FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

An optimizing Compiler for the Abstract State Machine Language CASM

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Philipp Paulweber
Matrikelnummer 0727937

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Andreas Krall
Mitwirkung: Projektass. Dipl.-Ing. Roland Lezuo

Wien, 21.04.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

An optimizing Compiler for the Abstract State Machine Language CASM

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science (MSc)

in

Computer Engineering

by

Philipp Paulweber
Registration Number 0727937

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Andreas Krall
Assistance: Projektass. Dipl.-Ing. Roland Lezuo

Vienna, 21.04.2014

(Signature of Author) (Signature of Advisor)

Preface

Vorwort

Corinthian Abstract State Machine
Language, Interpreter and Compiler

The origin of the name Corinthian is unclear whether it is taken from *"the letters, the pillars, the leather, the place, or the mode of behavior"*
Puck, The Sandman by Neil Gaiman

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.

Förderung: Diese Arbeit wurde teilweise von der Österreichische Forschungsförderungsgesellschaft (FFG) unter der Vertragsnummer 827485, Correct Compilers for Correct Application Specific Processors, und von der Firma Catena DSP GmbH gefördert.

Declaration

Erklärung zur Verfassung der Arbeit

Philipp Paulweber
Guntherstraße 1 / 32, 1150 Wien

I hereby declare that this master thesis has been completed by myself by using the listed references only. Any sections, including tables, figures, etc. that refer to the thoughts, listed parts of the Internet or works of others are marked by indicating the sources. Moreover, I confirm that I did not hand in the present thesis at any other university or department.

Hiermit erkläre ich, dass ich diese Diplomarbeit 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. Weiters ist diese Arbeit zuvor keiner anderen Stelle oder Institution als Studiums- oder Prüfungsleistung von mir vorgelegt worden.

(Location, Date)   (Signature Author)

Acknowledgments

Danksagungen

First I want to thank my advisor, professor Andreas Krall, for his great support and feedback during this master's thesis. He always motivated me with challenging questions and interesting discussions regarding the development of the compiler. Especially for letting me develop the optimizing compiler and the optimization passes with such a high degree of freedom. I also want to thank Roland Lezuo for the support during the practical part and his technical comments. Furthermore, I want to thank both, Andreas Krall and Roland Lezuo, for giving me the opportunity to contribute as co-author in a scientific paper and to publish my ideas of the optimizing compiler implementation in [49]. Special thanks to my classmates and friends Stefan Mödlhamer and Jürgen Maier for their technical comments, proof-reading and friendship. Many thanks to my friends Jessica Cornils, Sandra Kirchner and Lukas Raneburger for their precise comments and time-consuming proof-reading as well as their long-term friendship. I also want to thank my parents, Adele and Peter Paulweber, for their continuous support during every educational footstep I went through and that they always believe in me. And last of all, I want to thank my girlfriend, Magdalena Rösch, for her encouraging support and understanding during the work of this thesis – it would not have been possible without you.

Zuerst möchete ich meinem Betreuer, Prof. Andreas Krall, für seine tolle Betreuung und sein Feedback während des gesamten Projekts danken. Die interessanten Diskussionen und herausfordernden Fragen haben mich während der Entwicklung des Compilers sehr motiviert und unterstützt. Ich möchte Roland Lezuo für die Unterstützung im praktischen Teil und die vielen technischen Kommentare bezüglich der schriftlichen Arbeit danken. Ich möchte beiden, Andreas Krall und Roland Lezuo, auch dafür danken, dass ich bei einer wissenschaftlichen Arbeit als Koautor meine Ideen vom optimierenden Compiler einbringen durfte und diese in [49] publiziert worden sind. Speziell möchte ich meinen Studienkollegen und Freunden Stefan Mödlhamer and Jürgen Maier meinen Dank aussprechen. Dafür, dass sie mir mit technischen Kommentaren zur Arbeit geholfen haben und generell für ihre Freundschaft. Ein weiteres großes Dankeschön geht an meine Freunde Jessica Cornils, Sandra Kirchner und Lukas Raneburger für die präzisen Kommentare und das sehr Zeit-Aufwändige Korrekturlesen sowie deren langjährige Freundschaft. Ich möchte auch meinen Eltern, Adele und Peter Paulweber, danken für die Förderung und Unterstützung meiner Ausbildungen und, dass sie immer an mich glauben. Zum Schluss möchte ich meiner Freundin, Magdalena Rösch, für ihre aufmunternde Unterstützung während der gesamten Zeit danken – diese Arbeit wäre ohne dich nicht möglich gewesen.

Abstract

Kurzfassung

The Abstract State Machine (ASM) is a well known formal method which is based on an algebraic concept. This thesis describes the Corinthian Abstract State Machine (CASM) language which is an implementation of an ASM-based general purpose programming language. Features of this language are its combination of sequential and parallel execution semantics and a statically strong type system. Furthermore, this thesis outlines an optimizing run-time and code generator which enables an optimized execution of CASM programs. The code generator is a CASM to C source-to-source translator and the run-time is implemented in C as well. Moreover the CASM optimizing compiler (run-time and code generator) includes a novel optimization framework with the specialized CASM Intermediate Representation (IR). The CASM IR enables powerful analysis and transformation passes which are presented in detail. The evaluation of this thesis shows that CASM is currently the best performing ASM implementation available. Benchmark results show that the CASM compiler is 2 to 3 orders of magnitude faster than other ASM implementations. Furthermore, the presented optimization passes eliminate run-time costs which increases the execution speed of CASM generated programs by a factor 2 to 3 even further.

Die Abstrakte Zustandsmaschine (Abstract State Machine, ASM) ist eine mathematisch basierte formale Methode. In dieser Arbeit wird die Corinthian Abstract State Machine (CASM) Programmiersprache vorgestellt, welche eine konkrete Implementierung einer ASM basierenden Allzweck-Programmiersprache ist. CASM unterstützt das Verschachteln von paralleler und sequenzieller Ausführungssemantik und ist eine statisch getypte Sprache. In dieser Arbeit wird die Laufzeit-Umgebung (Run-Time) und der Codegenerator von CASM vorgestellt, wobei beide Teile auf optimierte Ausführung des erzeugten Programms bezüglich der Ausführungszeit ausgelegt sind. Der Codegenerator erzeugt aus dem CASM Quellcode ein C Programm und die dazugehörige Run-Time ist ebenfalls in C implementiert. Der optimierende Compiler, bestehend aus Codegenerator und Run-Time, beinhaltet zusätzlich ein neues Optimierungs-Framework mit einer speziellen CASM Zwischendarstellung (Intermediate Representation, IR). Basierend auf dieser CASM IR werden verschiedene Analysen und Transformationen in dieser Arbeit beschrieben. Die Evaluierung zeigt, dass CASM gegenüber anderen ASM Implementierungen die Performanteste ist. Die Ergebnisse von Benchmark-Tests zeigen, dass die generierten Programme zwei bis drei Größenordnungen schneller ausgeführt werden als von anderen ASM Implementierungen. Das Optimierungs-Framework und die vorgestellten Transformationen verbessern die Ausführungszeit der generierten Programme noch weiter um den Faktor 2 bis 3.

Contents

CHAPTER 1

Introduction

"The Abstract State Machine (ASM) thesis states that, for every computer system A, there is an ASM B such that B is behaviorally equivalent to A and in particular step-for-step simulates A" [34]

An Abstract State Machine (ASM) is used to formalize the semantics of programming languages [37], to describe and simulate micro-processors [63] or to construct cycle-accurate Instruction Set Simulator (ISS) applications [48]. Latter is the main motivation and focus of this thesis where an ISS is verified against its specification with use of an ASM language implementation. This verification method is known as *compiler back-end verification* [45]. The problem is that everything has to be verified for safety-critical applications – the source code and the hardware. But the used compiler which translates the source code to machine code can introduce incorrect translations. A common solution to this problem is *translation validation* [45] where the source code and the produced machine code is checked for semantically equivalence.

There are a lot of different approaches, implementations and extensions of ASMs. The ASM thesis itself was defined by Gurevich in the Lipari Guide [32]. Gurevich's main idea was to improve and generalize the Turing's thesis, but later on he developed and proved the ASM thesis for sequential algorithms [10]. Nowadays, the ASM method is *"a practical and scientifically well-founded systems engineering method"* which closes the gap between soft- and hardware validation and verification techniques.

The Corinthian Abstract State Machine (CASM)[1] language is a concrete implementation of an ASM. It consists of a statically strong type system and supports mixing of sequential and parallel execution semantics which suits the needs for developing and modeling of an Instruction Set Architecture (ISA) for an ISS very well. CASM consists of an interpreter which supports numeric and symbolic execution of CASM programs. The interpreter is used mainly to evaluate small applications of system units of a larger project. Furthermore, CASM consists of

[1]the definition of *Corinthian* is given at page i

an optimizing compiler which supports code generation to C and numeric execution of CASM programs. The compiler is used especially for large programs with rather long execution times.

1.1 Terminology

This thesis contains besides the CASM language definition the complete description of the CASM optimizing compiler. To avoid misinterpretation of commonly used expressions and terms, their definition is given as followed:

domain A *domain* is a set of values and can be composed of arbitrary sub-domains. A composition of arbitrary sub-domains is also known as *relation*.

type A *type* is a specific value domain, e.g. an integer type includes all negative and positive integer values.

function A *function* f maps a domain D to a co-domain C. The domain is defined by a *relation*, which includes none, one or multiple types.

$$f : D \to C$$

state A *state* is a consistent view of a function's co-domain.

global state The *global state* represents the co-domain of all functions.

location A *location* l is a pair $l = (f, x)$, where f is a function name and x is a tuple of elements (function arguments) and the length of the tuple equals the dimension of the function f. [32]

update An *update* u is a pair $u = (l, v)$, where l is a location and v is a concrete value of a specific type. [32]

update-set An *update-set* is a container which handles arbitrary updates.

rule A *rule* describes state transitions by producing updates to function locations.

composition A *composition* enables different execution semantics – sequential and parallel.

side effect free Something is *side effect free* if no update is produced whatsoever.

conflict update A *conflict update* occurs if two or more updates contain the same location. This introduces an inconsistent view of the global state. In CASM an inconsistent view results in an error.

sub-machine state A *sub-machine state* is introduced if sequential composition is used. Each sequential operation result in a sub-machine state which is merged with the next sequential operation. [11]

step An ASM *step* writes all gathered updates from the update-set to the global state.

1.2 Motivation

Current implementations of ASM languages mentioned in Section 2.1 do not perform very well. Especially for large programs the execution times are too high and unfeasible. The mixing of parallel and sequential execution semantics in ASM languages introduces sub-machine states and partial updates which are not handled very well by available ASM tools.

1.3 Problem Statement

In the course of this work a new optimized run-time and code generator is implemented and compared to the prototype implementation [48] (aka legacy compiler), the interpreter and other available ASM tools. The syntax of the CASM language shall be completely supported. Furthermore, an optimizing compiler shall enable multiple passes to perform various analyses and transformations. The results shall increase the execution time by one order of magnitude.

1.4 Methodological Approach

First of all, the current prototype implementation must be analyzed to detect weak spots of performance. After that a new design for the code generator and run-time can be made. The design should handle functions with n-dimensional relations, an optimized handling of updates and an enhanced implementation for integer types with sub-ranges. The next step is to implement a compiler framework which enables optimizations for different passes.

1.5 Structure of this Thesis

This thesis is structured as follows: Chapter 2 gives an overview of the state-of-the-art regarding ASM tools, optimizations and compiler frameworks. Chapter 3 gives a detailed introduction into the CASM language and describes all syntax elements with their semantics.

Chapter 4 gives an overview of the new design and a detailed description of the implementation of the CASM run-time and code generator. It includes the analysis of the prototype implementation from [48] and then a full description of all used data-structures, components and Application Programming Interface (API) C functions and macro signatures of the run-time. Additionally the complete generation process of the code generator is described.

Chapter 5 first gives an overview of the CASM compiler optimization framework with a novel CASM Intermediate Representation (IR) and then the implementation of several passes are outlined.

Chapter 6 presents an empirical evaluation of the new CASM compiler. The focus is set on the execution time speedup compared to the prototype implementation. Furthermore, the new optimization framework with its passes is evaluated for a specific application in detail.

Chapter 7 gives a conclusion of the work and outlines some future work topics.

CHAPTER 2

Related Work

Over the past 20 years since the ASM theory was defined, several ASM language specifications and software tools have been designed and developed [9]. First this chapter gives a chronological overview of several ASM languages, interpreters and compilers. Second some related work regarding optimizations, analyses and transformations are described. Thereafter some compiler frameworks are outlined as well.

2.1 ASM Languages, Interpreters & Compilers

In 1993 Gurevich formulated the ASM thesis[1] in the Lipari Guide [32]. Together with a team at Microsoft Research he developed the major implementation of an ASM specification called AsmL [52]. The specification of the AsmL language was designed to fulfill several properties which are *simple*, *precise*, *executable*, *testable*, *inter-operable*, *integrated*, *scalable* and *analyzable* [33]. According to [33] these properties were the main reasons for the design and creation of AsmL, because their was no other language at the time which satisfied all of these features. The implementation of AsmL is directly integrated into the Microsoft .NET run-time. Therefore AsmL can use all the .NET objective and structuring language features like *"enumerations, delegates, methods, events, properties and exceptions"* [33]. The language is heavily influenced by imperative ones like C# (C Sharp) [35], Java [40], Haskell [36], Standard Meta-Language (SML) [30] and Vienna Development Method (VDM) [55]. The main core of the language is called AsmL-S where the S stands for *simple* which is a subset of the AsmL specification to provide only basis functionality. The only supported data-type in AsmL-S are maps. The main language properties for an ASM specification are the finite choice and parallelism which is stated in [32], but AsmL has also some extensions. It includes sequential composition, sub-machine states and *partial updates*. Gurevich and Tillmann explored that *partial updates* [34] have a huge impact on the performance of an ASM implementation which were produced during the execution of an ASM step. They have shown how it is possible to implement ASM data-structures efficiently

[1]prior known as evolving algebra

and also used this technique in the AsmL compiler. The resulting compiler and the software built on top are currently the most advanced available ASM tools.

In 1998 Castillo presented the ASM Workbench [19]. It is an open and extensible tool environment for ASM models. The kernel of this tool is implemented in SML [30] and consists of an interpreter, debugger, type-checker and a graphical user interface. The ASM models are described in the Abstract State Machine-based Specification Language (ASM-SL). Castillo introduced this language which was mainly influenced by SML but he added a *"type-system to structure the data-model"* [19]. The general focus of this ASM implementation is the extensibility and modularity.

Schmid [60] used ASM-SL from Castillo as input specification and developed a source-to-source compiler which translates the ASM language to C++ [64] in 1999. The overall motivation was to generate source code for a big railway simulation tool which was written and debugged in the ASM Workbench, but the execution speed of the ASM Workbench interpreter was too slow. Schmid translates with its compiler functions directly to C++ Standard Template Library (STL) [13] container classes. Besides this translation, Schmid introduces a *free-type* data-type which is a universal data-type to all present types from ASM-SL. Furthermore, Schmid uses a *double buffering* technique to redirect location updates and lookups of ASM functions. This approach can only deal with parallel execution semantics and does not include sub-machine steps which is a major drawback for this compiler implementation.

In 1999 Schmid also introduced the ASM-based interpreter AsmGofer [67] [59]. The interpreter is an extension to the programming language Gofer [39] which is a subset language of Haskell [36]. Schmid adopted the run-time of Gofer to support updatable functions, but no further changes where made. AsmGofer uses the same type-system which is given by the Gofer language. According to the author, the interpreter executes *scripts* with *"signatures, rules, functions and data structures"* [67]. It is important to mention that AsmGofer implements almost all features of the Lipari Guide [32]. Especially the multi-agent concept to execute or *fire* multiple rules in parallel. Furthermore, AsmGofer also includes TkGofer [66] to support and build graphical user interfaces. But the major drawback of this interpreter implementation is that it is heavily integrated into the Gofer environment and that it is only an interpreter. The resulting interpreting speed of an AsmGofer *script* is impractical in a *"performance critical application"* [59]. Schmid used AsmGofer also for the ASM-SL to C++ compiler for validation purposes [60].

A source-to-source compiler named eXtensible Abstract State Machine (XASM) was introduced by Anlauff in 2000 [2]. The difference to Schmid's compiler [60] is that Anlauff translates its own ASM language XASM to C [41]. Furthermore, this language is the first one which has a component-based design pattern to allow programmers to reuse and structure ASM components. To achieve more flexibility in XASM it is possible to declare a function as *external* and the implementation of this function can be in plain C. It is also possible to embed a XASM model in a C project. Anlauff extended the normal ASM specification by adding language constructs like *constructor terms*, *regular expressions* and *the "once"-rule* [2]. He provided a very rich-featured graphical debugging and animation interface to e.g. perform stepwise execution or visualize the parse tree etc. Unfortunately, there is no ongoing development of this project and the tools are not available any more.

In 2005 the CoreASM open source project started, which was another ASM language with full tool support [15]. Farahbod stated in [24] that this language was designed to be as *"close as possible to the mathematical definition of pure ASMs"*. Further his motivation derived from the fact that all current ASM languages are too detailed and CoreASM represents the so called *Ground Model* which is perfectly fitting the problem space of an abstract software model [24]. An important aspect of the design is the extensibility of the language in form of *plug-ins*. For example via a *plug-in Signals* it is possible to send and receive data between two running ASM agents. Other *plug-ins* are *Math*, *Graph*, *Observer*, *JASMine* and *Barun* [15]. All *syntactic sugar* is implemented in CoreASM through *plug-ins* as well. To cover the requirement to have a real abstract model, the language is untyped. The execution of a CoreASM model is done via the CoreASM interpreter. All tools are freely available on the project website [15].

Another general approach to ASM systems is the ASM mETAmodelling framework (ASMETA) project [57]. It is a meta-model based specification designed by Gragantini [27]. He describes ASMETA as an instance of the underlying Object Management Group (OMG) meta-model framework which uses the XML Metadata Interchange (XMI) format and parts of the Model Driven Architecture (MDA) framework to create a unifying abstract mathematical model. ASMETA has its own compiler called AsmetaL with a graphical environment, an interpreter (simulator) AsmetaS and an integrator to other ASM languages e.g. CoreASM. AsmetaL is also the name for the ASMETA language. The language is based on the principle of mapping Meta-Object Facility (MOF) to Extended Backus-Naur Form (EBNF) grammar. Similar to other languages, the syntax is divided into *"a header, a body, a main rule and an initialization"* specification [27]. ASMETA is relying on a static type-system. Due to the generic concept of AsmetaL, the composition of ASM components is supported as well. Drawbacks of the language are that constructs like *choose* or *forall* are not implemented. Benefits are advanced logging of ASM transactions, *random simulation*, multiple extension features via Java interfaces and very feature-rich Eclipse [21] tools.

In 2007 Ouimet introduced the Timed Abstract State Machine (TASM) [56]. This approach extends the normal ASM theory to have functional and non-functional constraints in sequential and parallel execution semantics by specifying an execution time property for a rule evaluation. Therefore, modeling of real-time system behavior is possible with TASM. Language features are *hierarchical composition* like in [2] and the mixing of sequential and parallel compositions. TASM supports for multiple ASM systems *synchronization channels* in the Calculus of Communication Systems (CCS) style, to synchronize two machines through a *receiver* and *sender* channel [56]. Furthermore, the language has integrated a *resource* concept which allows specifying and modeling of non-functional resource consumptions e.g. power consumption and memory consumption.

Since 2012, Lezuo [48] has developed the new ASM language CASM in the course of the Correct Compilers for Correct Application Specific Processors (C3Pro) project [14]. CASM was mainly derived from the CoreASM language, because first evaluations were made in this specification to verify compilers and instruction set simulators [47]. The first version of the CASM interpreter and compiler are written in Python [65]. The interpreter supports numeric and symbolic execution whereas the compiler is translating the ASM model directly into a numeric execution-only C++ program. For the translation to C++, Lezuo uses a simple direct translation and ASM objects like functions are represented via STL container classes. Limitations of the first compiler are the restriction of functions with only 0-, 1- or 2-ary domains.

In 2013 Inführ presented a new interpreter for the CASM language [38]. This interpreter is written in C++ and re-implements the Python-based CASM interpreter of Lezuo [48]. The lexing, parsing, numeric and symbolic interpreting of CASM programs is covered by this implementation. This interpreter is the origin of this thesis. All adoptions of the language, the new compiler and run-time where combined with this tool to have an all-in-one CASM interpreter and compiler.

2.2 Other Languages, Compilers & Optimizations

Currently there are no explicit mentioned ASM analyses and transformations to achieve more performance of the resulting program. The transformation approaches of this thesis are mainly influenced by redundant memory access elimination, data-structure modifications and reducing run-time costs. Analysis algorithms are derived from classic sequential Data-Flow Analysis (DFA) procedures [20].

The major focus of optimizing CASM is set to reduce run-time costs similar to *load/store elimination* transformation algorithms presented by Barik [7] or Lo [51]. Another idea similar to load/store elimination is *copy avoidance* presented by Debray [18]. All these transformations are minimizing the run-time costs.

Besides the classical DFA [42] through dependency graphs, modern compilers use the Single Static Assignment (SSA) graph [53] to represent a program. Glesner presented in 2004 a SSA representation for formal ASM specifications [31].

In 2004, Edwards presented the Shim language, which is a mix of *"imperative C-like semantics for software and RTL-like semantics for hardware"* to describe soft-/hard-ware components in a unified language [22]. For this language a compiler exists, which can either emit C or Very High Speed Integrated Circuit Hardware Description Language (VHDL) source code. An example I^2C bus controller is demonstrated by the author.

The CASM compiler and its optimizations are mainly influenced by the following compiler frameworks: A prime example for a compiler framework is the Low Level Virtual Machine (LLVM). Lattner introduced it in 2004 and outlined a very flexible way to organize analyses and transformations into a unified pass structure [43]. Furthermore, the LLVM IR enables a decoupling of source and target languages. A lot of the present passes from LLVM inspired the pass structure of the CASM compiler [50]. Additionally the IR of LLVM inspired the CASM IR as well.

Another well-known compiler is the GNU Compiler Collection (GCC) [62] [29]. It was the first open-source cross compiler toolchain. Originally it was designed for the GNU's Not Unix (GNU) Operating System (OS) but nowadays it is the de facto open-source standard compiler on a Unix/Linux OS. It supports over 20 source languages and over 70 target computer architectures. Since 2003 the compiler has introduced a new mid-end IR with SSA support which is called GENERIC and the sub-set GIMPLE [54].

Wilson presented in [68] the SUIF compiler system. It is a flexible, extensible and according to the author easy to use compiler framework. The design of SUIF is based on a so called kernel which is never changed and the passes built on top are constantly replaced through more powerful ones. There exists a lot of libraries and packages which can be integrated into SUIF e.g. a Control Flow Graph (CFG) or bit-vector package [61]. The SUIF compiler has its own 2-layer IR. The SUIF IR is used to represent a program in form of statements and expressions. The second layer is the Machine-SUIF IR. The Machine-SUIF IR enables *"to develop machine-specific optimizations for existing or future machine models"* [61].

Besides all the existing frameworks, the ROSE [58] source-to-source and/or binary-to-binary compiler framework uses the parsed Abstract Syntax Tree (AST) as internal IR. Internal compiler behavior is optimized for AST transformations and traversals. The analysis part of ROSE is using several concepts at once. It is possible to create CFG and even include Satisfiability Modulo Theories (SMT) solver, symbolic execution *"and abstract interpretation"* [58]. In the transform part ROSE offers AST modification functions which are not only modifying the AST itself, they even keep the symbol tables in a consistent state. One focus of ROSE is program visualization, another focus is support of the language OpenMP [16].

The CETUS compiler infrastructure [17] is a more specialized framework for analyses and transformations of parallel programs. The target language is C. CETUS is written in Java and the IR is constructed through a Java class hierarchy. Due to the parallel focus, CETUS has loop and array analyses and transformations.

CHAPTER 3

CASM Language

Originally CASM was influenced by the CoreASM language [48]. Nowadays, the language has diverged and CASM evolved a lot of new features. This chapter describes in detail the syntax and semantics of the CASM language.

3.1 Overview

Before the details of the syntax and semantics of CASM are presented, some examples are outlined to get familiar with the CASM language and its behavior. The first example is the paragon of every language to outline the basics, the *hello world* example:

```
 1 CASM HelloWorld                 // program/module name
 2
 3 init example                    // set top-level program rule to 'example'
 4                                 // equals an update: program(self) := @example
 5
 6 rule example =                  // definition & declartion of rule 'exmaple'
 7 {
 8     print "Hello, World!"       // simple print statement
 9
10     program(self) := undef      // set top-level program rule to 'undef'
11 }                               // equals a termination of the program
```

Listing 3.1: CASM Hello World Example

Every CASM program starts with the module header specification in line 1 which declares the module name of the current program. This module name will be used to extend CASM in a later implementation with a module-based/component-based hierarchy design. In line 3 the top-level rule of the execution agent, is set by the *init* specification. When the program starts, the *init*-rule will be called as soon as the CASM kernel has been loaded.

In line 6 the rule *example* is declared and defined, which is the only rule in this small example. The rule defines a parallel block statement from line 7 to 11 with a print and an update statement. The print statement in line 8 outputs the *hello world* message to the standard output stream. Line

10 contains an update of the location *program* at *self* with the value *undef*. The function *program* is a special CASM function to control the execution ASM agent's top-level rule which is called after an ASM calculation step.

Currently CASM supports a single-agent ASM executable model and that is the reason why the argument of the function *program* uses the *self*-reference CASM-keyword. So this program is calling the rule *example* at the beginning and prints out the *hello world* string and after the first ASM step the program terminates. This happens because the top-level rule equals **`undef`**. The following listing shows the output of the *hello world* program:

```
Hello, World!
1 step later...
```

Listing 3.2: Output of the Hello World Example

In the above output besides the *hello world* string also the amount of the calculated ASM steps is printed out. To get an even better impression of the capabilities of CASM, the next example is more complex:

```
 1 CASM Swap init foo
 2
 3 function x : -> Int initially {1}
 4 function y : -> Int initially {2}
 5
 6 rule printf = print "x = " + x + ", y = " + y
 7
 8 rule foo =
 9 {|
10     call printf
11
12     {
13         x := y
14         y := x
15     }
16
17     call printf
18
19     let tmp = x in {|
20         x := y
21         y := tmp
22     |}
23
24     call printf
25
26     program(self) := undef
27 |}
```

Listing 3.3: CASM Swap Example

This *swap* example introduces several other CASM concepts. First of all in line 3 and 4 two functions are defined with an initial value of $x := 1$ and $y := 2$. Functions in an ASM language are not limited in their parameter and value domain, because of the mathematical function model. In theory e.g. the function x can have a value in the range $[-\infty, \infty]$. To print out the current sub-machine state of these two functions, a rule with the name *printf* is defined in line 6. The *init*-rule is set to the rule *foo* which is defined in line 8.

The rule *foo* starts by declaring a sequential block, so every execution step inside, is processed sequentially. Furthermore, a *printf* rule is called three times to output the current sub-machine state of the two functions. Between the first two calls a nested parallel block is declared. This block performs a swap algorithm in parallel execution semantics. Because the two updates are surrounded by a parallel block, CASM fetches the actual state value of the two functions and produces an update-set after line 13 with $\{x = 2\}$. After line 14 the update-set becomes $\{x = 2, y = 1\}$. In line 15 the parallel block ends and so the update-set is merged into the parent update-set from the sequential block. To show that the swap has happened the *printf* rule is called to output the sub-machine state of the functions x and y.

To get the origin state for the functions x and y, another swap algorithm is performed in line 19. This time the swap is performed in a sequential execution semantics. A binding is needed to store the intermediate value of one function. CASM provides a *let* statement which binds an expression to an identifier. The bound expression is read-only. In the swap example the *let* $tmp = x$ is defined and creates a nested sequential block to perform the swap. It results into the update-set at line 21 with $\{x = 1, y = 2\}$. The *let* statement ends in line 22 and the update-set is merged with the upper update-set which results in $\{\cancel{x = 2}, x = 1, \cancel{y = 1}, y = 2\}$. The console output is shown in the following listing:

```
x = 1, y = 2
x = 2, y = 1
x = 1, y = 2
1 step later...
```

Listing 3.4: Output of the Swap Example

3.2 Syntax & Semantics

The following subsections outline in detail the CASM language syntax in proper EBNF [28]. First some EBNF production rules are described which are used all over the CASM language definition.

```
identifier      ::= (* C variable naming convention *) ;
identifier-list ::= identifier { "," identifier } ;
expression      ::= (* see Section 'Expressions' *) ;
expression-list ::= expression { "," expression } ;
type            ::= (* see Section 'Type System' *) ;
parameter       ::= identifier [ ":" type ] ;
parameter-list  ::= parameter { "," parameter } ;
```

Like in other modern languages an *identifier* is an alpha-numeric string. In general there is no constraint on the *identifier* label, but due to the fact that CASM is compiled to C, the label should respect the C naming convention, that a variable never starts with one or two underscores, because these are reserved for language implementers and compiler engineers. Additionally the use of two underscores at the end of a variable name has also some limitations, because the compiler uses them to represent e.g. temporal calculations, etc. (see Chapter 4).

The CASM syntax is divided into a literal, type system, specification, statement and expression part.

3.2.1 Literals

```
literal ::= "undef" | "self" | "true" | "false" | number | string | range | ruleref | list ;
number  ::= (* C signed integer value in decimal or hexadecimal notation *) ;
string  ::= (* C character array aka string surrounded by "" *) ;
range   ::= "[" number ".." number "]"
          | "[" identifier ".." identifier "]" ;
ruleref ::= "@" identifier ;
list    ::= "[" [ expression-list ] "]" ;
```

Literals represent a concrete value of a specific data-type. CASM consists of primitive and non-primitive types which are described in the following Section 3.2.2. Currently CASM supports 8 different constant literals. They are defined as follows:

Undefined (`undef`) The `undef` literal is the universal value which every data-type includes in its value domain. Functions, lets, etc. can have any time the 'value' `undef`.

Agent Self Reference (`self`) A special literal in CASM is the `self` keyword. It returns the reference to the current executing ASM agent. Currently CASM supports only one ASM agent. This implies that the `self` reference is always the same agent. By supporting this `self` reference mechanism, the extension of CASM to a multi-agent ASM system in the future is simplified. Furthermore, the `self` reference is mainly used to retrieve the location of the *program* function which corresponds to the ASM agent top-level rule. The *program* function is predefined by CASM (see Section 3.2.3).

Boolean Values (`true`, `false`) Basic Boolean values are expressed in CASM through the `true` and `false` keyword.

Number A valid *number* in CASM corresponds to a valid number in C e.g. `123`, `-456`, `0x789ace`, etc. Either decimal or hexadecimal notation can be used.

String The CASM *string* literal corresponds to the C, C++ or Java string literal e.g. `"text"`, `"this is a string in CASM"`, etc.

Range To represent a *range* of integer numbers CASM provides the static or dynamic *range* syntax. It is possible to determine the range during run-time by loading the value of the given variable name through an *identifier* label. Ranges are mainly used in the later defined `forall` statement (see section 3.2.4).

Rule Reference The rule reference (short *ruleref*) is a pointer to a specified rule in the CASM program. It is created by using the 'at' (`@`) character and a rule *identifier* name. These references are mainly used e.g for the later defined *indirect call* of rules (see Section 3.2.4).

List A main feature of CASM is the support of tuple- and list-based data-types. The *list* literal is used to initialize such structures. Empty square brackets (`[ ]`) result either in an empty tuple- or list-based data-type depending on the annotation (see Section 4.1.1).

3.2.2 Type System

```
type     ::= identifier
           | identifier "(" type { "," type } ")"
           | identifier "(" number ".." number ")" ;
```

The type system of the CASM language is straight forward. The identification of the types is done after the parsing. This enables a flexible compiler structure and allows extending the type system with new types without changing the syntax of the language. The grammar above defines three possible patterns for a type. The first and last pattern is the syntax for primitive types and the second one is for the non-primitive data-types.

It is important to mention that all CASM types can either have a value of their specific value domain or they can be undefined. Furthermore, the conversion between different data-types is done in CASM explicitly through the later introduced *built-in* intrinsics (see Section 3.2.5).

Primitive Types

The CASM language contains 6 different primitive data-types, which are:

Boolean (`Boolean`) The `Boolean` type is the simplest type in most programming languages and the value domain B is defined as:

$$B \quad = \quad \{\, \texttt{undef},\ \texttt{true},\ \texttt{false}\, \}$$

Enumeration (`enum`) Enumerations can be defined through a unique name `N` and with a distinct identifier set $M = \{M_0, M_1, \ldots, M_{n-1}\}$ of length n which implies a direct ordered relation $r : 0 \mapsto M_0, 1 \mapsto M_1, \ldots, n-1 \mapsto M_{n-1}$ (see Section 3.2.3). The resulting value domain $E_{\texttt{N}}$ for the specific enumeration with name `N` is given by:

$$E_{\texttt{N}} \quad = \quad \{\, \texttt{undef}\, \} \ \cup \ \left\{ \bigcup_{i=0}^{n-1} r(i) \right\}$$

Integer (`Int`) The integer value domain I of the data-type `Int` is statically defined through:

$$I \quad = \quad \{\, \texttt{undef}\, \} \ \cup \ Z\ , \ \ Z\ \ldots\ \textit{set of integers}$$

Ranged Integer (`Int()`) For a ranged integer type `Int(a..b)` where a and b are numerical literals, the resulting value domain $I_{[a,b]}$ is defined as:

$$I_{[a,b]} \quad = \quad \{\, \texttt{undef}\, \} \ \cup \ \{\, a \ \le\ x \ \le\ b \mid x \in Z\, \}\ , \ \ Z\ \ldots\ \textit{set of integers}$$

String (`String`) The value domain S of the type `String` is either `undef` or all possible strings. The set *Strings* forms an infinite set. S is defined as:

$$S \quad = \quad \{\, \texttt{undef}\, \} \ \cup \ \textit{Strings}$$

Rule Reference (`RuleRef`) Every CASM program has a static known amount of rules. *RuleReferences* is the set of all possible *ruleref* literals of the existing rules in the program. So the value domain R of the data-type `RuleRef` corresponds to:

$$R \quad = \quad \{\, \texttt{undef}\, \} \ \cup \ \textit{RuleReferences}$$

Non-Primitive Types

The non-primitive data-types in CASM are similar to pointer/reference/object types. They have a specific behavior and to modify this behavior, CASM provides data-type specific intrinsics.

Tuple (`Tuple`) In CASM it is possible to declare `Tuple`-based data-types of arbitrary sub-types. For example the type `Tuple(Int,String,Int,Boolean)` results in a quadruple data-type of two integers, one Boolean and one String sub-type. A `Tuple` in CASM can have the value `undef`. The value domain T_H where H is the value domain of all sub-types of the `Tuple` is defined as:

$$T_H = \{\, \texttt{undef} \,\} \cup H$$

List (`List`) A `List`-based data-type in CASM can be declared with a generic sub-type e.g. `List(Int)` results in an integer-based list. The value domain L_H of the `List` data-type can either be `undef` or the list itself. H is the set of all possible list elements.

$$L_H = \{\, \texttt{undef} \,\} \cup H$$

Internal Types

In CASM there are two types not visible for the programmer, because these types are only used compiler internally - the *Undef* and *Self* data-type.

Undefined (`Undef`) The compiler uses the `Undef` data-type for unclear type relations e.g. the `undef` literal is by default of type `Undef` if the annotation can not resolve the type correctly. Therefore the value domain U of `Undef` consists only of the value `undef`.

$$U = \{\, \texttt{undef} \,\}$$

Agent Reference (`Self`) Currently CASM supports only the single agent execution of an ASM model and to represent this single agent the `Self` data-type is used. This special data-type consists only of the *self* reference literal and its value domain A (for agent) is defined as:

$$A = \{\, \texttt{self} \,\}$$

3.2.3 Specifications

```
CASM ::= header body { body } ;
body ::= init | enum | derived | function | provider | rule ;
```

A CASM model is structured into a module *header* and a *body* specification section. The *body* specifications can have an arbitrary order and thus it is possible e.g. to split a CASM program into multiple files. The CASM interpreter/compiler currently supports only a single compilation file, but the split specification parts can be concatenated to one compilation file. Therefore a simple code reuse and structuring can be applied to a CASM program.

Module Header Specification

```
header ::= "CASM" identifier ;
```

The module header specification[1] is used to group all CASM objects e.g. functions to one module. Currently the CASM header is not directly used, because the component-based approach is not implemented yet. But the module header is indispensable so that all current CASM programs can be also used in the future.

Init-Rule Specification

```
init ::= "init" identifier ;
```

The *init*-rule specification[1] defines the starting top-level rule of the ASM agent. If this specification is not defined, the CASM engine is not able to execute the model and will return an error.

Enumeration Specification

```
enum ::= "enum" identifier "=" "{" identifier-list "}" ;
```

Through the *enum* specification a new named enumeration type can be defined. The *identifier* of the *enum* has to be unique just like the defined enumerated values e.g.:

```
enum Channel = { Alice, Bob, Charlie, Dan, Eve }

enum Weekday = { Monday, Thuesday, Wednesday, Thursday, Friday, Saturday, Sunday }
```

Listing 3.5: Enumeration Examples

Derived Specification

```
derived ::= "derived" identifier [ "(" [ parameter-list ] ")" ] [ ":" type ] "=" expression ;
```

The *derived* specification can be used to define reusable side effect free expressions. The *derived* name has to be unique and it can be defined with typed[2] or not typed parameters and/or return type. If no return type and/or parameter types are used, the CASM compiler front-end annotates the type from the defined expression otherwise the specified type will be used e.g.:

```
derived constant_value = 2014

derived isNotEqual(a, b) = ( a != b )

derived calculate(a : Int, b : Int) : Int = a + b
```

Listing 3.6: Derived Expression Examples

The *derived constant_value* gets a return type of integer data-type, because of the constant integer number. The *isNotEqual derived*'s return type will be annotated with a Boolean data-type, because the root expression is a *not equal* comparison. The parameters are not annotated, because multiple types can be checked for inequality. In the last example *calculate*, the return type and parameters are directly defined.

[1] an example usage is shown in Listing 3.1
[2] valid CASM types are described in Section 3.2.2

Function Specification

```
function    ::= "function" [ property ] identifier ":" relation [ initially ]
property    ::= identifier
              | "(" [ identifier-list ] ")"
relation    ::= [ type { "*" type } ] "->" type
initially   ::= "initially" "{" init-list "}" ;
init-list   ::= init-list "," [ expression ]
              | expression ;
```

The *function* specification in CASM is very expressive. A *function* f has a distinct name and a mathematical relation. The relation defines domain D in which the function can be used and a co-domain (target set) C which holds the corresponding values. Due to the mathematical concept in ASM languages all specified functions are by default undefined over the complete domain. The following equation expresses the function specification as:

$$f : D \rightarrow C, \; \forall x \in D \mid x \mapsto \texttt{undef} \;, \; \texttt{undef} \in D$$

In some cases in a CASM model a function should have predefined values at distinct locations in the domain. For this use case the optional *initially* syntax is provided by the language. It allows defining the co-domain for a sub-set of the domain of a function with arbitrary expressions. The co-domain can even depend on another function's co-domain.

Furthermore, a function in CASM can have optional *property* behaviors which can be specified through a list of identifiers. So it is possible to set multiple properties for a function. The current supported properties are:

`controlled` Every function is by default *controlled* in CASM, which means that this function can be updated any time during the execution.

`defined` A function is *defined* means that all elements of the domain are mapped to the default value C_0 of the type domain C: $f : D \rightarrow C, \; \forall x \in D \mid x \mapsto C_0$
For example if a function has an **`Int`** co-domain the default value is 0 which would result in: $f : D \rightarrow I, \; \forall x \in D \mid x \mapsto 0$
With this property it is e.g. possible to initialize a modeled memory block.

`static` Declaring a function *static* results in a read-only or not updatable function for the whole execution of the model. The only way to set specific values for a function is to initialize it with the *initially* keyword.

`symbolic` The *symbolic* property enables symbolic execution of a specific function only in the CASM interpreter. The interpreter ignores this property if it executes the program in numeric mode. Lezuo [45] describes this in more detail.

`undead` By setting a function to *undead*, the CASM compiler will not optimize out this function even if he detects that this function will never be written or read. This is a special *property* for the later introduced *Dead Function Elimination* pass which is described in section 5.4.4.

The following Listing 3.7 shows some examples of function specifications:

```
function x : -> Int

function (controlled) y : Int * Int -> Int

function (static) z : Int -> Int initially { 0 -> 10, 10 -> 0 }

function a : -> Int initially { z( 0 ) }

function (static, defined) b : Int -> Int
```

Listing 3.7: Function Examples

Function x has no domain which implies that it is a 0-ary function whereas the function y is a binary function which results in a 2D-domain from $(-\infty, -\infty)$ to (∞, ∞). The z function creates a 1D-domain and initializes some distinct points to a constant value through the *initially* keyword. Function a initializes its co-domain with an initial value from function z. Function b uses the *defined* keyword and this initializes the co-domain of function b over the whole domain with the value 0 which corresponds to $b : D \rightarrow C, \forall x \in D \mid x \mapsto 0$. All of those example functions have an **Int** type co-domain.

One function is always defined by default – the *program* function. As mentioned earlier the *program* function stores the current top-level rule of the single ASM execution agent. It is always added to the input program before the annotation process starts (see Section 4.1.1). The following Listing 3.8 shows the internally defined *program* function described in the CASM language:

```
function program : Self -> RuleRef initially { @/* init rule */ }
```

Listing 3.8: Program Function

Rule Specification

```
rule ::= "rule" identifier [ "(" [ parameter-list ] ")" ] [ dump ] = statement { statement } ;
dump ::= "dumps" "(" identifier-list ")" "->" identifier
         [ { "," "(" identifier-list ")" "->" identifier } ] ;
```

Another prime specification in CASM is the *rule*. The ASM method defines rules as the only valid structure to modify the global state. Rules act as state transitions which can be *fired* one after another. A rule in CASM has a distinct name and at least a single *statement*.

Through the introduction of rule calling in section 3.2.4, a rule can have optional parameters to e.g. control the rule behavior. As with derived specifications, rule parameters are annotated through the compiler if the type was not specified[3]. A rule in CASM has no return value, because rules can only create updates to the update-set which will be later applied to the global state.

Rules can have an optional *dump* specification which allows defining multiple debug output streams to print out all updates of the listed functions. Listing 3.9 shows an example for *rule* specifications with and without a *dump* specification:

[3]more information about annotation is provided in Section 4.1.1

```
rule foo = skip

rule run( arg : Int ) = print arg

rule loop dumps ( cnt ) -> trace =
{
    cnt := cnt + 1

    call run( cnt )

    if cnt = 10 then program( self ) := undef
}
```

Listing 3.9: Rule Examples

The first *rule* is the smallest possible rule in CASM with a name and a *skip* statement. In the second *rule*, a parameter *arg* is defined and the rule performs a *print* statement to output the content of this argument. For the last *rule* a function *cnt* is assumed which stores a counter value and this rule *loop* increases the *cnt* by creating an update to it. Furthermore, a call to the rule *run* is performed. The *if* statement checks if the current state value of the *cnt* is equal to 10 and creates an update to terminate the program if this condition is true.

Provider Plug-in Specification

```
provider ::= "provider" identifier ;
```

A *provider* is the CASM external interface to include user-defined e.g. functions, rules, etc. The detail of this mechanism is described in Section 4.2.7. It is important that the provider plug-in is not available in the CASM interpreter. If a CASM program is executed with the CASM interpreter and a *provider* is used, the execution will be aborted.

3.2.4 Statements

```
statement     ::= trivial | conditional  | compositional ;
trivial       ::= skip    | diedie | impossible | print     | debuginfo | update
                | push    | pop    | forall     | iterate   | objdump ;
conditional   ::= assert | assure | case       | if ;
compositional ::= call    | let    | seqblock   | parblock ;
```

The statements in CASM are grouped in three categories – trivial, conditional and compositional.

Skip Statement

```
skip ::= "skip" ;
```

Every language defines at least on syntactic element which performs no operation at all and CASM uses the *skip* statement for that. It can be used e.g. to specify a rule which has no behavior.

Diedie, Impossible, Assert and Assure Statement

```
diedie     ::= "diedie" ;
impossible ::= "impossible" ;
assert     ::= "assert" expression ;
assure     ::= "assure" expression ;
```

To abort an ongoing execution at a specific point in the program, *diedie* or *impossible* can be used. The difference between *diedie* and *impossible* is that latter aborts a symbolic trace in the interpreter without an error. Only for compiled execution *diedie* and *impossible* are equivalent.

The statements *assert* and *assure* are used to check if an expression and/or condition holds in a specific point in the program. If the condition is false, the execution of the program will be aborted and an appropriate error message will show up. The difference between *assert* and *assure* is that latter creates path conditions along the execution path during symbolic interpretation. Only for compiled execution *assert* and *assure* are equivalent.

Print and Debuginfo Output Statement

```
print     ::= "print" expression { "+" expression } ;
debuginfo ::= "debuginfo" identifier expression { "+" expression } ;
```

The *print* statement in CASM enables a standard output to the command line. If an output of distinct messages should not always be visible it is possible to define a so called *debuginfo channel* with the *debuginfo* statement, which is deactivated by default and can be activated through a command line parameter of the compiler. Both statements use the plus character as concatenation operator of multiple expressions.

If-Then-Else and Case Statement

```
if        ::= "if" expression "then" statement [ "else" statement ] ;
case      ::= "case" expression "of" case-list { case-list } "endcase" ;
case-list ::= [ "default" | identifier | number | string ] ":" statement ;
```

To branch to a specific point in the program, the *if* statement can be used. According to [32] a *choose* statement was defined to evaluate a statement set in a non-deterministic behavior. CASM uses a *case* statement similar to other high-level languages to have a deterministic behavior and add syntactic sugar to the language which provides an alternative to the *if* statement. The following Listing 3.10 example below shows an example *case* statement:

```
function bar : -> Boolean

rule foo =
{
    case bar of
      true:    print "yes"
      default: print "?"
      false:   print "no"
    endcase
}
```

Listing 3.10: Case Example

Let Statement

```
let ::= "let" identifier "=" expression "in" statement ;
```

In CASM there are no local variables or local states. The *let* binding enables to bind the value of an expression to an identifier. This identifier acts like a local state for the underlying statement and is read-only.

Push and Pop Statement

```
push ::= "push" expression "into" identifier ;
pop  ::= "pop"  identifier "from" identifier ;
```

The *push* and *pop* statement can be used to modify a list-based data-type in CASM. The *push* statement does not only insert a new value to a list, it creates a new update of the list. The *pop* statement does not only return the first value of a list, it removes the value from the list and creates an update of the list. Therefore the list-based parameter in the *push* and *pop* statement has to be an updatable location.

Forall Statement

```
forall ::= "forall" identifier "in" expression "do" statement ;
```

The *forall* statement in CASM allows evaluating the specified statement block in a parallel execution semantics. The parallelism of the statement block is controlled either through a given numerical range or a list of elements. For example it is possible to initialize a function to a specific value by creating multiple updates to the locations which can be expressed with the *forall* statement as:

```
function array : Int -> Int

rule foo =
{
    forall i in [ 10 .. 20 ] do array( i ) := i * i
}
```

Listing 3.11: Forall example

In the example of Listing 3.11, a unary function *array* is updated to its sub-domain from $[10, 20]$ with the square product of the range element. It is important that the updates are produced in parallel execution semantics. Due to the fact that in the sequentialization of such code it may occur that the update $array(15) := 15*15$ is produced before $array(11) := 11*11$ and so on.

Fixpoint Iteration Statement

```
iterate ::= "iterate" statement ;
```

The *iterate* statement can be used in CASM to perform a fixpoint iteration. The iteration aborts when there are no updates produced any more. The statement block of the *iterate* has a sequential execution semantics. Note that if constantly updates were produced, the *iterate* will never reach the fixpoint and the execution results in an infinite loop.

Rule Invocation Statement

```
call ::= identifier
       | "call" identifier [ "(" expression-list ")" ]
       | "call" "(" expression ")" [ "(" expression-list ")" ] ;
```

The *call* statement is used to invoke another rule inside a rule specification. Three different call forms are available. If a rule has no parameter, the *identifier* name of the rule is interpreted as the syntax *call identifier*. These latter forms are known as *direct* rule calling. An *indirect* rule call is possible by passing a value of a rule reference type. It is important to mention that the CASM *call* statement semantics has a call-by-value semantics. The parameters of a rule call are evaluated before the actual call.

Function Update Statement

```
update ::= identifer [ "(" identifier-list ")" ] ":=" expression ;
```

The key statement in CASM is the function *update*. By assigning an expression to a distinct location *loc* := *expr* a new update is produced and properly inserted into the current update-set. Nevertheless, producing and inserting of updates is a non-trivial operation which will be described in detail in Section 4.2.3.

Parallel and Sequential Composition Statement

```
parblock ::= "par" statement { statement } "endpar"
           |   "{" statement { statement } "}" ;
seqblock ::= "seqblock" statement { statement } "endseqblock"
           |       "{|" statement { statement } "|}" ;
```

In CASM parallel and sequential execution semantics can be mixed by using the nested composition of parallel and sequential blocks. Each block can contain one or more statements.

Objdump Statement

```
objdump ::= "objdump" "(" identifier ")" ;
```

Through the *objdump* statement, information about a specific *identifier* can be printed at run-time. Furthermore, it allows printing specific debug information about the implemented object e.g. the address or address space location etc. This statement is mainly used for the CASM compiler development.

3.2.5 Expressions

```
expression ::= "(" expression ")"
             | unaryoperator expression
             | expression binaryoperator expression
             | literal
             | location ;
```

In CASM an expression is composed of either an expression surrounded by parenthesis, an *expression* and a unary *operator*, two expressions combined with a binary *operator*, a *literal* or a *location*.

Operators

```
unaryoperator  ::= "not"
binaryoperator ::= "and" | "or" | "xor" |  "+"  | "-"  |  "*"  | "%"  | "/"
                 |  "="  | "!=" |  "<"  | "<="  | ">"  | ">=" ;
```

The operations are similar to other high-level languages structured into arithmetic, logical and Boolean operators. Due to the use of the additional value `undef`, every operator has an extended operator semantics. There are also some type restrictions for almost all operators, except for the equality and inequality operators. Table 3.1 and 3.2 outline the operator semantics of CASM.

not	`undef`	Boolean
	`undef`	*not* Boolean

Table 3.1: Unary Logic Operator Semantics

$BINOP = \{and, or, xor\}$	`undef`	Boolean
`undef`	`undef`	`undef`
Boolean	`undef`	Boolean *BINOP* Boolean

$BINOP = \{<, >\}$	`undef`	Number
`undef`	`undef`	`undef`
Number	`undef`	Number *BINOP* Number

$BINOP = \{<=, >=\}$	`undef`	Number
`undef`	`true`	`undef`
Number	`undef`	Number *BINOP* Number

$=$	`undef`	Literal
`undef`	`true`	`false`
Literal	`false`	Literal == Literal

$!=$	`undef`	Literal
`undef`	`false`	`true`
Literal	`true`	Literal ! = Literal

$BINOP = \{+, -, *, \%, /\}$	`undef`	Number
`undef`	`undef`	`undef`
Number	`undef`	Number *BINOP* Number

Table 3.2: Binary Logic and Arithmetic Operator Semantics

Location Expression

```
location ::= identifer [ "(" expression-list ")" ] ;
```

The *location* syntax has multiple functionality. The obvious use is the *function lookup*. By specifying a function name and the optional point in its domain the co-domain or *location* can be retrieved e.g. the expression `program(self)` corresponds to a *lookup* to the function *program* at the *self* position. It is important that a *lookup* of a function value has to respect the context in which it was evaluated. By context means either if it was a parallel or a sequential execution block.

If the *identifier* does not equal a function name it can either be a *let* binding name, a CASM *built-in* intrinsic or a CASM *shared* utility operation. In the case of a *let* identifier the value is always statically known during run time, because an expression is bound to the identifier before it is used.

Built-ins

The CASM compiler currently defines the following *built-in* intrinsics:

`die()` This built-in allows aborting the program inside an expression.

`String hex( Int )` Translates an integer decimal number into the corresponding hexadecimal notation. The return value is a string.

`Int pow( Int, Int )` The *pow* intrinsic can be used to calculate the nth power of a given variable base a and exponent n:

$$\texttt{pow}(a, n) = \begin{cases} \texttt{undef} & : a = \texttt{undef} \ \vee \ n = \texttt{undef} \\ a^n & : otherwise \end{cases}$$

`Int rand( Int, Int )` CASM supports through the *rand* built-in an integer based random value generator. The lower and upper bound of the uniform distributed random value is given by an integer value a and b e.g. `rand(21,34)` equals the range $[21, 34]$.

`Boolean symbolic( <T> )` This intrinsic returns the value *true* if an expression of a generic type `T` is symbolic, otherwise the value *false* is returned. This built-in is only implemented in the interpreter.

`Int Boolean2Int( Boolean )` Explicit cast from the value b of type `Boolean` to an `Int` i is given by the translation:

$$\texttt{Boolean2Int}(b) = \begin{cases} \texttt{undef} & : b = \texttt{undef} \\ 0 & : b = \texttt{false} \\ 1 & : b = \texttt{true} \end{cases}$$

`Boolean Int2Boolean( Int )` Explicit cast from the value i of type `Int` to value b of type `Boolean`. The translation convention is based on the C programming language which is:

$$\texttt{Int2Boolean}(i) = \begin{cases} \texttt{undef} & : i = \texttt{undef} \\ \texttt{false} & : i = 0 \\ \texttt{true} & : \textit{otherwise} \end{cases}$$

`Int Enum2Int( <N> )` Explicit cast from an enumeration value n of type `N` to the integer i of type `Int` is given by their order of the identifier set M. The translation is defined as:

$$\texttt{Enum2Int}(n) = \begin{cases} \texttt{undef} & : n = \texttt{undef} \\ i & : n \in M, n \mapsto i \mid 0 \leq i \leq |M| - 1 \end{cases}$$

`<N> Int2Enum( Int )` Explicit cast of an integer number i of type `Int` to an enumeration identifier M_n type `N` with the identifier set M is translated according to:

$$\texttt{Int2Enum}(i) = \begin{cases} M_i & : 0 \leq i \leq |M| - 1 \\ \texttt{undef} & : \textit{otherwise} \end{cases}$$

`List(<T>) app( List(<T>), <T>)` To append an element e of a generic type `T` to a list l of type `List(T)`, CASM provides the built-in *app*. A new list with the appended element is returned after the evaluation of this intrinsic.

$$\texttt{app}(l, e) = \begin{cases} \texttt{undef} & : l = \texttt{undef} \vee e = \texttt{undef} \\ l \cup e & : \textit{otherwise} \end{cases}$$

`List(<T>) cons( <T>, List(<T>) )` The *cons* built-in has the same behavior as the above *app* built-in. The only difference is the arrangement of the built-in parameter signature:

$$\texttt{cons}(e, l) \quad = \quad \texttt{app}(l, e)$$

`List<T> tail( List(<T>) )` The *tail* built-in removes the first element l_0 of generic type `T` from a list l of type `List(T)` and returns the resulting 'tail' of the list:

$$\texttt{tail}(l) = \begin{cases} \texttt{undef} & : l = \texttt{undef} \\ l \setminus \{l_0\} & : \textit{otherwise} \end{cases}$$

`<T> nth( List(<T>), Int )  <T> nth( Tuple(<T>), Int )` To access a specific element e_i of generic type `T` at an index i of type `Int` either in a list l of type `List(T)` or a tuple l of type `Tuple(T)`, the *nth* built-in can be used. The behavior of *nth* is defined as:

$$\texttt{nth}(l, i) = \begin{cases} e_i & : e_i \in l, 1 \leq i \leq |l| \\ \texttt{undef} & : \textit{otherwise} \end{cases}$$

`<T> peek( List(<T>) )` *peek* allows retrieving the first element l_0 of a generic type `T` form a list l of type `List(T)`. The built-in *peek* can also be described through the *nth* built-in:

$$\texttt{peek}(l) \;=\; \texttt{nth}(l, 1) = \begin{cases} \texttt{undef} & : l = \texttt{undef} \\ l_0 & : \textit{otherwise} \end{cases}$$

Shareds

Another important functionality of CASM are *shared* expressions (*shareds*). It is an extensible structure with a distinct interface to define multi-purpose procedures which are shared between the interpreter and the compiler. It is important that *shareds* have no side-effects and produce no updates whatsoever. Currently the major *shareds* are named with a prefix `BV*`. *BV* stands for bit vector. These *BV* operations allow bit-true bit vector manipulations/operations. An example of a bit vector operation is:

`Int BVand( Int, Int, Int )` Performs a bitwise and operation on a given bit width. The first parameter is the width and the last two are the first and the second operand:

$$\texttt{BVand}(w, op1, op2) = \begin{cases} die() & : w = \texttt{undef} \\ \texttt{undef} & : op1 = \texttt{undef} \lor op2 = \texttt{undef} \\ (op1)_2 \land (op2)_2 & : otherwise \end{cases}$$

All available *shareds* are listed in the dissertation from Lezuo in [45].

CHAPTER 4

CASM Run-Time & Code Generator

This chapter describes in detail the new implementation of the CASM compiler. As mentioned in the related work chapter the new run-time and code generation is integrated into the CASM interpreter implementation from Inführ [38]. Since this implementation, a lot of improvements and adoptions were made to design, develop and integrate the new optimized CASM run time and the new code generation which performs a typed AST to C source-to-source translation. At the beginning of the design process of this work, the python-based prototype compiler from [46] was integrated into the interpreter structure to have a first reference implementation and a baseline for later measurements to evaluate the new design. This prototype implementation will be referred from now on as the *legacy compiler*[1]. The legacy compiler has several drawbacks e.g. inefficient translation of ASM functions and the prototypic status made it necessary to re-design the run-time and the code generation to achieve an optimized execution speed of the generated CASM program.

4.1 Overview

Before the details of the legacy compiler are outlined, a general summary of the current interpreter/compiler is given. Like all compilers, the CASM tool (interpreter and compiler) consists of a front-end and a back-end. The front-end is composed of a combined lexer and parser structure which is generated via the open-source tools *(f)lex* and *yacc* [44]. The result is an AST according to the defined syntax from Chapter 3. The AST is processed in the front-end by the annotation process to a typed AST. In the annotation process all used types are identified and checked if they are used correctly in the input program. With a correct and complete typed AST the CASM tool can either interpret or generate code in the back-end for the input CASM program. The following Figure 4.1 presents the current structure of the CASM tool:

[1]includes the legacy run-time and legacy code generator implementation

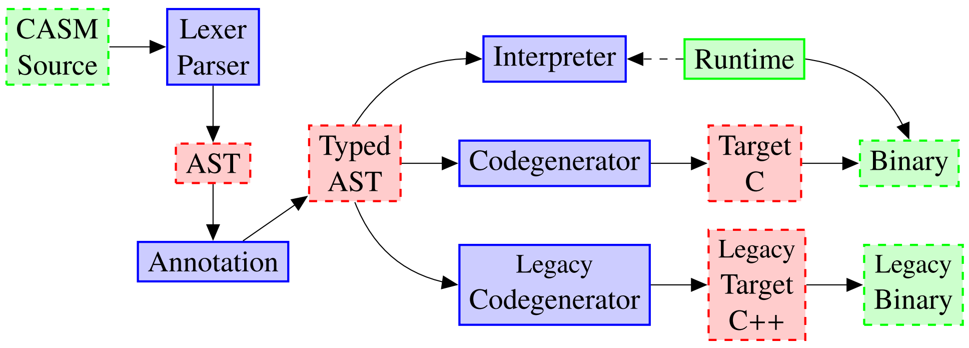


Figure 4.1: CASM Interpreter and Compiler Structure

4.1.1 AST, Annotation and Typed-AST

The CASM AST is a tree representation of the input program. A CASM AST node consists of the following properties: a distinct type, a data content field, a left and a right child pointer, a static and an inferred type, a symbol table entry pointer and a unique counter value. The AST node type is not hierarchic, because of the use of this generic type field which is defined through a C enumeration type.

The annotation process derives for every node which is not typed the corresponding type. This is done by an integrated type *merge* utility which is described by Inführ in [38]. Furthermore, the annotation assures the correct type usage for function values and arguments and it also makes sure that the rule parameters and derived parameters are correct. The result of this annotation is the typed AST which the code generator uses to generate the target C program.

4.1.2 Typed-AST Interpreter

The typed AST interpreter of [38] was constantly adapted by Lezuo [48] to make it compatible to the current run-time implementation. Especially the *shared* mechanism which will be described in Section 4.2.6 is fully integrated into the interpreter. Furthermore, by default the interpreter is activated and is interpreting a CASM input program. By using the command line option `-s` the interpreter performs a symbolic execution of the CASM input program. It should be mentioned, that there is no other ASM tool which enables symbolic execution of ASM specifications. For the sake of correctness the numeric interpretation is used as another CASM reference implementation to analyze the new compiler implementation (see Chapter 6).

4.1.3 Analysis of Legacy Compiler

First of all, the legacy compiler is a proof-of-concept implementation from Lezuo [46]. The run-time is written in C++ and the generated code of a program is C++ too. Due to the heavy use of C++ STL container classes a lot of inefficient behaviors regarding execution time are introduced. The most time consuming part of the execution is spent on in library routines e.g. to re-balance AVL trees, or re-size hash maps etc. Additionally to that the update-set is directly translated to a STL set implementation. CASM functions are only supported up to a 2D domain. For each function domain (0-ary, unary or binary) there is a template implementation. Furthermore, the update and lookup behavior of a function and the update-set modifications are very tightly coupled with the function implementation itself.

The *provider* plug-in which is described in Section 4.2.7 was originally introduced by Lezuo [46]. The new run-time API for the provider differs from the legacy compiler, so all providers are ported to the new run-time implementation. Regarding types, the only generated ones are enumerations. The rest is implemented directly in the run-time or mapped to a corresponding STL container e.g. `List` to `std::list` or `Tuple` to `boost::tuples::tuple`. Another drawback of the C++ run-time and generated code is that the compilation time from C++ to an execution binary is very high. This is because of the heavy use of the C++ template mechanism.

4.2 Run-Time

The new run-time of CASM does not have a clear separation between generated and non-generated components, because some functionality is statically implemented and some of it has to be generated via the code generator. For example special CASM types like `Tuple` are generated and not implemented statically at all. Statical parts of the run-time are implemented in multiple header files. By default every implemented C function in the run-time is marked `static inline`. This property is essential, otherwise the overhead of function calls in the translated program is bigger than the actual computation. Furthermore, everything that is not implemented as a C function is defined through C preprocessor macros. The run-time defines several interfaces for the generated code, external plug-ins, debugging and tracing facilities. Figure 4.2 outlines the composition of the CASM run-time, its generated components and their dependencies.

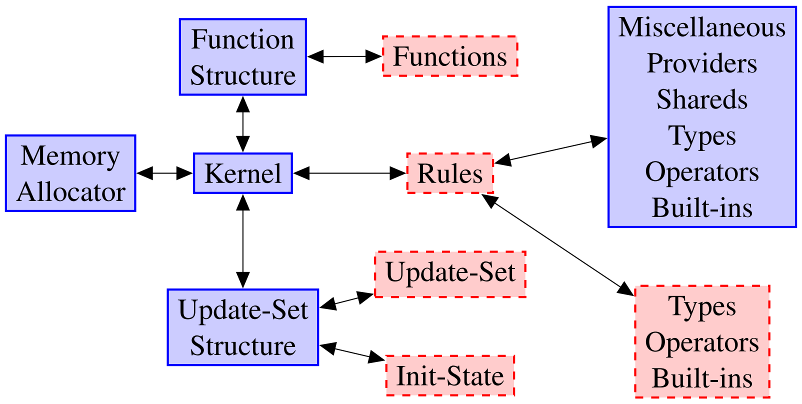


Figure 4.2: CASM Run-Time Components

All components except for the dashed ones are statically implemented in C and the rest is generated through the CASM code generator which is described in Section 4.3. The following subsections describe in detail every listed statically implemented run-time component from the figure above.

4.2.1 Memory Allocator

The need for a specific memory allocator in the run-time implementation arises from the different requirements which an optimized run-time should have - fast allocation of updates, fast deallocation of all updates at once and dynamic memory allocation for `List` and `Tuple` types. In CASM there are two separate regions where memory is allocated. The first one is reserved for static defined variables which are either located in the data section of the binary or the allocation is done directly in a local C function scope. The second region is intended for dynamic memory allocation requests. This is supported in the run-time through a specialized, simple and fast memory allocation algorithm. Three different memory allocation behaviors are supported by this allocation algorithm: *allocate-never-release*, *allocate-release-everything* and *allocate-deallocate*.

The first behavior *allocate-never-release* addresses the need to allocate a bunch of data, kernel memory, etc. dynamically at start-up. Due to the fact that this memory can not be released during run-time, this memory allocation will not be released until to the program is terminated. In the run-time implementation this memory block is called `_casm_mem_global`.

Secondly, the *allocate-release-everything* behavior is used for dynamic memory allocations which can be deallocated at once at a given time instance. In the CASM run-time this moment is exactly after the update-set is applied to the global state, because all updates which were allocated with this behavior are irrelevant after the apply. In the run-time this memory area is named `_casm_mem_stack`, because it is constantly increased and then decreased to an amount of 0. This allocation behavior is very fast, because at start-up a huge memory block is allocated and during run-time only a pointer has to be increased by the allocation byte amount. To perform this allocation algorithm only a few Central Processing Unit (CPU) cycles are needed.

The last behavior *allocate-deallocate* is used for the data-types `List` and `Tuple` to hold their dynamically created data elements and release them if it is necessary. So it can be interpreted as a heap memory and that is why it is called `_casm_mem_heap` in the run-time.

4.2.2 Function Structure

To implement the function specification which is defined in Section 3.2.3 different approaches were made. The basic problem, to represent an n-ary function relation in a bounded linear address memory space is well known for functional programming languages. The first approach was to represent a CASM function with a prefix compressed trie data-structure [6]. But that was discarded very soon in the development, because the memory and execution time overhead for n-ary functions with $n > 2$ was not suitable for an optimized and performance critical run-time, were every CPU cycle is important. Some data-structures inspired by Bagwell [4] [5] have improved the general idea, but the lookup of a multi-dimensional location was too slow. Furthermore, during the development the idea came up to bind a CASM location directly to a fixed memory address in the Random Access Memory (RAM) to directly apply a value to an updated location of a function without re-fetching the same memory address. This idea is named *branding* in the run-time.

The current implementation is based on a simple array hash-map with linear probing [3]

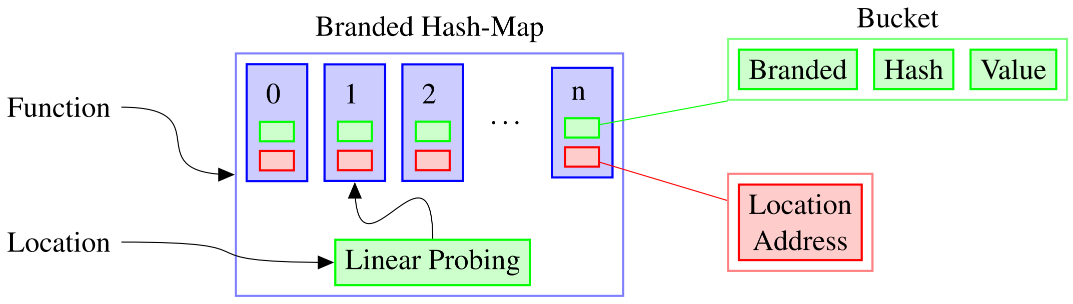


Figure 4.3: CASM Branded Hash-Map Function Structure

and a geometric hashing [69] algorithm to project the n-ary function arguments to a unique integer hash value key. The geometric hashing scheme is generated for every CASM function separately through the code generator (see Section 4.3.1). To achieve the *branding* behavior of a function, an inverse strategy is implemented to the classic hash-map data-structures, where a bucket of the hash-map is marked as used (*branded*), when the bucket is looked up and not when it is set. This behavior is based on the fact that during run time a function has always to be looked up first, even when it is updated, because an *update* in the CASM run-time has to lookup the unique memory address location of the updated function (see 4.2.3). Therefore the data-structure is named *branded hash-map*. Figure 4.3 outlines the *branded hash-map* data-structure. This generalized approach to represent n-ary function relations is used especially for domains with a large size like **`Int`** and **`String`** data-types which are commonly used in CASM programs. But the other types which have a statically known size like **`Boolean`**, **`RuleRef`**, etc. can be optimized by translating them directly to C arrays to avoid the hashing overhead and to achieve more performance (see Section 4.3.1).

To support different CASM function representations, a C structure named **`casm_function`** is used which is currently only composed of a **`void`** pointer field *state*. This *state* field stores the memory address of the statically (data section) or dynamically (CASM *global* memory area) allocated function structure which equals in ASM terms the global state of the function. All functions together form the global state of the CASM input program and are generated in the code generator through a C array of the **`casm_function`** type. The address of one function is obtained by the macro **`CASM_LOCATION(NAME)`** where **`NAME`** is the function name. Furthermore, the run-time also defines the macros **`CASM_TABLE_READ(LOCATION, KEY, VALUE)`** and **`CASM_TABLE_WRITE(LOCATION, KEY, VALUE)`** which are wrapper to the underlying *branded hash-map* structure to the function location **`LOCATION`** that can be fetched through **`CASM_LOCATION()`**. The **`KEY`** parameter is the geometric hashed function argument value and the parameter **`VALUE`** is either the read or the written CASM value from the *branded hash-map* function structure.

A function getter and setter interface enables direct access to the global state of a function. The code generator uses this interface to define the specified functions of the input program. The getter interface should not be confused with a lookup of a location, because a lookup should not directly access the global state before the update-set is checked first for partial updates (see Section 4.2.3). For a function also a *printer* and *objdump* interface can be defined (objdump statement, see Section 3.2.4). The getter, setter, printer and object dump interface is defined through the following EBNF syntax:

```
setter  ::= "DEFINE_CASM_FUNCTION_SET" "(" identifier "," param { "," param } ")" "{" "}" ;
getter  ::= "DEFINE_CASM_FUNCTION_GET" "(" identifier "," param { "," param } ")" "{" "}" ;
printer ::= "DEFINE_CASM_FUNCTION_PRINT" "(" identifier "," "void" "*" "value" ","
                                          "uint8_t" "def"  ")" "{" "}" ;
objdump ::= "DEFINE_CASM_FUNCTION_OBJDUMP" "(" identifier ")" "{" "}" ;
param   ::= "ARG" "(" type "," identifier ")" ;
```

4.2.3 Updates, Pseudo State & Update-Set

Besides the function management the creation and handling of updates are the major and crucial operations for the run-time.

Updates

A CASM *update* is represented through a C structure named `casm_update` that has to be declared with the macro `CASM_UPDATE_TYPE()`. Because the update structure has a pointer parameter field *args* and this field represents a *uint64_t* array and the actual size is calculated in the code generator. It is using it to store a compressed form of the argument values (packed arguments) to the update structure. The setter function uses it to speed up the access to a function location. Furthermore, this field also is used in the function *printer* interface to decode the update arguments (see Section 4.2.2). The other `casm_update` structure fields are a `Generic` type value named *value*, an Unique Identifier (UID) for the updated function named *func* and the *line* number from the input program where the update was contained for debug and error information purposes. To perform an update the *statement* macro `CASM_UPDATE()` is used which will be described in the following sub-section.

Pseudo State

An important property of an update is the so called *pseudostate* counter. It tracks the parallel and sequential composition nesting depth of a CASM program during run-time and is stored inside the *update-set* structure which will be described in the following sub-section. An even *pseudostate* counter value equals a parallel execution block, an odd value equals a sequential execution block. This property is assured inside the *fork* and *merge* statements of the run-time (see Section 4.2.3). Additionally the fork/merge statements of the run-time are incrementing/decrementing the *pseudostate* counter by every entering/exiting of a parallel to sequential or a sequential to parallel composition block. Listing 4.1 illustrates the pseudo state in- and decrementing.

```
rule r =
{                  // <-- 0
    {|             // <-- 1
    |}             // <-- 0
    {|             // <-- 1
        {          // <-- 2
            {|     // <-- 3
            |}     // <-- 2
        }          // <-- 1
    |}             // <-- 0
}
```

Listing 4.1: Pseudo State Counter

Update-Set

Through an update-set structure all updates are handled in the correct parallel and sequential execution semantics. The run-time implements a single update-set structure which supports the following functionality:

function update create new updates and add them to the update-set

forking every entry from the parallel to sequential or from sequential to parallel execution semantics the update-set has to create (*fork*) a new update-set for the underlying composition block

merging every exit from the parallel to sequential or from sequential to parallel execution semantics the update-set has to *merge* the inner update-set with the outer one

function lookup a lookup of a function location has to check the update-set first if a sequential update has been inserted before

applying writes all created updates after the execution of one step to the global state

The update-set itself is implemented as a C structure (`casm_updateset`) with an `uint64_t` field *pseudostate* and a `linked_hashmap*` field *set*. A *linked hash-map* [23] is a hybrid data-structure. It is a combination of a hash-map and a linked list data-structure. Every new inserted value in the hash-map is linked to a previous value. For an optimized execution behavior a single linked list data-structure is used pointing where new elements are inserted to the start of the list. The *hash-map* implementation maps an `uint64_t` *key* to a `void*` *value*. The hash function for the hash-map *key* is an optimized case for CASM. It is constructed by setting the upper 48 bits to the memory address location of an updated function and the lower 16 bits to the current pseudo state counter. This hash-function implies that the pseudo state counter is limited to $2^{16} - 1$ nested composition blocks. If this maximum is reached, the CASM program aborts with an error message. The 48 bits for the location memory address is justified, because modern CPU architectures currently use a maximum virtual address size of 48 bits. Figure 4.4 on page 36 visualizes the update-set data-structure.

This `casm_updateset` structure is defined and allocated in the CASM kernel *initialization* phase (see Section 4.2.4) and the address of this update-set is passed to every rule call and every update-set API function.

The `CASM_UPDATE()` macro implements the *function update* functionality (update statement, see Section 3.2.4). A new update is allocated always in the CASM *stack* memory area (see Section 4.2.1). This macro initializes an update with the new value, an UID of the updated function and debug information. The function location gets packed into the *args* field of a `casm_update` structure. The hash value of the *args* field is used to access the update-set structure. There are two cases – parallel and sequential access. If the update-set is accessed in a parallel execution block and the key already exists, the run-time has detected a conflicting update. If the update-set is accessed in a sequential execution block, an existing key in the update-set

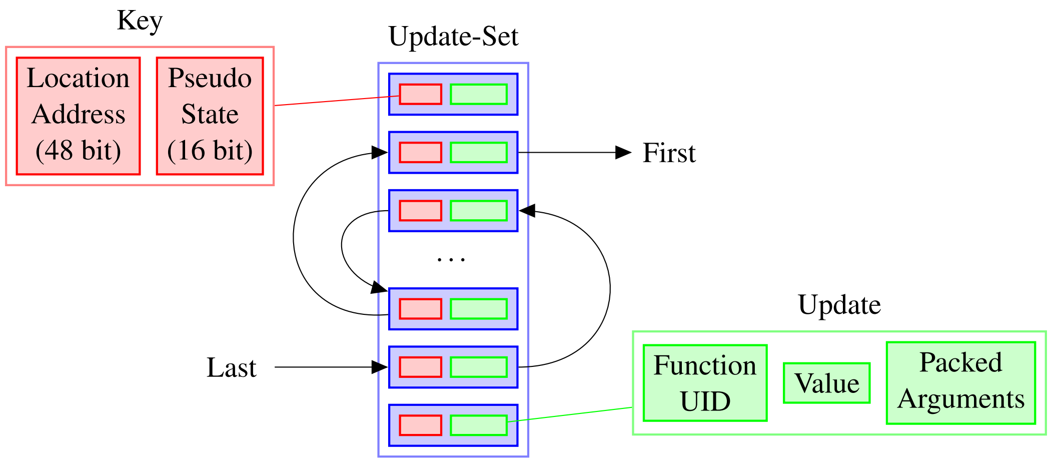


Figure 4.4: CASM Linked Hash-Map Update-Set Structure

overwrites the value. The update-set implements directly the sequential merge at update creation. This speeds up the later mentioned *merge* operations.

The **`CASM_UPDATESET_FORK_PAR()`** and **`CASM_UPDATESET_FORK_SEQ()`** *fork* functions implement the *forking* functionality (parallel and sequential composition, see Section 3.2.4) of the update-set. If the current execution semantics is parallel, the **`SEQ`** *fork* has to be used and vice versa. Both functions increment the *pseudostate* counter value of the update-set structure. Additionally they check via a C **`assert()`** if the *fork* is valid. An invalid fork would be if e.g. the current composition is parallel and a nested parallel composition block forked. The same applies for the sequential case.

The **`CASM_UPDATESET_MERGE_PAR()`** and **`CASM_UPDATESET_MERGE_SEQ()`** *merge* functions implement the *merging* functionality (parallel and sequential composition, see Section 3.2.4) of the update-set. Both functions' decrement the *pseudostate* counter value from the update-set structure. Those functions assure valid merges from either **`SEQ`** to **`PAR`** or **`PAR`** to **`SEQ`**. A merge modifies the key of the updates. If the pseudo state from a key is greater or equal the current pseudo state it gets decremented by one. If an update has a smaller pseudo state the merge is done. After a key is decremented it gets inserted into the update-set again. If a merge from sequential to parallel is performed, *conflicting updates* can occur. Otherwise the keys can be overwritten.

Besides the *function update*, the *function lookup* functionality (location expression, see Section 3.2.5) of the update-set is the most critical operation regarding execution time. Because the amount of lookups of function locations is always greater or equal to the amount of updating function locations. Even if only one update is performed in a CASM program, the run-time has to lookup the location memory address to perform the update. The **`CASM_LOOKUP()`** macro retrieves the current semantically correct (sub-machine) state of a function location. Semantically correct (sub-machine) state means that if the lookup is executed in a nested composition block, the lookup has to check the update-set first if there was a sequential update to the lookup location. This behavior is implemented by getting the location memory address and the global state first. With the retrieved address the lookup can check the update-set for possible sequential updates. If an update is found the lookup returns the value. Otherwise the lookup returns the global state.

The *applying* functionality of the update-set is done by the C function `CASM_UPDATESET_APPLY()`. It iterates over all updates in the *updateset* structure by using the linked list behavior. Every iteration writes the update value to the location address. The *apply* of an update-set is called from the CASM kernel in the *step* phase after the top-level rule has returned (see Section 4.2.4).

In the CASM kernel *start-up* phase it is possible to pass a command line parameter which enables a step-based output of the update-set before it is applied to the global state. This *printing* functionality is implemented in the C function `CASM_UPDATESET_PRINT()`. It iterates over all updates and calls the generated *printer* of the corresponding updated function.

The `CASM_DUMPING_UPDATES()` macro prints the produced updates at the end of a rule for the used `dumps` specification (see Section 3.2.3). It simply iterates over all updates and checks if an update has the same function `NAME` and if so an internal `debuginfo` message is printed (see Section 4.2.8).

The code generator uses the macro `CASM_UPDATE_INITIALLY()` to directly write into the global state of a CASM program by generating sequences of this macro calls into the initialization file (see Section 4.3.1).

4.2.4 Kernel

The *kernel* of the CASM run-time controls the execution of the translated CASM program. Therefore the kernel has to perform the following phases: *start-up*, *initialization*, *step* and *finalizing* phase.

In the *start-up* phase of the CASM kernel, some data-values e.g. to track the executed CASM steps are defined. Additionally the command line passed to the CASM program is checked. The available commands of the kernel are listed in the following section. After the *start-up* the *initialization* phase takes over. It defines and allocates CASM memory regions, allocates and initializes the update-set, allocates the global state, initializes the global state from the generated *initially* component (see Section 4.3.1) and sets the top-level rule to the specified `init` rule (see Section 3.2.3). After that the kernel enters the main execution loop which is the *step* phase. This phase checks first if the current execution step counter is smaller than the maximal defined one. If the maximal value is reached the loop is aborted, otherwise the current top-level rule is called. After the return of the top-level rule, the resulting update-set is applied to the global state. The top-level rule is loaded through a lookup of the function `program(self)` and if the value is undefined the main loop aborts. If not, the step counter is incremented and the body of the main loop is repeated. When the loop aborts the kernel enters the *finalizing* phase. This phase first prints the amount of performed CASM steps and then it releases all allocated memory blocks and terminates the CASM program.

To include the kernel in a C program the run-time provides the macro `CASM_MAIN(INIT_RULE)`. This macro defines the C `main()` function and it includes all the CASM kernel phases which are described in the above paragraph. Furthermore, through the parameter `INIT_RULE` the `init` rule identifier can be specified.

Kernel Command Line Options

`-u` Prints after every return of the top-level rule the calculated update-set with the current step counter value to the standard output stream.

`-d <arg>` Enables the `debuginfo` channel-based messages. The <*arg*> parameter can be a comma separated string with multiple channels of the CASM program. If <*arg*> equals the string "all", all `debuginfo` channels are activated at once. Per default all channels are deactivated.

`-M <arg>` This option allows defining an upper bound of executed CASM steps. If <*arg*> equals 0, the top-level rule will not be executed. Per default this maximal step counter value is defined as the greatest 64 bit unsigned integer value (`UINT64_MAX`).

`-updateset-size <arg>` Through this command line option the compiled CASM program can use the specified start size <*arg*> for the update-set initialization. Otherwise a default value will be used by the kernel.

`-q` Disables all outputs of the compiled CASM program.

`-s` Outputs after every CASM step the memory consumption of the three memory areas and a statistic of the update-set structure. If the CASM program is also generated with a special command additional access information and timings are printed (see Section 4.2.8).

`-v` Enables the verbose mode of the run-time. For example it outputs the update-set with an internal representation after every CASM step.

`-V` Outputs the source CASM file name, the used command line of the code generator to generate the CASM program, the time stamp and the used include paths and libraries for compilation.

4.2.5 Types, Operators & Built-ins

The only assumption in the run-time implementation is a 64 bit host computer architecture to represent pointers and every numeric value in one generic C `void*` data field.

Types

Upon the architecture assumption an internal type named `Generic` is defined which is implemented as a C structure of a `void` pointer (64 bit) *value* and an `uint8_t` unsigned integer (8 bit) *defined* field:

```c
typedef struct _Generic
{
    void* value;
    uint8_t defined;
} Generic;
```

Listing 4.2: Generic Type

So the *Generic* type can fulfill all the desired properties from the CASM syntax, that a type can be either undefined or defined through the *defined* structure field. By setting the *defined* field to 0 (`FALSE`) the type is undefined. Setting it to 1 (`TRUE`) equals a defined type. The run-time uses the upper 7 bits of the *defined* field to perform internal checks, so the overhead of this memory consumption is negligible. For the *value* field, primitive types can be directly casted and stored into this memory location, only reference-based data-types use this *value* field as a C pointer and follow the indirection. The run-time supports via type interface functions direct translations from `Generic` to a specific type and vice versa. Every CASM type except the enumeration, the ranged integer, the *List* and the *Tuple* type is statically implemented in the run-time. The other types are either partial or not implemented at all, because they are generated.

Table 4.1 shows the different used C types (*CT*) to represent all other CASM types (`T`) in the same way as the defined C structure in the above Listing 4.2 for the `Generic` type.

		`Undef`					
		`Self`	`Int`				`List(T)`
`T`	`Generic`	`Boolean`	`Int()`	`String`	`RuleRef`	`enum N`	`Tuple(T)`
CT	void*	uint8_t	int64_t	char*	(*)()	enum N_type	T*

Table 4.1: C type mapping of CASM value field types

Undefined (`Undef`) As mentioned in the language chapter the `Undef` type is only used internally to represent an unresolvable annotation of an AST node. For example if the `undef` literal can not be annotated in a given expression the type `Undef` is used. The *value* of this field is always 0 and is unused during execution.

Agent Reference (`Self`) Currently the `Self` type is just a representation of one single ASM execution agent. The *value* field is unused and has always the value 0. In a later implementation this type can evolve to a type e.g. `Agent` to represent the complete set of all possible running agents.

Boolean (`Boolean`) The *value* field of the `Boolean` type follows the C Boolean expression convention. A 0 equals *false* and not 0 equals *true*.

Integer (`Int, Int()`) The normal full range integer type `Int` is currently implemented through a 64 bit signed integer type. As for the ranged integer type `Int()` there are two implementations. One uses the same C type as the `Int` type and another one generates a special ranged integer type with the only necessary bit-with of either `uint8_t`, `int8_t`, `uint16_t`, `int16_t`, `uint32_t`, `int32_t`, `uint64_t` or `int64_t`. A ranged integer is defined by the code generator through the use of the macro `DEFINE_Int( FROM,TO,MIN,MAX,CTYPE)` (see Section 4.3.1).

String (`String`) Strings in CASM are directly mapped to character pointers. Furthermore, the content of a `String` is besides a calculated `String` at run-time always constant data. The C compiler translates C strings directly as constants into the text section of the binary.

Rule Reference (`RuleRef`) As the type name implies, a `RuleRef` stores rule references. The exact type in the run-time is a function pointer to a generated rule of the input program (see Section 4.3.1).

Enumeration (`enum`) The enumeration type and their operators and built-ins are implemented via a generic macro. The code generator uses the macros `DEFINE_Enum(NAME, SIZE)` and `DECLARE_Enum(NAME, SIZE)` to define/declare a specific type and an implementation of the specified enumeration `NAME` is integrated into the target C program (see Section 4.3.1).

Tuple (`Tuple`) For the `Tuple` type there is no support from the run time at all, because this type itself, its operators and built-ins are completely generated (see Section 4.3.1).

List (`List`) The `List` type in CASM is implemented via a generic macro. The code generator creates for every list-based data-type a separate instance of the list implementation by using the `DEFINE_List(TYPE)`. The parameter `TYPE` is the sub-type of the list data-type e.g. `List(Int)` or `List( Tuple(Int,String,RuleRef) )`.

Type Helper Functions

The run-time type helper functions provide a dedicated interface to initialize, modify, cast, allocate and print CASM data-type C variables. Especially the code generator uses this interface to handle all different CASM types in a unified way. The naming schemes of the interface is defined as `CASM_<OP>_<T>()`. *<OP>* stands for the operation and *<T>* for the implemented type. Currently CASM consists of 7 type helper functions which are:

`void CASM_undef_<T>(<T>* reg)` The *undef* API function can be used to set a CASM object to undefined. It sets the *defined* field of the parameter *reg* of type *<T>* to `FALSE` and the *value* field to 0.

`void CASM_const_<T>(<T>* reg, <CT> val)` To set a specific defined value for a CASM type the *const* function can be used. This function is defined for every CASM type with the explicit C type except for the `Undef` type. The placeholder *<CT>* corresponds to the mapping which is mentioned in Table 4.1 e.g. if `T = Int` then is *<CT>* = `int64_t`. The function writes the constant value *val* directly into the *value* field of the parameter *reg*. The *defined* field is set to `TRUE`.

`void CASM_assign_<T>(<T>* reg, <T>* val)` Through the *assign* function a type-based assignment is implemented. All primitive types perform a simple C structure assignment. For non-primitive types the assignment is not a trivial operation. Furthermore, a different implementation of the assignment for e.g. the `List` type can improve the performance of an executed CASM program. This function is not provided for the `Undef` type.

`<T>* CASM_alloc_<T>(<T>* reg)` The *alloc* function returns the actual memory address of the parameter *reg* address. For primitive types this function just returns the incoming address. Non-primitive types can use this function to make their data which is located in the CASM *stack* memory area persistent by transferring the data to the CASM *heap* memory area.

The function returns as result the new address (see Section 4.2.1). This function is called in the update implementation for a CASM function (see Section 4.2.2) which is generated through the code generator (see Section 4.3.1). This function is unsupported for the `Undef` type.

`void CASM_print_<T>(<T>* reg)` The *print* function enables a formatted output of a CASM value. If a value is undefined the string “undef” is written to the standard output stream. Otherwise the formatted value is printed. By format means that the internal numeric value of a type is printed in a readable format. For example an enumeration type `Flags` defined as `enum Flags = { A, B, C }` returns the string “B” if the *value* field of the parameter *reg* contains 1. This function is not provided for the `Undef` type.

`void CASM_cast_<T>(<T>* reg, void* value, uint8_t defined)` Casting in the CASM run-time is defined and performed explicitly. With the clear definition of the CASM types through C structures a type mismatch can be avoided. In some points in the program the run-time needs a cast from the `Generic` to a specific CASM type.

`CASM_<T>_CAST` The inverse explicit cast from a CASM type to a `void*` value is supported through a partial type-based macro. It is defined statically for all types and corresponds to the C cast from a CASM type <T> to a 64 bit integer value. The run-time uses this macro in several places to cast a *value* field of a CASM type <T> to a `void*` value e.g. a cast from an `Int` value `int64_t` to a `Generic` data-type value `void*` is implemented over the *value* field of the data-types as `(void*) CASM_Int_CAST reg->value` where *reg* is an `Int` C variable.

Logical & Arithmetic Operators

The logical and arithmetic operators in the CASM run-time are implemented accordingly to the specified semantics from Section 3.2.5. The naming scheme of the operator functions is defined as `CASM_<OP>_<T>_<U>_<V>()`. *<OP>* stands for the operation, *<T>* and *<U>* are the operand types and *<V>* is the return type.

`void CASM_not_Boolean_Boolean(Boolean* op1, Boolean* target)` The *not* operator is the only unary operator in CASM.

`void CASM_<OP>_Boolean_Boolean_Boolean(Boolean* op1, op2, target)` This function signature represents in the run-time the `Boolean` logical operators where the placeholder *OP* can either be `and`, `or` or `xor`.

`void CASM_<OP>_<T>_<T>_Boolean(<T>* op1, <T>* op2, Boolean* target)` The comparison of all types in CASM for equality and inequality is implemented through this function signature where the placeholder *OP* can be either `eq` (equal) or `neq` (not equal).

`void CASM_<OP>_Int_Int_Boolean(Int* op1, Int* op2, Boolean* target)` Logical operators of the `Int` type are grouped into this function signature. The *OP* placeholder can be either `les` (less), `leq` (less or equal), `gre` (greater) or `geq` (greater or equal).

`void CASM_<OP>_Int_Int_Int(Int* op1, Int* op2, Int* target)` The arithmetic operators in CASM are only defined for the `Int` type. Placeholder *OP* can be either `add` (addition), `sub` (subtraction), `mul` (multiplication), `div` (division) or `mod` (modulo). In the division operator a run-time error is raised if the denominator value is defined and the numerical value equals 0.

Built-ins

As described in Section 3.2.5, the run-time defines all the specified built-ins except for the `Tuple` type, because this type and its operators are generated. The run-time implements the built-ins as:

`void CASM_die(void)` To abort inside an expression the intrinsic *die* is used. Before the program is aborted an error message is printed to the standard output stream.

`void CASM_hex_Int_String(Int* i, String* s)` The run-time converts the incoming integer to a string and forms the decimal value into a hexadecimal notation. The *char** inside of the `String` is allocated in the CASM *stack* memory area.

`void CASM_pow_Int_Int_Int(Int* base, Int* exp, Int* target)` To perform an integer exponentiation the *built-in pow* (to the power of) can be used. If the *base* value or the *exp* (exponent) value is undefined the *built-in* returns an undefined value, otherwise a defined *target* value is calculated and returned as:
`target->value = (int64_t) pow((double)base->value, (double)exp->value);`

`void CASM_rand_Int_Int_Int(Int* start, Int* end, Int* target)` The run-time supports the integer random variable generator through the *rand* intrinsic. It is important that this *built-in* relies on the C function `rand()` and the run-time *kernel* seeds the C random value generator with the current time stamp the beginning of the execution in the *initialization* phase with the C function `srand()` (see Section 4.2.4). If the *start* value and/or the *end* value are undefined the intrinsic returns an undefined value, otherwise it generates a defined *target* value accordingly to the following formula:
`target->value = (int64_t)(start->value + (rand()%(end->value - start->value + 1)));`

`void CASM_Boolean2Int_Boolean_Int(Boolean* b, Int* i)` The cast from `Boolean` to an `Int` is always possible, because the value domain of the `Boolean` type is a sub-set of the `Int` type.

`void CASM_Int2Boolean_Int_Boolean(Int* i, Boolean* b)` For the cast from an `Int` to a `Boolean` type in CASM the run-time performs the C Boolean expression convention where 0 equals *false* and not 0 equals *true*. The following calculation shows the implementation in the run-time: `b->value = (i->value != 0 ? TRUE : FALSE)`

`void CASM_Int2Enum_Int_<N>(Int* i, <N>* e)` The cast from an `Int` to an enumeration type `N` is statically defined in the run-time. Enumeration types are generated and every enumeration size is defined through an extra label *<N>_SIZE* which is the last element in a C enumeration type. The index of the label *<N>_SIZE* in the enumeration directly corresponds to size of the enumeration type. If the incoming integer *i* is outside of the range [0, *<N>_SIZE*[then the result value *e* is undefined, otherwise the index of the enumeration label is assigned.

`void CASM_Enum2Int_<N>_Int(<N>* e, Int* i)` The mapping of a generic enumeration type `N` to an `Int` is trivial. The index of the enumeration value equals the integer value. If the enumeration *e* is undefined, then the integer *i* is undefined as well.

`void CASM_app_List_<T>_<T>_List_<T>(List_<T>* l_in, <T> *e, List_<T>* l_out)` The intrinsic *app* (append) adds an element *e* of type `T` to an incoming `List` *l_in* with the sub-type `T` and writes the result to *l_out* which has the same list data-type as *l_in*. Currently the run-time just copies internally the list and extends it by the element *e*. The allocation of the copied list is located in the CASM *heap* memory area (see Section 4.2.1).

`void CASM_cons_<T>_List_<T>_List_<T>(<T>* e, List_<T>* l_in, List_<T>* l_out)` As mentioned in the Section 3.2.5 the *cons* (constructs) intrinsic performs the same operation like *app*. The only difference is that *cons* uses a mirrored parameter signature.

`void CASM_tail_List_<T>_List_<T>(List_<T>* l_in, List_<T>* l_out)` The *tail* intrinsic removes the first element of an incoming `List` *l_in* with sub-type `T` and returns the resulting list by setting the *l_out* parameter. Internally the list is copied from the second element at index 1 to the end. The allocation of the copied list is located in the CASM *heap* memory area (see Section 4.2.1).

`void CASM_nth_List_<T>_Int_<T>(List_<T>* l, Int* i, <T>* e)` To access a specific position *i* of type `Int` in a `List` *l* of sub-type `T` the intrinsic *nth* can be used. The run-time accesses the list element *e* of type `T` and returns it.

`void CASM_peek_List_<T>_<T>(List_<T>* l, <T>* i)` The *peek* built-in retrieves the first element of a `List` *l* of sub-type `T` and returns it via the parameter *i* of type `T`. This intrinsic is implemented via the *nth* built-in by accessing the first position of the list *l*. The internal call equals `nth(l, 1)`.

4.2.6 Shareds

```
shared ::= "DEFINE_CASM_SHARED" "(" identifier "," param { "," param } ")" "{" "}" ;
param  ::= "ARG" "(" type "," identifier ")" ;
```

The CASM *shared* interface is a side effect free way to extend the CASM run-time statically with additional operators or calculation procedures directly in C. Side effect free means, that it is impossible to modify the CASM global state within a *shared* procedure call, because *shareds* equal an expression in CASM.

In the above EBNF *shared* syntax the *identifier* has to be unique in the program. The first parameter in the *shared* interface is the return value. Additional parameters can be added to the definition. To call a *shared* the run-time provides the macro `CASM_CALL_SHARED(NAME, VALUE, ARGS...)`.

Currently this interface is used for defining a bit-vector bit-true arithmetic operator library for the `Int` type. These *shared* functions are prefixed with *BV*. *BV* is the abbreviation for bit-vector. This library implements about 40 to 50 operations which are used by Lezuo in [45] to have a computer architecture model independent bit-vector operator set.

Section 3.2.5 presents the language syntax and semantics of the *shared BVand*. Listing 4.3 outlines the run-time implementation of this *BVand* with the *shared* interface definition.

```
DEFINE_CASM_SHARED(BVand, ARG(Int,ret), ARG(Int,width), ARG(Int,op1), ARG(Int,op2))
{
    // implementation
}
```

Listing 4.3: Shared BVand Definition Example

4.2.7 Providers

A *provider* in CASM is equal to a program extension but directly implemented in C. The difference to the *shared* interface is that a *provider* functionality can have side-effects. A *provider* can implement external functions, rules and define new *debuginfo* channels for the program where the provider is used. Furthermore, it is possible to tell the compiler that the *provider* needs additional source files to compile the provider correctly. It is also possible to specify pre- and post-hooks which will be called before/after the execution of the CASM program in the run-time kernel. The *provider* identifier (see Section 3.2.3) is used for the provider C header file name where all the intended *provider* functionality shall be defined. The following subsections describe in detail all provider functionality.

Provided Functions

```
provFunc  ::= { funcGetH [ funcSetH ] } ;
funcGetH  ::= "DEFINE_CASM_FUNCTION_SET" "(" identifier "," param { "," param } ")" "{" "}" ;
funcSetH  ::= "DEFINE_CASM_FUNCTION_GET" "(" identifier "," param { "," param } ")" "{" "}" ;
funcObjH  ::= "DEFINE_CASM_FUNCTION_OBJDUMP" "(" identifier ")" "{" "}" ;
param     ::= "ARG" "(" type "," identifier ")" ;
```

Sometimes the programmer of a CASM model has more knowledge about a specific *function* behavior then the compiler itself. For example, sometimes a function is only defined in special sub-ranges or only on distinct values in the domain. Through the *funcGetH* and *funcSetH* syntax 'external' functions with a unique *identifier* name in the used input program can be defined and implemented. It is important that this is exactly the same run-time interface as the code generator uses to generate the functions from the input program (see Section 4.2.2 and 4.3.1).

Furthermore, the *printer* function interface has not to be defined in a provider, because the code generator generates the *printer* function. The *objdump* interface specified through *funcObjH* is optional. The setter (*funcSetH*) specification is optional. If it is not defined, the compiler will treat the provided function as a read-only one which equals the function properties *static* and *controlled* (see Section 3.2.3). Listing 4.4 shows an example provided function *thesisYear* which is a defined constant read-only 0-ary function and returns always the constant value 2014.

```
DEFINE_CASM_FUNCTION_GET(thesisYear, ARG(Int, ret))
{
    static int64_t year = 2014;
    CASM_const_Int(ret, year);
    return &year;
}
```

Listing 4.4: Provided function example

If a setter is provided too, the memory location of the function should be in a separate compilation unit e.g. a provided rule or an initialization pre-hook.

Provided Rules

```
provRule ::= { ruleH ruleC } ;
ruleH    ::= "DECLARE_CASM_RULE" "(" identifier { "," param } ")" ";" ;
ruleC    ::= "DEFINE_CASM_RULE"  "(" identifier { "," param } ")" "{" "}" ;
param    ::= "ARG" "(" type "," identifier ")" ;
```

Through the provided rule specification *ruleH* in the *provider* header file the compiler integrates the rule *identifier* name into the rule reference set of the generated program. The implementation of a provided rule has to be in a separately C source file with the same name as the rule *identifier* which contains the *ruleC* specification implementation. The annotation process checks if there is an implementation provided and if there is none the compiler reports an error.

Provided Debuginfo Channels

```
provDebugChannel ::= { provDebugChannel } ;
debugChannelH    ::= "DECLARE_CASM_DEBUG_CHANNEL" "(" identifier ")" ";" ;
```

If a *provider* uses inside a provided rule a *debuginfo* statement, the code generator can not see it and has no knowledge about the used *debuginfo* channel if it is not used in the input program. The *debugChannelH* syntax makes the used channel visible to the code generator. This definition has to be in the *provider* header file.

Provided Pre- and Post-hooks

```
provHook  ::= { preHookH preHookC } { postHookH postHookC } ;
prehookH  ::= "DECLARE_CASM_INIT_FUNCTION" "(" identifier ")" ";" ;
prehookC  ::= "DEFINE_CASM_INIT_FUNCTION"  "(" identifier ")" "{" "}" ;
posthookH ::= "DECLARE_CASM_DEINIT_FUNCTION" "(" identifier ")" ";" ;
posthookC ::= "DEFINE_CASM_DEINIT_FUNCTION"  "(" identifier ")" "{" "}" ;
```

If there is a request to e.g. allocate special memory, load huge chunk of data, pre-set the global state in a specific pattern the init functions can be created through the *prehookH* syntax. The inverse operation to finalize or deallocate a special memory block the *posthookH* syntax can be used inside the *provider* header specification file. For every defined hook function there exists a C source file with the same name as the *identifier*. The implementation of the hook is done via the syntax *prehookC* and/or *posthookC*. Internal in the run-time the init and de-init functions are C functions with no parameters and no return type. The CASM kernel invokes all provided init functions in the *initialization* phase and all provided de-init functions in the *finalizing* phase by invoking an array of function pointers to the defined hooks (see Section 4.2.4).

Provided Additional Sources

```
provAddSrc ::= { addSrcH } ;
addSrcH    ::= "DECLARE_CASM_ADDITIONAL_SOURCE" "(" filename ")" ";" ;
```

The *addSrcH* syntax is used by the compiler to include the mentioned source *filename* into the compilation from C to binary. This definition has to be in the *provider* header file.

The MIPS Provider

The MIPS provider is a reimplementation of the original MIPS provider from Lezuo in [48]. This provider includes the loading of Executable and Linking Format (ELF) files, a RAM representation and an instruction decoding facility of the MIPS computer architecture. In the current version of the CASM tool the MIPS provider consists of the functions *PMEM* (program memory), *PARG* (program arguments) and *MEMORY*. Through an init pre-hook function named *mips_load_elf* it is possible to load via the provider defined command line argument *–mips <arg>* an ELF file into the modeled RAM memory block function *MEMORY*. Further the lookup of the global state from the function *PARG* and *PMEM* activates helper C functions in the *mips* provider to perform the instruction decoding. Furthermore, there is a *debuginfo* channel *monitor* and a provided rule *mips_monitor* defined. This rule is used to model and translate the MIPS syscall[2] (system call) instruction to the host computer system call.

4.2.8 Printing, Debugging & Tracing

The run-time supports the `print` statement (see Section 3.2.4) by type based *print* functions which are described in Section 4.2.5. Furthermore, the run-time has an equivalent macro to the C *printf* function with the name `CASM_PRINTF(FORMAT, ARGS...)` which can output formatted text. This macro checks internally if the *-q* option from the CASM kernel is disabled, otherwise the text will be suppressed. All output facilities use this macro as its basis. Additionally to end a `print` statement and to output a line feed the run-time defines the macro `CASM_PRINT_END`.

For the `debuginfo` statement inside a CASM program the macros `CASM_ DEBUG_START(CHANNEL)` and `CASM_DEBUG_END(CHANNEL)` are provided. By surrounding a C code-block with these macros, it only gets executed if the channel `CHANNEL` was activated in the CASM kernel *start-up* phase (see Section 4.2.4). The code generator uses a combination of `CASM_PRINTF()` and the `CASM_DEBUG_*()` macros to generate a `debuginfo` statement.

The debugging of the run-time is currently implemented by a simple macro-based approach where in every interface, operator, function, built-in, shared, etc. The macro `CASM_RT(FORMAT, ARGS...)` is used to print every event of an ongoing CASM execution. By default this debug output is deactivated. Through a CASM tool command line option (*–gdebug*) the code generator activates this debugging facility in the generated code.

Another run-time facility is tracing. The run-time only defines the macro `CASM_PRINTF_TRACE( FORMAT, ARGS...)` which is deactivated by default and not included into the source code. Through a CASM tool command line option (*–gtrace*) the code generator defines this macro and it is

[2]system calls are used to implement e.g. basic file operations like open(), write(), read(), close(), etc.

compiled and activated in the translated CASM program. Currently only tracing of top level expressions and update statements is supported.

4.2.9 Miscellaneous

Temporary C Variables

The macro `CASM_REGISTER(TYPE, NAME)` declares and defines a C variable with the CASM type `TYPE` and the identifier name `NAME`. Furthermore, the variable will be initialized to the `undef` value.

Let Bindings

For the `let` statement (see Section 3.2.4) the run-time provides the macro `CASM_LET(TYPE, VARIABLE, VALUE)`. This macro defines and declares a C variable of type `TYPE` with name `NAME`, but it directly assigns the provided CASM value `VALUE`. There is another form of this macro defined as `CASM_LET_CONST(TYPE, VARIABLE, VALUE)` which does not assign a CASM type but rather assigns a constant C value of type *CT* (see Section 4.1).

Diedie & Assert

The `diedie` and `assert` statement (see Section 3.2.4) implementation in the run-time are using the macro `CASM_ERROR(FORMAT, ARGS...)` to print an error message and abort the ongoing execution. The `assert` statement additionally checks via the macro `CASM_ASSERT(VALUE, LINE)` if the Boolean value `VALUE` is defined and *true*. If the value is either *false* or undefined the macro calls internally the `CASM_ERROR()` macro with the CASM input program `LINE` number information.

If-Then-Else & Case

The `if` statement (see Section 3.2.4) is supported by the run-time by the following macros - `CASM_IF(VALUE)` and `CASM_ELSE`. The value `VALUE` will be checked if it is defined and *true*.

Similar to the `CASM_IF()` is the `CASM_CASE(VALUE)` macro. It is a direct translation to the C `case` keyword. The value `VALUE` can either be a Boolean, enumeration, numeric or string literal. Boolean, enumeration and numeric values are provided by the `CASM_CASE_CONST(VALUE)` macro. String values are provided by the `CASM_CASE_VAR(VAR, REG)` macro. The value `VAR` is a unique hashed value of the string content. The default case is supported through the `CASM_CASE_DEFAULT` macro. Furthermore, the `CASM_CASE_BREAK` macro is used to provide a wrapper of the C `break` keyword.

Deriveds & Rule Calls

All `derived` specifications in a CASM program are generated through the code generator (see Section 4.3.1). The `derived` C macro signature is defined in the run-time as `CASM_DERIVED_<N>()` where <*N*> is the name from the specification (see Section 3.2.3). The code generator uses this signature to directly use and define a CASM `derived`.

Just like the deriveds, all `rule` specifications are generated (see Section 3.2.3). The run-time provides the macros `CASM_CALL_RULE_PLAIN(NAME)` and `CASM_CALL_RULE(NAME, RULE_ARGS)`. The first one can be used to call a `rule` with the name `NAME` without any parameters. The second function provides a parameter `RULE_ARGS` which is of a `Generic*` type. The code generator uses this parameter to pass a `Generic` array address to the called `rule` (see Section 4.3.1).

Forall & Iterate

The `forall` statement (see Section 3.2.4) allows performing a parallel execution of a statement. The parallelism is given by a numeric range, list of elements, all elements of a `Tuple` value or all elements of an `enum` specification. All `forall` macros are translations to C `for` loops.

The `CASM_FORALL(TYPE, VALUE, BOTTOM, TOP)` macro defines a `forall` identifier with the name `NAME` of type `TYPE` and initializes it with the defined value of the `BOTTOM` parameter. The created C `for` loop is bounded by the upper value of the `TOP` value. In this macro the `BOTTOM` and `TOP` parameter are of a C type `int64_t` to represent a number literal range. The macro `CASM_FORALL_REG(TYPE, VALUE, BOTTOM, TOP)` can be used to perform a range over two `Int` values. The code generator sometimes needs an inverse C `for` loop iteration order. The run-time provides therefore the macros `CASM_FORALL_REVERSE()` and `CASM_FORALL_REG_REVERSE()`.

As mentioned before the `forall` statement can also operate over `Tuple` and `List` value domains by executing the `forall` statement over all elements of these types. `CASM_FORALL_LIST(SUBTYPE, VAR, TYPE, LIST)` and `CASM_FORALL_TUPLE(SUBTYPE, VAR, TYPE, LIST)` are provided by the run-time to iterate through the elements. To end a `forall` statement the `CASM_END_FORALL` macro has to be called.

The run-time supports the `iterate` statement (see Section 3.2.4) by the following two macros. `CASM_ITERATE` starts the iteration block and `CASM_END_ITERATE` ends the block. These two macros form a C `do while` loop which aborts if no further update is produced inside the `iterate` statement.

4.3 Code Generator

The CASM code generator uses the typed AST and translates each AST node directly to C. Before the translation process starts, every node of the typed AST is uniquely labeled by a preorder traversal. The label is an `uint64_t` counter value which is later used in the generation process to construct unique C variables for temporary calculations of CASM expressions. The approach is similar to the *three-address code* [1] representation where e.g. an expression of a program is linearized into multiple sub-expressions with temporary results and each temporary has a unique name. Figure 4.5 outlines the generation process of the code generator.

The code generation process is divided into five generation phases. The last phase of the code generator invokes a C compiler to compile the generated code (aka Target C) which is linked together with the run-time library to an executable. The generated files of the code generator are summarized in the following Table 4.2 where the placeholder <*P*> stands for the CASM input program name and the placeholder <*> stands for all rule identifier names:

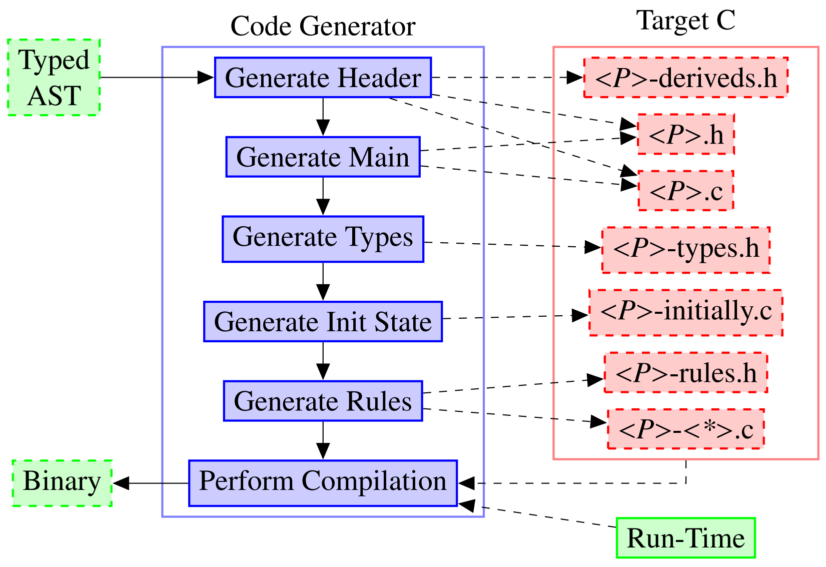


Figure 4.5: CASM Code Generation Process

File	**Phase**	**Run-Time Component**
<*P*>.h	Header, Main	Debuginfo Channels, Functions, Glue
<*P*>.c	Header, Main	Update-Set, Kernel
<*P*>-rules.h	Rules	Rules (Declarations)
<*P*>-<*>.c	Rules	Rules (Definitions), Statements, Expressions
<*P*>-types.h	Types	Enum, Tuple, List, Operator, Built-ins
<*P*>-deriveds.h	Header	Deriveds
<*P*>-initially.c	Init-State	Initial Updates

Table 4.2: CASM Target C File Layout

4.3.1 Generation Phases

Generate Header

In the *Generate Header* phase the code generator creates the main header file *<P>.h* and the main source file *<P>.c*. The header file contains all includes to other headers and to the run-time library. Then the code generator traverses the AST of all rules and searches for the **`debuginfo`** statements. The generator collects the *debuginfo channels* in an internal data-structure. If a *debuginfo* channel was provided (see Section 4.2.7) the code generator also adds the declared channels to this structure. All channels are then declared in the main header file and defined in the main source file.

After that the code generator starts with the generation of the functions. For all functions in the main header and main source file the code generator declares/defines a numeric UID which will be used to map a function number to a function interface call.

Then all specified CASM functions are evaluated one-by-one. To define a CASM function the code generator uses the function *getter*, *setter*, *printer* and *objdump* interface which is provided by the run-time (see Section 4.2.2). The code generator distinguishes three different types of functions - *static domain*, *dynamic domain* and *mixed domain* functions. By a *static domain* all function parameters are bounded, *dynamic domain* all are not bounded and *mixed domain* is composed of a *static* and a *dynamic domain*. The code generator sorts the different types of the function domain accordingly to the value domains (see Section 3.2.2) in the following order:

$$\underbrace{A < B < E_{\mathbb{N}} < R < I_{[a,b]}}_{static\ domain} < \underbrace{I < S < T_H < L_H}_{dynamic\ domain} \tag{4.1}$$

The code generator separates a function domain into a *static domain* and a *dynamic domain*. For example the function f with the following defined relation:

$$f : \underbrace{I \times R \times B \times I_{[a,b]}}_{domain} \rightarrow \underbrace{S}_{co\text{-}domain}$$

The code generator rearranges the *domain* of the function f accordingly to the defined order in Formula 4.1 to:

$$f : \underbrace{B \times R \times I_{[a,b]}}_{static\ domain} \times \underbrace{I}_{dynamic\ domain} \rightarrow \underbrace{S}_{co\text{-}domain}$$

The code generator selects one of the four possible CASM function implementation generation schemes. The first scheme is selected by the code generator if a function has no *dynamic domain*. Due to the fact that a *static domain* is bound to a finite n-dimensional space the code generator translates the domain to a linear C array of the co-domain's data-type. If the size of the C array is too large and will not fit in the BSS section of the executable, the code generator uses the second generation scheme. This scheme generates run-time calls to dynamically allocate the linear array.

The third generation scheme is used if there exists no *static domain* in the separated domain. The code generator generates an extended n-dimensional geometric hashing [69] procedure for the function arguments. Each function argument is casted to an `uint64_t` value. The bitmap (`args_bitmap`) to perform the geometric hashing is a generated random variable C array of type `uint64_t` . The code generator concatenates all function argument values *args* to a hash value *h* accordingly to the formula:

$$\texttt{key} \;=\; (\texttt{uint64\_t}) \sum_{i=0}^{|args|-1} (-1) * args_i * args\_bitmap_i$$

The value *key* is used in the function implementation to access the *branded hash-map* function structure (see Section 4.2.2). The function structure is allocated by the CASM kernel (see Section 4.2.4).

The fourth function generation scheme is used when a function is specified with a *mixed domain*. The code generator applies the same translation as in the third generation scheme with the extension that the *static domain* values are packed into groups of 64 bit values via the type `uint64_t`. These packed values are concatenated with the hash value *key*.

The code generator also generates the predefined *program* function into the main header file. And if a `provider` specification (see Section 3.2.3) is used, all function implementations from the provider are directly copied into the main header file, because a provided function has to be implemented with the same function interface (see Section 4.2.2).

After the function generation the code generator declares/defines the glue code between the update-set and the functions. This is done by mapping the used functions to a previously generated UID number. The glue logic consists of the macro `DECLARE_CASM_UPDATESET_APPLY( FUNCTION_VARIABLES, FUNCTION_MAPPING)` and `DECLARE_CASM_UPDATESET_PRINT( FUNCTION_MAPPING)` which define the apply and print mechanism from the update-set structure (see Section 4.2.3).

The code generator creates the `deriveds` header file *<P>-deriveds.h* and writes all `derived` specifications into this file. A *derived* is defined directly via the `CASM_DERIVED_<N>()` *derived* macro signature (4.3.1). The C preprocessor will replace the *derived* expression with its implementation.

Generate Main

In the *Generate Main* phase the code generator continues with the generation in the main source file. First it generates the before mentioned random variable bitmap array which is used for the geometric hashing procedure of *dynamic domain* function implementation generation. The code generator continues with generating all the static known information of the CASM input program. It generates the rule names, enumeration names and *debuginfo* channel names. In the next step all provided pre- and post-hook functions are defined (see Section 4.2.7) and afterwards the statically allocated CASM functions are defined and the `casm_function` structure (see Section 4.2.2) is used to generate the global state of the program. Last the code generator generates the CASM kernel into the source main file by using the macro `CASM_MAIN(RULE)`. `RULE` is the name of the `init` rule (see Section 4.2.9).

Generate Types

In the *Generate Types* phase the code generator creates the types header file *<P>-types.h*. This file declares all types which are not implemented directly in the run-time. The generation order of the types is enumeration-, ranged integer-, `Tuple`- and then `List`-based types. The code generator recognizes all used types from the CASM input program in the *Generate Header* phase.

All specified enumerations (`enum`, see Section 3.2.3) are directly generated into the types header file as C enumeration types. After the last element of an enumeration an additional entry with the name `<N>_SIZE` is added to the definition. `<N>` stands for the name of the generated enumeration and this size field corresponds to the size of the enumeration type. This field is used e.g. in the casting facility of the run-time (see Section 4.2.5). The code generator generates then CASM type of the enumeration with the `DECLARE_Enum()` macro. An enumeration with type name `<N>` is then declared in C with the same name.

A ranged integer type `Int()` is declared by the generic macro `DEFINE_Int()` which is provided by the run-time (see Section 4.2.5) to define new ranged integer types. This macro creates a new CASM type with a C structure name `Int_<A>_<B>`. The place holder `<A>` and `<B>` stand for the lower and upper bound of the ranged integer type. If negative numbers are used the type is defined by using a lowercase 'n' (negative) character. For example the CASM type `Int(-12..34)` is defined as a C structure `Int_n12_34`. It is important that the code generator only has to define this macro and all *built-in*, *operator* and *type interface* C functions are implicitly created in the target program. Furthermore, the macro defines a range checking *type function* with the name `void CASM_check_Int_<A>_<B>(Int_<A>_<B>* i)`. This function is called in the later generated rule if a calculated `Int` type value is assigned to a ranged integer type. There are only two possibilities in a CASM program where this can happen, either the ranged integer type is used as a co-domain of a function or the type is used as a parameter of a function domain, a derived parameter or a rule parameter. CASM always calculates in the `Int` value domain and when the assignment or usage of this value has to fit the ranged integer definition then the *check type interface* function is used to check if the integer value is in the correct range.

The `Tuple` type is the only CASM type which for every sub-type signature is completely generated from scratch by the code generator, because the run-time has no knowledge about this type at all. From scratch means that besides the `Tuple` C structure definition all *built-in*, *operator* and *type helper* functions (see Section 4.2.5) are generated for the specific used sub-type. The resulting CASM `Tuple(<H>)` type name in C corresponds to `Tuple_<H>` where `<H>` is the underscore separated sub-type signature of the tuple. For example the type `Tuple(Int, String, RuleRef)` is generated into a C structure type `Tuple_Int_String_RuleRef`.

Similar to the `Int()` type the `List` is predefined in a generic macro in the run-time. The code generator just has to generate each list-based type with the macro `DEFINE_List()` and a new CASM `List` type is created. For example if a `List(Int(0..7 )` is used in the CASM input program the code generator creates through the macro the new type `List_Int_0_7` in the types header file.

Generate Init-State

In the *Generate Init-State* phase the code generator creates the initially source file *<P>-initially.c*. This file contains a C function which will be called from the CASM kernel in the *initialization* phase (see Section 4.2.4) to preset the global state of the CASM functions accordingly to the defined `initially` expressions from the CASM input program. First the code generator generates the *static initial state* and then the *dynamic initial state*. The *static initial state* generation assigns all specified constant expressions to the CASM function's global state. This is necessary to generate afterward the *dynamic initial state*, because the `initially` specifications can depend on the *static* ones. The compiler resolves the dependencies within the *dynamic* `initially` specifications in the annotation process (4.1.1). The code generator uses the provided `CASM_UPDATE_INITIALLY()` macro from the run-time (see Section 4.2.3) to set functions to an initial state.

Generate Rules

The last step in the generation process is the *Generate Rules* phase. Before the code generator translates every CASM `rule` specification into statements, expressions etc. it creates the rule header file *<P>-rules.h*. All specified `rule` C declarations are written to this file. The macro `DECLARE_CASM_RULE()` declares a rule specification in the rule header file (see Section 4.2.9). The run-time represents rules as C functions. Afterwards the rule generation starts by executing the following procedure for each `rule` specification. First the generator creates for each rule a separate C source file with the name *<P>_<*>.c* where <*> stands for the `rule` name identifier. This file contains the actual implementation of a `rule` which is defined through the `DEFINE_CASM_RULE()` macro. This macro defines the C function of the `rule`.

The code generator then calls a recursive function to traverse in preorder over the AST of a rule to generate all statements and expressions to the rule implementation file. It is important that the code generator assures that the composition hierarchy of parallel and sequential execution blocks is always generated in the nesting order of `PAR` $\rightarrow$ `SEQ`, `SEQ` $\rightarrow$ `PAR`, `PAR` $\rightarrow$ `SEQ`, etc. This property minimizes the *fork* and *merge* operation calls to the update-set. Furthermore, the update-set structure (see Section 4.2.3) needs this property, because the semantics of the *fork* and *merge* operations assumes only execution semantics block transitions either from `PAR` $\rightarrow$ `SEQ` or `SEQ` $\rightarrow$ `PAR`.

The code generator always generates by default each rule into a separate file. If large programs with thousands of rules are compiled the code generator will create a C compilation unit for each rule. The compilation time of a C compiler to translate the complete CASM program will be huge. Another problem is that if one rule C file has so many Lines Of Code (LOC) that a C compiler is either unable to compile the file or the compile time is not feasible. The code generator can perform two optimized generations to achieve more compilation speed.

The first optimization is called *packed* where the code generator packs multiple C rule implementations into one compilation file bounded by a maximal C LOC amount. This can be specified with the command line option *–gpf* (generate packed file). The size can be configured with *–gpf-size <arg>* where *<arg>* stands for the amount of LOC per compilation rule file.

The second optimization is called *split rules*. The code generator splits a rule into multiple files to support the C compiler with small compilation units. The split procedure is based on a heuristic. The longest connected statement chain of a rule AST is split into multiple *split rules*. Before the rule generation process starts the code generator calculates the number of nodes in the rule AST. Then the longest statement chain in the AST is detected. This statement chain is divided by the containing AST nodes which can be specified with the *–gsplit <arg>* command line into sub-trees. *<arg>* stands for the upper bound of maximal AST nodes in a sub-tree. This generation optimization applies for flat rules with huge consecutive statements very well.

Perform Compilation

In the last process step the code generator executes the *Perform Compilation* phase. Therefore, an external C compiler is used. The code generator calls the external C compiler and compiles all generated files from previous phases. Afterwards the code generator links all object files to the final executable.

4.3.2 Generation Example

The following Listings 4.5 and 4.6 show an example code generation from the code generator. The CASM program *example* implements a simple counter application. The rule *main* increments every step the content of a function by one. If the content of the function is greater than a specific value, a sequential block is executed. This block prints the content of the function and performs the termination update to the *program* function. Due to the lack of space only the generated *main* rule implementation file is shown in Listing 4.6.

```
CASM example init main

function counter : -> Int initially { 0 }

rule main =
{
    counter := counter + 1

    if counter > 0xfeed then
    {|
        print counter
        program( self ) := undef
    |}
}
```

Listing 4.5: CASM Counter Example

```
DEFINE_CASM_RULE(main)
{
    /* example.casm:7 | UPDATE */
    CASM_REGISTER(Int, r22__);
    CASM_REGISTER(Int, r23__);
    CASM_FUNCTION_LOOKUP(counter, Int, r23__);
    CASM_REGISTER(Int, r24__);
    CASM_const_Int(&r24__, 1);
    CASM_add_Int_Int_Int(&r23__, &r24__, &r22__);
    CASM_UPDATE(7, counter, Int, r22__);
    /* example.casm:9 | IF */
    CASM_REGISTER(Boolean, r27__);
    CASM_REGISTER(Int, r28__);
    CASM_FUNCTION_LOOKUP(counter, Int, r28__);
    CASM_REGISTER(Int, r29__);
    CASM_const_Int(&r29__, 65261);
    CASM_gre_Int_Int_Boolean(&r28__, &r29__, &r27__);
    CASM_IF(r27__)
    {
        /* example.casm:10 | SEQBLOCK */
        CASM_UPDATESET_FORK_SEQ(updateset);
            /* example.casm:11 | PRINT */
            CASM_REGISTER(Int, r34__);
            CASM_FUNCTION_LOOKUP(counter, Int, r34__);
            CASM_print_Int(&r34__);
            CASM_PRINT_END;
            /* example.casm:12 | UPDATE */
            CASM_REGISTER(Self, r40__);
            CASM_const_Self(&r40__, SELF);
            CASM_REGISTER(RuleRef, r41__);
            CASM_UPDATE(12, program, RuleRef, r41__, &r40__);
        CASM_UPDATESET_MERGE_SEQ(function, updateset);
    }
}
```

Listing 4.6: Counter Example Rule main C Code

CHAPTER 5

CASM Optimization Framework

The CASM compiler is composed of a new optimized run-time and new optimized code generator. Furthermore, the compiler consists of an optimization framework. This chapter introduces the CASM optimization framework, the CASM IR, several analyses and transformations.

5.1 Overview

First of all the optimization framework uses the typed AST as input representation and performs its *passes* on the typed AST. The output of the optimization framework is always a correct and valid typed AST. Therefore, both the code generator and the interpreter can profit from all transformations, because the transformations modify directly the typed AST. Figure 5.1 gives an overview of the optimization framework.

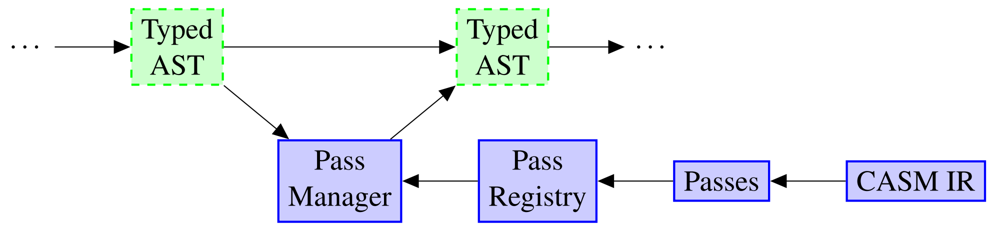


Figure 5.1: CASM Optimization Framework

The main component of the optimization framework is the *pass manager*. It performs different passes on the input typed AST and returns a typed AST as output. The implementation of the pass manager, all including components and the pass interface structure were inspired by the LLVM compiler framework implementation [43].

5.2 Pass Manager, Registry & Pipeline

The first function of the three main tasks of the pass manager concerns the acquisition of all selected passes from the command line, storage of all available passes via pass registry and comparison of those with the (already) registered passes. The second task causes the assignment to one of the four predefined *pass pipeline* buckets for all *passes*, which leads to the third and last task, the execution of all passes for all rules.

Passes can be assigned to four distinct *buckets*. The assignment is defined in the *pass information* description of a pass implementation (see Section 5.2.1). The pipeline buckets are defined as:

preprocessing Passes which shall run before the pipeline fixpoint mechanism are located in the *preprocessing* bucket. For example all analysis or information dumping passes are located in this bucket. If a pass in the pass information is not assigned to a bucket it will be assigned by default to the *preprocessing* bucket.

trivial fixpoint Analysis and transformation passes which are standard compiler optimizations are assigned to the *trivial fixpoint* bucket.

non-trivial fixpoint Specialized CASM passes are contained in the *non-trivial fixpoint* bucket. For example this bucket contains the CASM *Redundant Lookup Elimination* transformation pass (see Section 5.4.4). The separation between *trivial* and *non-trivial* is needed to optimize the pass results of the *non-trivial* passes, because those passes are costly compared to the *trivial* passes.

postprocessing All passes which should be executed after the fixpoint mechanism have to be assigned to the *postprocessing* bucket.

After the execution of bucket assignment, the pass manager schedules all buckets according to a scheduling algorithm. The execution order of passes inside a bucket is defined by their dependency to other passes. This dependency is defined by a specific pass interface. The pass manager guarantees for a distinct set of selected passes always the same schedule. As the name *pass pipeline* implies the four buckets form an execution pass pipeline. Figure 5.2 shows the implemented *pass pipeline* scheduling algorithm in the pass manager.

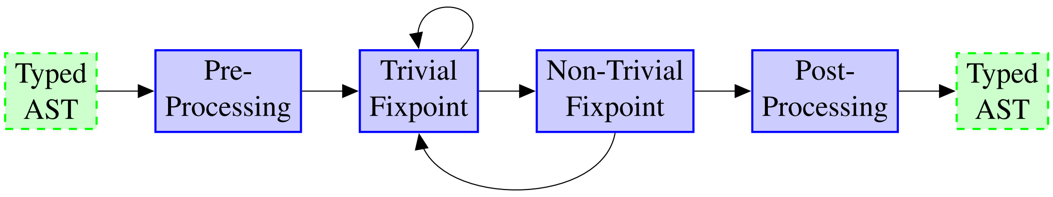


Figure 5.2: CASM Pass Pipeline Execution

After the pass manager executes the *preprocessing* pass set, the pass manager enters the fixpoint mechanism and all *trivial fixpoint* passes are executed. As long as the AST is modified by at least one pass the pass manager repeats the *trivial fixpoint* bucket, otherwise the pass manager executes the *non-trivial fixpoint* pass bucket. If the AST is modified by at least one pass, the pass manager repeats the *trivial fixpoint* bucket, otherwise the pass manager continues with the *postprocessing* pass bucket.

During the execution of a single pass no matter what in which pass pipeline state the pass was called, the pass manager perform a fixpoint iteration of this pass as well. Before the pass manager calls the actual pass procedure, all passes the pass depends on are executed first. To optimize the optimization speed of the compiler, the pass manager implements a history (linear list) of previously executed passes which also store the last AST changes of the passes. If the pass manager finds the same pass which is scheduled next and its last AST changes in the history was zero, the pass is skipped. If the pass manager finds any other pass with a last AST change which is greater than zero, the scheduled pass has to be executed.

A complete run of the pass pipeline is repeated for every CASM input program rule. There are command line options to select just a sub-set of rules to perform the selected optimization passes.

5.2.1 Pass Information & Interface

Similar to LLVM every *pass* in CASM consists of a statically defined *pass information*, a pass header and a pass implementation file. The information contains a string for the unique command line option, the pass description, its name and the setting for the *pass pipeline* into which bucket a pass belongs. Furthermore, the *pass* implementation has to fulfill a pass interface which defines specific entry points for the pass manager to *initialize*, *run*, *finalize* and *verify* a pass implementation.

Furthermore, the pass interface defines a *usage* function which allows defining for a pass a dependency relation to other passes. This usage function is called in the scheduling process in the pass manager (see Section 5.2).

5.2.2 Pass Statistics

The optimization framework tracks for each pass the performed AST modifications and spend time. By enabling through a command line option, the compiler outputs after the pass pipeline execution a text-based table. Listing 5.1 shows an example pass statistic output.

```
| PASS STATISTICS                   | SCHED | TIME [ms] |  [%]  | CHANGES |  [%]  |
|-----------------------------------+-------+-----------+-------+---------+-------|
| ASTPrinterPass                    |     1 |     0.127 | 20.73 |       0 |  0.00 |
| IRPrinterPass                     |     1 |     0.242 | 39.62 |       0 |  0.00 |
| CFGPrinterPass                    |     1 |     0.151 | 24.68 |       0 |  0.00 |
| IntermediateRepresentationPass    |     1 |     0.092 | 14.96 |       0 |  0.00 |
| Total                             |     4 |     0.612 |       |       0 |       |
```

Listing 5.1: Example Pass Statistics

5.3 CASM Intermediate Representation

Besides the typed AST intermediate representation, the optimization framework defines an additional intermediate representation - the CASM IR. This IR was motivated by the fact that it is intricate and complex to perform all the desired analysis and transformations on the typed AST intermediate representation. Especially the need of parallel aware analysis constructs made it indispensable to design and implement a CASM specific representation. The CASM IR is an n-ary tree-based representation which is generated by the *Intermediate Representation* analysis pass (see Section 5.4.1) of the typed AST. Each structural element in the CASM IR has a direct connection (pointer) to the typed AST. The structure of the IR is divided into five main classes which are visualized in Figure 5.3.

Instruction An *Instruction* is a representation of a single temporary (sub-)expression or a statement of the typed AST. The idea behind the *Instruction* class is to form *three-address code* [1] instructions to create a SSA form of all used CASM language constructs just like the code generator creates its intermediate code.

Basic Block The *Basic Block* is an abstract class which is used to create the actual n-ary tree of the CASM IR.

Statement A *Statement* is inherited of *Basic Block* to represent all possible CASM statements. A *Statement* class consists of a linear list of *Instruction* objects.

Scope The class *Scope* represents the parallel and sequential composition blocks is also inherited from *Basic Block*. Every *Scope* has knowledge about its execution semantics, pseudo state and consists of a linear list of *Basic Block* objects. So the elements in a *Scope* can either be *Statements* or nested *Scope* objects.

Rule The *Rule* class is the top-level node of the CASM IR. It is composed of a by default parallel *Scope*.

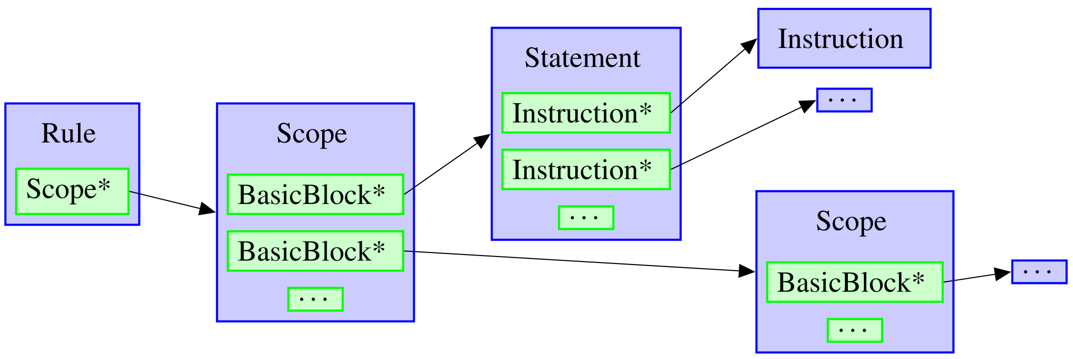


Figure 5.3: CASM IR Structure

The *Rule* class supports several auxiliary functions to access the CASM IR elements in different behaviors. These auxiliary functions implement traversal and iteration schemes. The pass developer can decide between pre-, in- and post-order traversals and forward or backward iterations. Furthermore, the auxiliary functions return either all CASM IR objects or only *Statements*, *Scopes* or *Instructions*. It is for example possible to traverse over the IR *Instruction* objects, or just over all *Scope* objects. The forward and backward iteration enables a sequential forward or backward flow through the IR. This allows a classical *forward flow* through the program statements like in a CFG. These auxiliary functions are used e.g. in the *IR Printer* and *CFG Printer* pass (see Sections 5.4.2).

The approach described in this work is a generalization of the *PAR/SEQ Control Flow Graph* which was presented by Lezuo, Paulweber and Krall in [49]. Instead of creating from the typed AST a CFG representation which includes the parallel and sequential composition formation directly, the CASM IR generalizes this with its parallel sequential tree structure and auxiliary functions.

5.3.1 Instructions

For each CASM statement, expression and literal syntax (see Section 3.2) there exists a separate Instruction class which is inherited from *Instruction*. For example to represent a constant number literal in the CASM IR a typed *Int Const Instruction* is used. Furthermore, the *Instruction* IR provides full knowledge about the underlying AST of a statement, expression, etc. which means that when the CASM IR is created an identifier of the CASM input program is mapped to a concrete *Instruction* class. This procedure avoids reinterpreting of identifiers in analysis and transformation passes which use the CASM IR. Listing 5.2 shows an example update statement. Listing 5.3 shows the CASM IR of the example update statement with its underlying instructions.

```
...
bar( baz, BVand( 32, 0xdead, 0xbeef ) ) := bar( "txt", 0xfeed )
...
```

Listing 5.2: CASM Example Update

```
...
+UpdateStatement                            // UpdateStatement
  r21__ = const 'txt'                       //   StringConstInstruction
  r23__ = const int 65261                   //   IntConstInstruction
  r18__ = lookup bar(r21__, r23__)          //   LookupInstruction
  r8__ = derived baz                        //   DerivedInstruction
  r13__ = const int 32                      //   IntConstInstruction
  r15__ = const int 57005                   //   IntConstInstruction
  r17__ = const int 48879                   //   IntConstInstruction
  r10__ = BVand r13__, r15__, r17__         //   SharedInstruction
  update bar(r8__, r10__), r18__            //   UpdateInstruction
...
```

Listing 5.3: CASM IR of Example Update

By default the *Instruction* class defines an instruction return value where the instruction operation result is stored. This return value is also known as the *register* of the CASM IR instruction. Every register gets a unique number and a prefix ‘r’ character (see Section 3.2). Only a few *In-*

struction classes do not return a register value which are the *Update Instruction* and the *Call Instruction*, because they do not modify a temporary register.

The *Update Instruction* and *Lookup Instruction* are special in the CASM IR. In Listing 5.3 the lookup of a function is specified directly by a *location* signature (line 3). Similar applies for the first parameter of the *Update Instruction* (line 11). This location is created through an internal *Location Instruction* which encapsulates the location signature. It is important that from every *Instruction* object it is possible to acquire the enclosing *Statement* object.

5.3.2 Statements

The implemented *Statement* classes are containers for the underlying instructions. The CASM IR distinguishes between *Trivial Statement*, *Branch Statement* and *Loop Statement* sub-classes. Trivial statements are update, call, assert, diedie, print, debuginfo, push, pop and skip statements. For each *Trivial Statement* exists a concrete sub-class in the *Statement* hierarchy. These statements are trivial, because they consist only of a linear list of *Instruction* objects. *Branch Statement* classes consist also of the linear list for the instructions, but they consist additionally of one or more inner *Scope* objects. So it is possible to represent lets, if-then-else and case statements in concrete sub-classes.

Furthermore, the *Loop Statement* includes beside the linear list of instructions one *Scope* object which is the actual ‘loop body’. This distinction is necessary to implement the different auxiliary function behaviors (see Section 5.3). Similar to *Instruction* objects, it is possible to get the enclosing *Scope* object from *Statement* objects which they belong to.

5.3.3 Scopes

The CASM IR *Scope* classes represent the parallel and sequential execution block and therefore exists a class *Parallel Scope* and a class *Sequential Scope*. From every *Scope* it is possible to get the enclosing parent *Scope* until the top-level *Scope* object is reached.

Common Scope

A very important property of two objects in the CASM IR is the so called *common scope*. A *common scope* is the scope intersection of two CASM IR objects. Every *Instruction* belongs to a *Statement* and every *Statement* belongs to a *Scope*. Every *Scope* has a parent *Scope* unless it is the root node. Therefore this chain can be constructed for every CASM IR object. The *common scope* property is important for the *Use Definition* analysis pass to determine of an update statement and a lookup instruction reside in a parallel or sequential *common scope* (see Section 5.4.3).

5.4 Passes

The following sub-sections outline all current implemented passes of the optimizing compiler.

5.4.1 Framework Internal Passes

Intermediate Representation Pass

The *Intermediate Representation* pass generates the CASM IR from a typed AST of a rule body specification. This pass is a compiler internal one, which can not be selected in the command line of the CASM tool.

Register Renaming Pass

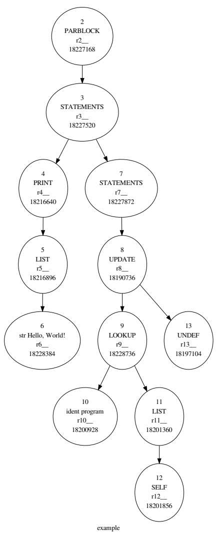
The *Register Renaming* pass re-writes all identifiers in a **`rule`** body specification. This is especially needed to achieve a SSA form for the CASM IR. The *Intermediate Representation* and *Loop Unwinding* pass use this pass to rename let and local AST node identifier.

5.4.2 Printer Passes

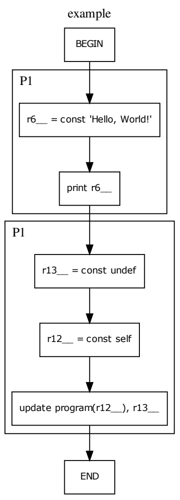
As the name implies the printer passes can be used to output the internal representation of the CASM input program in a specific format or view. Figure 5.4a, 5.4b and 5.4c show the *hello world* example as AST, a CFG and the IR.

AST Printer Pass

The *AST Printer* pass prints the typed AST rule body specification to a file in *Graphviz dot* format [26].



(a) AST



(b) CFG

```
example:
  *PAR@1
    +PrintStatement
      r6__ = const 'Hello, World!'
      print r6__
    +UpdateStatement
      r13__ = const undef
      r12__ = const self
      update program(r12__), r13__
```

(c) IR

Figure 5.4: CASM AST, CFG and IR of the Hello World Example

CFG Printer Pass

In the *CFG Printer* pass the CASM IR is iterated through an auxiliary function which performs a forward flow iteration over the containing *Statement* objects. The printer generates a *Graphviz dot* file and writes the CFG into this file.

IR Printer Pass

The *IR Printer* pass emits the CASM IR of a rule to the standard output stream.

5.4.3 Analysis Passes

Possible Update Pass

This pass acquires all update location function names. The pass is based on the typed AST and simply traverses over the AST and searches for AST update nodes. Furthermore, if the analysis finds a AST call node the analysis is aborted and all possible function names are returned.

Reaching Definition Pass

The *Reaching Definition* pass performs a classic DFA algorithm [1]. Whereas this analysis does not detect the definition of variables, it is a modified implementation to acquire the sequential view of all update statements in the CASM IR. The pass iterates in a forward flow through an auxiliary function over all *Statement* objects. If a *Statement* object is a *Update Statement* a new location definition is generated and all previous ones of the same name are deleted. This pass does not consider the parallel execution semantics.

Use Definition Pass

In the *Use Definition* pass function lookups (uses) are mapped to the correct update (def) definitions. The definitions in the program flow are analyzed through the *Reaching Definition* pass. The *Use Definition* pass iterates over all CASM IR instructions which considering the sequential and parallel execution semantics and decides for each lookup if it has a *global*, *local* or *merge* property. A *global* property means that a lookup has no definition in the *Reaching Definition* result set and at run-time the lookup will always read the global state of the function. If there is exactly one definition in the *Reaching Definition* result set for a lookup, the *local* property is detected. At run-time a *local lookup* will receive the same value an update had stored to the update-set before. If the *Use Definition* pass detects multiple definition sites for a function lookup, a *merge* property is assigned to the lookup. If a function is updated twice or only in one branch of an if-then-else statement, a later lookup will see both definition sites.

For the lookup *local* property it is important that the pass always checks in which *common scope* the lookup and update are located (see Section 5.3.3). If the common scope is parallel, the *local* lookup gets promoted to a *global* lookup, because when the only definition in a program is detected to be a parallel definition, the lookup will never see this value, and will always retrieve the global state of the function. If the *common scope* is sequential the *local* property

```
...
rule r =
{|
   {
      x := x + 1
   }

   if x = 10 then
   {
      x := 20
   }

   print x

   print x
|}
...
```

```
 1 ...
 2 r:
 3   *SEQ@1
 4     *PAR@2
 5       +UpdateStatement
 6         r9__ = lookup x         // global lookup of x
 7         r10__ = const int 1     //
 8         r8__ = add r9__, r10__  //
 9         update x, r8__          // definition of x <---+ <-+ <--+
10     +IfStatement                //                     |   |    |
11       r14__ = lookup x          // local lookup of x --+   |    |
12       r15__ = const int 10      //                         |    |
13       r13__ = equ r14__, r15__  //                         |    |
14       r12__ = if r13__          //                         |    |
15       *PAR@3                    //                         |    |
16         +UpdateStatement        //                         |    |
17           r20__ = const int 20  //                         |    |
18           update x, r20__       // definition of x <---+ <---+  |
19     +PrintStatement             //                     |   | |  |
20       r24__ = lookup x          // merge lookup of x --+---+ |  |
21       print r24__               //                           |  |
22     +PrintStatement             //                           |  |
23       r28__ = lookup x          // merge lookup of x  -------+--+
24       print r28__
25 ...
```

Figure 5.5: Use Definition Pass Example

stays a local property. If the definition is not an update but a call statement, the *Use Definition* determines through the *Possible Update* pass if a statically known rule call has updates of a specific function. If the lookup function is not contained in the result set, the *Use Definition* pass uses the next definition of the location.

Figure 5.5 shows an example (left) and its *Use Definition* pass result (right). The first lookup of x (line 6) has no previous definition and therefore this lookup equals a *global* lookup. In line 11 the lookup of x is a *local* lookup, because in line 9 is a definition (update to x) and this definition and the lookup have a sequential *common scope* which means that this lookup will read exactly this updated value. In line 20 and 28 the lookups of x are both *merge* lookups, because both lookups see both definitions of x at line 18 and 9. The reason for that is that the *Reaching Definition* pass propagates both definitions to the two *print* statements, because of the *if* statement with the optional update.

Definition Use Pass

The *Definition Use* pass is the inverse of the *Use Definition* pass which simply creates a pass result with a map of *Update Statement* objects to multiple *Lookup Instruction* objects.

Conflict Update Pass

This pass is a simple verification pass which emulates the run-time update-set behavior and can detect conflicting updates. If locations are fully specified, or the signature of an update equals another update, the *Conflict Update* pass returns an error of the conflicting updates.

5.4.4 Transformation Passes

Case to If Conversion Pass

This AST-based transformation converts all AST case statement nodes into equivalent if-then-else AST statements. Figure 5.6 shows an example *Case To If Conversion* transformation.

```
...
case data of
    1: diedie
    2: print "2"
    default: skip
endcase
...
```

$\Longrightarrow$

```
...
if data = 1 then
    diedie
else
    if data = 2 then
        print "2"
    else
        skip
...
```

Figure 5.6: Case To If Conversion Pass Example

Skip Removal Pass

The *Skip Removal* pass eliminates all AST **skip** statement nodes by traversing over the rule AST body specification and removes all the AST nodes of type *skip*. This pass can be used to clean up the AST of other transformations which insert **skip** statements to avoid multiple implementations of the *Skip Removal* pass. Figure 5.7 outlines an example where a sequence of **skip** statements is reduced to one **skip** statement. This transformed result can be optimized further by the *Trivial Block Removal* pass.

```
...
{|
    skip
    skip
    skip
|}
...
```

$\Longrightarrow$

```
...
{|
    skip
|}
...
```

Figure 5.7: Skip Removal Pass Example

Trivial Block Removal Pass

Through the *Trivial Block Removal* pass it is possible to remove unnecessary nesting of parallel and sequential composition blocks, or remove **if**, **forall**, **iterate**, etc. statements which only consist of a **skip** statement. Therefore, a trivial block is replaced by a single **skip** statement which enables the *Skip Removal* pass to remove this even further. Figure 5.8 presents an example were three different trivial blocks are removed by the *Trivial Block Removal* pass. First, the **forall** statement can be reduced to a **skip** statement since the inner statement is a **skip** statement. Second, the sequential block can be removed because the only statement is a **skip** statement. Third, the parallel block can be removed and replaced by a **skip** statement.

```
...
{
    {|
        forall i in [0 .. 100] do
            skip
    }|
}
...
```

$\Longrightarrow$

```
...
skip
...
```

Figure 5.8: Trivial Block Removal Pass Example

Dead Code Elimination Pass

This AST-based transformation removes every local definition which is not used inside a composition block. The pass removes especially **`let`** and *local* statements[1] which were introduced through the *Constant Propagation* pass (see Section 5.4.4). Figure 5.9 shows the result of an example *Dead Code Elimination* pass.

```
...
let tmp = 10 in
{
    print "no tmp usage"
}
...
```

$\Longrightarrow$

```
...
{
    print "no tmp usage"
}
...
```

Figure 5.9: Dead Code Elimination Pass Example

Debuginfo Removal Pass

The *Debuginfo Removal* pass traverses over the AST and removes every **`debuginfo`** statement specification by replacing it with a **`skip`** statement. The advantage of this transformation is that by eliminating the **`debuginfo`** statement, all its expressions are eliminated too. If e.g. a function is only used inside a **`debuginfo`** statement, it can be determined with the *Use Definition* analysis that a function is only updated but never used so the complete function can be removed by the *Dead Function Elimination* pass (see 5.4.4). Figure 5.10 shows an example transformation.

```
...
debuginfo channelA x + y + z
debuginfo channelB (x + y + z)
...
```

$\Longrightarrow$

```
...
skip
skip
...
```

Figure 5.10: Debuginfo Removal Pass Example

[1] *local* statements are currently not valid CASM syntax

Dead Function Elimination

The *Dead Function Elimination* pass allows to remove 'updated-only' functions from the program. This transformation determines with the *Use Definition* pass if a function is used somewhere in the CASM program. If the function has no use in any rules, it can be removed. If a function was defined in CASM and it is used in a provider it should be defined in the function specification as *undead* (see Section 3.2.3) otherwise this transformation will remove the complete function, because the compiler framework can only analyze the CASM input program. Figure 5.11 shows an example transformation.

```
CASM DF init foo
function x : -> Int
rule foo = skip
```

$\Longrightarrow$

```
CASM DF init foo
rule foo = skip
```

Figure 5.11: Dead Function Removal Pass Example

Inline Derived Expression Pass

All **derived** specifications are per-definition read-only and just simple expressions. The *Inline Derived Expression* pass performs a 'find-and-replace' procedure which swaps the used **derived** expression and the optional arguments AST node with a copy of the **derived** specification AST and replaces the variable parameter with the expression arguments. Figure 5.12 illustrates an example transformation.

```
derived macro = 1234
...
let tmp = macro + macro in skip
...
```

$\Longrightarrow$

```
derived macro = 1234
...
let tmp = 1234 + 1234 in skip
...
```

Figure 5.12: Inline Derived Expression Pass Example

Inline Rule Call Pass

Inline Rule Call pass replaces all static rule calls by the **rule** body specification AST. So this transformation is AST-based. The arguments are bound to a *let* statement and are renamed. The *let* statements inside of the rule is also renamed through the *Register Renaming* pass. Furthermore, always the original AST specification gets inserted. By original it is meant that previous optimized rules have always the original rule body AST specification and the optimized AST stored in the memory separately. Figure 5.13 shows an example transformation.

```
...
rule bar(a : Int, b : Int) =
    let x = a in
    let y = b in
        print (x + y)
...
rule foo =
{
    call bar(1, 2)
}
...
```

$\Longrightarrow$

```
...
rule bar(a : Int, b : Int) =
...
rule foo =
{
    let l0 = 1 in
    let l1 = 2 in
    {
        let l2 = l1 in
           let l3 = l2 in
               print (l2 + l3)
    }
}
...
```

Figure 5.13: Inline Rule Call Pass Example

Loop Unwinding Pass

Static known sizes of the **forall** statement can be unwind with the *Loop Unwinding* pass into multiple equal inner statement blocks. Every **forall** statement block variable gets replaced by a concrete number or enumeration literal, because the **forall** statement can be specified with number ranges or e.g. with the type name of an enumeration type which equals the range over the enumeration. The **forall** statement gets first replaced by a parallel composition block, because the execution semantics of a **forall** statement is always parallel. Figure 5.14 illustrates an example transformation.

```
...
forall i in [1 .. 3] do
    reg(i) := reg(i-1)
...
```

$\Longrightarrow$

```
...
{
    reg(1) := reg(1-1)
    reg(2) := reg(2-1)
    reg(3) := reg(3-1)
}
...
```

Figure 5.14: Loop Unwinding Pass Example

Constant Propagation Pass

In the *Constant Propagation* pass all constant literals are detected and hoisted to the beginning of a CASM program. To do so, this pass uses a new AST node type *local* to declare local rule states which are constant in this case. By iterating over the CASM IR all constant literals are directly visible through the *Const Instruction* pattern. A new constant gets created, hoisted to the top and another occurrences will get replaced with the previously created *local* identifier. This pass introduces a new AST node type namely *local*. A local state bounds an expression to an identifier similar to a **let** statement. Figure 5.15 shows an example transformation.

```
...
rule r =
{
    let x = 10 in
    let y = 20 in
    let z = 5 in
       print (5 + z + 20 + y + 10
             + x)
}
...
```

$\Longrightarrow$

```
...
rule r =
local c0 = 10 in
local c1 = 20 in
local c2 = 5 in
{
    print (c2+c2 + c1+c1 + c0+c0)
}
...
```

Figure 5.15: Constant Propagation Pass Example

Constant Folding Pass

The implementation of the *Constant Folding* pass itself is straight forward. In the CASM IR a special interface is defined which allows direct implementation of the folding process at the *Instruction* class definition. A similar concept is used in the LLVM compiler [43]. The interface is called *Foldable Instruction* and the *Constant Folding* pass calls the implemented folding behavior if a CASM IR object uses this interface. Constant expressions, constant *Shareds*, constant *Built-ins* and static function lookups are folded by this transformation. Figure 5.16 outlines an example transformation.

```
function (static) x : -> Int
     initially { 10 }
...
rule r =
{
    print hex(BVand(x, x, x / 2))
}
...
```

$\Longrightarrow$

```
function (static) x : -> Int
     initially { 10 }
...
rule r =
{
    print "F"
}
...
```

Figure 5.16: Constant Folding Pass Example

Dead Branch Elimination

The *Dead Branch Elimination* pass removes in this IR-based transformation the dead branch of an **if** statement. Furthermore, if the condition of a **case** statement is constant, the **case** statement is replaced with the correct case of the **case** statement. Figure 5.17 illustrates an example transformation.

```
...
if true then
{
    if false then
        assert false
    else
        assert true
}
else
    assert false
...
```

$\Longrightarrow$

```
...
{
    assert true
}
...
```

Figure 5.17: Dead Branch Elimination Pass Example

Redundant Lookup Elimination Pass

The CASM *Redundant Lookup Elimination* pass presented by Lezuo, Paulweber and Krall in [49] removes all redundant lookups which were detected by the *Use Definition* pass. The pass introduces local states to prevent further lookups of the same state. If two or more lookups have a *global* property then they are hoisted to the first common scope all *global* lookups. If two or more *merge* lookups are pointing to the same definition sites these lookups are replaced by inserting one local state lookup and all other lookups retrieve the identifier of this local lookup. This transformation is very powerful because it increases the performance of the compiled CASM program by minimizing the accesses to the update-set structure which are costly.

```
...
rule r =
{|
    {
        x := x + 1
    }

    if x = 10 then
    {
        x := 20
    }

    print x

    print x
|}
...
```

```
...
rule r =
local l0 = x in
{|
    local l1 = l0 + 1
    {|
        {
            x := l1
        }

        if l1 = 10 then
        {
            x := 20
        }
    |}

    local l2 = x in
    {|
        print l2

        print l2
    |}
|}
...
```

Figure 5.18: Redundant Lookup Elimination Pass Example

Figure 5.18 outlines an example transformation. The same example was used to visualize the *Use Definition* analysis pass (see Section 5.4.3). The *Use Definition* pass result is that the first lookup (left, line5) is a *global* lookup, the second lookup (left, line 8) is a *local* lookup to the previous update (left, line 5) and the third and fourth lookup (left, line 13 and 15) are *merge* lookups, because of the *if* statement. The *global* lookup gets transformed to the *local l0* (right, line 3) and the actual lookup gets replaced by the *l0* identifier. The update expression (left, line 5) gets also replaced by a *local l1* (right, line 5 and 8). The *local* lookup uses now the bound update expression from the *local l1*. The two *merge* lookups are redundant the *local l2* is used (right, line 17) to bind the lookup of *x* and reuse it in the two print statements.

Redundant Update Elimination Pass

The *Redundant Update Elimination* pass presented by Lezuo, Paulweber and Krall in [49] removes all unnecessary updates which will be overridden by other updates or update-set merges anyway. The current implementation iterates backwards over the CASM IR and starts to memorize update functions. If the pass detects in a sequential composition an update with the same location it gets removed. The implementation aborts if a branch or something equal is detected. There are ongoing compiler developments to improve the update elimination by providing an accurate analysis which calculates out the unnecessary updates. This pass allows keeping the update-set of the run-time as small as possible. The performance increases significantly which is shown in the following Chapter.

Figure 5.19 shows an example transformation. The functions *x*, *y* and *z* are updated twice in a sequential block. Therefore the first three updates can be removed.

```
...
rule r =
{|
   x := x + 1
   y := y + x + 1
   z := z + y + x + 1

   x := x * 2
   y := y * x / 2
   z := z - y * x / 2
|}
...
```

$\Longrightarrow$

```
...
rule r =
local l0 = x + 1 in
local l1 = y + l0 + 1 in
local l2 = z + l1 + l0 + 1 in
{|
   local l3 = l0 * 2 in
   {|
      x := l3

      local l4 = l1 * l3 / 2 in
      {|
         y := l4
         z := l2 - l4 * l3 / 2
      |}
|}
...
```

Figure 5.19: Redundant Update Elimination Pass Example

CHAPTER 6

Evaluation

The evaluation chapter consists of three parts. Section 6.1 presents the performance of the new compiler run-time and code generation. Section 6.2 covers the performance aspects of a not optimized and an optimized CASM program through the compiler optimization framework. The last Section 6.3 summarizes the performance of a real application.

6.1 Compiler

6.1.1 CASM Interpreter vs Legacy Compiler vs Compiler

In this section the quality of the new CASM run-time is evaluated by comparing it to the prototype implementation from Lezuo [46] (aka *legacy compiler*) and to the interpreter implementation from Inführ [38]. All three implementations are using the same input programs to stress several aspects of the CASM run-time.

The program *matrix.casm* implements simple 2-ary function modification with very small update amounts. It operates over the function domain of 10×10 to stress the performance of handling functions with multiple arguments. This program performs only 1000 ASM steps. The program *bubblesort* is a naive implementation of the bubble sort algorithm. It is used to analyze the performance of updating CASM function locations with a small update set. The *array* which is sorted is a unary function with relation `Int` $\rightarrow$ `Int` and the used value domain of the `Int` is $[0, 30]$. The program *bubblesort-large* implements the same program as *bubblesort* but the used *array* range is $[0, 90]$. This *large* data-set shall outline the effect of larger update-set sizes per top-level rule invocation. The program *fibonacci* is a simple implementation of the recursive definition of the fibonacci numbers. This program prints all fibonacci numbers until the program reaches the fibonacci number 2014. The program *rulecalls* evaluates the indirect rule call behavior. A function holds a counter which is interpreted as an address and through this address a program memory function can be accessed which stores a rule reference which shall be executed next. Like in a micro-architecture the called rule performs some operation and writes the result back to the memory. This program executes 4095 ASM steps. The last example program is *trivial*. It consists only of one rule and an update to terminate the program. The execution time of this program equals the setup time of the run-time.

For benchmarking a 64-bit Gentoo Linux-based host system is used with an Intel Core i5 at 1.80 GHz[1]. Additionally the CASM tool[2] (interpreter, legacy compiler and compiler) and GCC C compiler[3] is used. In this section no CASM optimizations are performed, because the focus is only one the run-time and code generation of the new implementation. The influence of the performed C optimizations is considered. All the above mentioned programs are executed with the interpreter (abbr. *casmi*), the legacy compiler (abbr. *casmc++*) and the CASM compiler (abbr. *casm*). The generated code is compiled twice – without (abbr. *-O0*) and with (abbr. *-O3*) C compiler optimizations.

As mentioned in the CASM kernel description (see Section 4.2.4), in the new run-time it is possible to configure the update-set size. In the following evaluation the *casm* uses the default update-set size and *casm-us* uses the specific update-set size. The configured update-set size value is 1000 updates. Table 6.1 shows the execution times of all generated programs for their *-O0* and *-O3* setting and also the execution time of the CASM interpreter.

Benchmark	[s] **casmi**	[s] **casmc++ -O0**	[s] **casmc++ -O3**	[s] **casm -O0**	[s] **casm-us -O0**	[s] **casm -O3**	[s] **casm-us -O3**
matrix	21.2951	4.7028	0.7245	0.6987	0.6261	0.2273	0.1442
bubblesort-large	3.6065	1.0286	0.1285	0.1688	0.0849	0.1112	0.0275
fibonacci	0.7351	0.0181	0.0075	0.0865	0.0032	0.0863	0.0025
rulecalls	0.1711	0.0486	0.0133	0.0893	0.0129	0.0871	0.0099
bubblesort	0.0238	0.0077	0.0034	0.0868	0.0020	0.0844	0.0017
trivial	0.0069	0.0025	0.0026	0.0839	0.0015	0.0851	0.0016

Table 6.1: Benchmark of CASM Interpreter/Legacy Compiler/Compiler

The programs are sorted descending to the execution time from *casmi*. The interpreter (as expected) needs the most time to perform all the benchmark programs. Depending on the produced update-set sizes the execution time of the *casmc++* and *casm* differ a lot. If small update-set sizes are involved (*fibonacci*, *rulecalls*, *bubblesort* and *trivial*) the *casmc++ -O0*, *casmc++ -O3* and *casmi* are faster than the *casm -O0* and *casm -O3*. But if the measured setup time in *trivial* is almost equally big as the execution time it can be observed that the creation of a predefined huge update-set (*matrix* and *bubblesort-large*) needs all the execution time. This is confirmed by the execution times from *casm-us -O0* and *casm-us -O3*, because now the setup is really short. The used starting update-set size is 1000.

In the *trivial* program the difference from *casmi* to *casm-us -O3* is about a factor of 4.3 and from *casmc++ -O3*) to *casm-us -O3* about a factor of 1.6. The simple *fibonacci* program differs

[1]**uname -a:** GNU/Linux air 3.9.0-gnu #2 SMP x86_64 Intel(R) Core(TM) i5-3427U CPU @ 1.80GHz
[2]**casm –version:** 2724b59 Mon Mar 17 12:35:36 CET 2014
[3]**gcc –version:** gcc (Gentoo 4.7.1 p1.5, pie-0.5.3) 4.7.1

from *casmi* to *casm-us -O3* about a factor of 294 (!) and the fastest compilation from legacy and normal compiler about a factor of 3. More impressive are the result differences for bigger update-set sizes. In the example *matrix*, a factor of *147* is between *casmi* and *casm-us -O3*. *casmc++ -O3* and *casm-us -O3* differ about a factor of 5. Figure 6.1 shows the comparison of all executions with *casm-us -O3* as baseline.

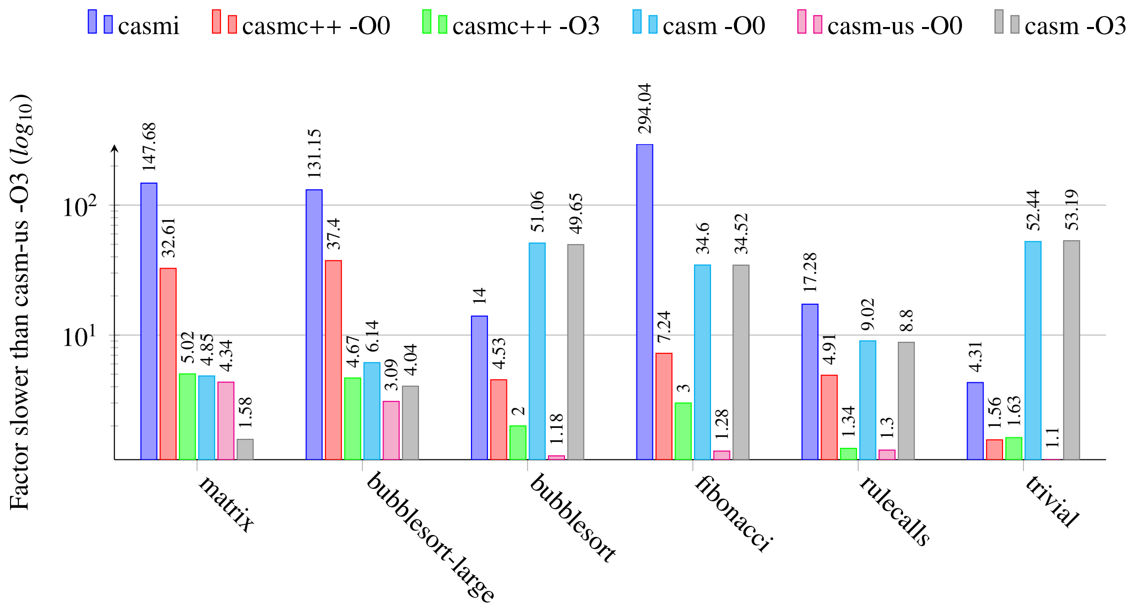


Figure 6.1: Execution Time Comparison of CASM Interpreter/Legacy Compiler/Compiler

6.1.2 AsmL vs CoreASM vs CASM

Lezuo, Paulweber and Krall present in [49] an evaluation of the not optimizing CASM compiler (abbr. *casm*) by comparing it to the CASM interpreter (abbr. *casmi*), the AsmL compiler and the CoreASM interpreter. Similar programs are used as in Section 6.1.1.

The program *quicksort* implements a quick sort algorithm to sort an array. It performs many steps with small update-set sizes. The program *sieve* implements Eratosthenes famous prime number sieve. The benchmark *gray* calculates Gray codes of arbitrary word lengths. The program *fibonacci* uses a dynamic programming approach to calculate the Fibonacci numbers. *bubblesort* implements the bubble sort algorithm to sort an array. The program *trivial* implements no behavior at all. It is used to evaluate the setup time of the ASM implementations. Furthermore, for all examples except *trivial* exists two variants. One with a small and a large data set. The small data set variants are used to evaluate CoreASM and CASM. The large data set variants are used to evaluate AsmL and CASM. Table 6.2 shows the measured execution times.

The results of the benchmark especially for the AsmL compiler are varying a lot. For sorting and simple programs like *bubblesort*, *quicksort* and *fibonacci* the AsmL compiler is slower only

	[s]	[s]	[s]	[s]
Benchmark	**AsmL**	**CoreASM**	**casmi**	**casm**
quicksort	-	32.51	0.021	0.0842
quicksort-large	3.063	-	35.41	0.5860
sieve	-	13.82	0.100	0.0857
sieve-large	74.39	-	1.050	0.0822
gray	-	57.61	0.229	0.0882
gray-large	24.37	-	40.83	0.7702
fibonacci	-	67.24	0.011	0.0854
fibonacci-large	4.175	-	79.17	3.0436
bubblesort	-	213.6	0.047	0.0859
bubblesort-large	5.275	-	95.43	2.5458
trivial	0.129	1.360	0.005	0.0865

Table 6.2: Execution Times of AsmL, CoreASM and CASM [49]

about a factor of 2 to 5. But the *sieve* benchmark results show that the CASM compiler performs over a factor of 900 better than AsmL. The interpreter CoreASM does not vary so much but e.g. the *bubblesort* program is over 2480 times slower than the CASM compiler. Figure 6.2 shows the relative performance of the benchmark programs with the CASM compiler as baseline.

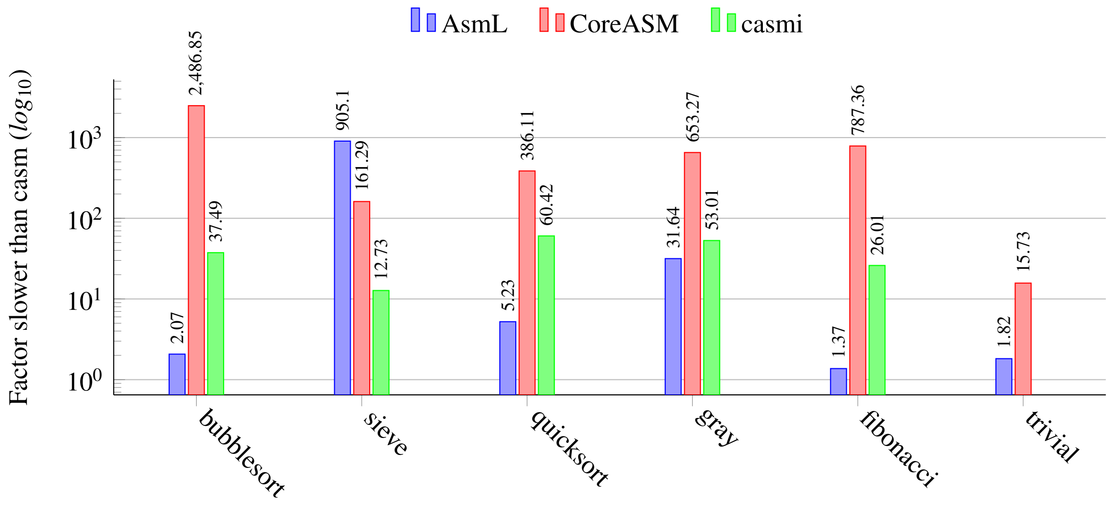


Figure 6.2: Execution Time Comparison of AsmL, CoreASM and CASM [49]

6.2 Optimizing Compiler

The same host system is used for the evaluation of the optimizing compiler. The CASM compiler and GCC C compiler are the same as in Section 6.1.

The execution time baseline is the *casm -O0* and with no activated CASM optimizations. The optimization focus is especially on the *Redundant Lookup Elimination* and *Redundant Update Elimination* pass. All other passes are also activated to achive maximal lookup and update eliminations, because some passes are needed to resolve a `forall` for better copy propagation etc.

The benchmark program *compsim* (compiled simulation or synthesized simulation [25]) is a very important benchmark application. It is a conceptual model of an ISS of a pipelined micro-architecture [12]. Furthermore, the compiled ISS model does not execute arbitrary input programs. A specific application is translated (compiled) to CASM source code and combined with the ISS which is also described in CASM source code. Such models are used by Lezuo [45] to evaluate different MIPS micro-architectures.

The ISA of the example program *compsim* consists of three instructions which are modeled through rules (abbr. *instr.*). The instruction cycle is modeled by the rules *fetch*, *execute* and *step*. Such compiled simulation models consists of *basic blocks* which contain a sequence of rule calls to the instruction cycle rules in a sequential composition block. The *compsim* benchmark consists of only one rule named *bb_X* which represents an example *basic block* rule with six times invoking the instruction cycle rules. This rule is called multiple times from the CASM kernel which simulates the model cycles. This rule is the focus of the optimization. Table 6.3 summarizes the run-time costs of lookups and updates in the different rules and the total for the *bb_X* rule.

Rule	**Lookups**	**Updates**	**Calls**
fetch	0	1	0
execute	3	0	2
step	2	2	0
instr.	2	2	0
bb_X	*144*	*48*	-

Table 6.3: Run-Time Costs of compsim Rule bb_X

Due to the sequences of the rule calls to the instruction cycle rules in the *bb_X* rule, the optimizing compiler is able to remove redundant lookups and updates of the used pipeline which is modeled as a CASM function. Table 6.4 presents the pass statistics (see Section 5.2.2) and optimization results of the used passes to optimize the program *compsim*.

During the optimization of the *compsim* program, the compiler spends over 75% of the time in analyses. Additionally, 17.50 % of the time is spent in constructing the CASM IR. The results show that over 1000 AST nodes were modified (abbr. *Modif.*) in this small example application.

The *Redundant Lookup Elimination* pass removes 38 lookups and the *Redundant Update*

Compiler Pass	[#] **Runs**	[ms] **Time**	[%] **Time**	[#] **Modif.**	[%] **Modif.**
Redundant Update Elimination	2	0.340	0.18	15	1.39
Redundant Lookup Elimination	5	1.669	0.88	38	3.52
Use Definition	6	75.916	40.15	0	0.00
Dead Code Elimination	7	0.545	0.29	3	0.28
Reaching Definition	6	69.271	36.63	0	0.00
Inline Rule Statement	10	0.395	0.21	48	4.44
Constant Propagation	16	2.671	1.41	398	36.82
Inline Derived Expression	8	0.706	0.37	0	0.00
Case To If Conversion	8	0.113	0.06	0	0.00
Loop Unwinding	9	0.268	0.14	6	0.56
Intermediate Representation	21	33.096	17.50	0	0.00
Constant Folding	18	2.831	1.50	227	21.00
Dead Branch Elimination	10	0.768	0.41	75	6.94
Trivial Block Removal	18	0.461	0.24	271	25.07
Skip Removal	5	0.041	0.02	0	0.00
Total	*151*	*189.117*	*100.00*	*1081*	*100.00*

Table 6.4: Pass Statistics of Optimized compsim Rule bb_X

Elimination removes 15 updates. These numbers seem quite small compared to the calculated run-time costs of produced updates and performed lookups in the *bb_X* rule from Table 6.3. But the impact is impressive which is shown in the following paragraphs. All other changes of the AST like the *Copy Propagation* pass which modifies 398 AST nodes is needed to support the redundant elimination passes. Especially the *Dead Branch Elimination* promotes the *Use Definition* analysis pass results. Figure 6.3 visualizes the usage in percent of pass iterations, optimization times and modifications of AST nodes[4].

The generated program, not optimized (abbr. *casm*) and optimized (abbr. *opt. casm*), is compiled to C twice with use of the optimization flags *-O0* and *-O3*. The binaries are executed several times with different upper bound of CASM kernel steps (see Section 4.2.4). The ASM kernel steps equal the ISS model cycles. Table 6.5 summarizes the execution times of all four different compiled *compsim* programs.

By comparing *casm -O0* and *opt. casm -O0* it can be observed that the presented transformations (see Section 5.4) perform very well. The optimized version is 4 times faster than the not optimized version. Both examples do not use any C compiler optimization. By comparing *casm -O0* and *casm -O3* it can be observed that the C compiler with all optimizations enabled achieves only a speedup factor of 3.5. By enabling all optimizations (*opt. casm -O3*), CASM compiler and C compiler, the execution is about a factor 10 faster.

The direct comparison of *casm -O3* and *opt. casm -O3* concludes that the CASM optimiza-

[4]the passes *Case To If Conversion*, *Definition Use* and *Skip Removal* are not visualized in the figure

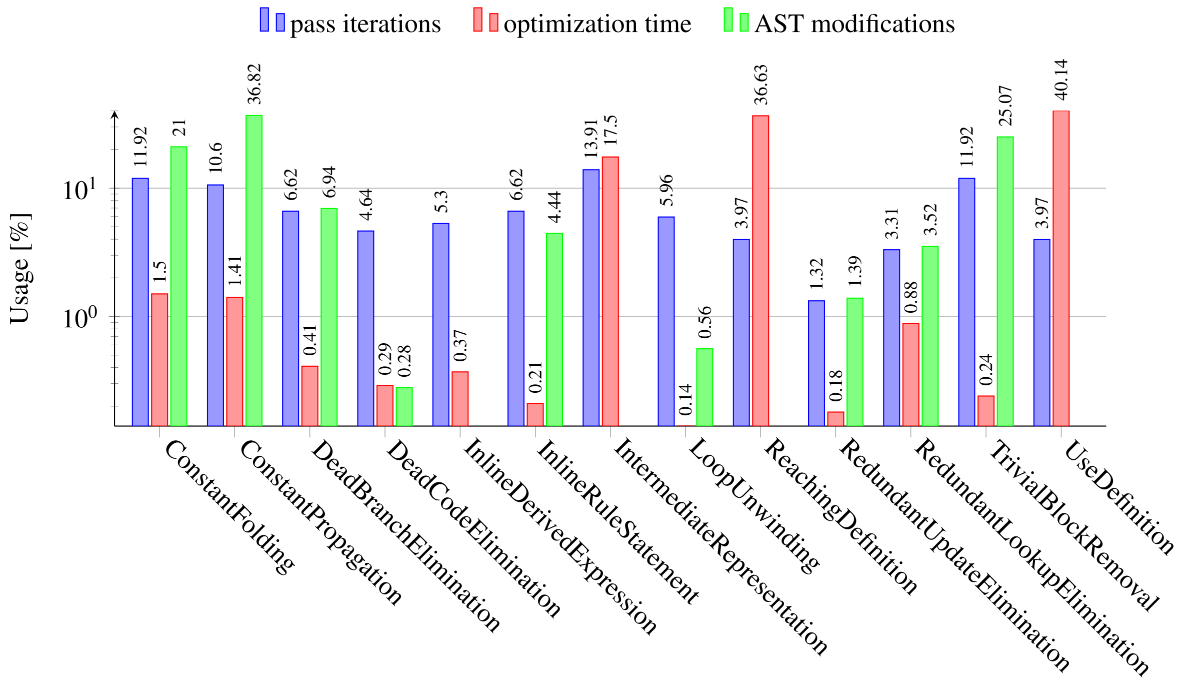


Figure 6.3: Pass Statistics of Optimized compsim

[model cycles] **compsim**	[s] **casm -O0**	[s] **casm -O3**	[s] **opt. casm -O0**	[s] **opt. casm -O3**
100000	2.0728	0.6403	0.5505	0.2774
250000	5.0680	1.4976	1.2325	0.5109
500000	10.0045	2.8505	2.3760	0.9296
1000000	19.8885	5.6083	5.6300	1.8677
2500000	49.4895	13.7084	11.7069	4.3579
5000000	100.5347	28.0327	23.0430	8.6518
10000000	200.6449	55.5215	50.1825	17.1121

Table 6.5: Benchmark of compsim

tions improve the execution speed by a factor 3. Furthermore, by increasing the CASM kernel steps the speedup factor especially for the *opt. casm -O3* execution has an increasing tendency. Figure 6.4 visualizes the speedup factors of all *compsim* versions compared to the not optimized version (*casm -O0*) as baseline.

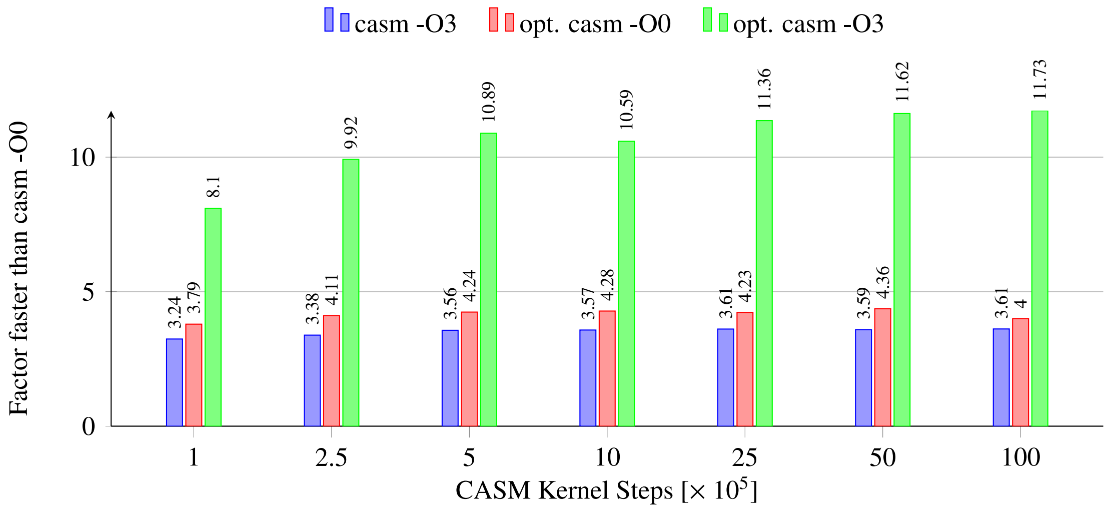


Figure 6.4: Execution Time Speedup of compsim

6.3 MIPS Instruction Set Simulator

Lezuo uses in [45] a MIPS ISS functional (abbr. *smips*), forwarded pipeline (abbr. *mips*) and bubbled pipeline (abbr. *bmips*) model. First evaluations were performed by Lezuo in [48] with the CASM legacy compiler. Those benchmark results achieved for the *smips* $\approx$ 1 MHz, the *mips* $\approx$ 50 kHz and the *bmips* $\approx$ 40 kHz simulation speed. The new CASM compiler has been used in [45] to re-evaluate all the benchmarks from [48]. The results with the new CASM compiler achieves for the *smips* $\approx$ 2.47 MHz, the *mips* $\approx$ 256 kHz and the *bmips* $\approx$ 224 kHz simulation speed.

Furthermore, Lezuo presents in [45] the *compiled simulation* of the three MIPS models. All models are optimized by the CASM optimization framework and the evaluation of those really huge programs (over 1000 to 5000 rules) show performance gains of factor 2 to 3.

CHAPTER 7

Conclusion

First this thesis presents the ASM-based general purpose programming language CASM. For this language there exists an all-in-one tool which includes numeric and symbolic execution, and an optimizing source-to-source compiler which emits C code. The new run-time and code generator supports the complete CASM syntax.

The main contribution of this thesis is the design and implementation of the new CASM run-time, code generator, optimization framework and its optimization analysis and transformation passes.

For the run-time and code generator part the novel idea is to represent n-ary CASM functions through a *branded hash-map* data-structure (see Section 4.2.2). Furthermore, the idea to implement a specialized update-set structure similar to a *linked hash-map* data-structure improved the overall update handling, forking, merging, applying and the general run-time significantly (see Sections 4.2.3 and 6.1).

In the optimization framework part, the idea to construct a specialized CASM IR from the typed AST enables very powerful analyses and transformations. The CASM IR includes a *PAR/SEQ Control Flow Graph* forward flow iteration which enables the very powerful CASM specific *Use Definition* analysis pass (see Section 5.4.3). This analysis information is used in the *Redundant Lookup Elimination* transformation pass (see Section 5.4.4) to minimize the amount of location lookups into the update-set of the run-time.

The evaluation in Chapter 6 shows the new compiler implementation performance and optimization impact. With the optimization framework the execution speed of a CASM program increases significantly.

7.1 Future Work

The following sections are outlining further improvements and ideas for the CASM compiler and its optimization framework.

7.1.1 Run-Time & Code Generation

In the run-time, the memory allocator (see Section 4.2.1) can be even further improved by different memory allocation/deallocation algorithms. Furthermore, the current implementation of the `List` type is copy-based which introduces huge drawbacks regarding memory consumption.

Another idea is to generate the C code from the CASM IR, because currently the C code is generated from the typed AST. If the AST changes, the construction of the CASM IR in the *Intermediate Representation* pass (see Section 5.4.2) and the C code generator (see Section 4.3) has to be adopted, because both use the AST as input representation.

7.1.2 Optimizations

During the work of this thesis some new optimization ideas came up. The following subsections give a short overview of new analyses and transformations for the CASM compiler.

Top-Level Rule Pass A *Top-Level Rule* pass could analyze the top-level rule calls of the CASM kernel. It could detect if the top-level rule never gets undefined and the execution behavior results in an infinite loop.

Integer Range Detection Pass To further improve the lookup and update behavior of not ranged `Int` types an *Integer Range Detection* pass similar to the *ABCD* algorithm [8] can be implemented which determines the smallest possible range for a CASM function over all rules.

Integer Range Check Elimination Pass With the analysis information from the *Integer Range Detection* pass, the *Integer Range Check Elimination* pass removes partially the built-in range checking for accessing function arguments and updating the co-domain of a function.

Integer Function Domain Modification Pass The *Integer Function Domain Modification* pass can use the analysis information from the *Integer Range Detection* pass to limit globally the function domain to a specific range, which will improve the generated code and the execution time significantly.

Location Address Propagation Pass Due to the concept of memory location fixed addresses in the run-time (see Section 4.2.2) a specific location address can be ‘buffered’ to avoid re-calculations in the run-time.

Update Pseudo-State Modification Pass Every update is inserted into the update-set with a corresponding pseudo-state counter value. This value has to be incremented in each merge which implies that if an update is nested very deeply it is decremented by the nesting depth. The *Update Pseudo-State Modification* could ‘mark’ an update to be inserted at a specific position in the update-set *linked hasp-map* structure to minimize the decrement cycles at every update-set merge operation (see Section 4.2.3).

A Appendix

A.1 List of Acronyms

API	Application Programming Interface
ASM	Abstract State Machine
ASM-SL	Abstract State Machine-based Specification Language
ASMETA	ASM mETAmodelling framework
AST	Abstract Syntax Tree
C3Pro	Correct Compilers for Correct Application Specific Processors
CASM	Corinthian Abstract State Machine
CCS	Calculus of Communication Systems
CFG	Control Flow Graph
CPU	Central Processing Unit
DFA	Data-Flow Analysis
EBNF	Extended Backus-Naur Form
ELF	Executable and Linking Format
GCC	GNU Compiler Collection
GNU	GNU's Not Unix
IR	Intermediate Representation
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator
LLVM	Low Level Virtual Machine
LOC	Lines Of Code
MDA	Model Driven Architecture
MOF	Meta-Object Facility
OMG	Object Management Group
OS	Operating System
RAM	Random Access Memory
SML	Standard Meta-Language

SMT	Satisfiability Modulo Theories
SSA	Single Static Assignment
STL	Standard Template Library
TASM	Timed Abstract State Machine
UID	Unique Identifier
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VDM	Vienna Development Method
XASM	eXtensible Abstract State Machine
XMI	XML Metadata Interchange
XML	Extensible Markup Language

A.2 List of Listings

A.3 List of Figures

A.4 List of Tables

Bibliography

[1] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, & Tools*, volume 1009. Pearson/Addison Wesley, 2007.

[2] Matthias Anlauff. XASM - An Extensible, Component-Based Abstract State Machines Language. In *Abstract State Machines*, Lecture Notes in Computer Science, pages 69–90. Springer, 2000.

[3] Nikolas Askitis. Fast and Compact Hash Tables for Integer Keys. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science-Volume 91*, pages 113–122. Australian Computer Society, Inc., 2009.

[4] Phil Bagwell. Fast And Space Efficient Trie Searches. Technical report, 2000.

[5] Phil Bagwell. Ideal Hash Trees. *Es Grands Champs*, 1195, 2001.

[6] Masanori Bando and H Jonathan Chao. FlashTrie: Hash-based Prefix-Compressed Trie for IP Route Lookup Beyond 100Gbps. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[7] Rajkishore Barik and Vivek Sarkar. Interprocedural Load Elimination for Dynamic Optimization of Parallel Programs. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 41–52. IEEE, 2009.

[8] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *ACM SIGPLAN Notices*, volume 35, pages 321–333. ACM, 2000.

[9] Egon Börger. The Origins and the Development of the ASM Method for High Level System Design and Analysis. *Journal of Universal Computer Science*, 8(1):2–74, 2002.

[10] Egon Börger. The Abstract State Machines Method for High-Level System Design and Analysis. In *Formal Methods: State of the Art and New Directions*, pages 79–116. Springer, 2010.

[11] Egon Börger and Joachim Schmid. Composition and Submachine Concepts for Sequential ASMs. In *Computer Science Logic*, pages 41–60. Springer, 2000.

[12] Florian Brandner, Nigel Horspool, and Andreas Krall. DSP instruction set simulation. In *Handbook of Signal Processing Systems*, pages 945–974. Springer, 2013.

[13] Ulrich Breymann. *Designing Components with the C++ STL.* Addison-Wesley, 1998.

[14] C3Pro - Correct Compilers for Correct Application Specific Processors. `http://www.complang.tuwien.ac.at/c3pro`. Accessed: 2013-02-15.

[15] CoreASM - Main Page. `http://sourceforge.net/apps/mediawiki/coreasm`. Accessed: 2014-02-14.

[16] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[17] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. CETUS: A SOURCE-TO-SOURCE COMPILER INFRASTRUCTURE FOR MULTICORES. *Computer*, 42(12):36–42, 2009.

[18] Saumya K Debray. On Copy Avoidance in Single Assignment Languages. In *ICLP*, pages 393–407, 1993.

[19] Giuseppe Del Castillo. The ASM Workbench: an Open and Extensible Tool Environment for Abstract State Machines. In *Workshop on Abstract State Machines*, pages 139–154. Citeseer, 1998.

[20] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):992–1030, 1997.

[21] Eclipse Project Homepage. `http://www.eclipse.org`. Accessed: 2014-02-24.

[22] Stephen A Edwards. SHIM: A Language for Hardware/Software Integration. *Synchronous Programming-SYNCHRON'04*, pages 1–6, 2004.

[23] Daniel J Ernst, Daniel E Stevenson, and Paul J Wagner. Hybrid and Custom Data Structures: Evolution of the Data Structures Course. *ACM SIGCSE Bulletin*, 41(3):213–217, 2009.

[24] Roozbeh Farahbod, Vincenzo Gervasi, and Uwe Glässer. CoreASM: An Extensible ASM Execution Engine. *Fundamenta Informaticae*, 77(1):71–103, 2007.

[25] Stefan Farfeleder, Andreas Krall, and Nigel Horspool. Ultra fast cycle-accurate compiled emulation of inorder pipelined architectures. *Journal of Systems Architecture*, 53(8):501–510, 2007.

[26] Emden R Gansner. Drawing Graphs with Graphviz. Technical report, Technical report, AT&T Bell Laboratories, Murray, 2009.

[27] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A Metamodel-based Language and a Simulation Engine for Abstract State Machines. *J. UCS*, 14(12):1949–1983, 2008.

[28] Lars Marius Garshol. BNF and EBNF: What are they and how do they work? *acedida pela última vez em*, 16, 2005.

[29] GCC, the GNU Compiler Collection. `http://gcc.gnu.org/`. Accessed: 2014-03-08.

[30] Arthur M Geoffrion. The SML Language for Structured Modeling: Levels 1 and 2. *Operations Research*, 40(1):38–57, 1992.

[31] Sabine Glesner. An ASM Semantics for SSA Intermediate Representations. In *Abstract State Machines 2004. Advances in Theory and Practice*, pages 144–160. Springer, 2004.

[32] Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. *Specification and Validation Methods*, pages 9–36, 1995.

[33] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. Semantic essence of AsmL. *Theoretical Computer Science*, 343(3):370–412, 2005.

[34] Yuri Gurevich and Nikolai Tillmann. Partial Updates: Exploration. *Journal of Universal Computer Science*, 7(11):917–951, 2001.

[35] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Adobe Press, 2006.

[36] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, et al. Report on the Programming Language Haskell: A Non-strict, Purely Functional Language Version 1.2. *ACM SigPlan notices*, 27(5):1–164, 1992.

[37] James K Huggins and Wuwei Shen. The Static and Dynamic Semantics of C. In *Local Proc. Int. Workshop on Abstract State Machines*, pages 272–284, 2000.

[38] Dominik Inführ. AST interpreter for CASM. Bachelor's thesis, Vienna University of Technology, Karlsplatz 13, 1040 Vienna, 2013.

[39] Mark P. Jones. GOFER Gofer 2.28 release notes. *Departement of Computer Science, Yale University, Februar*, 1993.

[40] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *{Java}(TM) Language Specification*. Addison-Wesley, 2000.

[41] Brian W Kernighan, Dennis M Ritchie, and Per Ejeklint. *The C Programming Language*, volume 2. Prentice-Hall Englewood Cliffs, 1988.

[42] David J Kuck, Robert H Kuhn, David A Padua, Bruce Leasure, and Michael Wolfe. DEPENDENCE GRAPHS AND COMPILER OPTIMIZATIONS. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–218. ACM, 1981.

[43] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[44] Michael E Lesk and Eric Schmidt. Lex: A Lexical Analyzer Generator, 1975.

[45] Roland Lezuo. *Scalable Translation Validation*. Dissertation, Vienna University of Technology, Karlsplatz 13, 1040 Vienna, 2014.

[46] Roland Lezuo, Gergö Barany, and Andreas Krall. CASM: Implementing an Abstract State Machine based Programming Language. In *Software Engineering (Workshops)*, pages 75–90, 2013.

[47] Roland Lezuo and Andreas Krall. A Unified Processor Model for Compiler Verification and Simulation Using ASM. In *ABZ*, Lecture Notes in Computer Science, pages 327–330. Springer, 2012.

[48] 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.

[49] Roland Lezuo, Philipp Paulweber, and Andreas Krall. CASM - Optimized Compilation of Abstract State Machines. ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES), 2014.

[50] LLVM's Analysis and Transform Passes. `http://llvm.org/docs/Passes.html`. Accessed: 2013-02-20.

[51] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores. In *ACM SIGPLAN Notices*, volume 33, pages 26–37. ACM, 1998.

[52] Microsoft Research - AsmL: Abstract State Machine Language. `http://research.microsoft.com/en-us/projects/asml`. Accessed: 2014-02-16.

[53] Steven S. Muchnick. Advanced compiler design implementation. 1997.

[54] Diego Novillo. Tree SSA A New Optimization Infrastructure for GCC. In *Proceedings of the 2003 GCC Developers' Summit*, pages 181–193, 2003.

[55] Gerard O'Regan. Vienna Development Method. *Mathematical Approaches to Software Quality*, pages 92–108, 2006.

[56] Martin Ouimet, Kristina Lundqvist, and Mikael Nolin. The Timed Abstract State Machine Language: An Executable Specification Language for Reactive Real-Time Systems. *RTNS'07*, page 15, 2007.

[57] Overview: Abstract State Machine Metamodel (AsmM, Asmeta). `http://asmeta.sourceforge.net/`. Accessed: 2014-02-13.

[58] Dan Quinlan and Chunhua Liao. The ROSE Source-to-Source Compiler Infrastructure. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT 2011*, October 2011.

[59] J. Schmid. *Introduction to AsmGofer*, 2001.

[60] Joachim Schmid. Compiling Abstract State Machines to C++. *Journal of Universal Computer Science*, 7(11):1068–1087, 2001.

[61] Michael D Smith and Glenn Holloway. An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization. *Division of Engineering and Applied Sciences, Harvard University*, 2002.

[62] Richard M Stallman. Using and Porting the GNU Compiler Collection. *Free Software Foundation*, 51:02110–1301, 1989.

[63] Jürgen Teich, Philipp W Kutter, and Ralph Weper. Description and Simulation of Microprocessor Instruction Sets Using ASMs. In *Abstract State Machines-Theory and Applications*, pages 266–286. Springer, 2000.

[64] Stefanus Du Toit. Working Draft, Standard for Programming Language C++. Technical report, Technical Report, 2013.

[65] Guido Van Rossum and Fred L Drake. *Python Language Reference Manual*. Network Theory, 2003.

[66] Ton Vullinghs, Wolfram Schulte, and Thilo Schwinn. *An Introduction to TkGofer*. Univ., Fak. für Informatik, 1996.

[67] What is AsmGofer. `http://www.tydo.de/doktorarbeit/asmgofer`. Accessed: 2014-02-15.

[68] Robert P Wilson, Robert S French, Christopher S Wilson, Saman P Amarasinghe, Jennifer M Anderson, Steve WK Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W Hall, Monica S Lam, et al. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM Sigplan Notices*, 29(12):31–37, 1994.

[69] Haim J Wolfson and Isidore Rigoutsos. Geometric Hashing: An Overview. *Computing in Science and Engineering*, 4(4):10–21, 1997.

CASM Run-Time API

Index