instead of vanHelsing). While validation is a good thing during development a productive tool would not need to perform this step. The remaining time is spent in preprocessing of the problem files (38

	preprocessing	execution
CASM	10 s	55 s
liveness, constness	16 s	
problem files	12 s	149 s
total	38 s	204 s
validation		643 s

Table 10.10: Register Allocation Time Spent (8 cores)

	preprocessing	execution
CASM	102 s	411 s
vanHelsing	11 s	35 s
total	113 s	446 s
validation		107 s

Table 10.11: VLIW Scheduling Time Spent (8 cores)

s) which evenly distribute to creation of the CASM files (10 s), performing the analysis (very simple, inefficient python implementations, 16 s) and generating initial state formulae and the conjecture (12 s). Creation of the symbolic traces needs 55 s, and the majority is spent proving the problem files (149 s). The *bzip2* source code is 4639 LOC C code (excluding header files).

10.3.3 VLIW Scheduling

The prototype implementation for the VLIW scheduler is based on our research compiler again. We modified the compiler to dump the results of its own snake block analysis along the sLIR CASM models for each function. An productive version of the validator would need to calculate the snake blocks as part of the trusted code base. From this representation we create CASM code for each of the sLIR snake blocks and also their cLIR counterparts.

This per snake blocks CASM programs are then executed to generate the trace files. For this validator the initial conditions and conjectures are trivial to compute, the observable states (i.e. registers and memory) must match exactly. To handle predicated execution the wrapper for predicated execution of cLIR code is used (see section 4.3.2).

We report on the validation of the *rijndael* reference implementation, which consists of 683 snake blocks. The validation of the 841 LOC sources needs 11 minutes and 32 seconds. Table 10.11 breaks this down to the single tasks.

102 seconds are spent to create the CASM representation of the snake blocks, which is implemented as a python script for this prototype. Symbolic execution takes the majority of the time (411 s), but as a side-effect also validates that no resource conflicts exist in the bundles (they would trigger a conflicting update). Creation of the problem files is straight forward (11 s) and vanHelsing very efficiently handles this kind of proofs (35 s). The validation step (which is not needed for a productive implementation) is again based on the Vampire prover.

16 problem files fail to be proven, we provide an analysis in the next section, to demonstrate the effectiveness of vanHelsing's proof debugging facilities.

10.3.4 Analyzing a Translation Failure

We observed 16 failing proofs during validation of the VLIW scheduling pass on the *rijndael* source code. The smallest failing problem file has 9280 lines of first-order formulae and a quite large conjecture. Vampire also fails to find a proof, but the only output it prints is *satisfiable*.

```
fof(cj0,conjecture,$true
        & stslirREG_Flag(0,1053,symclir1309)
        & stslirREG_Flag(0,1054,symclir1314)
        & stslirREG_Flag(0,1055,0)
        & stslirREG_AddressRegister(0,14,symclir1326)
        & stslirREG_AddressRegister(0,284,symclir1339)
        & stslirREG_AddressRegister(0,285,symclir1384)
        & stslirREG_DataRegister(0,282,symclir1296)
        & stslirREG_DataRegister(0,351,symclir1304)
        & stslirREG_GuardBits(0,175,symclir1321)
        & stslirHWSTACK(0,symclir1367)
        & stslirMEM(0,symclir1332,symclir1337)
        & stslirMEM(0,symclir1345,symclir1347)
        & stslirMEM(0,symclir1353,symclir1349)
        & stslirMEM(0,symclir1372,symclir1375)
        & stslirMEM(0,symclir1381,symclir1377)
        & stclirREG_Flag(0,1053,symslir1318)
        & stclirREG_Flag(0,1054,symslir1323)
        & stclirREG_Flag(0,1055,0)
        & stclirREG_AddressRegister(0,14,symslir1296)
        & stclirREG_AddressRegister(0,284,symslir1354)
        & stclirREG_AddressRegister(0,285,symslir1368)
        & stclirREG_DataRegister(0,282,symslir1305)
        & stclirREG_DataRegister(0,351,symslir1313)
        & stclirREG_GuardBits(0,175,symslir1330)
        & stclirHWSTACK(0,symslir1357)
        & stclirMEM(0,symslir1302,symslir1337)
        & stclirMEM(0,symslir1353,symslir1359)
        & stclirMEM(0,symslir1365,symslir1361)
        & stclirMEM(0,symslir1374,symslir1376)
        & stclirMEM(0,symslir1382,symslir1378)).
```

Listing 10.1: Failing conjecture

Utilizing vanHelsing's debugging facilities (*-F* command line switch) creates 4 DOT graph files containing the clauses of the conjecture which can not be shown. 4 of the memory side-effects clauses can not be proven. A corresponding pair of failing clauses is shown in the collage (figure 10.8). vanHelsing has colored values which were not unified in orange, which is a great help to start with. The graphs can be zoomed (e.g. using the xdot tool[3]). Inspection of the orange node (enlarged in center of figure 10.8) shows that it is the value of a memory location, the address symbol has been unified (with symbols from the sLIR and cLIR traces). The definition of the orange symbol (enlarged at the top) reveals the symbol was initially read from the *REG_AddressRegister* function (at logical time 1 and register number 285). The graph of the corresponding failing clause (right side) is higher than the graph on the left side, which already

[3] `https://github.com/jrfonseca/xdot.py`

indicates a problem. Inspecting the marked area (enlarged at bottom) shows that the (also not unified) symbol of the right graph was initially read from an instruction field (at logical time 1 and instruction with id 25).

The problem so far seems to be that a value written into memory is read from a register (on cLIR side) but from an instruction field (on sLIR side). From the graph we learned the identifier of the involved instruction (25) and can now inspect the cause. It turns out that an immediate move and a store instruction have been combined into a bundle. The architecture allows this, as the immediate move writes to the register file a pipeline stage before the store reads its data source registers. In the witness information used to create the cLIR CASM code those two instructions end up swapped, i.e. the store operation is executed *before* the move. It therefore stores the old register contents to the memory, while the sLIR code (correctly) reads the moved immediate value. This is exactly what we learned by inspection of the failing proofs graphs. The 15 other failing proofs turned out to be caused by the very same bug.

While this bug was *just* caused by an erroneous witness information it must not have gone unnoticed by the validator, a real data dependency bug in the scheduler would show exactly the same behavior. Finding and isolating this bug demonstrates the effectiveness of our validation tools.

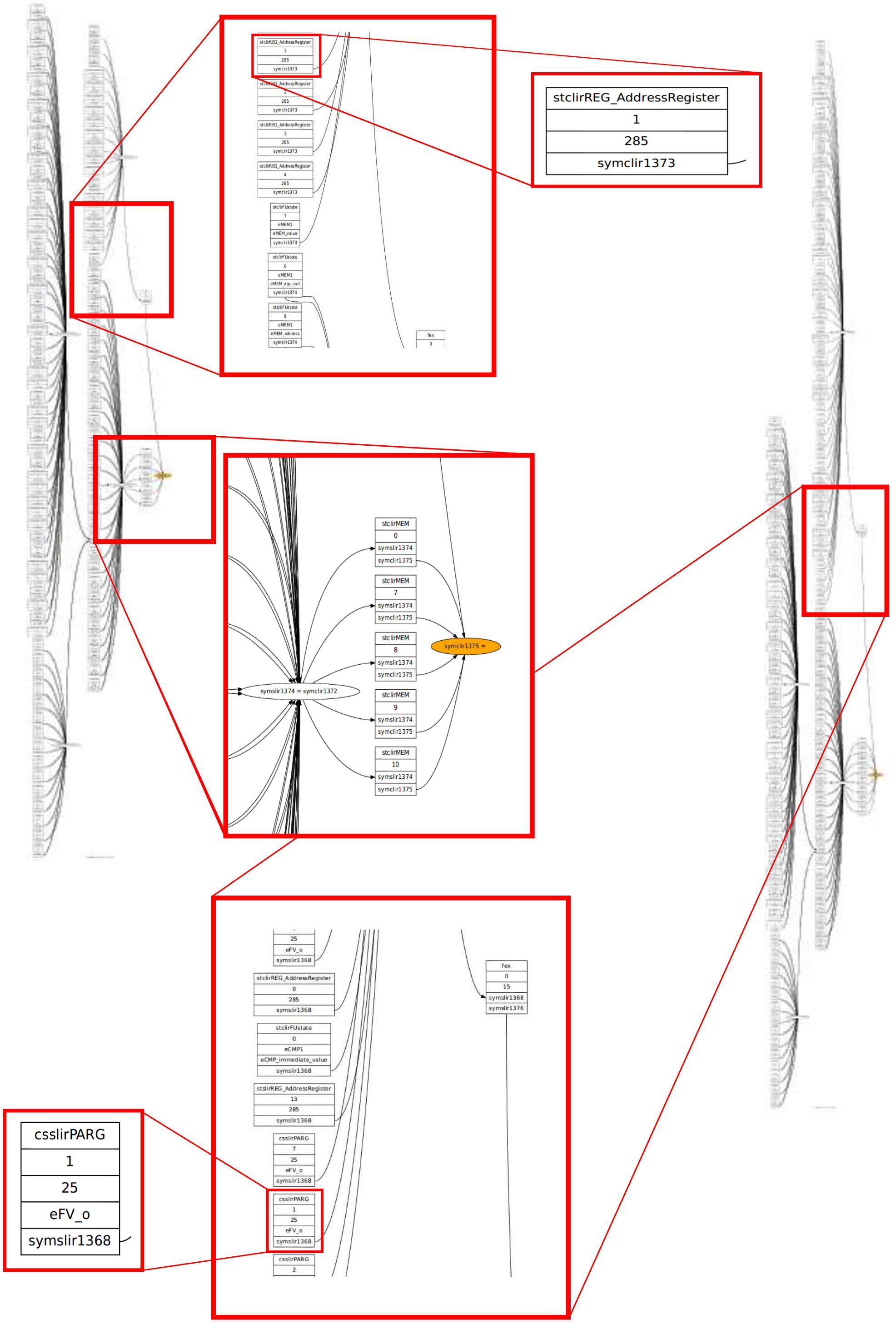


Figure 10.8: Visual debugging a VLIW Scheduling Bug

11 Future Work

Die Wissenschaft fängt eigentlich erst da an interessant zu werden, wo sie aufhört.
Justus von Liebig

This chapter sketches interesting ideas which came up but have not been further investigated in this thesis.

11.1 CASM Object Model

The CASM language currently operates on one global state. To create composable models, it would be advantageous to have objects. Rules would then operate on their object's state. An open research question however is the composition of objects, especially considering the transactional semantics of ASM. When will modification of an object's state be visible from other objects? One idea is to have a top-level rule (as in the current implementation) which orchestrates all existing objects. The global state would then be updated when the top-level rule *concludes* (returns). Following this approach objects would implement namespaces for the global state and not be independent STS. It remains to be shown whether this approach is good suited to model complex hardware or if independent STS (and a different object model or a different mechanism) is desirable.

11.2 Update Placement Optimization for the CASM Compiler

The CASM run-time needs to merge collected updates when leaving a *pseudo state*. For correctly detecting conflicting updates this merging is needed for each *pseudo state*. Most updates will never conflict, though, and in many cases this property can be analyzed statically. The idea is to initially put updates into the lowest possible *pseudo state* which would reduce the number of merged updates significantly (and thus directly boost performance as update set operations are very expensive). By placing the updates in the lowest pseudo state the size of the update set would decrease, the main reason why our CS is not performing well.

11.3 Translation Validation of the CASM Compiler

The CASM compiler performs the optimizations directly on the AST itself. Because the AST is shared between the compiler and interpreter the translation validation approach presented in this thesis could directly be applied to the CASM compiler itself. The modified rules would just need to be symbolically executed by the interpreter before and after the transformation.

11.4 Synthesization of the Compiler Specification

The approach proposed by in this thesis removes a lot of redundancy from the ADL by modeling the architecture in CASM. Because the CASM models capture the semantics of each instruction, it should be possible to derive the compiler specification (matcher tree patterns). The naïve approach would be to *guess* a tree pattern for a mMIR input and validate whether the pattern is a valid translation (i.e. perform instruction selection validation). Accumulating correctly guessed patterns would finally result in a complete compiler specification. Brandner [13] has shown that the completeness of a generated compiler specification can be shown automatically. So at least a criterion to stop the search for patterns can be given. A realistic implementation however should use a better algorithm and also the quality of the generated compiler specification must be measured. Brandner's results [14] on deriving tree patterns from a micro-instruction based ADL may be directly applicable to our ADL. As the benefit of such a synthesization a large redundant and tedious to write part of the ADL (the tree patterns) could be removed.

12 Conclusion

Everything that has a beginning has an end, Neo.
Agent Smith, The Matrix Revolutions

We have presented the CASM language, a statically typed implementation of Abstract State Machine. Due to its parallel execution semantics CASM is well-suited to specify micro-processors. This has been demonstrated by modeling a proprietary micro-processor and two variants of MIPS implementations (forwarding and bubbling). A compiler for CASM has been developed which allows synthesization of efficient Instruction Set Simulation based on the instruction set specification (up to 2.47 MHz). The highly efficient optimizer (increasing performance up to 264%) can be used to synthesize Compiled Simulation tools. Thus we were able to remove two highly redundant (ISS and CS) specifications from our research compiler's Architecture Description Language and have CASM as the *unified specification*.

Due to the formal nature of ASM the micro-processor specifications have a concise (formal) meaning. By using a set of *common semantic vocabulary* to specify compiler IRs *and* the instruction set, simulation proofs can be performed on these specifications. Technically we represent these *common semantic operations* as traces of first-order logic predicates. These traces are created by our novel method of *direct symbolic execution* (of ASM).

Based on this formal foundation a translation validation framework is defined. Each compiler pass is validated in isolation, applying a *local* correctness criterion. Certain properties of a translation must be validated during later passes (so called *constraints*). Our proposed framework validates each single pass of the compiler and makes sure that the output of the preceding pass is used as input to the succeeding one (*chain of trust*). In addition it collects the *constraints* and provides them to later passes which ultimately validate these properties.

The focus of this thesis is on the compiler back-end. We have discussed validators for all back-end passes of our research compiler and implemented prototypes for the crucial ones (instruction selection, register allocation and VLIW scheduling). All those validators operate on individual basic blocks which limits the complexity of the proofs and allows a high degree of parallelization. To evaluate the effectiveness of our approach we applied our validators to realistic industrial programs (AES implementation and bzip2) and are able to achieve convincing validation times (some minutes).

An additional contribution is the vanHelsing tool, a highly specialized first-order prover which handles the problems arising from translation validation very efficiently. A distinct feature of vanHelsing is its capability to provide graphical debugging aids which allow to quickly investigate translation error and pinpoint the issue. We have demonstrated this ability on a realistic bug by identifying the causing instructions without manual inspection of the problem file (more than 9000 lines).

We hope that the tools, techniques and results presented in this thesis act as a stimulation to launch translation validation projects for industrial compilers.

Bibliography

[1] Iso/iec jtc1/sc22/wg14. 9899:tc3: Programming languages-c. Technical report, 2007. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

[2] Kunal Agrawal, Jeremy T. Fineman, and Jim Sukha. Nested parallelism in transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 163–174, New York, NY, USA, 2008. ACM. doi:10.1145/1345206.1345232.

[3] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '83, pages 177–189, New York, NY, USA, 1983. ACM. doi:10.1145/567067.567085.

[4] Matthias Anlauff. XASM- an extensible, component-based abstract state machines language. In Yuri Gurevich, PhilippW. Kutter, Martin Odersky, and Lothar Thiele, editors, *Abstract State Machines - Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 69–90. Springer Berlin Heidelberg, 2000. doi:10.1007/3-540-44518-8_6.

[5] Jürgen Avenhaus, Jörg Denzinger, and Matthias Fuchs. Discount: A system for distributed equational deduction. In *RTA*, pages 397–402, 1995.

[6] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In *Handbook of Automated Reasoning*, pages 19–99. 2001.

[7] Mike Barnett, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Towards a tool environment for model-based testing with AsmL. In *Formal Approaches to Software Testing, FATES 2003, volume 2931 of LNCS*, pages 264–280. Springer, 2003.

[8] John Bergin and Stuart Greenfield. Teaching parameter passing by example using thunks in C and C++. *SIGCSE Bull.*, 25(1):10–14, March 1993. doi:10.1145/169073.169083.

[9] Yves Bertot, Pierre Castéran, Gérard (informaticien) Huet, and Christine Paulin-Mohring. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts in theoretical computer science. Springer, Berlin, New York, 2004. URL: `http://opac.inria.fr/record=b1101046`.

[10] Manuel Blum, Sampath Kannan, Comp Sci, and Comp Sci. Designing programs that check their work, 1989.

[11] Egon Börger and Joachim Schmid. Composition and submachine concepts for sequential ASMs. In *Computer Science Logic (Proceedings of CSL 2000), volume 1862 of LNCS*, pages 41–60. Springer-Verlag, 2000.

[12] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT - a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the international conference on Reliable software*, pages 234–245, New York, NY, USA, 1975. ACM. `doi:10.1145/800027.808445`.

[13] F. Brandner. Completeness of automatically generated instruction selectors. In *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, pages 175–182, 2010. `doi:10.1109/ASAP.2010.5540994`.

[14] Florian Brandner, Dietmar Ebner, and Andreas Krall. Compiler generation from structural architecture descriptions. In *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '07, pages 13–22, New York, NY, USA, 2007. ACM. `doi:10.1145/1289881.1289886`.

[15] Florian Brandner, Andreas Fellnhofer, Andreas Krall, and David Riegler. Fast and accurate simulation using the LLVM compiler framework. In *RAPIDO '09: 1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, 2009.

[16] Florian Brandner, Nigel Horspool, and Andreas Krall. DSP instruction set simulation. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 945–974. Springer New York, 2013. `doi:10.1007/978-1-4614-6859-2_29`.

[17] Giuseppe Del Castillo. The ASM workbench - A tool environment for computer-aided analysis and validation of abstract state machine models. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 578–581, London, UK, UK, 2001. Springer-Verlag.

[18] Lori A. Clarke. A program testing system. In *Proceedings of the 1976 annual conference*, ACM '76, pages 488–491, New York, NY, USA, 1976. ACM. `doi:10.1145/800191.805647`.

[19] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzé. Using symbolic execution for verifying safety-critical systems. *SIGSOFT Softw. Eng. Notes*, 26(5):142–151, September 2001. `doi:10.1145/503271.503230`.

[20] Maulik A. Dave. Compiler verification: A bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, November 2003. doi:10.1145/966221.966235.

[21] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] Roozbeh Farahbod. CoreASM language user manual. URL: http://coreasm.svn.sourceforge.net/viewvc/coreasm/engine-carma/trunk/doc/user_manual/CoreASM-UserManual.pdf.

[23] Roozbeh Farahbod, Vincenzo Gervasi, and Uwe Glässer. CoreASM: An extensible ASM execution engine. *Fundamenta Informaticae*, 77:1–33, 2007.

[24] Stefan Farfeleder, Andreas Krall, and Nigel Horspool. Ultra fast cycle-accurate compiled emulation of inorder pipelined architectures. *J. Syst. Archit.*, 53(8):501–510, August 2007. doi:10.1016/j.sysarc.2006.11.003.

[25] Stefan Farfeleder, Andreas Krall, Edwin Steiner, and Florian Brandner. Effective compiler generation by architecture description. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*, LCTES '06, pages 145–152, New York, NY, USA, 2006. ACM. doi:10.1145/1134650.1134671.

[26] Nicu G. Fruja and Robert F. Stärk. The hidden computation steps of Turbo Abstract State Machines. In *Abstract State Machines — Advances in Theory and Applications, 10th International Workshop, ASM 2003*, pages 244–262. Springer-Verlag, 2003.

[27] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[28] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. Model-driven language engineering: The asmeta case study. In *Proceedings of the 2008 The Third International Conference on Software Engineering Advances*, ICSEA '08, pages 373–378, Washington, DC, USA, 2008. IEEE Computer Society. doi:10.1109/ICSEA.2008.62.

[29] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 2Nd International Conference on Supercomputing*, ICS '88, pages 442–452, New York, NY, USA, 1988. ACM. doi:10.1145/55364.55407.

[30] Gerhard Goos and Wolf Zimmermann. Verifying compilers and ASMs. In *Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*, ASM '00, pages 177–202, London, UK, 2000. Springer-Verlag.

[31] Yuri Gurevich. *Evolving algebras 1993: Lipari guide*, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995.

[32] Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Logic*, 1(1):77–111, July 2000. doi:10.1145/343369.343384.

[33] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. Semantic essence of AsmL. *Theor. Comput. Sci.*, 343(3):370–412, October 2005. doi:10.1016/j.tcs.2005.06.017.

[34] Yuri Gurevich and Nikolai Tillmann. Partial updates. *Theoretical Computer Science*, 336(2):311–342, 2005.

[35] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society. doi:10.1109/WWC.2001.15.

[36] Dominik Inführ. AST interpreter for CASM. 2013. URL: http://publik.tuwien.ac.at/files/PubDat_227295.pdf.

[37] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: A survey. *ACM Trans. Des. Autom. Electron. Syst.*, 4(2):123–193, April 1999. doi:10.1145/307988.307989.

[38] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *In Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer, 2003.

[39] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. doi:10.1145/360248.360252.

[40] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES*, pages 207–220. ACM, 2009.

[41] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *CAV*, pages 1–35, 2013.

[42] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 318–328, New York, NY, USA, 1988. ACM. doi:10.1145/53990.54022.

[43] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52:107–115, 2009. doi:10.1145/1538788.1538814.

[44] Raya Leviathan and Amir Pnueli. Validating software pipelining optimizations. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '02, pages 280–287, New York, NY, USA, 2002. ACM. doi:10.1145/581630.581676.

[45] Roland Lezuo, Gergö Barany, and Andreas Krall. CASM: Implementing an Abstract State Machine based Programming Language. In Stefan Wagner and Horst Lichter, editors, *Software Engineering 2013 Workshopband, 26. Februar - 1. März 2013 in Aachen*, volume 215 of *GI Edition - Lecture Notes in Informatics*, pages 75–90, February 2013. (6. Arbeitstagung Programmiersprachen (ATPS'13)).

[46] Roland Lezuo and Andreas Krall. A unified processor model for compiler verification and simulation using ASM. In *Proceedings of the Third international conference on Abstract State Machines, Alloy, B, VDM, and Z*, ABZ'12, pages 327–330, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-30885-7_24.

[47] Roland Lezuo and Andreas Krall. Using the CASM language for simulator synthesis and model verification. In *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '13, pages 6:1–6:8, New York, NY, USA, 2013. ACM. doi:10.1145/2432516.2432522.

[48] Roland Lezuo, Philipp Paulweber, and Andreas Krall. CASM - Optimized Compilation of Abstract State Machines. LCTES '07, New York, NY, USA, 2014 (to appear). ACM.

[49] Z Manna, Anuchit Anuchitanukul, Nikolaj Bjorner, Anca Browne, Edward Chang, Michael Colon, Luca de Alfaro, Harish Devarajan, Henny Sipma, and Tomas Uribe. STeP: The Stanford Temporal Prover. Technical report, Stanford, CA, USA, 1994. URL: http://theory.stanford.edu/~zm/papers/step.ps.Z.

[50] William McCune. Otter 3.3 reference manual. *CoRR*, cs.SC/0310056, 2003.

[51] William McCune and Larry Wos. Otter - the CADE-13 competition incarnations. *Journal of Automated Reasoning*, 18:211–220, 1997. doi:10.1023/A:1005843632307.

[52] Tom Mens. A state-of-the-art survey on software merging. *Software Engineering, IEEE Transactions on*, 28(5):449–462, 2002.

[53] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 83–94, New York, NY, USA, 2000. ACM. doi:10.1145/349299.349314.

[54] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In *Handbook of Automated Reasoning*, pages 371–443. 2001.

[55] Lawrence Paulson, Tobias Nipkow, and Markus Wenzel. The isabelle reference manual. Technical report, 1995.

[56] Philipp Paulweber. Masterthesis. 2014 (to appear).

[57] A. Pnueli, M. Siegel, and F. Singerman. Translation validation. pages 151–166. Springer, 1998.

[58] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, pages 63–74, New York, NY, USA, 1994. ACM. doi:10.1145/192724.192731.

[59] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Commun.*, 15:91–110, August 2002.

[60] J. Schmid. Compiling abstract state machines to C++. *Journal of Universal Computer Science*, 7(11):1068–1087, 2001.

[61] Joachim Schmid. Introduction to AsmGofer, 2001. URL: http://www.tydo.de/AsmGofer.

[62] Stephan Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2,3):111–126, August 2002.

[63] Richard Stallman and GCC Developer Community. *GNU Compiler Collection Internals*. Free Software Foundation, 2010.

[64] G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.

[65] Geoff Sutcliffe, Stephan Schulz, Koen Claessen, and Allen Van Gelder. Using the TPTP language for writing derivations and finite interpretations. In *Proceedings of the Third international joint conference on Automated Reasoning*, IJCAR'06, pages 67–81, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11814771_7.

[66] Jürgen Teich, Philipp W. Kutter, and Ralph Weper. Description and simulation of microprocessor instruction sets using ASMs. In *Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*, ASM '00, pages 266–286, London, UK, 2000. Springer-Verlag.

[67] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In RenateA. Schmidt, editor, *Automated Deduction – CADE-22*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer Berlin Heidelberg, 2009. doi:10.1007/978-3-642-02959-2_10.

[68] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM. doi:10.1145/1993498.1993532.

[69] Han-Saem Yun and Jihong Kim. Power-aware modulo scheduling for high-performance VLIW processors. In *Proceedings of the 2001 international symposium on Low power electronics and design*, ISLPED '01, pages 40–45, New York, NY, USA, 2001. ACM. `doi:10.1145/383082.383091`.

[70] Wolf Zimmermann and Thilo Gaul. On the construction of correct compiler back-ends: An ASM approach. *Journal of Universal Computer Science*, 3:504–567, 1997.

A Common Semantic Vocabulary and Axiomatization

... has caused more and bloodier wars than anything else in the history of creation.
Douglas Adams on the poor babel fish

The following listing contains all semantic operations used to model the (integer parts only) of the MIPS instruction set and a proprietary DSP micro-processor. Each item is the name of a predicate, in braces the arguments and a short description of its semantic meaning.

- *ex (from, to, val, res)* extracts bits *from* to *to* from *val* and puts them into *res*
- *ze (old, new, val, res)* zero-extends *val* from a *old*-bit vector to a *new*-bit vector and puts into *res*.
- *se (old, new, val, res)* sign-extends *val* from a *old*-bit vector to a *new*-bit vector and puts into *res*.
- *sb (from, to, tmpl, val, res)* sets bits *from* to *to* in *tmpl* to *val* and puts results in *res*.
- *or (w, o1, o2, res)* logical or of width *w*.
- *and (w, o1, o2, res)* logical and of width *w*.
- *xor (w, o1, o2, res)* logical xor of width *w*.
- *max (w, o1, o2, res)* the maximum of the signed *w*-bit wide values *o1* and *o2* is put into *res*.
- *max_select (w, o1, o2, res) res* is 0 is *o1* is max (see above), 1 otherwise
- *min (w, o1, o2, res)* the minimum of the signed *w*-bit wide values *o1* and *o2* is put into *res*.
- *min_select (w, o1, o2, res) res* is 0 is *o1* is min (see above), 1 otherwise
- *saturate (w, ov, os, val, res)* saturates *w*-bit wide value *val* depended on overflow condition (*ov* = 1) and whether overflow was towards negative values (*os* = 1).

- *equal (w, o1, o2, res) res* = 1 if *w*-bit wide values *o1* and *o2* are equal, 0 otherwise.
- *unequal (w, o1, o2, res) res* = 1 if *w*-bit wide values *o1* and *o2* are not equal, 0 otherwise.
- *gez (w, o1, res) res* = 1 if signed *w*-bit wide value *o1* is greater or equal zero, 0 otherwise.
- *gtz (w, o1, res) res* = 1 if signed *w*-bit wide value *o1* is greater than zero, 0 otherwise.
- *lez (w, o1, res) res* = 1 if signed *w*-bit wide value *o1* is less or equal zero, 0 otherwise.
- *ltz (w, o1, res) res* = 1 if signed *w*-bit wide value *o1* is less than zero, 0 otherwise.
- *testbit (w, o1, o2, res) res* 0 1 if bit *o2* is 1 in *w*-bit wide value *o1*, 0 otherwise.
- *add_result (w, o1, o2, res) res* contains result of unsigned addition of *w*-bit wide values *o1* and *o2*.
- *add_overflow (w, o1, o2, res) res* = 1 if above addition overflows, 0 otherwise.
- *add_overflowsign (w, o1, o2, res) res* = 1 if above addition overflows towards negative values, 0 otherwise.
- *shift_result (w, r, s, val, nrbits, res) res* contains the results of a shift operation. If *r* = 1 a right shift is performed, left otherwise. A right shift may be arithmetic (*s* = 1) or logical. *val* will be shifted *nrbits* bits.
- *shift_overflow (w, r, s, val, nrbits, res) res* = 1 if above shift overflows, 0 otherwise.
- *shift_overflowsign (w, r, s, val, nrbits, res) res* = 1 if above shift overflows towards negative values, 0 otherwise.
- *rotate (w, r, val, nrbits, res) res* contains results of a *nrbits* rotation of a *w*-bit wide value *val*. If *r* = 1 the rotation direction is right, left otherwise.
- *abs_result (w, val, res) res* contains the absolute value of the signed *w*-bit wide value *val*.
- *abs_overflow (w, val, res) res* = 1 if above operation overflows, 0 otherwise.
- *negate_result (w, val, res) res* contains the negation of the signed *w*-bit wide value *val*.
- *negate_overflow (w, val, res) res* = 1 if above operation overflows, 0 otherwise.
- *not (w, val, res)* logical negation of *w*-bit wide value *val*.
- *clz (w, val, res)* counts leading zero bits of the *w*-bit wide value *val*, puts results in *res*.
- *clz (w, val, res)* counts leading one bits of the *w*-bit wide value *val*, puts results in *res*.
- *multiply_fract (w, op1, op2, res) res* contains result of fractional multiplication of *w*-bit wide values *op1* and*op2*.

- *multiply_result (w, s1, s2, op1, op2, res) res* contains the lower half of the result of a multiplication of *w*-bit wide values *op1* and *op2*. If *s1 (s2)* = 1 *op1 (op2)* is treated as signed value.

- *multiply_result_hi (w, s1, s2, op1, op2, res) res* contains the upper half of the result of a multiplication of *w*-bit wide values *op1* and *op2*. If *s1 (s2)* = 1 *op1 (op2)* is treated as signed value.

- *multiply_overflow (w, s1, s2, o1, o2, res) res* = 1 if above multiplication overflows, 0 otherwise.

- *multiply_overflowsign (w, s1, s2, o1, o2, res) res* = 1 if above multiplication overflows towards negative values, 0 otherwise.

- *divide_result (w, s1, s2, o1, o2, res) res* contains the results of the division of the *w*-bit wide value *op1* by the *w*-bit wide value *op2*. If *s1 (s2)* = 1 *op1 (op2)* is treated as signed value.

- *divide_remainder (w, s1, s2, o1, o2, res) res* contains the remainder of the above division.

- *mir_if (w, c, ai, ae, res) res* contains the value *ai* if the *w*-bit wide value *c* does not equal zero, *ae* otherwise. This predicate is used to model conditional control flow in mMIR.

The following listing contains the axiomatization of the semantic operations used in the validator prototypes.

$$\forall A0, A1, A2, X0, X1 : \\ fex(A0, A1, A2, X0) \land fex(A0, A1, A2, X1) \implies (X0 = X1)$$

$$\forall A0, A1, A2, X0, X1 : \\ fze(A0, A1, A2, X0) \land fze(A0, A1, A2, X1) \implies (X0 = X1)$$

$$\forall A0, A1, A2, X0, X1 : \\ fse(A0, A1, A2, X0) \land fse(A0, A1, A2, X1) \implies (X0 = X1)$$

$$\forall A0, A1, A2, A3, X0, X1 : \\ fsb(A0, A1, A2, A3, X0) \land fsb(A0, A1, A2, A3, X1) \implies (X0 = X1)$$

$$\forall A0, A1, A2, X0, X1 : \\ for(A0, A1, A2, X0) \land for(A0, A1, A2, X1) \implies (X0 = X1)$$

$$\forall A0, A1, A2, X0, X1 : \quad fand(A0, A1, A2, X0) \land fand(A0, A1, A2, X1) \implies (X0 = X1)$$

$$\forall A0, A1, A2, X0, X1 : \quad fxor(A0, A1, A2, X0) \land fxor(A0, A1, A2, X1) \implies (X0 = X1)$$

$$\forall A0, A1, A2, X0, X1 : \quad fmax(A0, A1, A2, X0) \land fmax(A0, A1, A2, X1) \implies (X0 = X1)$$

$$\forall A0, A1, A2, X0, X1 : \quad fmax\_select(A0, A1, A2, X0) \land fmax\_select(A0, A1, A2, X1) \implies (X0 = X1)$$

$$\forall A0, A1, A2, X0, X1 : \quad fmin(A0, A1, A2, X0) \land fmin(A0, A1, A2, X1) \implies (X0 = X1)$$

$$\forall A0, A1, A2, X0, X1 : \quad fmin\_select(A0, A1, A2, X0) \land fmin\_select(A0, A1, A2, X1) \implies (X0 = X1)$$

$$\forall A0, A1, A2, A3, X0, X1 : \quad fsaturate(A0, A1, A2, A3, X0) \land fsaturate(A0, A1, A2, A3, X1) \implies (X0 = X1)$$

$$\forall A0, A1, A2, X0, X1 : \quad fequal(A0, A1, A2, X0) \land fequal(A0, A1, A2, X1) \implies (X0 = X1)$$

$$\forall A0, A1, A2, X0, X1 : \quad funequal(A0, A1, A2, X0) \land funequal(A0, A1, A2, X1) \implies (X0 = X1)$$

$$\forall A0, A1, X0, X1 : \quad fgez(A0, A1, X0) \land fgez(A0, A1, X1) \implies (X0 = X1)$$

$$\forall A0, A1, X0, X1 :$$

$$fgtz(A0, A1, X0) \wedge fgtz(A0, A1, X1) \implies (X0 = X1)$$

$$\forall A0, A1, X0, X1 :$$

$$flez(A0, A1, X0) \wedge flez(A0, A1, X1) \implies (X0 = X1)$$

$$\forall A0, A1, X0, X1 :$$

$$fltz(A0, A1, X0) \wedge fltz(A0, A1, X1) \implies (X0 = X1)$$

$$\forall A0, A1, X0, X1 :$$

$$ftestbit(A0, A1, X0) \wedge ftestbit(A0, A1, X1) \implies (X0 = X1)$$

$$\forall A0, A1, A2, X0, X1 :$$

$$fadd\_result(A0, A1, A2, X0) \wedge fadd\_result(A0, A1, A2, X1)$$
$$\implies (X0 = X1)$$

$$\forall A0, A1, A2, X0, X1 :$$

$$fadd\_overflow(A0, A1, A2, X0) \wedge fadd\_overflow(A0, A1, A2, X1)$$
$$\implies (X0 = X1)$$

$$\forall A0, A1, A2, X0, X1 :$$

$$fadd\_overflowsign(A0, A1, A2, X0) \wedge$$
$$fadd\_overflowsign(A0, A1, A2, X1 \implies (X0 = X1)$$

$$\forall A0, A1, A2, X0, X1 :$$

$$fsub\_result(A0, A1, A2, X0) \wedge fsub\_result(A0, A1, A2, X1)$$
$$\implies (X0 = X1)$$

$$\forall A0, A1, A2, A3, A4, X0, X1 :$$

$$fshift\_result(A0, A1, A2, A3, A4, X0) \wedge$$
$$fshift\_result(A0, A1, A2, A3, A4, X1) \implies (X0 = X1)$$

$$\forall A0, A1, A2, A3, A4, X0, X1 :$$

$$fshift\_overflow(A0, A1, A2, A3, A4, X0) \wedge$$
$$fshift\_overflow(A0, A1, A2, A3, A4, X1) \implies (X0 = X1)$$

$$\forall A0, A1, A2, A3, A4, X0, X1 : \\ fshift\_overflowsign(A0, A1, A2, A3, A4, X0) \wedge \\ fshift\_overflowsign(A0, A1, A2, A3, A4, X1) \implies (X0 = X1)$$

$$\forall A0, A1, A2, A3, X0, X1 : \\ frotate(A0, A1, A2, A3, X0) \wedge frotate(A0, A1, A2, A3, X1) \\ \implies (X0 = X1)$$

$$\forall A0, A1, A2, X0, X1 : \\ fabs\_result(A0, A1, X0) \wedge fabs\_result(A0, A1, X1) \implies (X0 = X1)$$

$$\forall A0, A1, A2, X0, X1 : \\ fabs\_overflow(A0, A1, X0) \wedge fabs\_overflow(A0, A1, X1) \\ \implies (X0 = X1)$$

$$\forall A0, A1, A2, X0, X1 : \\ fabs\_diff(A0, A1, A2, X0) \wedge fabs\_diff(A0, A1, A2, X1) \\ \implies (X0 = X1)$$

$$\forall A0, A1, X0, X1 : \\ fnegate\_result(A0, A1, X0) \wedge fnegate\_result(A0, A1, X1) \\ \implies (X0 = X1)$$

$$\forall A0, A1, X0, X1 : \\ fnegate\_overflow(A0, A1, X0) \wedge fnegate\_overflow(A0, A1, X1) \\ \implies (X0 = X1)$$

$$\forall A0, A1, X0, X1 : \\ fnot(A0, A1, X0) \wedge fnot(A0, A1, X1) \implies (X0 = X1)$$

$$\forall A0, A1, X0, X1 : \\ fclz(A0, A1, X0) \wedge fclz(A0, A1, X1) \implies (X0 = X1)$$

$$\forall A0, A1, X0, X1 : \\ fclo(A0, A1, X0) \wedge fclo(A0, A1, X1) \implies (X0 = X1)$$

$$\begin{aligned}&\forall A0, A1, A2, X0, X1 :\\ &\qquad fmultiply\_fract(A0, A1, A2, X0) \wedge\\ &\qquad\quad fmultiply\_fract(A0, A1, A2, X1) \implies (X0 = X1)\end{aligned}$$

$$\begin{aligned}&\forall A0, A1, A2, A3, A4, X0, X1 :\\ &\qquad fmultiply\_result(A0, A1, A2, A3, A4, X0) \wedge\\ &\qquad\quad fmultiply\_result(A0, A1, A2, A3, A4, X1) \implies (X0 = X1)\end{aligned}$$

$$\begin{aligned}&\forall A0, A1, A2, A3, A4, X0, X1 :\\ &\qquad fmultiply\_result\_hi(A0, A1, A2, A3, A4, X0) \wedge\\ &\qquad\quad fmultiply\_result\_hi(A0, A1, A2, A3, A4, X1) \implies (X0 = X1)\end{aligned}$$

$$\begin{aligned}&\forall A0, A1, A2, A3, A4, X0, X1 :\\ &\qquad fmultiply\_overflow(A0, A1, A2, A3, A4, X0) \wedge\\ &\qquad\quad fmultiply\_overflow(A0, A1, A2, A3, A4, X1) \implies (X0 = X1)\end{aligned}$$

$$\begin{aligned}&\forall A0, A1, A2, A3, A4, X0, X1 :\\ &\qquad fmultiply\_overflowsign(A0, A1, A2, A3, A4, X0) \wedge\\ &\qquad\quad fmultiply\_overflowsign(A0, A1, A2, A3, A4, X1) \implies (X0 = X1)\end{aligned}$$

$$\begin{aligned}&\forall A0, A1, A2, A3, A4, X0, X1 :\\ &\qquad fdivide\_result(A0, A1, A2, A3, A4, X0) \wedge\\ &\qquad\quad fdivide\_result(A0, A1, A2, A3, A4, X1) \implies (X0 = X1)\end{aligned}$$

$$\begin{aligned}&\forall A0, A1, A2, A3, A4, X0, X1 :\\ &\qquad fdivide\_remainder(A0, A1, A2, A3, A4, X0) \wedge\\ &\qquad\quad fdivide\_remainder(A0, A1, A2, A3, A4, X1) \implies (X0 = X1)\end{aligned}$$

$$\begin{aligned}&\forall A0, A1, A2, A3, X0, X1 :\\ &\qquad fmir\_if(A0, A1, A2, A3, X0) \wedge fmir\_if(A0, A1, A2, A3, X1)\\ &\qquad\quad \implies (X0 = X1)\end{aligned}$$

$$\begin{aligned}&\forall A0, A1, A2, X0, X1 :\\ &\qquad fbinop(A0, A1, A2, X0) \wedge fbinop(A0, A1, A2, X1) \implies (X0 = X1)\end{aligned}$$

$$\forall A0, A1, X0, X1 : \quad fcons(A0, A1, X0) \wedge fcons(A0, A1, X1) \implies (X0 = X1)$$

$$\forall A0, A1, X0, X1 : \quad fpush(A0, A1, X0) \wedge fpush(A0, A1, X1) \implies (X0 = X1)$$

$$\forall A0, Y1, Y2, X1, X2 : \quad fpop(A0, X1, X2) \wedge fpop(A0, Y1, Y2)) \implies ((X1 = Y1) \wedge (X2 = Y2))$$

$$\forall A, A1, A2, D, E : \quad fex(0, 15, A, A1) \wedge fex(16, 31, A, A2) \wedge fsb(16, 31, 0, A2, D) \wedge fsb(0, 15, D, A1, E) \implies (A = E)$$

$$\forall A, W1, W2, B, W3 : \quad fsb(0, 7, 0, A, W1) \wedge fsb(8, 15, W1, B, W2) \wedge fsb(0, 15, 0, W2, W3) \implies (W2 = W3)$$

$$\forall A, B, C : \quad fse(16, 32, A, B) \wedge fex(0, 19, B, C) \implies (A = C)$$

$$\forall A, B, C : \quad fse(16, 32, A, B) \wedge fex(0, 31, B, C) \implies (A = C)$$

$$\forall A, W, X : \quad fadd\_result(W, A, 0, X) \implies (X = A)$$

$$\forall W, A, B, X1, X2 : \quad fadd\_result(W, A, B, X1) \wedge fadd\_result(W, X1, 0, X2) \implies (X1 = X2)$$

$$\forall W, X, Y, Z1, Z2 : \quad fadd\_result(W, X, 2, Y) \wedge fadd\_result(W, Y, 1, Z1) \wedge fadd\_result(W, X, 3, Z2) \implies (Z1 = Z2)$$

$$\forall W, A, X \quad : \quad fsub\_result(W, A, 0, X) \quad \implies \quad (X \quad = \quad A)$$

$$\forall W, A, B, X1, X2 : \quad fxor(W, A, B, X1) \wedge fxor(W, B, A, X2) \implies (X1 = X2)$$

$$\forall A, B, C, D, E, F : \quad fadd\_result(20, A, B, C) \wedge fex(0, 15, C, D) \wedge fadd\_result(20, A, B, E) \wedge fex(0, 15, E, F) \implies (D = F)$$

$$\forall W, A, B, X1, X2 : \quad fadd\_result(W, A, B, X1) \wedge fadd\_result(W, B, A, X2) \implies (X1 = X2)$$

$$\forall B, A \quad : \quad fequal(A, B, 0, 1) \quad \implies \quad (B \quad = \quad 0)$$

$$\forall B, A \quad : \quad fequal(A, B, 0, 0) \quad \implies \quad (B \quad \neq \quad 0)$$

$$\forall B, A \quad : \quad funequal(A, B, 0, 1) \quad \implies \quad (B \quad \neq \quad 0)$$

$$\forall B, A \quad : \quad funequal(A, B, 0, 0) \quad \implies \quad (B \quad = \quad 0)$$

$$\forall A, C, D, E \quad : \quad fmir\_if(A, 0, C, D, E) \quad \implies \quad (D \quad = \quad E)$$

$$\forall A, C, D, E \quad : \quad fmir\_if(A, 1, C, D, E) \quad \implies \quad (C \quad = \quad E)$$

$$\forall W, X, Xn, Z : \quad fnot(W, X, Xn) \wedge fadd\_result(W, 1, Xn, Z) \implies fnegate\_result(W, X, Z)$$

$$\forall A, X, Y \quad : \quad fze(A, A, X, Y) \quad \implies \quad (X \quad = \quad Y)$$

$$\forall A, X, Y \quad : \quad fse(A, A, X, Y) \quad \implies \quad (X \quad = \quad Y)$$

B The CASM Language

A EBNF grammar definition of the CASM language.

⟨*specification*⟩ ::= ⟨*header*⟩ ⟨*body_elements*⟩

⟨*header*⟩ ::= **’casm’** ⟨*identifier*⟩

⟨*body_elements*⟩ ::= ⟨*body_elements*⟩ ⟨*body_element*⟩
| ⟨*body_element*⟩

⟨*body_element*⟩ ::= ⟨*provider*⟩
| ⟨*enum*⟩
| ⟨*function_definition*⟩
| ⟨*derived*⟩
| ⟨*init*⟩
| ⟨*rule*⟩

⟨*init*⟩ ::= **’init’** ⟨*identifier*⟩

⟨*provider*⟩ ::= **’provider’** ⟨*identifier*⟩

⟨*enum*⟩ ::= **’enum’** ⟨*identifier*⟩ **’=’** **’{’** ⟨*identifier_list*⟩ **’}’**

⟨*derived*⟩ ::= **’derived’** ⟨*identifier*⟩ **’(’** ⟨*param_list*⟩ **’)’** **’=’** ⟨*expression*⟩
| **’derived’** ⟨*identifier*⟩ **’(’** ⟨*param_list*⟩ **’)’** **’ :’** ⟨*type*⟩ **’=’** ⟨*expression*⟩
| **’derived’** ⟨*identifier*⟩ **’=’** ⟨*expression*⟩
| **’derived’** ⟨*identifier*⟩ **’ :’** ⟨*type*⟩ **’=’** ⟨*expression*⟩
| **’derived’** ⟨*identifier*⟩ **’(’** **’)’** **’=’** ⟨*expression*⟩
| **’derived’** ⟨*identifier*⟩ **’(’** **’)’** **’ :’** ⟨*type*⟩ **’=’** ⟨*expression*⟩

⟨*function_definition*⟩ ::= **’function’** **’(’** ⟨*identifier_list*⟩ **’)’** ⟨*identifier*⟩ ⟨*function_signature*⟩ ⟨*initializers*⟩
| **’function’** **’(’** ⟨*identifier_list*⟩ **’)’** ⟨*identifier*⟩ ⟨*function_signature*⟩
| **’function’** ⟨*identifier*⟩ ⟨*function_signature*⟩ ⟨*initializers*⟩
| **’function’** ⟨*identifier*⟩ ⟨*function_signature*⟩

⟨*identifier_list*⟩ ::= ⟨*identifier*⟩ **’,’** ⟨*identifier_list*⟩
| ⟨*identifier*⟩
| ⟨*identifier*⟩ **’,’**

⟨*function_signature*⟩ ::= **’:’** **’->’** ⟨*type*⟩
| **’ :’** ⟨*type_identifier_starlist*⟩ **’->’** ⟨*type*⟩

⟨*param*⟩ ::= ⟨*identifier*⟩ **’:’** ⟨*type*⟩
| ⟨*identifier*⟩

⟨*param_list*⟩ ::= ⟨*param*⟩ **’,’** ⟨*param_list*⟩
| ⟨*param*⟩ **’,’**
| ⟨*param*⟩

⟨*type_identifier_starlist*⟩ ::= ⟨*type*⟩ **’*’** ⟨*type_identifier_starlist*⟩
| ⟨*type*⟩

⟨*type*⟩ ::= ⟨*identifier*⟩
| ⟨*identifier*⟩ **’(’** ⟨*type_list*⟩ **’)’**
| ⟨*identifier*⟩ **’(’** ⟨*number*⟩ **’..’** ⟨*number*⟩ **’)’**

⟨*type_list*⟩ ::= ⟨*type*⟩ **’,’** ⟨*type_list*⟩
| ⟨*type*⟩ **’,’**
| ⟨*type*⟩

⟨*tuple_list*⟩ ::= ⟨*identifier*⟩ **’,’** ⟨*tuple_list*⟩
| ⟨*identifier*⟩ **’,’**
| ⟨*identifier*⟩

⟨*initializers*⟩ ::= ⟨*initially*⟩ **’{’** ⟨*initializer_list*⟩ **’}’**
| ⟨*initially*⟩ **’{’** **’}’**

⟨*initializer_list*⟩ ::= ⟨*initializer_list*⟩ **’,’** ⟨*initializer*⟩
| ⟨*initializer_list*⟩ **’,’**
| ⟨*initializer*⟩

⟨*initializer*⟩ ::= ⟨*atom*⟩
| ⟨*atom*⟩ **’->’** ⟨*atom*⟩

⟨*atom*⟩ ::= ⟨*function*⟩
| ⟨*value*⟩
| ⟨*bracket_expression*⟩

⟨*value*⟩ ::= ⟨*ruleref*⟩
| ⟨*number*⟩
| ⟨*strconst*⟩
| ⟨*listconst*⟩
| ⟨*number_range*⟩
| **’self’**
| **’undef’**
| **’true’**
| **’false’**

⟨*number*⟩ ::= **’+’** ⟨*intconst*⟩
| **’-’** ⟨*intconst*⟩
| ⟨*intconst*⟩
| **’+’** ⟨*floatconst*⟩
| **’-’** ⟨*floatconst*⟩
| ⟨*floatconst*⟩
| **’+’** ⟨*rationalconst*⟩
| **’-’** ⟨*rationalconst*⟩
| ⟨*rationalconst*⟩

⟨*ruleref*⟩ ::= **’@’** ⟨*identifier*⟩

⟨*number_range*⟩ ::= **’[’** ⟨*number*⟩ **’..’** ⟨*number*⟩ **’]’**
| **’[’** ⟨*identifier*⟩ **’..’** ⟨*identifier*⟩ **’]’**

⟨*listconst*⟩ ::= **’[’** ⟨*expression_list*⟩ **’]’**
| **’[’ ’]’**

⟨*expression_list*⟩ ::= ⟨*expression*⟩ **’,’** ⟨*expression_list*⟩
| ⟨*expression*⟩ **’,’**
| ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*expression*⟩ **’+’** ⟨*expression*⟩
| ⟨*expression*⟩ **’-’** ⟨*expression*⟩
| ⟨*expression*⟩ **’!=’** ⟨*expression*⟩
| ⟨*expression*⟩ **’=’** ⟨*expression*⟩
| ⟨*expression*⟩ **’<’** ⟨*expression*⟩
| ⟨*expression*⟩ **’>’** ⟨*expression*⟩
| ⟨*expression*⟩ **’<=’** ⟨*expression*⟩

```
  | ⟨expression⟩ '>=' ⟨expression⟩
  | ⟨expression⟩ '*' ⟨expression⟩
  | ⟨expression⟩ '/' ⟨expression⟩
  | ⟨expression⟩ '%' ⟨expression⟩
  | ⟨expression⟩ 'div' ⟨expression⟩
  | ⟨expression⟩ 'or' ⟨expression⟩
  | ⟨expression⟩ 'xor' ⟨expression⟩
  | ⟨expression⟩ 'and' ⟨expression⟩
  | 'not' ⟨expression⟩
  | ⟨atom⟩

⟨bracket_expression⟩ ::= '(' ⟨expression⟩ ')'

⟨function⟩ ::= ⟨identifier⟩
  | ⟨identifier⟩ '(' ')'
  | ⟨identifier⟩ '(' ⟨expression_list⟩ ')'

⟨rule⟩ ::= 'rule' ⟨identifier⟩ '=' ⟨statement⟩
  | 'rule' ⟨identifier⟩ '(' ')' '=' ⟨statement⟩
  | 'rule' ⟨identifier⟩ '(' ⟨param_list⟩ ')' '=' ⟨statement⟩
  | 'rule' ⟨identifier⟩ 'dumps' ⟨dumpspec_list⟩ '=' ⟨statement⟩
  | 'rule' ⟨identifier⟩ '(' ')' 'dumps' ⟨dumpspec_list⟩ '=' ⟨statement⟩
  | 'rule' ⟨identifier⟩ '(' ⟨param_list⟩ ')' 'dumps' ⟨dumpspec_list⟩ '=' ⟨statement⟩

⟨dumpspec_list⟩ ::= ⟨dumpspec⟩ ',' ⟨dumpspec_list⟩
  | ⟨dumpspec⟩

⟨dumpspec⟩ ::= '(' ⟨identifier_list⟩ ')' '->' ⟨identifier⟩

⟨statement⟩ ::= ⟨assert⟩
  | ⟨assure⟩
  | ⟨diedie⟩
  | ⟨impossible⟩
  | ⟨debuginfo⟩
  | ⟨print⟩
  | ⟨update⟩
  | ⟨case⟩
  | ⟨call⟩
  | ⟨kw_seqblock⟩
  | ⟨seqblock⟩
  | ⟨kw_parblock⟩
  | ⟨parblock⟩
  | ⟨ifthenelse⟩
  | ⟨let⟩
  | ⟨push⟩
```

| ⟨*pop*⟩
| ⟨*forall*⟩
| ⟨*iterate*⟩
| **'skip'**
| ⟨*identifier*⟩
| **'objdump'** '(' ⟨*identifier*⟩ ')'

⟨*assert*⟩ ::= **'assert'** ⟨*expression*⟩

⟨*assure*⟩ ::= **'assure'** ⟨*expression*⟩

⟨*diedie*⟩ ::= **'diedie'**
| **'diedie'** ⟨*expression*⟩

⟨*impossible*⟩ ::= **'impossibe'**

⟨*debuginfo*⟩ ::= **'debuginfo'** ⟨*identifier*⟩ ⟨*debug_atom_list*⟩

⟨*debug_atom_list*⟩ ::= ⟨*atom*⟩ **'+'** ⟨*debug_atom_list*⟩
| ⟨*atom*⟩

⟨*print*⟩ ::= **'print'** ⟨*debug_atom_list*⟩

⟨*update*⟩ ::= ⟨*function*⟩ **':='** ⟨*expression*⟩

⟨*case*⟩ ::= **'case'** ⟨*expression*⟩ **'of'** ⟨*case_label_list*⟩ **'endcase'**

⟨*case_label_list*⟩ ::= ⟨*case_label*⟩ ⟨*case_label_list*⟩
| ⟨*case_label*⟩

⟨*case_label*⟩ ::= ⟨*case_label_default*⟩
| ⟨*case_label_number*⟩
| ⟨*case_label_ident*⟩
| ⟨*case_label_string*⟩

⟨*case_label_default*⟩ ::= default **':'** ⟨*statement*⟩

⟨*case_label_number*⟩ ::= ⟨*number*⟩ **':'** ⟨*statement*⟩

⟨*case_label_ident*⟩ ::= ⟨*identifier*⟩ **':'** ⟨*statement*⟩

⟨*case_label_string*⟩ ::= ⟨*strconst*⟩ **':'** ⟨*statement*⟩

⟨*call*⟩ ::= **'call'** '(' ⟨*expression*⟩ ')' '(' ⟨*expression_list*⟩ ')'
| **'call'** '(' ⟨*expression*⟩ ')'
| **'call'** ⟨*identifier*⟩ '(' ⟨*expression_list*⟩ ')'
| **'call'** ⟨*identifier*⟩

⟨*kw_seqblock*⟩ ::= **’seqblock’** ⟨*statements*⟩ **’endseqblock’**

⟨*seqblock*⟩ ::= **’{|’** ⟨*statements*⟩ **’|}’**

⟨*kw_parblock*⟩ ::= **’parblock’** ⟨*statements*⟩ **’endparblock’**

⟨*parblock*⟩ ::= **’{’** ⟨*statements*⟩ **’}’**

⟨*statements*⟩ ::= ⟨*statements*⟩ ⟨*statement*⟩
| ⟨*statement*⟩

⟨*ifthenelse*⟩ ::= **’if’** ⟨*expression*⟩ **’then’** ⟨*statement*⟩
| **’if’** ⟨*expression*⟩ **’then’** ⟨*statement*⟩ **’else’** ⟨*statement*⟩

⟨*let*⟩ ::= **’let’** ⟨*identifier*⟩ **’=’** ⟨*expression*⟩ **’in’** ⟨*statement*⟩
| **’let’** ⟨*identifier*⟩ **’:’** ⟨*type*⟩ **’=’** ⟨*expression*⟩ **’in’** ⟨*statement*⟩

⟨*push*⟩ ::= **’push’** ⟨*expression*⟩ **’into’** ⟨*identifier*⟩

⟨*pop*⟩ ::= **’pop’** ⟨*identifier*⟩ **’from’** ⟨*identifier*⟩

⟨*forall*⟩ ::= **’forall’** ⟨*identifier*⟩ **’in’** ⟨*expression*⟩ **’do’** ⟨*statement*⟩

⟨*iterate*⟩ ::= **’iterate’** ⟨*statement*⟩

C vanHelsing Input Language

A EBNF grammar definition of the vanHelsing input language.

⟨*top_level*⟩ ::= ⟨*tptp_file*⟩
| ϵ

⟨*tptp_file*⟩ ::= ⟨*tptp_file*⟩ ⟨*tptp_input*⟩
| ⟨*tptp_input*⟩

⟨*tptp_input*⟩ ::= ⟨*tff_annotated*⟩
| ⟨*fof_annotated*⟩

⟨*name*⟩ ::= ⟨*atomic_word*⟩
| ⟨*integer*⟩

⟨*tff_annotated*⟩ ::= **’tff’ ’(’** ⟨*name*⟩ **’,’** ⟨*formulae_role*⟩ **’,’** ⟨*tff_formula*⟩ **’)’ ’.’**

⟨*tff_formula*⟩ ::= ⟨*tff_typed_atom*⟩

⟨*tff_typed_atom*⟩ ::= ⟨*tff_untyped_atom*⟩ **’:’** ⟨*tff_top_level_type*⟩
| **’(’** ⟨*tff_typed_atom*⟩ **’)’**

⟨*tff_untyped_atom*⟩ ::= ⟨*functor*⟩
| ⟨*system_functor*⟩

⟨*tff_top_level_type*⟩ ::= ⟨*tff_atomic_type*⟩
| ⟨*tff_mapping_type*⟩
| **’(’** ⟨*tff_top_level_type*⟩ **’)’**

⟨*tff_atomic_type*⟩ ::= ⟨*atomic_word*⟩
| ⟨*defined_type*⟩

```
  | ⟨atomic_word⟩ ’(’ ⟨tff_type_arguments⟩ ’)’
  | ⟨variable⟩

⟨tff_type_arguments⟩ ::= ⟨tff_atomic_type⟩
  | ⟨tff_type_arguments⟩ ⟨tff_atomic_type⟩

⟨tff_mapping_type⟩ ::= ⟨tff_unitary_type⟩ ’>’ ⟨tff_atomic_type⟩

⟨tff_unitary_type⟩ ::= ⟨tff_atomic_type⟩
  | ’(’ ⟨tff_xprod_type⟩ ’)’

⟨tff_xprod_type⟩ ::= ⟨tff_unitary_type⟩ ’*’ ⟨tff_atomic_type⟩
  | ⟨tff_xprod_type⟩ ’*’ ⟨tff_atomic_type⟩

⟨fof_annotated⟩ ::= ’fof’ ’(’ ⟨name⟩ ’,’ ⟨formulae_role⟩ ’,’ ⟨fof_formula⟩ ’)’ ’.’

⟨fof_formula⟩ ::= ⟨fof_logic_formula⟩

⟨fof_logic_formula⟩ ::= ⟨fof_binary_formula⟩
  | ⟨fof_unitary_formula⟩

⟨fof_binary_formula⟩ ::= ⟨fof_binary_nonassoc⟩
  | ⟨fof_binary_assoc⟩

⟨fof_binary_nonassoc⟩ ::= ⟨fof_unitary_formula⟩ ⟨binary_connective⟩ ⟨fof_unitary_formula⟩

⟨fof_binary_assoc⟩ ::= ⟨fof_or_formula⟩
  | ⟨fof_and_formula⟩

⟨fof_or_formula⟩ ::= ⟨fof_unitary_formula⟩ ’|’ ⟨fof_unitary_formula⟩
  | ⟨fof_or_formula⟩ ’|’ ⟨fof_unitary_formula⟩

⟨fof_and_formula⟩ ::= ⟨fof_unitary_formula⟩ ’&’ ⟨fof_unitary_formula⟩
  | ⟨fof_and_formula⟩ ’&’ ⟨fof_unitary_formula⟩

⟨fof_unitary_formula⟩ ::= ⟨fof_quantified_formula⟩
  | ⟨fof_unary_formula⟩
  | ⟨atomic_formula⟩
  | ’(’ ⟨fof_logic_formula⟩ ’)’

⟨fof_quantified_formula⟩ ::= ⟨fol_quantifier⟩ ’[’ ⟨fof_variable_list⟩ ’]’ ’:’ ⟨fof_unitary_formula⟩

⟨fof_variable_list⟩ ::= ⟨variable⟩
  | ⟨fof_variable_list⟩ ’,’ ⟨variable⟩

⟨fof_unary_formula⟩ ::= ’~’ ⟨fof_unitary_formula⟩
  | ⟨fol_infix_unary⟩
```

⟨*fol_infix_unary*⟩ ::= ⟨*term*⟩ **’!=’** ⟨*term*⟩
| ⟨*term*⟩ **’=’** ⟨*term*⟩

⟨*fol_quantifier*⟩ ::= **’!’**
| **’?’**

⟨*term*⟩ ::= ⟨*function_term*⟩
| ⟨*variable*⟩

⟨*function_term*⟩ ::= ⟨*plain_term*⟩
| ⟨*defined_term*⟩

⟨*plain_term*⟩ ::= ⟨*constant*⟩
| ⟨*functor*⟩ **’(’** ⟨*arguments*⟩ **’)’**

⟨*constant*⟩ ::= ⟨*functor*⟩

⟨*defined_term*⟩ ::= ⟨*defined_atom*⟩
| ⟨*defined_atomic_term*⟩

⟨*defined_atom*⟩ ::= ⟨*number*⟩
| ⟨*distinct_object*⟩

⟨*number*⟩ ::= ⟨*integer*⟩

⟨*defined_plain_term*⟩ ::= ⟨*defined_constant*⟩
| ⟨*defined_functor*⟩ **’(’** ⟨*arguments*⟩ **’)’**

⟨*defined_constant*⟩ ::= ⟨*defined_functor*⟩

⟨*defined_functor*⟩ ::= ⟨*atomic_defined_word*⟩

⟨*defined_atomic_term*⟩ ::= ⟨*defined_plain_term*⟩

⟨*defined_atomic_formula*⟩ ::= ⟨*defined_plain_formula*⟩

⟨*defined_plain_formula*⟩ ::= ⟨*defined_plain_term*⟩

⟨*defined_infix_formula*⟩ ::= ⟨*term*⟩ ⟨*defined_infix_pred*⟩ ⟨*term*⟩

⟨*defined_infix_pred*⟩ ::= **’=’**

⟨*atomic_formula*⟩ ::= ⟨*plain_atomic_formula*⟩
| ⟨*defined_atomic_formula*⟩

⟨*atomic_defined_word*⟩ ::= ⟨*dollar_word*⟩

⟨*plain_atomic_formula*⟩ ::= ⟨*plain_term*⟩

⟨*distinct_object*⟩ ::= ⟨*double_quoted*⟩

⟨*arguments*⟩ ::= ⟨*term*⟩
| ⟨*arguments*⟩ **’,’** ⟨*term*⟩

⟨*binary_connective*⟩ ::= **’=’**
| **’=>’**
| **’<=’**
| **’<~>’**
| **’~|’**
| **’~&’**

⟨*functor*⟩ ::= ⟨*atomic_word*⟩

⟨*atomic_word*⟩ ::= ⟨*single_quoted*⟩
| ⟨*lower_word*⟩

⟨*variable*⟩ ::= ⟨*upper_word*⟩
| ⟨*upper_word*⟩ **’:’** ⟨*dollar_word*⟩

⟨*defined_type*⟩ ::= ⟨*dollar_word*⟩

⟨*system_functor*⟩ ::= ⟨*atomic_system_word*⟩

⟨*atomic_system_word*⟩ ::= ⟨*dollar_dollar_word*⟩

⟨*formulae_role*⟩ ::= ⟨*lower_word*⟩

D Colophon

Image Copyrights

The chapter heading image for chapter 2 is used with the friendly permission of Marc Hennekes, the author of `http://www.mandree.de/auf-den-schultern/`. The chapter heading image for chapter 8 (*Satanic Rites of Dracula*) is licensed under CC-BY-3.0 by ONTV from `http://ontv.deviantart.com/art/Satanic-Rites-of-Dracula-100391613`. The chapter heading image for chapter 3 has been created by the author and Phillip Paulweber. The other images are public domain (from wikipedia.org for chapters 1 and 6 and from openclipart.org for chapter 9).

Version Information

Official Discordian Document Number: *f870053 Di 18. Mär 13:28:22 CET 2014*

E Curriculum Vitae

Personal Data

Name	Roland Lezuo
Date of Birth	18-10-1979
Place of Birth	Innsbruck, Austria
Address	Burggasse 35/1, 1070 Wien
Email	roland.lezuo@tuwien.ac.at

Professional Experience

2001 - 2003	Cura Marketing GmbH, Innsbruck *Web Development*
2004 - 2006	RISE Schwechat, Schwechat *Firmware Development, Austrian eHealth set-top boxes* *Software Development, Austrian Citizen Card*
2008 - 2010	Secure Business Austria, Wien *Firmware Development, Smartcard Reader* *Software Architect, German eHealth set-top boxes* *Firmware Development, Road Tolling Equipment* *Software Development, Austrian eHealth*
2010 - 2013	Vienna University of Technology *Project Assistant, Correct Compilers for Correct Application Specific Processors (C3Pro)*

Education

1994 - 1999	HTL (Higher Technical Institute for electrical engineering), Innsbruck *Reifeprüfung and Diploma with excellent success*
1999 - 2002	Computer Science, Fernuniversität in Hagen *„Diplomvorprüfung II“ passed on 25.3.2002*
2002 - 2004	Bachelor’s programme Computer Engineering, TU Wien
2004 - 2007	Master’s programme Computer Engineering, TU Wien *Diploma Thesis: "Porting the CACAO Virtual Machine to POWERPC64 and Coldfire", with distinction*
2010 - current	Doctoral programme in Engineering Sciences , TU Wien *PhD Thesis: Scalable Translation Validation*

List of Publications

- Roland Lezuo and Felix Breitenecker. A Programmed Solution to ARGESIM Comparison C 6 ‘Emergency Department’ with DSOL, a Java- based Suite. In *Simulation News Europe SNE 16/1*, page 33, München, 2006, ARGESIM and ASIM. url: `http://www.sne-journal.org/publications/?tx_pubdb_pi1%5Bpubid%5D=52&tx_pubdb_pi1%5Bppid%5D=87`
- Roland Lezuo and Andreas Krall. A unified processor model for compiler verification and simulation using ASM. In *Proceedings of the Third international conference on Abstract State Machines, Alloy, B, VDM, and Z*, ABZ’12, pages 327–330, Berlin, Heidelberg, 2012. Springer-Verlag. `doi:10.1007/978-3-642-30885-7_24`
- Roland Lezuo, Gergö Barany, and Andreas Krall. CASM: Implementing an Abstract State Machine based Programming Language. In Stefan Wagner and Horst Lichter, editors, *Software Engineering 2013 Workshopband, 26. Februar - 1. März 2013 in Aachen*, volume 215 of *GI Edition - Lecture Notes in Informatics*, pages 75–90, February 2013. (6. Arbeitstagung Programmiersprachen (ATPS’13))
- Roland Lezuo and Andreas Krall. Using the CASM language for simulator synthesis and model verification. In *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO ’13, pages 6:1–6:8, New York, NY, USA, 2013. ACM. `doi:10.1145/2432516.2432522`
- Roland Lezuo, Philipp Paulweber, and Andreas Krall. CASM - Optimized Compilation of Abstract State Machines. LCTES ’07, New York, NY, USA, 2014 (to appear). ACM
- Roland Lezuo and Andreas Krall. Simulation Proofs by Direct Symbolic Execution of Abstract State Machines (under submission).
- Roland Lezuo, Ioan Dragan and Andreas Krall. vanHelsing: Tool and Proof Debugger for Expression Equivalence Problems (under submission).