CASM: Implementing an Abstract State Machine based Programming Language [*]

Roland Lezuo, Gergö Barany, Andreas Krall

Institute of Computer Languages (E185)
Vienna University of Technology
Argentinierstraße 8
1040 Vienna, Austria
{rlezuo,gergo,andi}@complang.tuwien.ac.at

Abstract: In this paper we present CASM, a general purpose programming language based on abstract state machines (ASMs). We describe the implementation of an interpreter and a compiler for the language. The demand for efficient execution forced us to modify the definition of ASM and we discuss the impact of those changes. A novel feature for ASM based languages is symbolic execution, which we briefly describe. CASM is used for instruction set simulator generation and for semantic description in a compiler verification project. We report on the experience of using the language in those two projects. Finally we position ASM based programming languages as an elegant combination of imperative and functional programming paradigms which may liberate us from the von Neumann style as demanded by John Backus.

1 Introduction

Most of the well known programming languages have a sequential execution semantics. Statements are executed in the order they appear in the source code of the program (imperative programming). The effects of a statement – changes to the program's state – are evaluated before the next statement is considered. Examples of such programming languages are assembly languages, C, Java, Python and many more. Actually there are so many languages based on the imperative paradigm that it may even feel ”natural”.

However, there seems to be a demand for alternative programming systems. John Backus even asked whether programming can be liberated from the von Neumann style [Bac78]. John Hughes tries to convince the world that functional programming matters [Hug89]. Hudak et al. conclude that, although not used by the masses, aspects of functional programming are being incorporated into main stream imperative languages [HHPJW07].

Functional languages describe side-effect free application of functions to their arguments which makes them presumably easier to understand. They struggle to gain real-world

[*]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.

acceptance however. While not claiming to be a representative source, but merely an indicator, the popular source code repository hoster github[1] (more than 4 million repositories) lists no functional language amongst its 10 most popular ones. Actually, even assembly languages (17th most popular) are more popular than Haskell (18th most popular with approx. 16000 projects using it) according to github's language statistics.

In this paper we report on CASM a general purpose programming language based on Abstract State Machines (ASMs). ASMs were introduced by Gurevich (originally named evolving algebras) in the Lipari Guide [Gur95]. One of the core concepts of ASM (as the name indicates) is the state. Another core concept of ASM is the concept of a rule, which describes exactly how the state is changed by means of *updates* applied to the state. Application of a rule itself is side-effect free, one of the core concepts of functional programming.

The CASM language was originally designed to describe programming language semantics, a purpose for which ASMs are known to be well suited [SSB01, ZG97, Gau95, KP97, Gle04, HS00, GH93]. We perform correctness proofs in a compiler verification project using symbolic execution. Having the ASM models of a machine language at hands it suggests itself to reuse them for implementing an instruction set simulator [BHK10]. To create a fast simulator we added a type system and developed a compiler for the CASM language.

The remainder of this paper is structured as follows. Section 2 discusses previous work on ASM based programming languages, in section 3 we describe the CASM language, section 4 briefly introduces the implemented type system, section 5 explains implementation details of the interpreter and compiler, in section 6 we report on our experience using CASM in various projects and section 7 finally concludes the paper and gives directions for further work.

2 Related Work

The ASM method is well known and there have been quite a few efforts to create a widely accepted framework for ASM tools. However, most of the projects ceased to exist. One reason it seems to be so difficult to provide a generally accepted framework and language for all ASM users may be the fact that ASM's applications are so broad. On the one hand it is used to create very high level models [Bö03], and on the other it is used to model very low level aspects of hardware (like we do). The demands of the users vary greatly and this may be the reason there is quite a number of attempts to create ASM tools. In a sense this work adds to the misery, but it is driven by the very specific needs of compiler verification (symbolic execution) and simulator synthesis (very fast performance). This section tries to distinguish the CASM language from other ASM based languages.

Schmid introduced AsmGofer in [Sch01b]. AsmGofer is an interpreter for an ASM based language. It is written in the Gofer[2] language (a subset of Haskell) and covers most of the

[1]http://github.com

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

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

Castillo describes the ASM Workbench in [Cas01]. Similar to CASM he added a type system to his language. The ASM Workbench is implemented in ML[3] in an extensible way. Castillo describes an interpreter and a plugin for a model checker, which allows to translate certain restricted classes of abstract state machines to models for the SMV[4] model checker.

Schmid also describes compiling ASM to C++ [Sch01a]. His compiler uses the ASM Workbench language as input. He proposes a double buffering technique avoiding collection update sets to increase runtime performance. There is no report of the achieved performance. CASM uses a so called *pseudo state* (more details are in section 5.1.3) to implement *update sets* efficiently.

Anlauff introduces XASM, a component based ASM language compiled to C [Anl00]. The novel feature of XASM is the introduction of a component model, allowing implementation of reusable components. XASM supports *functions* implemented in C using an *extern* keyword. CASM does not feature modularization, but can be extended using C code as well. XASM was used as the core of the gem-mex system, a graphical language for ASMs.

Farahbod designed CoreASM, an extensible ASM execution engine [FGG05]. The CoreASM project is actively maintained and early prototypes of our compiler verification proofs even used CoreASM. Unfortunately performance of the CoreASM interpreter is very poor, which ultimately lead to the development of CASM. Execution speed was increased by a factor of 3000 [LK12]. The CASM language is heavily inspired by the CoreASM language, but over time they have diverged.

3 The CASM language

In this section we introduce the most important aspects of the CASM language. The Lipari guide [Gur95] contains a formal definition of the constructs presented in this section. Also the CoreASM handbook may be a useful reference [Far] as CASM roots in the CoreASM language. In section 3.7 the most important differences between the two languages are pointed out.

3.1 State and Execution model

The state of an ASM is a so called *static algebra* over a set of universes (types) [Gur95]. These universes form a *superuniverse* X in which at least 3 distinct elements (*true*, *false* and *undef*) are defined. The state contains *r-ary* functions (mapping X^r to X) and relations

[3]`http://en.wikipedia.org/wiki/Standard_ML`
[4]`http://www.cs.cmu.edu/~modelcheck/smv.html`

(mapping X^r to *true, false*). For all *locations* of a function, unless not explicitly defined otherwise, the value *undef* is assigned. CASM programs describe a (potentially infinite) sequence of state changes by alternating calculation and application of *update sets*.

The state of a CASM program is formed by a number of *functions* (possibly 0). An example is given in listing 1.

```
function foo : -> Boolean
function bar : Int -> Int
```

Listing 1: A state

Function *foo* is a *0-ary* function, and is very similar to a global boolean variable in common programming languages. Function *bar* is a *1-ary* function, mapping an *Int* to another *Int*. This can be interpreted as an array, although there are no bounds and the function may be only partially defined. The special value *undef* will be returned for all *locations* where *bar* is undefined. In that sense a *1-ary* CASM *function* is more like a hash-map. The arguments to a function are called a *location*.

CASM programs are formulated as a set of *rules*. There is a top-level rule which will be executed repeatedly until the program terminates itself. A rule can't change the state while being executed, which implies that all calculations are side-effect free. The rule returns the changes to the state, as a so called *update set*. An *update set* is a set of *updates*, each of which describes a *function*, a *location* and the new value. *Update sets* are applied to the state whenever the top-level rule *concludes* (returns). Listing 2 shows an example of a rule (and the state it operates on).

```
function x : -> Int
function y : -> Int
function z : -> Int

rule swap = {
  x := y
  y := x
}
```

Listing 2: Parallel swap

Assuming an initial state of $\{x = 3, y = 2, z = 1\}$ the calculated *update set* of a (single) invocation of *swap* is $\{(x, 2), (y,3)\}$. The update is then applied to the state, yielding $\{x = 2, y = 3, z = 1\}$.

When a *function* is assigned different values (for the same *location*) the update set is said to be *inconsistent*. In CASM a non-revocable runtime error is raised when an update set becomes inconsistent. Listing 3 shows an example for a rule triggering a runtime error.

```
function b : -> Boolean
rule inconsistent = {
  b := true
  b := false
}
```

Listing 3: Inconsistent update set - runtime error

3.1.1 Update Rule and Function Signatures

The previous examples informally introduced the *update* rule (*:*=) and function *signatures*.

A *function signature* defines the types of the function arguments (comma-separated between *:* and ->) and the type the function maps to (after ->).

The general form of an update rule is $f(l) := v$, where f is a *function* and l a *location*. A *location* has to match the *argument types* of the *function signature*. v is an expression of the function's type which will be evaluated and assigned to the *function* at the given *location*. The state will not be changed when the *update rule* is executed, but the effects of the *update* will be added to the *update set* of the environmental rule. Only the *update rule* can change the state (or more precisely, add *updates* to an *update set*).

3.1.2 Types, Composition and Enumerations

CASM offers 3 built-in primitive types. They are *Boolean* for boolean values (*true, false*), *Int* for integer values and *String* for character strings.

An enumeration can be defined using the *enum* keyword. Each enumeration defines a new type with the same name as the enumeration. Each member of the enumeration becomes a new globally unique identifier of the enumeration's type. An example is given in listing 4.

```
enum MyEnum = { one, two, three }
function x : -> MyEnum

rule example =
  x := three
```

Listing 4: Enumeration Type

CASM offers two kinds of type composition. A *List* is a sequence of zero or more elements of equal type. The *Tuple* is a sequence with a fixed number of elements of possibly different type. CASM disallows user-provided recursive data types in the current version. An example is given in listing 5.

```
function stack : -> List(Int)
function aMapping : -> List(Tuple(String,Int))
```

Listing 5: Composition Type

3.2 Expressions, Variables and Derived Values

For *Int* types the usual set of operators is provided via built-ins. Expressions are very similar to the C programming language (without any side-effects however).

CASM does not offer a notion of local state, so there are no local variables in the common sense. There is however the possibility to bind expressions to a local name using the *let* rule. In contrast to CoreASM *let* rules are statically scoped.

```
derived d(a : Int, b : Boolean) = (a >= 3) and b
function foo : Int -> Boolean

rule bar =
  let x = 2*3 in
    let y = 3*x in
      foo(y) := d(x, true)
```
Listing 6: Nested Let Rules

Some expressions may be used extensively in a program and in more than one rule. Such expressions can be declared globally using the *derived* keyword. A *derived* accepts typed arguments, just like *functions*, and is a single expression. Listing 6 gives an example of a *derived* and nested *let* rules (assigning *true* to *function foo* at *location 18*).

3.3 Block Rule and Control Flow

The block rule, syntactically expressed by curly brackets, combines the *update sets* of all enclosed rules into one single *update set*. Each of the enclosed rules is invoked on the same state and because all rules are side-effect free the order in which the rules are invoked does not matter. The resulting *update set* is formed by applying the *union* operator on all calculated *update sets*. When merging is performed *inconsistent updates* need to trigger a runtime error. Listing 2 already made use of the block rule.

The CASM language offers an *if-then-else* rule (an example can be seen in listing 11). In the context of an ASM based language it is an *if-then-else rule* (not a statement). Thinking of *if-then-else* as a rule also helps to comprehend the semantics. *if-then-else* produces an *update set* (like any other CASM rule does). The update set is either the update set produced by the rule in the *if-branch* or the one in the *else-branch*, depending on the boolean value of the *expression* following the *if* keyword.

There also is a *switch-case* statement, with the usual semantics and an optional *default* case label.

```
rule r1(a:Int) = skip

rule r2 =
  call r1(3)

rule r3 =
  let rr = @r1 in
    call (rr)(5)
```
Listing 7: Direct and indirect call

The *call* rule invokes another rule. There are two flavors of the call rule, a direct and an indirect one. In the direct case the rule to be invoked is known statically and directly coded in the source file, while in the indirect case the rule to be invoked is calculated by an *expression* of type *RuleRef*. A *RuleRef* can be produced by the @ operator (similar to C's & operator). Listing 7 shows a direct and an indirect call (note the additional brackets around the expression). One can also see that rules can have typed arguments. CASM's *call* rule differs from the definition given in the Lipari guide, see section 3.7 for more details.

3.4 Sequential Block Rule

Some problems can naturally be described by imperative programming. CASM supports sequential programming by means of the *seqblock* rule. Statements enclosed by a *seqblock* are executed in exactly the specified order. Additionally the state change induced by previous rules is visible to subsequent ones. The *update set* of a *seqblock* rule is calculated by subsequently merging the *update sets* of the enclosed rules. Two updates to the same *function* and *location* from different rules do not conflict however, the later overwrites the previous one. Effectively the *update set* describes what would have happened if the rules were applied using a sequential semantics.

It is important to note that the state is not actually changed. The rules are evaluated using the state which would result from applying the previous update sets. A temporary state is created and each calculated update set is applied to it. This temporary state is discarded at the end of the *seqblock*.

Listing 8 shows an implementation of swap using a *seqblock* rule. Please note the use of the temporary variable *t*, because the update to *x* is visible to the second *update rule* (compare to listing 2). Again assuming an initial state of $\{x = 3, y = 2, z = 1\}$ the *update set* of the first *update rule* is $\{(x,2)\}$. This update set is applied to the initial state resulting in the temporary state $\{x = 2, y = 2, z = 1\}$. The second *update rule* results in the update set $\{(y,3)\}$. Applying to the temporary state the gives $\{x = 2, y = 3, z = 1\}$, this state is discarded however. Merging the update sets gives the update set of the *seqblock* rule itself, it is $\{(x,2), (y,3)\}$. This final update set will be applied to the state when the *swap* rule concludes.

```
function x : -> Int
function y : -> Int
function z : -> Int

rule swap =
  let t = x in
    seqblock
      x := y
      y := t
    endseqblock
```

Listing 8: Sequential swap

3.5 Forall and Iterate

The *forall* rule is used in combination with an iterate-able expression (e.g. a range of integers, an enumeration). For each element of the iterate-able expression the rule given in the body is invoked. The *update sets* produced by the body are understood to be executed in parallel. An example is given in listing 9 where it is used to create a list of 4 elements.

```
function x : -> Int

rule create =
  forall i in [0..3] do
    x(i) := i
```

Listing 9: Forall

The *iterate* rule on the other hand repeatedly invokes its rule body until the produced *update set* is empty. Each invocation is understood to be executed in a sequential manner, so each rule is invoked using the *temporary state* produced by the previous iteration. An example is given in listing 10 where the rule is used to atomically perform a fold operation (using addition) on a list. The update set returned by the *iterate* rule is {*(i,10),(f,45)*}.

```
function a : Int -> Int initially { 0->0, 1->1, /* skipped */ , 9->9 }
function i : -> Int initially { 0 }
function f : -> Int initially { 0 }

rule fold =
  iterate
    if i < 10 then {
      i := i + 1
      f := f + a(i)
    }
```

Listing 10: Iterate

3.6 Stacks and Lists

There are rules and built-in functions which can be used with *List* types. Rules *pop* and *push* are used to implement stacks. The built-in functions *cons*, *peek* and *tail* construct and consume lists. There also is a *nth* function to access the nth element of a list (or tuple). These functions are (parametric) polymorphic in their nature and need to be handled correctly by the type system. Listing 11 gives an example.

```
function list : -> List(Int)

rule foo =
  seqblock
    push 3 into list
    if peek(list) != 3 then assert false
    let x = nth(list, 1) in list := cons(x, list)
  endseqblock
```

Listing 11: List built-ins and rules

3.7 Differences to other ASM based languages

CASM differs in two major aspects from other ASM based programming languages (i.e. CoreASM).

The Lipari guide defines rule arguments to be passed-by-name. Passing by-name has some interesting features, but is difficult to be implemented efficiently [BG93]. In CASM the semantics of the *call* rule specify arguments to be passed by-value. Please note that pass-by-value is semantically equivalent to pass-by-name when all arguments are constants. Therefore the restricted *call* rule of CASM can be simulated by evaluating all argument expressions (e.g. using a *let* rule) before invoking an unrestricted *call*. We merely enforce this policy on the CASM programmer by performing this evaluation before invoking the called rule.

The other difference is the scope of variables introduced by the *let* rule. CoreASM installs variable names into an environment passed to rules invoked by a *call* rule. So listing 12 is valid in CoreASM, but not in CASM. Environments are not passed to rules invoked by a *call* rule in CASM. The main reason for this is that the type inference and type checking would be unable to handle this.

```
function y : -> Int

rule callee = y := x
rule caller =
  let x = 3 in
    call callee()
```

Listing 12: Only valid in CoreASM

4 CASM Type System

CASM is a static[5], strongly typed language and no implicit type conversion is performed. To reduce the often redundant notation of types the programmer is allowed to omit the type if it can be deduced automatically. CASM only demands types for the arguments of *functions*, *rules* and *derived* as well as for a *function's type*. Optionally the programmer can provide type information for the type of a *derived* and a named expression type bound via a *let* rule. This may improve readability of the source code and may be needed to guide the type inference system in some corner cases. These corner cases often result from the special value of *undef* which is compatible to every type.

```
function assign : -> List(Int)

rule foo =
  let uList : List(Int) = undef in
    let uElem = nth(1, uList) in
      assign := [ uElem ]
```

Listing 13: Type Annotation needed

[5]except indirect calls

Listing 13 shows an example the CASM type system implementation can not handle without annotation. The first *let* binds the name *uList* to the value *undef*. The second *let* binds the name *uElem* to the *nth* (1st in this case) element of a list or tuple. *nth* (a built-in function) returns *undef* if any of its arguments is *undef*. *uElem's* type could not be computed (locally) when no additional type information would be provided by the programmer. Although limitations in the implementation of the type system exist, they very rarely occur in real world programs.

To specify the argument types of *rules* and *deriveds* may be superfluous, but increases readability and documentation. But especially for *derived* it prevents parametric polymorphism, which may be a useful feature after all. We are considering to change this in a future version of CASM.

5 Interpreter and Compiler

We have developed an interpreter and a compiler for CASM. The interpreter is capable of concrete and symbolic execution of CASM programs. We have also developed a compiler generating C++ code.

The CASM interpreter is a simple abstract syntax tree (AST) interpreter. For creating the parser traditional compiler tools like lex and yacc have been used. Type inference and type checking is performed on the AST. The program is rejected if any types can't be calculated or any type mismatch is detected.

During symbolic execution some (or all) values of the state can hold symbolic instead of concrete values. Evaluation of operators then depends on whether all operands are concrete values or if there is at least one symbolic one. As long as all operands have concrete values the operator is evaluated as usual. But if there is at least one symbolic operand the operator itself returns a new symbol. This returned symbol is linked to the fact that it had been calculated by applying the operator to the specific operands. E.g. an addition of the symbol *s3* and the concrete value *23* will result in a new symbol *s4*. *s4* will be linked to the fact *s4 := s3 + 23*. Should *s4* ever be used as an operand a new symbol will be created as result, linked to the fact that *s4* was used as an operand. Any symbol appearing as the result of a program can that way be traced back to an initial symbol provided as input to the program.

Things get interesting when control flow branches on a symbolic value [Kin76]. For the sake of brevity we only consider *if-then-else* rules here. The *conditional expression* must be of *Boolean* type in CASM, so it can only have two possible values (*true, false*). When the boolean value is symbolic, both possible values need to be considered. The program *forks* assuming the *expression* to be *true* in one case and *false* in the other one. These assumptions become part of the facts known about the symbols. The sum of all facts learnt about symbols along a path of execution is called path condition. Path conditions can be used to automatically generate test cases [VPK04]. The CASM interpreter does not directly support this, but it can easily be implemented by using the generated trace files.

5.1 Compilation scheme and efficient runtime

The typed AST is also used to compile the CASM program to C++. Our current CASM compiler implementation performs only a limited set of optimizations to keep the generated C++ files small. The runtime system makes heavy use of C++ template mechanism and inlining and hence the generated machine code is therefore quite compact resulting in satisfying performance for common CASM code patterns.

5.1.1 Efficient Memory Allocation

During evaluation of a rule a number of updates and update sets have to be generated dynamically. The lifetime of these objects is limited however. All update sets and the updates they contain can safely by deleted when the top-level rule has concluded and all updates have been committed to the state. We therefore allocate these objects in a dump memory area. When the top-level rule concludes, the dump memory is reset and all objects it contained are invalid. New updates are allocated overwriting the old ones. This technique reduces overhead for dynamic memory allocation to almost zero.

5.1.2 Optimization for very small update sets

For the workloads we operate on we observed that most update sets are very small ($<= 4$ in most cases). Our current implementation for update sets therefore use a small number (4) of pre-allocated slots. Only after all pre-allocated slots of an update set are occupied a hash-map is used to store any additional updates part of the update set. A hash-map needs to be used as update sets are frequently tested for membership (to detect conflicting updates).

5.1.3 Pseudo State

Handling of the *update sets* is crucial for the performance of the compiled code. CASM uses hash-maps to implement *update sets*. The underlying assumption justifying this design decision is that *update sets* are small and the global state is large. A concept called *pseudo state* is used to realize temporary states needed to implement *sequential* execution semantics (see section 3.4). When reading a state *function* from within a *seqblock* the *update set* is queried first. If there has been an update to that *function* (and *location*) by a preceding rule, the value from the *update set* is returned. Otherwise the value is read from the global state. All rules return the update set produced according to their semantics (described in section 3). Using this mechanism the update sets only needs to be applied to the state when the top-level rule *concludes*. Merging and querying hash-maps is efficient as long as they are reasonably small.

6 Evaluation of the CASM language

6.1 Hardware modeling

We successfully used CASM in two projects to model CPU architectures and will briefly report the results here. Details can be found in the cited papers.

One project focused on fast design space exploration synthesizing cycle-accurate simulators from a CASM model of a microprocessor. The microprocessor model is proven to be coherent to the CASM specification of its instruction set architecture. We were able to model the instruction set and two variants of pipelined MIPS processors in just a few hundred lines of code. This is in size similar to a MIPS model formulated in a specialized hardware description language and demonstrates the expressiveness of the CASM language. The synthesized simulator is capable of executing benchmark programs of the SPECInt suite with a very satisfying peak performance of 1 MHz. This roughly translates into 15 million (basic) CASM rules to be executed per second. More details are reported in [LK13].

In a compiler verification project CASM is used to model a complex (non-interlocking) DSP processor. Combining parallel and sequential execution modes emerged as the crucial feature to describe cycled circuits. All hardware blocks are described in a parallel execution block, whereas their internal operations are described by a sequential execution block. We primarily use symbolic execution to perform simulation proofs in a translation validation approach. Some details are reported in [LK12].

6.2 Functional and Imperative programming style

In the introduction it was claimed that ASM based languages in a way combine imperative and functional programming styles. This property of the CASM language proved to be very useful in our experience. In this section we want to point out this property giving an illustrative example.

Börger and Bolognesi give a recursive ASM version of quicksort in [BB03]. Their implementation is very short and concise, like it is in most functional languages, but is not in-place as well. Imperative implementations swap elements directly (in-place) in the array avoiding copies of the whole input data. Functional languages need to construct a new list containing the resulting array (they can not destroy the input data) which induces $\mathcal{O}(n)$ additional space requirements. ASM based languages calculating quicksort in one computation step return an update set describing the new state of the array. This update set also uses $\mathcal{O}(n)$ additional space. We present an in-place, non-recursive version of quicksort.

```
function stack : -> List(Tuple(Int, Int)) initially { [] }
function array : Int -> Int
function p : -> Int initially { undef }
function l, r, ll, rr, pivot : -> Int
function need_pop, need_partition : -> Boolean
```

Listing 14: Quicksort (state)

```
 1  rule partition_one_step =
 2  seqblock
 3    iterate
 4      if array(ll) < array(pivot) then
 5        ll := ll + 1
 6    iterate
 7      if array(rr) >= array(pivot) and
 8         ll < rr then
 9           rr := rr - 1
10    if ll < rr then {
11      array(ll) := array(rr)
12      array(rr) := array(ll)
13    }
14    else
15      need_partition := false
16  endseqblock
17
18  rule partition =
19  if pivot = undef then {
20    pivot := r
21    rr := r - 1
22    ll := l
23    need_partition := true
24  }
25  else if need_partition then
26    call partition_one_step
27  else {
28    p := ll
29    if pivot != ll then {
30      array(pivot) := array(ll)
31      array(ll) := array(pivot)
32    }
33  }
```

Listing 15: Partition

```
34  rule quicksort_one_step =
35    if p = undef then
36      call partition
37    else seqblock
38      if l < p-1 then
39        push [l,p-1] into stack
40      if p+1 < r then
41        push [p+1,r] into stack
42      need_pop := true
43    endseqblock
44
45  rule quicksort =
46    if need_pop then {
47      let top = nth(stack, 1) in
48        if top != undef then {
49          stack := tail(stack)
50          l := nth(top, 1)
51          r := nth(top, 2)
52          pivot := undef
53          p := undef
54          need_pop := false
55        }
56        else
57          program(self) := undef
58    }
59    else
60      call quicksort_one_step
61
62  rule init = {
63    push [0,SIZE] into stack
64    need_pop := true
65    program(self) := @quicksort
66  }
```

Listing 16: in-place, non-recursive Quicksort

Listing 14 shows the state needed to perform the algorithm. It uses a *stack* to keep track of the parts of the *array* still to be sorted. The *quicksort* rule pops a new part to be sorted of the *stack* and stores the left and right indices into *l* and *r*. Line 48 tests if there is still further work to do and if so lines 49-45 contain the necessary initializations. The CASM idiom in line 57 terminates the whole computation. As long as a part of the array still needs to be sorted the rule *quickstep_one_step* will be executed. The rule *partition* will be called until a partition has been found. If the remaining parts are not trivial small they are pushed onto the *stack* and a new part needs to be popped from the stack *need_pop*.

The *partition* rule initially determines a pivot element (the last element of the part to be sorted here) and either swaps two elements of the array (rule *partition_one_step*) or swaps the pivot element to its final position *p*. Because the *partition_one_step* rule concludes after swapping two elements of the array the resulting update set is size bound and not dependent on the input data. Otherwise the update sets would need up to $\mathcal{O}(n)$ memory (i.e. on input $y_0, y_1, \dots y_n, x_0, x_1, \dots x_m, p \ : y_i < p \land x_j > p \ : 0 \leq i \leq n, 0 \leq j \leq m$).

To keep the observable computations small and the program simpler *partition_one_step* utilizes the *iterate* rule in lines 4 and 7 when searching for elements to be swapped. Otherwise it would need to keep track of the search similar to *need_partition*.

An ASM based language shares many features of a functional programming language. Large parts of the program are executed side-effect free, and while executing them the global state is read-only. One could think of the state as an explicit argument to each rule, the returned *update set* would then be the result of a functional application. The state only changes between invocations of the top-level rule, so the assumed implicit state argument changes for the next invocation. In that sense each single application of an ASM top-level rule is functional.

By combining parallel and sequential execution modes a programmer can chose the granularity of the computations steps seen by an observer of the program. This can, as demonstrated, be used to implement an in-place version of quicksort while basically programming a functional style.

The beauty of ASM based languages is the clear separation of invoking rules and applying changes to the state. They closely resemble what Backus called an applicative state transition system (AST system), on a much smaller scale though. What he calls a *formal system for functional programming* (FFP system) correspond to a rule and his *SYSTEM* state is just the state of the program.

7 Conclusion and Further Work

In this paper we presented CASM, a general purpose programming language based on the abstract state machine (ASM) method. We presented core features of the language and described how we are able to efficiently compile CASM to C++. The CASM compiler was successfully used to generate an instruction set simulator for the MIPS architecture capable of executing SPECInt benchmark with a peak performance of 1 MHz. We also developed an interpreter capable of symbolic execution which is successfully used in an ongoing compiler verification project.

The CASM language in a way combines functional and imperative programming aspects. We found this feature very useful and convenient and tried to showcase it using a small example.

While using the language common programming pattern arise which lead to new *rules* being implemented and new built-in functions being added. We are also working on a faster compilation and runtime implementation further increasing the performance of generated simulators.

References

[Anl00] Matthias Anlauff. XASM- An Extensible, Component-Based Abstract State Machines Language. In Yuri Gurevich, PhilippW. Kutter, Martin Odersky, and Lothar Thiele, editors, *Abstract State Machines - Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 69–90. Springer Berlin Heidelberg, 2000.

[Bö03] Egon Börger. Abstract State Machines: A Method for High-Level System Design and Analysis, 2003.

[Bac78] John Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[BB03] Egon Börger and Tommaso Bolognesi. Remarks on Turbo ASMs for Functional Equations and Recursion Schemes. In Egon Börger, Angelo Gargantini, and Elvinia Riccobene, editors, *Abstract State Machines 2003*, volume 2589 of *Lecture Notes in Computer Science*, pages 218–228. Springer Berlin Heidelberg, 2003.

[BG93] John Bergin and Stuart Greenfield. Teaching parameter passing by example using thunks in C and C++. *SIGCSE Bull.*, 25(1):10–14, March 1993.

[BHK10] Florian Brandner, Nigel Horspool, and Andreas Krall. DSP instruction set simulation. In Shuvra S. Bhattacharyya, Ed Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 679–705. Springer, August 2010.

[Cas01] Giuseppe Del Castillo. The ASM Workbench - A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models Tool Demonstration. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 578–581, London, UK, UK, 2001. Springer-Verlag.

[Far] Roozbeh Farahbod. CoreASM Language User Manual.

[FGG05] Roozbeh Farahbod, Vincenzo Gervasi, and Uwe Glässer. CoreASM: An extensible ASM execution engine. In *Proc. of the 12th International Workshop on Abstract State Machines*, pages 153–165, 2005.

[Gau95] Thilo S. Gaul. An Abstract State Machine specification of the DEC-Alpha Processor Family, 1995.

[GH93] Yuri Gurevich and James K. Huggins. The Semantics of the C Programming Language. In Egon Börger, Gerhard Jäger, Hans Kleine Büning, Simone Martini, and Michael M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–308. Springer, 1993.

[Gle04] Sabine Glesner. An ASM Semantics for SSA Intermediate Representations. In Wolf Zimmermann and Bernhard Thalheim, editors, *Abstract State Machines 2004. Advances in Theory and Practice*, volume 3052 of *Lecture Notes in Computer Science*, pages 144–160. Springer Berlin / Heidelberg, 2004.

[Gur95] Yuri Gurevich. *Evolving algebras 1993: Lipari guide*, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995.

[HHPJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[HS00] James K. Huggins and Wuwei Shen. The Static and Dynamic Semantics of C, 2000.

[Hug89] John Hughes. Why functional programming matters. *The Computer Journal*, 32:98–107, 1989.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[KP97] Philipp W. Kutter and Alfonso Pierantonio. The Formal Specification of Oberon. *Springer Journal of Universal Computer Science*, 3(5):443–503, 1997.

[LK12] Roland Lezuo and Andreas Krall. A Unified Processor Model for Compiler Verification and Simulation Using ASM. In *ABZ*, pages 327–330, 2012.

[LK13] 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 ?–? ACM, 2013.

[Sch01a] J. Schmid. Compiling Abstract State Machines to C++. *Journal of Universal Computer Science*, 7(11):1068–1087, 2001. `http://www.jucs.org/jucs_7_11/compiling_abstract_state_machine`.

[Sch01b] Joachim Schmid. Introduction to AsmGofeer, 2001.

[SSB01] Robert Stärk, Joachim Schmid, and Egon Börger. Java and the Java Virtual Machine - Definition, Verification, Validation, 2001.

[VPK04] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, July 2004.

[ZG97] Wolf Zimmermann and Thilo Gaul. On the Construction of Correct Compiler Back-Ends: An ASM Approach. *Journal of Universal Computer Science*, 3:504–567, 1997.