FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

AST interpreter for CASM

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Dominik Inführ
Matrikelnummer 0925697

an der
Fakultät für Informatik der Technischen Universität Wien

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

Wien, 03.05.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

AST interpreter for CASM

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Dominik Inführ
Registration Number 0925697

to the Faculty of Informatics
at the Vienna University of Technology

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

Vienna, 03.05.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Dominik Inführ
Schubertstraße 10, 3442 Langenrohr

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

(Ort, Datum) (Unterschrift Verfasser)

Abstract

An Abstract State Machine (ASM) is a method to formalize languages, which can be used for compiler backend verification to verify translations. CASM is a statically strong typed implementation of an ASM, targeted for this application. In this bachelor thesis an efficient Abstract Syntax Tree (AST) interpreter is written for CASM in C++, since for translation validation, many small files need to be executed. CASM supports symbolic execution with the output of symbolic traces in TPTP format.

Kurzfassung

Eine Abstract State Machine (ASM) kann genutzt werden, um Programmiersprachen zu formalisieren. Dies wird bei der Verifizierung von Compiler-Backends genutzt. CASM ist eine statisch getypte Implementierung einer ASM für dieses Anwendungsgebiet. In dieser Bachelorarbeit wird ein effizienter Abstract Syntax Tree (AST)-Interpreter für CASM in C++ geschrieben. Ein Interpreter ist erforderlich, da für die Verifizierung von Übersetzern viele kleine Dateien ausgeführt werden müssen. CASM unterstützt die symbolische Ausführung mit der Ausgabe der Traces im TPTP-Format.

Contents

CHAPTER 1

Introduction

1.1 Compiler Verification

The motivation of this thesis is compiler backend verification [6]. Nowadays, safety-critical applications run verified source code on verified hardware, but bugs in the compiler can lead to the incorrect translation from source code to machine code. Critical systems are only able to detect bugs introduced by the compiler by testing. A solution to this problem is compiler verification with translation validation. For this purpose the generated machine code is verified, not the whole compiler itself.

For compiler verification the formal semantics of the source code is proven to be equal to the semantics of the generated machine code with a first-order theorem prover.

1.2 Abstract State Machine

An Abstract State Machine (ASM) can be used to formalize a programming language or an instruction set of a microprocessor. CASM is a statically strong typed implementation of an ASM and was designed to be used in translation validation.

Since for translation validation a large number of small and short-running files needs to be executed, an interpreter is needed for CASM. A compiler would spend most of the time compiling rather than executing, but it has advantages for long-running files. CASM is also used to verify an instruction set simulation against its specification, where longer running files are more common, so CASM has the need for both interpreter and compiler.

CoreASM

At the beginning, CoreASM [3, 4] was used instead of CASM, but performance was not good enough with the dynamically typed CoreASM interpreter. Since efficient compilation is difficult for a dynamically typed language like CoreASM, the statically typed CASM was created. CASM is a subset of CoreASM with some additions and changes. An important difference to

CoreASM is that arguments are passed by value in CASM, while in CoreASM arguments are passed by name.

1.3 Mission Description

casmintr and *casm2cpp* were the first prototypes for the interpreter and compiler of CASM. Both programs were written in Python and used the same Abstract Syntax Tree (AST), one for executing and one for generating C++ code. The new interpreter should be 10 to 100 times faster than *casmintr* and is called *casmi*. The compiler is developed in a separate project but should use the AST of the interpreter. Test coverage should also be improved for the new interpreter.

CHAPTER 2

CASM

CASM is an implementation of an Abstract State Machine (ASM) as defined by Gurevich in [5], the definitions and equations in this chapter are from [2]. The main concept of an ASM is the state, which contains the current environment of the program. An ASM consists of a set of transition rules, where every rule describes how the current state is changed.

In this chapter $\mathfrak{A}$ is used for a state, while R and S are used for rules. An ASM allows to bind variable names to values in the variable interpretation ζ. It is important to point out that in an ASM the variable bindings are not part of the state $\mathfrak{A}$. The state is only composed of functions, where each function maps its parameters to a value.

Therefore the state is a set of $(location, value)$ pairs in CASM, where $location$ is a function name with a sequence of n-elements (the arguments for the function) as in $f\langle a_1, ..., a_n\rangle$ and *n* is the arity of the function. The given function name with its parameters is mapped to the element $value$. For example a state $\{(f\langle 1\rangle, 3), (f\langle 5\rangle, 10), (g, 2)\}$, where f and g are defined functions, maps $f(1)$ to 3, $f(5)$ to 10 and g to 2. But which value is returned by $f(2)$, where $f\langle 2\rangle$ is not a location in the state? In mathematics, a functions has to map all elements of the domain to an element of the codomain. An ASM returns $undef$ in this situation to fulfill this requirement. It is neither allowed to access undefined functions nor invoke a function with the wrong arity as in $h(1)$ or $f(1, 2)$. An ASM function is similar to a global variable in imperative languages.

As mentioned above, executing a transition rule changes the current state of the program in an ASM. The current state $\mathfrak{A}$ and the variable interpretation ζ are needed to execute a rule R, which is described by $next_R(\mathfrak{A}, \zeta)$. This function returns the new or the next state $\mathfrak{A}$‘ of the program, after executing the rule R in the state $\mathfrak{A}$ with the variable interpretation ζ. $\mathfrak{A}$‘ is used as the current state for the next rule S, which is repeatedly done until there are no more rules to execute.

In an ASM rules do not directly return the next state, but an update set u which contains all updates to form the next state $\mathfrak{A}$‘ starting from the current state $\mathfrak{A}$. The update set is also a set of $(location, value)$ pairs, but usually much smaller than the set of the state since it only contains updated function locations. $[\![R]\!]^{\mathfrak{A}}_{\zeta}$ returns the produced update set after executing the rule R. For an assignment rule $f(2) := 3$ in the example given above, it returns an update set

with a single update: $\{(f\langle 2\rangle, 3)\}$. The **skip** keyword has no effect and returns an empty update set: $[\![skip]\!]^{\mathfrak{A}}_{\zeta} = \emptyset$.

If an update set contains more than one update for a single location, it is called inconsistent. An update set $\{(f\langle 2\rangle, 1), (f\langle 2\rangle, 2)\}$ is inconsistent, while $\{(f\langle 2\rangle, 1), (f\langle 3\rangle, 2)\}$ is consistent. The function $Locs$ returns all locations occurring in an update set u as shown in equation 2.1.

$$Locs(u) = \{loc | \exists val : (loc, val) \in u\} \tag{2.1}$$

The function $fire_{\mathfrak{A}}(u)$ is used to merge an update set u into the state $\mathfrak{A}$, where $fire$ is only defined for consistent update sets. $\mathfrak{A}$‘ is equal to $\mathfrak{A}$, except to the locations defined in u, where $f^{\mathfrak{A}'}(a) = val$ for each $(f\langle a\rangle, val) \in u$. That means, both states are equal except to those function locations which are updated in u, if u is empty and therefore contains no updates, both states are the same. $next_R(\mathfrak{A}, \zeta)$ can also be expressed with $fire$:

$$next_R(\mathfrak{A}, \zeta) = fire_{\mathfrak{A}}([\![R]\!]^{\mathfrak{A}}_{\zeta}) \tag{2.2}$$

2.1 Hello World

A simple Hello World-program is shown in listing 2.1 to get a first impression of CASM:

```
1  CASM SpecificationName
2
3  // header block
4
5  init initRule
6
7  rule initRule = {
8    print "Hello World!"
9    program( self ) := undef
10 }
```

Listing 2.1: Hello World in CASM

The first line states the name of the CASM specification. Then the header block follows in which *function*s, *derived*s and *enum*s are declared globally, but these declarations are explained later in this chapter. The first rule to be executed is *initRule*, which is defined by the **init** keyword and the name of the rule on line 5. The rule *initRule* is defined on line 7, which prints to *stdout* and terminates the program. In CASM rules can be compared to procedures in imperative programming languages, but all of these statements are explained later in more detail.

2.2 Parallel and Sequential Composition

In a block of CASM rules, all rules are invoked independently of each other, which is a big difference to imperative programming languages. In CASM, a parallel block is encapsulated in curly brackets, e.g. `{ R S }` or semantically equivalent **par** `R S` **endpar**. Invoking independently means that each rule in the block is evaluated with the same state as shown in equation 2.3. The update sets of all rules in the block are merged, a resulting inconsistent update set leads

to a runtime error in *casmi*. Although it is called parallel block, the rules are not really executed in parallel (although this would be possible) in CASM. Parallel means that all rules are called for the same state in this context.

$$[\![R \textbf{ par } S]\!]^{\mathfrak{A}}_{\zeta} = [\![R]\!]^{\mathfrak{A}}_{\zeta} \dot{\cup} [\![S]\!]^{\mathfrak{A}}_{\zeta} \tag{2.3}$$

There are sometimes situations where it is very useful for the developer to invoke rules in a more imperative way. Therefore CASM also supports a sequential block with `seqblock R S endseqblock`. In this example *R* would be invoked with state $\mathfrak{A}$, but *S* in contrast to the parallel block with state $\mathfrak{A}$‘ (equation 2.4). $\mathfrak{A}$‘ is the state after invoking rule *R* in state $\mathfrak{A}$, expressed by by $next_R(\mathfrak{A}, \zeta)$. This means each rule is executed with the resulting state of the previous rule, so the rules are not independent in a sequential block. The update sets are merged with $u \oplus v$ as shown in equation 2.5.

u and v cannot be inconsistent in the first place, because otherwise the execution would have already failed. The resulting update set cannot be inconsistent, since updates for locations in u are replaced by their newer entries in v.

$$[\![R \textbf{ seq } S]\!]^{\mathfrak{A}}_{\zeta} = [\![R]\!]^{\mathfrak{A}}_{\zeta} \oplus [\![S]\!]^{\mathfrak{A}‘}_{\zeta} \quad \text{where } \mathfrak{A}‘ = next_R(\mathfrak{A}, \zeta) \tag{2.4}$$

$$u \oplus v = \{(loc, val) | (loc, val) \in u \wedge loc \notin Locs(v)\} \cup v \tag{2.5}$$

CASM needs to clone the current state into a temporary state for a sequential block. All rules within a sequential block are invoked with this state. After executing a rule in the sequential block the resulting update set is merged back to the temporary state, with the result that the next rule can be invoked with it. All update sets are also merged into a single update set, which is returned at the end of the sequential block and the temporary state can be safely deleted at the end of the block. It is not feasible to change the current state directly instead of the temporary state, because a sequential block can also be used inside a parallel block.

Both types of blocks can easily contain more than two rules:

$$[\![\{R\ S\ T\}]\!]^{\mathfrak{A}}_{\zeta} = [\![(R \textbf{ par } S) \textbf{ par } T]\!]^{\mathfrak{A}}_{\zeta} \tag{2.6}$$

$$[\![\textbf{seqblock } R\ S\ T \textbf{ endseqblock}]\!]^{\mathfrak{A}}_{\zeta} = [\![(R \textbf{ seq } S) \textbf{ seq } T]\!]^{\mathfrak{A}}_{\zeta} \tag{2.7}$$

2.3 Types

CASM supports the types *List*, *Tuple*, *Int*, *Boolean*, *String* and *RuleRef*. *Int* is a 64-bit signed integer, floating point values are not supported. *RuleRef* is explained in section 2.4. In CASM, types are used to declare functions and parameters of rules and deriveds, but this section should only give an overview of the existing types.

A list can save an arbitrary count of elements of the same type and is defined with e.g. `List (Int)`. In this example the list can only contain integer values, `Int` is considered as subtype of

the list. In the contrast to a list, a tuple needs at least two subtypes as in `Tuple(Int,String, Boolean)`. In many programming languages, the syntax is different for lists and tuples, however both types are defined within square brackets in CASM. While it is easy to recognize the type of `[1,"a",true]`, it is more difficult for `[1,2,3]` since it could either be a list or a tuple and so the type has to be deduced through the usage of the constant by the type inference unit.

2.4 Statements

function declaration

All n-ary functions need to be globally defined with the `function` keyword. In CASM all parameters and return values need to be typed as shown in listing 2.2, so the arity of each function is fixed. CASM differentiates between *controlled* and *static* functions, *static* functions are read-only and cannot be updated after their definition. In this example x and y are *controlled* functions in contrast to z. If neither *controlled* nor *static* is given, the function is assumed to be *controlled*. The `*` operator is used to separate parameter types, the return type follows after the `->` operator. `initially` can optionally be used to initialize a function with the given values. When a function has more than one parameter all arguments need to be encapsulated in square brackets as in `[1,2]->"a"`, where 1 and 2 are used as parameters.

```
1 function x: -> Int initially { 1 }
2 function controlled y: Int -> String initially { 1 -> "a", 2 -> "b" }
3 function static z: Int * Int -> String initially { [ 1, 2 ] -> "a" }
```

Listing 2.2: Defining functions

For the declarations in listing 2.2, `x` would return 1 and `y(1)` returns `"a"`. If a function location is accessed and no value is stored for it, *casmi* returns *undef* like for `y(5)` or `z(1,3)`.

A controlled function can be updated in the program with the `:=` operator. The update set of the statement is a single update with the evaluated expressions. Note that it is only able to update functions, but not variables. Listing 2.3 shows updating functions:

```
x := 3
y( 5 ) := "e"
z( 1, 5 ) := "b" // error, z is static
```

Listing 2.3: Updating functions

The first line updates x to 3, the second updates $y(5)$ to 10. Line 3 would fail, because updating static functions is not allowed. Since *undef* is compatible with every type, this value can also be assigned to the function x, with `x := undef`. When x is now read, it returns *undef* and there is no way to determine whether x was uninitialized or not. If this differentiation is needed, either do not use *undef* for assignments or save this information in another function. Since *undef* is compatible with every type it can also be used as parameter as in `y(1,undef) := undef` or `y(1,undef) := 3`.

The update set for an assignment is a single update with the given function location and value as shown in 2.8:

$$[\![f(t_1, ..., t_n) := t]\!]^{\mathfrak{A}}_{\zeta} = \{(f\langle [\![t_1]\!]^{\mathfrak{A}}_{\zeta}, ..., [\![t_n]\!]^{\mathfrak{A}}_{\zeta}\rangle, [\![t]\!]^{\mathfrak{A}}_{\zeta})\} \tag{2.8}$$

let statement

```
let x [: type] = t in R
```

The `let` statement binds the evaluated value of *t* to the variable *x*, where *x* can be used in the rule *R*. The type of *x* can be deduced from the expression *t* in most situations by type inference, but can also be explicitly declared by the developer when this is not possible with the : operator. Although *x* is called a variable in CASM, all variables are immutable and are semantic names for values. It is not allowed to update *x* with the := operator. *let* does not induce any updates, the update set returned by *let* is the update set of *R*, where all occurrences of *x* are substituted with the value of *t* as shown in equation 2.9.

$$[\![\textbf{let } x = t \textbf{ in } R]\!]^{\mathfrak{A}}_{\zeta} = [\![R]\!]^{\mathfrak{A}}_{\zeta \frac{x}{v}} \text{ where } v = [\![t]\!]^{\mathfrak{A}}_{\zeta} \tag{2.9}$$

let can only be used inside rules, it is not possible to declare global variables with it.

if statement

```
if t then R [else S]
```

If the condition *t* is *true*, the update set of *R* will be returned. Otherwise the update set of *S* is returned. If the condition is *undef*, the execution will fail.

$$[\![\textbf{if } t \textbf{ then } R \textbf{ else } S]\!]^{\mathfrak{A}}_{\zeta} = \begin{cases} [\![R]\!]^{\mathfrak{A}}_{\zeta}, & \text{if } [\![t]\!]^{\mathfrak{A}}_{\zeta} = true \\ [\![S]\!]^{\mathfrak{A}}_{\zeta}, & \textbf{otherwise} \end{cases} \tag{2.10}$$

The `case` statement is very similar to *if*, but can have more than two possible branches as shown in listing 2.4. *case* evaluates the expression *t* and executes the rule from the branch with the matching value. The *default* label is optional and is executed, if no label matches the value of *t*. In CASM, there is no automatic fall-through for *case* as in C-like languages. The update set of *case* is the update set of the executed rule or empty, if no matching or *default* branch was found.

```
1 case t of
2   v1: R1
3   v2: R2
4   default: S
5 endcase
```

Listing 2.4: The case statement

forall statement

```
forall x in t do R
```

The `forall` statement is a parallel composition, the rule R is invoked for the same state however with different bindings for the variable x as show in equation 2.11. t needs to be a list and x is bound to each of its list elements. The update set of a *forall* statement is the parallel composition of all the passes.

$$[\![\textbf{forall } x \textbf{ in } t \textbf{ do } R]\!]^{\mathfrak{A}}_{\zeta} = \bigcup_{v \in V} [\![R]\!]^{\mathfrak{A}}_{\zeta \frac{x}{v}} \text{ where } V = [\![t]\!]^{\mathfrak{A}}_{\zeta} \tag{2.11}$$

When a name of an enumeration is given for t, a list with all possible values of the enumeration is used for t. When *t* is an integer value, that is the same as looping over `[1..t]` or `[-1..t]` if $t < 0$. If $t = 0$ or an empty list is given for *t*, the rule *R* is not executed at all.

rule declaration

CASM allows to define named rules with parameters:

```
rule <name>( <param1> : <type1>, ..., <paramN> : <typeN> ) = R
```

Every parameter of the named rule needs to be typed, if no type is given for a parameter, `Int` is assumed and a warning message is printed to *stdout* for the missing type of the parameter. A named rule can be called via the `call` statement.

```
call (<ruleExpr>)( <param1>, ..., <paramN> )
```

Listing 2.5: Calling a rule

As already mentioned, a CASM program needs to have at least one named rule to be valid, since the execution is always started for a named rule. The start rule is not allowed to have parameters and can be specified with `init <ruleName>`. A CASM rule can be compared to functions in imperative programming languages, but rules do not return values in CASM.

There is an important difference in the handling of rule parameters to CoreASM. CASM uses call-by-value for parameters, while CoreASM uses call-by-name. This means parameters are evaluated before they are passed to the rule in CASM, the usages of parameters are substituted with its expressions in CoreASM. So the parameter is evaluated each time it is used. CASM uses call-by-value instead of call-by-name, since of performance reasons for the compiler. One can create *let* statements for each parameter right before the rule is called to achieve a call-by-value behaviour in CoreASM. So the semantics of the rule call is defined as 2.13 in CASM as opposed to 2.12 in CoreASM. $\mathfrak{A}$ is a state, *R* a named rule defined by $\textbf{rule } R(a, ..., a_n)$.

$$[\![\textbf{call } R(t_1, ..., t_n)]\!]^{\mathfrak{A}}_{\zeta} = [\![R[t_1/a_1, ..., t_n/a_n]]\!]^{\mathfrak{A}}_{\zeta} \tag{2.12}$$

$$[\![\textbf{call } R(t_1, ..., t_n)]\!]^{\mathfrak{A}}_{\zeta} = [\![R]\!]^{\mathfrak{A}}_{\zeta_\emptyset \frac{a_0}{v_0} ... \frac{a_n}{v_n}} \text{ where } v_i = [\![t_i]\!]^{\mathfrak{A}}_{\zeta} \tag{2.13}$$

The type *RuleRef* can be used as a reference or pointer to a specific rule in CASM. A given function `function r: -> RuleRef` can be initialized with `r := @rule_name`. *call* can be used to dynamically invoke rules with `call (r)`.

push and *pop* statement

CASM also supports pushing into and popping from 0-ary functions defined as `List`, e.g. `function f: -> List(Int)`.

```
push <expr> into <func>
pop <var> from <func>
```

func needs to be the name of a controlled function, since static functions cannot be changed. The variable for the *pop* statement is created with the subtype of the list as type. Both statements will raise an error if the given function is *undef*. Moreover *pop* does not allow an empty list. *push* puts the given value to the beginning of the list, while *pop* returns and removes the first value of the list. These statements produce a single update to the function with the updated value.

Listing 2.6 demonstrates how to push into and pop from a list:

```
1  function list: -> List(Int) initially { [] }
2
3  rule main = seqblock
4    push 1 into list // [] -> [1]
5    push 2 into list // [1] -> [2,1]
6    pop x from list  // [2,1] -> [1], x = 2
7    pop y from list  // [1] -> [], y = 1
8
9    program( self ) := undef // termination
10 endseqblock
```

Listing 2.6: push/pop example

derived declaration

*derived*s are semantic shorthands of expressions with the possibility of parameters as shown in listing 2.7 and can be used anywhere in the program in an expression. A *derived* without arguments is just a name for a constant. It is only necessary to declare the return type of the *derived*, if the type inference of *casmi* cannot infer it. If no parameter type is given, it will be assumed to be *Int*. This means the definitions of *add1* and *add2* are semantically equivalent. Note that a warning message is printed, if a parameter is not typed in a *derived* statement.

```
1  derived add1( a, b ) = a + b
2  derived add2( a : Int, b : Int ) : Int = a + b
3  derived one = 1
```

Listing 2.7: Declaring deriveds

enum declaration

In CASM it is also possible to declare enumerations:

```
enum states { ready, running, blocked }
```

Listing 2.8: Declaring an enumeration

All enumerations are also types which can be used in the declaration of functions or parameters. Listing 2.8 defines the additional type `states` with the constant values `ready`, `running` and `blocked`. The values in the enumeration are integer constants starting with 0. An enumeration value can be automatically converted to an integer and vice versa.

2.5 Operators

All operators have their common semantic meaning in CASM, but there are some limitations compared to other programming languages.

+, -, *, / and %
: These operators only work for `Int` operands and always return `Int`. *undef* is returned as long as at least one operand has the value *undef*.

<, <=, > and >=
: These comparison operators only compare integer values and return either `true` or `false`. The result of these operators is *undef* as far as at least one operand has the value *undef*.

= and *!=*
: These operators can compare values of the same type and return a `Boolean` as well. So it is also possible to compare strings, lists or tuples by value but comparing different types would lead to an error. The return value is never *undef*, since $undef = undef$ is *true*, while $undef = 2$ is *false*.

2.6 Built-ins

CASM supports `print` to write to *stdout*. `+` can be used to concatenate the output of expressions as in `print "a = "+a`, `+` is here only used as separator, not as the `Int` operator and is also available for *diedie* and *debuginfo*. In expressions it is not possible to concatenate strings. `diedie` stops the execution and prints the given parameter as its error message.

`debuginfo` is a more mature way for printing debug messages than *print*. This statement expects an identifier as first parameter as in `debuginfo x1 "message"`, which can be used to filter output. By default all messages are filtered out, but can be activated with the `-d` parameter for *casmi*, `-dx1` activates messages with identifier *x1*. Multiple identifiers are separated with commas (e.g. `-dx1,x2`), while `-dall` and `-d*` activate all debug messages.

`assert` raises an error if the given expression does not return *true* and is heavily used within the test suite of *casmi*.

casmi also has some built-in functions primarily used for list handling, which operate on values and do not induce any updates:

hex

This function converts a number to its hexadecimal representation, e.g. `hex(28)` returns `"1c"`. *hex* returns `"undef"` for **`undef`**. The parameter has to be *Int*, return type is *String*.

len

len returns the length of a list, e.g. 2 for `[1,2]`. It gets a list as only parameter, return type is *Int*.

nth

This function is the only one which accepts a tuple. It returns an element of the given list or tuple according to the also given index, e.g. `nth(2,[2,3])` returns `3` and `nth(1,["a",2,true])` returns `"a"`. As shown in these small examples, the index starts with 1 as in CoreASM and not 0 as used in many other languages. *nth* returns *undef* if the index is out of bounds.

The second parameter can either be list or tuple. If the method is invoked for a tuple, the first parameter has to be an integer constant. Otherwise it is not possible to determine the return type of the function for the type inference unit. For lists, this parameter can be any valid *Int* expression, since the return type is always the subtype of the list.

cons

cons adds an element to the beginning of a list, so `cons(1,[2,3])` returns `[1,2,3]`. The second parameter needs to be a list. The subtype of the list and the type of the first parameter must match. Return type is a list, identical to the type of the second parameter. *cons* returns *undef*, if the given list is *undef*.

peek

peek returns the first element of a list. Return type of the parameter is the subtype of the list. `peek([3,2])` returns `3`.

tail

tail returns all but the first element from a list. A list is expected as only parameter, the return type is the type of the list. `tail([4,7,1,1])` returns `[7,1,1]`.

External functions

casmi can also call functions from a shared library. This functionality is used to invoke functions of a bit vector library. It searches the path of its executable for an *asm_engine* directory. In this directory a header file *arithmetics.hpp* is parsed, for all defined functions and arity. These functions are limited to *Int* parameters and *Int* as return type, so no more information is needed. When the header file could not be found, *casmi* prints an error message, but it does not stop execution immediately.

When an external function is called the first time, *casmi* dynamically loads the file *asm_engine.so*. The interpreter also has to convert the parameters to the type `shared_int` and pass the converted values to the function. *casmi* cannot use a simple *int* data type, since *Int*s in *casmi* can also be **`undef`**.

```
class shared_int {
public:
  uint64_t value;
  bool undef;
  uint64_t sym_t;
};
```

The external function also needs to return a `shared_int` value, which is converted back to a value suitable for the interpreter. The property *undef* is **`true`** for **`undef`**, *sym_t* is explained later in chapter 3.

2.7 Program Execution

In CASM a simple program can be written with just one line of code: **`rule`** `main = {` **`print`** `" Hello World"}`. This program prints `"Hello World"` to *stdout* infinitely in CASM. By default, the initial rule is executed repeatedly, this can be controlled with the *program* function, defined by **`function`** `program: Agent ->` **`RuleRef`**. In CoreASM, this function is used to start multiple agents. In CASM, this is not possible so the only possible parameter for *program* is the current agent **`self`**. *Agent* is used as a type only internally for this function, so it is not possible to use it for functions defined by the developer.

The agent has to be set to *undef* with `program(` **`self`** `) :=` **`undef`** to end the execution of the program. *program* can also be used to replace the top level rule, as shown in listing 2.9. *program* is used to call the rule `hello_world`, which prints to *stdout* and ends the execution of the program by setting the current agent to *undef*. The top level rule is not allowed to have parameters.

```
rule main = {
  program( self ) := @hello_world
}

rule hello_world = {
  print "Hello World"
  program( self ) := undef
}
```

Listing 2.9: Prints 'Hello World' once

CHAPTER 3

Symbolic execution

casmi also supports a symbolic mode or symbolic execution. Any valid CASM file can be executed in this mode. The syntax remains the same, while the semantic slightly changes. *casmi* has to be started with the parameter `--symbolic` or `-s` to run a file in symbolic mode. Otherwise *casmi* is in so-called concrete or normal mode. *casmi* can either be in one of these two modes, the mode is chosen on startup and cannot be changed at runtime.

In symbolic mode it is allowed that states or variables do not hold concrete values but symbolic values instead of them. Symbolic values cannot occur in concrete mode. Each symbolic value has an unique id, which is simply a counter started with 1. Creating a new symbolic value increases this counter, so symbolic values are equal if they have the same id. When a symbolic value is used in an operation, the return value is a new symbolic value.

Symbolic mode is used to verify an instruction set simulation against its specification [7], where the naive implementation of instructions is an easy task compared to writing pipelined models for the same instruction set. Pipelined models are proven to be equivalent to their naive counter part by theorem proving. Therefore running in symbolic mode also means that a trace file or output is written in TPTP format [9], which is suitable for many theorem provers. This output describes the transformations made to a state and connects the initial to the final states. The two generated trace files allow a theorem prover like *VAMPIRE* [8] to prove the correctness of the pipelined model.

It is enough to access an uninitialized function state to create a symbolic value, like in listing 3.1.

```
1 function a: -> Int
2
3 rule main = {
4   print "a = " + a
5   program( self ) := undef
6 }
```

Listing 3.1: Creating a symbolic value

The output shows `a = sym2`, this means that the value is a symbolic value with the id 2, however in concrete mode *casmi* prints `a = undef`.

3.1 Symbolic Output

TPTP output in *casmi* is printed to *stdout* by default, but with parameter `-x` or `--fileout` can be forced to be written into a *.trace* file. The statements **`straceon`** and **`straceoff`** can be used to activate and disable output for symbolic values for specific parts of the program in CASM, by default output is deactivated and has to be activated with **`straceon`**. In concrete mode, these commands have no effect. The parameter `--straceon` can be used for *casmi* to switch on the symbolic output at the beginning of the program. Listing 3.2 shows the output with parameter `--straceon` in symbolic mode. The first 3 lines would be missing, without this parameter.

```
1 tff(symbolNext, type, sym2: $int).
2 fof(id0,hypothesis,fa(0,sym2)). %CREATE: a
3 fof(id1,hypothesis,fprogram(1,0,undef)). %UPDATE: program self
4 fof(final0, hypothesis, fa(0,X) <=> fa(666,X) ).
5 fof(final1, hypothesis, fprogram(1,0,X) <=> fprogram(666,0,X) ).
```

Listing 3.2: Complete output of listing 3.1

Each update instruction is logged in the output. In symbolic mode, the first access to an uninitialized function does not return *undef* any more, but creates a symbolic value and initializes the function with it. Since *a* is uninitialized, a symbolic integer value *sym2* is created and assigned to *a* on line 1 and 2. The first line is needed to correctly type the problem for the theorem prover.

`program( self ):= undef` leads to the output on line 5, since `program( self )` was already initialized with `@main`, *casmi* prints `UPDATE` instead of `CREATE` for *a*. The first parameter in `fprogram(1, 0,undef)` is a generation counter, which is a logical timestamp pro location.

At the end, *casmi* dumps the final state. With *casmi* and the parameter `--prefix`, it is possible to prefix symbol and function names, so when executing listing 3.1 with `casmi --symbolic --straceon --prefix=PRE` symbol names are now prefixed. Use parameter `--symbolic_f` instead of `--symbolic` or `-s` to prefix function names, listing 3.3 shows the prefixed version of listing 3.2.

```
1 tff(symbolNext, type, symPRE2: $int).
2 fof(idPRE0,hypothesis,fPREa(0,symPRE2)). %CREATE: a
3 fof(idPRE1,hypothesis,fPREprogram(1,0,undef)). %UPDATE: program self
4 fof(final0, hypothesis, fPREa(0,X) <=> fPREa(666,X) ).
5 fof(final1, hypothesis, fPREprogram(1,0,X) <=> fPREprogram(666,0,X) ).
```

Listing 3.3: Prefixed output of listing 3.1

Assuming the declaration **`function`** `a: Int * Int -> Boolean` and its update with `a(1,2) := true`, leads to the output of `fof(id1,hypothesis,fa(1,1,2,1)). %UPDATE: a 1 2`. Before the first update of *a*, it is automtically initialized with a symbolic value, since *a* is uninitialized at this point. This means `fof(id0,hypothesis,fa(0,1,2,sym2)).%CREATE: a 1 2` is printed right before the update.

As already seen values are encoded a bit different for the theorem prover in *TPTP* output, `true` is printed as 1, `false` is 0. String constants and *RuleRef*s are escaped with `e` to e.g. `e"Hello World!"` or `emain`. As already mentioned above, `self` is printed as 0.

3.2 Evaluating Expressions with Symbolic Values

Since symbolic values can occur in expressions, the evaluation rules for expressions need to be adjusted for symbols. `+`, `-`, `*`, `/` and `%` still return *undef*, if at least one operand is *undef*. In symbolic mode these operators return a new symbol if at least one symbol is given as operand. Listing 3.4 shows the results of the possible combinations, *a* and *b* are two symbolic values. The order of the operands has no effect on the result, `a + undef` is the same as `undef + a`. Although this listing only shows the add operator, the results remain the same for the rest.

```
a // = sym2
b // = sym3

a + b     // = sym4
a + undef // = undef
a + 2     // = sym5
```

Listing 3.4: Evaluating expressions

The comparison operators `<`, `<=`, `=`, `!=`, `>` and `>=` work really similar with subtle differences. If two symbolic values with the same id are compared, these operators return `true` or `false` depending on the used operator. If the symbols have different ids, a new symbol will be created and returned. Comparing a symbol with *undef* returns `false` for the equality and `true` for the inequality operator, since a symbolic value and *undef* cannot be equal. The remaining comparison operators return *undef* in this situation.

```
a // = sym2
b // = sym2
c // = sym3

a = b     // = true
a = c     // = sym4
a = undef // = false
a < undef // = undef
a = 2     // = sym5
```

Listing 3.5: Comparing values

With the built-in function *symbolic*, an expression can be checked if it returns a concrete or symbolic value. This is used in listing 3.6 to assert a symbolic value for an uninitialized state. In *casmi* this function was useful for many tests in symbolic mode.

```
// symbolic

function a: -> Int

rule main = {
```

```
6     assert( symbolic( a ) )
7     program( self ) := undef
8 }
```

Listing 3.6: Checking for symbolic value

3.3 Symbolic Conditional Control Flow

External functions

As mentioned in section 2.6, *casmi* can use functions from a shared library. These functions can also be invoked with symbolic integer values, therefore the *sym_t* property in `shared_int` from listing 2.6 saves the id of the symbolic value and is greater than 0 for symbolic values.

In symbolic mode *casmi* does not load *asm_engine.so* but *asm_engine_symbolic.so*. Therefore these functions can behave differently depending on the current mode.

push and *pop* statement

push and **pop** also work for symbolic values. When a value (it does not matter if it is symbolic or not) is pushed into a state with a symbolic value, then a new symbol is created and saved in this state. *pop* works very similar and returns a new symbol and updates the state with a new symbolic value. Listing 3.7 shows a part of the output of two CASM files, which *push* into and *pop* from **function** `a: -> List(String)`. Since *a* is uninitialized, it is initialized with *sym2*. Pushing `"a"` changes *a* to *sym3*, while popping returns *sym3* and changes *a* to *sym4*.

```
1 % push "a" into a
2 fof(id0,hypothesis,fa(0,sym2)).%CREATE: a
3 fof(id1,hypothesis,fpush(sym2,e"a",sym3)).
4 fof(id2,hypothesis,fa(1,sym3)).%UPDATE: a
5
6 % pop x from a
7 fof(id0,hypothesis,fa(0,sym2)).%CREATE: a
8 fof(id1,hypothesis,fpop(sym2,sym3,sym4)).
9 fof(id2,hypothesis,fa(1,sym4)).%UPDATE: a
```

Listing 3.7: push and pop in symbolic output

if statement

If a symbolic value is given for an *if* statement, *casmi* does not know if the symbolic value is *true* or *false*. So *casmi* forks itself and assumes the value in one execution path as *true*, in the other as *false*. That means a symbolic program can have more than one symbolic output and execution path as in listing 3.9, which is the output for listing 3.8.

```
1 function a: -> Boolean
2 function b: -> Int initially { 0 }
3
4 rule main = {
```

```
5   if a then b := 1 else b := 2
6   program( self ) := undef
7 }
```

Listing 3.8: if with symbolic value

```
1 forklog: I
2 fof(id0,hypothesis,fa(0,sym2)).%CREATE: a
3 fof('idtests/sym_if.casm:5',hypothesis,sym2=1).
4 fof(id1,hypothesis,fb(1, 1)).%UPDATE: b
5
6 forklog: E
7 fof(id0,hypothesis,fa(0,sym2)).%CREATE: a
8 fof('idtests/sym_if.casm:5',hypothesis,sym2=0).
9 fof(id1,hypothesis,fb(1, 2)).%UPDATE: b
```

Listing 3.9: Symbolic output of if

Listing 3.9 shows the two execution paths. *I* is the output if the *if* branch is taken, otherwise *E* is the output. The important line in the TPTP output is:

```
fof('idtests/sym_if.casm:5',hypothesis,sym2=1).
```

This line assumes the symbolic value *sym2* in the *if* statement to be *true*, while for *else* it is assumed to be *false* with `sym2=0`.

If `-x` was used as additional parameter for executing this program and assuming the file name to be *if.casm*, *casmi* would create two trace files, *if_I.trace* and *if_E.trace* instead of writing to *stdout*.

CHAPTER 4

Type Inference

CASM is a statically typed language, that means all parameters, variables and return types need a static type. *casmi* can infer variable and return types since explicitly typing every element of the application is tedious. Type inference is a separate pass before the interpreter starts executing the program.

An Abstract Syntax Tree (AST) is generated for the program, when parsing and tokenizing the source code. The structure of a node in the tree is defined in listing 4.1.

```
1  typedef struct ast {
2    ast_type type;
3    struct ast *left, *right;
4
5    int line;
6    int col;
7
8    union {
9      int_type number;
10     const char *name;
11   };
12
13   itype *aprio, *inferred;
14   ientry *entry;
15   vector<itype*> *param_types;
16 } ast;
```

Listing 4.1: Node of the AST

After parsing, the whole source code of the input file can be referenced through one single pointer `ast *program`. In contrast to *casmintr* an AST node only has two children, the left and right successor, in *casmintr* each node could have a list of children. It was easier to generate the tree in this format with *Bison*, which was used for writing the parser and to generate the tree. Type inference is done on the AST of the CASM program.

aprio and *inferred* are the two main properties of a node for type inference. *aprio* is set for a node, if the node is or should be of this type. This property is calculated from the top to the

bottom. *inferred* is the calculated type of the node, which is computed from the bottom to the top of the tree. The properties *aprio* and *inferred* have to be compatible types, this is accomplished with the function `type_check` in listing 4.2, which is called for every node in an expression. If *aprio* is not set, the check cannot fail.

```
1  void iannotation::type_check( ast *node ) {
2    if( node->aprio == NULL ) {
3      return;
4    }
5
6    if( !itype::compatible( node->aprio, node->inferred ) ) {
7      ERR( "type " + node->aprio->inspect( ) +
8        " expected but got " + node->inferred->inspect( ), node );
9    }
10 }
```

Listing 4.2: Checking inferred and aprio properties of a node

Type inference consists of three different steps. At first the types of the initializers of *function*s are checked. After checking initializers all *derived*s are type annotated and their return type is computed. The last step is annotating all statements, that means the body or content of all defined *rules*. *derived*s need to be annotated before *rule*s, since the return type of a *derived* needs to be known when it is used inside a *rule*.

4.1 Annotating Declarations

Annotating declarations starts with annotating the initializers of all controlled and static *function*s, which is very easy since the interpreter knows all function signatures. Therefore the *aprio* property can be set for all values.

After checking initializers, all *derived*s are type annotated, so that the return type is computed and can later be used by rules when calling *derived*s. If a return type is given for the *derived*, the computed return type has to be compatible with it. *casmi* uses the types of the parameters, operators and built-in functions to infer a type. For example `+` and *len* return an *Int*, while *hex* returns a *String*.

It is not always possible to calculate the return type, e.g. for `undef` or `[]`. So in these rare cases, a return type has to be specified. In the contrast to local variables, *casmi* does not try to determine the return type of *derived*s through their usage. Of course *casmi* could try to do this, but it would add some complexity to the type inference unit for minor savings for the developer in very unusual cases. A naive implementation would have to annotate almost the whole source code again if a return type was detected to ensure that all usages are compatible with the return type. Moreover it is very easy to fix this error, through stating the return type of the *derived* explicitly.

Return types of *derived*s are calculated in order of their declaration in the source code. A current limitation of the type inference implementation does not allow a *derived* to use another *derived*, which return type is not already known. Listing 4.3 demonstrates this case, where *d1* calls *d2*, which leads to an error. This situation can be resolved by either specifying the return

type for *d2* with `derived d2( a : Int ): Int = a + 2` or placing the definition of *d2* before *d1*. *casmi* could make some efforts to detect return types in such situations, but this was not implemented for the sake of simplicity. The reasons were the same as mentioned above.

```
derived d1( a ) = d2( a ) // error
derived d2( a : Int ) = a + 2
```

Listing 4.3: Derived using another derived

4.2 Annotating Statements

After computing all return types of *derived*s, *casmi* starts checking the body of all named rules. This means all statements and expressions in rules in the source code are annotated and checked for type mismatches. The function `type_statement` is invoked for the body of the rule, which recursively calls the annotation function of every statement in the rule. Each instruction has its own function for annotation, since they all work in different ways. However, there are some statements such as `straceon` or `skip` which do not need to be annotated at all, so they are just skipped. Annotating rules starts by calling `type_statement` for each body of a *rule*. The parameter `table` references the symbol table, which holds all available parameters and local variables, but also all global elements, such as enumerations, *derived*s or *function*s. `table` is reinitialized for every *rule* with all global elements and all parameters of it.

```
void iannotation::type_statement( ast *node, isymtable &table ) {
  switch( node->type ) {
  case AST_SEQBLOCK:
  case AST_PARBLOCK:
    type_block( node, table );
    break;

  case AST_UPDATE: type_update( node, table ); break;
  case AST_LET: type_let( node, table ); break;
  case AST_FORALL: type_forall( node, table ); break;

  // ...

  default: ERROR( );
  }
}
```

Listing 4.4: Function type_statement

undef and the empty list

It is important to note that *undef* is a value and compatible with every type, but there is no own *undef* type to be used inside a program by the developer. *undef* always needs a specific type such as `Int` or `String` if bound to a variable. *undef* is only used as type internally to indicate that this part of the type is still unknown. `[]` is of type `List(undef)`, which means that the subtype of the list is unknown. If bound to a variable, this part of the type needs to be resolved.

The complete type is only needed for variables, since otherwise statements such as `len([])` or `len(undef)` would not work, where the unknown subtype of the list is no problem.

Although *undef* is compatible with all types, binding it to a variable does not mean that the variable can be used for all types, it still needs a static type. The following listing fails for type inference on line 7:

```
1 function a: -> Int
2 function b: -> String
3
4 // ...
5 let x = undef in {
6   a := x
7   b := x // error, x is Int
8 }
```

Annotating update instruction

One of the most important rules is the update instruction, since it often allows to determine the type of local variables. Due to the fact that it is only allowed to update *function*s and they are guaranteed to be explicitly typed, the types of all expressions in the statement are known. Therefore the update instruction on line 4 in listing 4.5 defines the type of x as `Boolean`, y as `String` and z as `Int`.The function `type_update` sets the *aprio* property for all parameter expressions and the value and calls `type_expression` for them. This function annotates an expression and returns the detected type of it. It also calls `type_check` (see listing 4.2), as a result *aprio* and *inferred* are checked for every single node in an expression.

```
1 function a: Boolean * String -> Int
2
3 // ...
4 a( x, y ) := z
```

Listing 4.5: Update function a

Assuming `type_expression` is called for a variable and the property *aprio* is already set (such as in listing 4.5 for x, y and z), the type of the variables can be determined. If the variable type is already known, its type will be assigned to *inferred* and therefore checked against *aprio* with `type_check`, so no explicit check is necessary for the return type. Listing 4.6 shows some code of `type_expression`. The function calls itself recursively for all operands.

```
itype *iannotation::type_expression( ast *node, isymtable &table ) {
  switch( node->type ) {
  case '+' || '-' || '*' || '/' || '%':
    // operands need to be Int
    node->left->aprio = node->right->aprio = itype::const_int;

    type_expression( node->left, table ); // call function for successors
    type_expression( node->right, table );

    node->inferred = itype::const_int; // return type is Int

  case AST_NUMBER: node->inferred = itype::const_int; break;
  case AST_STRING: node->inferred = itype::const_string; break;

  // ...

  default: ERROR( );
  }

  type_check( node );
  return node->inferred;
}
```

Listing 4.6: function type_expression

Annotating *let*

Since the `let` statement allows omitting the type of the variable, *casmi* has to be able to determine types of variables. The detection of the type in expressions such as `2`, `"Hello!"` or `false` is straightforward, while expressions as `undef` or `[]` are more trickier.Assigning `[]` to a variable means that it is some sort of `List`, but the subtype of the list remains unknown, while for `undef` the type of the variable remains completely unknown, since *undef* is compatible with every type.

If the type of a variable was not declared or could not be calculated from the binding, the type must be determined through the usage of the variable. In most cases, the type can be deduced from an update instruction, but there are also some other ways, e.g. through an equality operator. `push` and `pop` allow the type of the variable to be easily recognized, since *function*s are explicitly typed. The *let* statement adds a variable to `table`, annotates the scope where the variable is available and removes the variable again. After annotating the variable scope, the variable type should be known. Type inference fails and raises an error, if the type is still unknown.

If the type of the variable could be determined because of its usage, it is necessary to annotate the binding again with an updated *aprio* property. Otherwise some rare cases like in listing 4.7 do not work. Since x is *undef*, the type cannot be inferred from the binding alone and so the type of y is also unknown on line 2. z is explicitly declared as `Int`, so *aprio* can be set for the binding. Since y is used as `Int` on line 3, its type can be inferred at this point. This leads to the repetitive annotation of the binding on line 2 with *aprio* set to `Int` for x and therefore the type of x can now be inferred. If the binding would not be annotated again, the type of x could not be

computed and type inference would fail on line 1 for the variable *x*.

```
1 let x = undef in
2   let y = x in
3     let z : Int = y in skip
```

Listing 4.7: x, y and z as Int

Annotating the binding with an updated *aprio* property is also necessary, because of assignments like `z = cons( x, y )` in listing 4.8. When the binding is annotated again on line 6, both *x* and *y* get their types *Int* and *List(Int)* assigned.

```
1  function a: -> List(Int)
2
3  rule main = {
4    let x = undef in
5    let y = undef in
6    let z = cons( x, y ) in
7      a := z
8
9    program( self ) := undef
10 }
```

Listing 4.8: cons in assignment

Choosing Tuple or List for a Variable

As already mentioned, type inference has to differentiate between tuple and list in CASM. At first, list constants are treated as tuple unless the list constant has less than two elements or *aprio* is set to list. Note that tuple and list types are not compatible with each other in any case.

The list constant `[1,2,3]` is assumed to be of type `Tuple(Int,Int,Int)` at first, but could also be of type `List(Int)`, which is not possible for all list constants. A tuple is saved in the structure `itype_tuple` (shown in listing 4.9). The property `subtype` is set to *NULL* if the tuple cannot be used as a list. If the tuple can be used as a list, `subtype` references the subtype of the list. For the example above, `subtype` references `Int`.

Declaring a *function* or variable as a tuple, always sets `subtype` to *NULL*. Non-*NULL* values are only allowed for list constants.

```
1 class itype_tuple : public itype {
2 public:
3   vector <itype*> subtypes;
4   itype *subtype;
5 };
```

Listing 4.9: The structure of a tuple in *casmi*

It is not a must to treat list constants as a tupel first and later transform it to a list as required. The reverse would also be possible, it is only important that the type can change and the variable can either be used as tuple or list, but not both.

The built-in functions *len*, *peek*, *tail* and *cons* need some type of `List`, so using a variable in one of these functions also determines the type of it. Therefore the variable cannot be used as

`Tuple` again. *nth* is a bit special and more complex, since it is possible to call it for a `Tuple`, but only if the first parameter is a constant value. Otherwise the type inference could not determine the return type of *nth*, whereas for `List` it is always possible.

= and *!=*

These operators are by far the most complex part of type inference for CASM, since the two operands of the operators need to have the same type. Listing 4.10 shows a pseudo code for annotating these operators. This code serves as an overview for the explanations in this section.

```
1  void iannotation::type_equals( ast *node, isymtable &table ) {
2    ast *l = node->left;
3    ast *r = node->right;
4
5    bool re_r = false, re_l = false;
6
7    // first annotation of operands
8    l->aprio = r->aprio = NULL;
9    itype *tl = type_expression( l, table );
10   itype *tr = type_expression( r, table );
11
12   // check if tuple is compared to list and tuple can be converted
13   // to a list
14   if( is_list( tl ) && is_tuple_and_convertible_to_list( tr ) ) {
15     re_r = true; // annotate right operand again
16     tr = convert_to_list( tr );
17   } else if( is_tuple_and_convertible_to_list( tl ) && is_list( tr ) ) {
18     re_l = true; // annotate left operand again
19     tl = convert_to_list( tl );
20   }
21
22   // types must be compatible
23   if( !types_compatible( tl, tr ) )
24     ERROR;
25
26   itype *tm = merge( tl, tr );
27
28   // annotate operands again if their type changed
29   if( re_r || tm != tr ) {
30     r->aprio = tm;
31     type_expression( r, table );
32   }
33
34   if( re_l || tm != tl ) {
35     l->aprio = tm;
36     type_expression( l, table );
37   }
38
39   if( unknown_type( tm ) ) {
40     // save comparison for the current scope and rule
41
42     current_scope.add_comparison( node );
```

```
43      current_rule.add_comparison( node );
44    }
45
46    // operators always return bool
47    node->inferred = itype::const_bool;
48  }
```

Listing 4.10: Annotating = and !=

Because all types are allowed, the *aprio* property of the operands is set to *NULL*. After this, the method `type_expression` is invoked for both operands, to annotate these expressions. If both types are compatible, the type of the comparison is computed by merging the types of the operands. Merging types returns the common type of both given types. Note that $merge$ is only defined for compatible types. The following table demonstrates some results of the $merge$ function:

t_1	t_2	$merge(t_1, t_2)$
String	*String*	*String*
undef	*Int*	*Int*
List(undef)	*List(String)*	*List(String)*
Tuple(undef,String)	*Tuple(Int,undef)*	*Tuple(Int,String)*
List(undef)	*undef*	*List(undef)*
Int	*String*	merging fails

If the merged type is different than the type of an operand, `type_expression` will be invoked again for the operand with the *aprio* property set to the merged type. Listing 4.11 demonstrates this case, where **`undef`** and **`Int`** are merged to **`Int`** on line 2. The left operand of the = operator is annotated again, so that the type of *x* can be inferred. This must be done, otherwise it would not be possible to assign a type from one operand to the other.

```
1  let x = undef in {
2    if x = 4 then
3      print "x = 4"
4  }
```

Listing 4.11: x is used as Int

A special case occurs for the comparison when a tuple is compared to a list, since the tuple needs to be checked if it can be converted to a list. This case needs to be considered, since list constants are assumed to be tuples at first, but the type can change to a list as in listing 4.12. In this example on line 4, the types of the operands are **`Tuple(Int,undef,Int)`** and **`List(Int)`** first, type inference would fail for these types. Therefore the tuple needs to be converted to a list, in this example **`List(Int)`**. Now both operands are compatible. Since the merged type is **`List(Int)`**, the left operand is annotated again to assign the new type to *x*.

The same would happen for comparing function a with the list constant `[1,2]` as in `a = [1,2]`. `[1,2]` is assumed to be **`Tuple(Int,Int)`**, so a conversion is needed too.

```
1  function a: -> List(Int)
2
```

```
let x = [1,undef,3] in { // at first Tuple(Int,undef,Int)
  if x = a then
    print "x = a"
}
```

Listing 4.12: x is used as a list

As mentioned above, both operands need to have compatible types, but unfortunately the types of the operands are not always known at this time. So in *casmi*, each comparison with undefined merged type is saved and treated as a requirement, which has to be fulfilled. Later, these comparisons are annotated again, when the interpreter hopefully has more information to detect type errors.

The comparison on line 5 in listing 4.13 has `undef` as merged type, so it is saved in a list for the current scope. The type of x can then be inferred through the update for a on line 7. Before the scope is left on line 8, the comparison is annotated again, but at this point `Int` can be used as *aprio* type for the left operand and the type of y can now be inferred.

```
function a: -> Int

let x = undef in
let y = undef in {
  if x = y then print "x = y"

  a := x
}
```

Listing 4.13: Unknown type of comparison on line 5

It is also valid if the merged type of a comparison is still undefined at this point, since all comparisons are always added to the current rule too, where they are annotated again when leaving the rule. At this point, all variable types are known, so all type mismatches in comparisons are guaranteed to be detected.

In listing 4.14, the comparison on line 8 has an unknown type and when checking the comparisons for the scope of variable z, the type is still undefined. The type mismatch is detected, when the rule is left.

```
function a: -> List(String)
function b: -> List(Int)

// ...
let x = [ undef, undef ] in
let y = [ undef, undef ] in {
  let z = 10 in {
    if tail( x ) = tail( y ) then print "yeah" // error
    // ...
  }

  a := x
  b := y
}
```

Listing 4.14: Error in comparison could be undetected

To find all type mismatches, it is only necessary to check the comparisons at the end of the rule. It is important to point out that although all variable types are known, unknown compare types can still occur at the end of the rule and no error is raised for them. Otherwise comparisons such as `undef = []` would not be allowed.

Comparisons are verified again at the end of a scope, to detect variables types as in listing 4.13. If the check is not done, the type of variable x cannot be inferred and x would need to be explicitly typed.

Annotating *call*

Annotating is a single pass before the interpreting of the file starts. The single one exception is the *call* statement, since a dynamically called rule is only known at runtime. So the type annotation saves the types of all parameters in the node of the *call* instruction. Due to this, the node has a property `param_types`, which is a pointer to a type vector (`vector<itype*> *`). Then the interpreter only needs to compare the parameter types of the *call* instruction with the rule declaration.

For static rule calls, the parameters are checked right in the type inference unit. `param_types` is set to `NULL`, so that the interpreter does not check the parameter types again. It is faster to check the types in the type inference unit if this is possible, since the interpreter would have to check them at every call.

CHAPTER 5

Correctness of Implementation

The test suite of *casmi* was a very important part in the development. It is written in *Ruby* and simply executes all *CASM* files in the *tests* folder and checks its output and exit codes against the expected values. Since the test suite worked very well, the development has become more and more test-driven. At the time of writing *casmi* has 309 tests, the execution of these tests lasts for about four seconds. 42 preexisting *CASM* files were added to the tests to ensure compatibility with *casmintr*. These files reside in the directories `examples`, `tests/examples` and `tests/examples2`. Some of these files are also used for performance comparison with *casmintr*.

The rest of the test suite consists of small files, which check different features of *casmi*. Code coverage for the test suite was measured with *gcov* and is about 85%, that means 85% of the source code lines of *casmi* are executed while running the test suite. A large part of the not executed code is either unreachable or contains methods used for debugging. Therefore code coverage is good and the development of new features or bug fixing should not lead to major regressions.

5.1 Unit-Tests

Additional to the test suite, *casmi* also has unit tests to check internal functions of the interpreter. The unit tests are simple *C++* tests, which are compiled into its own binary *unittest*, with use of some simple helper functions from the *boost* test library. Executing tests also means invoking this binary.

There are unit tests for creating and comparing all different types of values, such as `Int`, `String` or `List`, but they also contain tests for types, where types are merged and compared.

5.2 Adding Test Cases

It is very easy to add test cases to the test suite of *casmi*, just create a `.casm`-file in the `tests` directory. When executing the test suite, *casmi* is invoked with every file in this directory. By

default only the return code of *casmi* is checked, which is assumed to be 0.

It is also possible to specify that the execution of a file should fail, through a single line comment in the first line of a file:

```
1 // error annotation @4
2
3 rule main = {
4   call (1) ( "hallo", "welt" )
5   program( self ) := undef
6 }
```

Listing 5.1: This test case expects an error on line 4

For listing 5.1 *casmi* should raise an error on line 4 in the type inference unit. It is essential that this is a single line comment on line 1. The test suite does not understand a multi-line comment or if this comment is not on the first line. The test will fail, if either the exit code or the line which raised the error, is different from the declared value.

Exit codes

It is important to note that *casmi* sets the exit code after the failing component. *casmi* has following different exit codes:

exit code 0
: Exit code is 0, if no error occurred during execution.

***lexer* for exit code 1**
: These are errors in the lexer, for example an unclosed string or comment, as well as illegal or unrecognized characters in the input file.

***parser* for exit code 2**
: All errors in the parser or structure of the program use exit code 2.

***annotation* for exit code 3**
: Exit code 3 is returned by all errors which are detected in the type inference unit. See chapter 4 for possible type annotation errors.

***interpreter* for exit code 4**
: All errors in the interpreter return exit code 4, such as division by zero or if the loading of external functions fails. Some annotation checks can only be performed at runtime, so these checks also return exit code 4 on error.

Checking the output

The test suite is also capable of checking the output of *casmi*, not only the exit code. For this purpose an *.expected* file with the same filename and path has to be created as the dedicated *.casm* file. Then the output of *casmi* is written into a *trace* file and compared to the expected output, which must be equal. After the comparison, the *trace* file is deleted. `-k` can be used as

parameter for running the test suite to keep the file on disk, which is very useful for debugging purposes.

Listing 5.2 and 5.3 demonstrate the check of the program output.

```
CASM helloWorld

init main

rule main = {
  print "Hello World!"
  program( self ) := undef
}
```

Listing 5.2: tests/basic/hello-world.casm

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

Listing 5.3: tests/basic/hello-world.expected

Symbolic execution mode

The test suite also tests the symbolic execution mode, `symbolic` is used in the comment on the first line to switch to this execution mode for this file.

```
// symbolic

function a: -> Int
function b: -> Int initially { 0 }

rule main = {
  straceon

  if a = 2 then {
    b := 1
  } else {
    b := 2
  }

  program( self ) := undef
}
```

Listing 5.4: tests/symbolic/eq1.casm

Since symbolic execution can lead to multiple execution paths and therefore to various different outputs, it is not more sufficient to have a single *.expected* file. All these outputs can be checked with the test suite, but the *.expected*-files use a slightly different naming convention. If *eq1.casm* is the filename, then *eq1_.expected* will be the expected output if no fork happened while executing. *tests/symbolic/eq1_I.casm* and *tests/symbolic/eq1_E.casm* are the expected output files, for listing 5.4. It is also possible to specify a function prefix and use parameter `--symbolic_f` with the line `// symbolic_f, prefix PRE` for a test case. The `prefix` option can also be used with `symbolic`.

5.3 Executing the Test Suite

The test suite should be as seamless and maintainable as possible, so just running `make` builds the program and runs all tests. Following output will be shown, if all tests run without errors:

```
executed 309 tests, 309 succeeded, 0 failed
```

Listing 5.5: Test suite output

If a test case fails, the file path and the error message from the test case will be shown:

```
exit code 0 expected but got 4
tests/basic/hello-world.casm failed

executed 309 tests, 308 succeeded, 1 failed
```

Listing 5.6: Test suite with a failed test case

`make test` executes all the tests alone including the unit test, while `make only` just builds the program.

It is also feasible to start the test suite with `ruby testsuite` (`make test` does that internally). Where the test suite also accepts files or directories as parameters. Then the test suite only executes the given files and all the files in the given directories. This feature is very handy for developing, to check only single or specific test cases. If no parameters are given, the test suite will execute all test cases in the `tests` directory.

CHAPTER 6

Evaluation

Performance was the major reason why the development of *casmi* was started. *casmintr*, the first prototype of a CASM interpreter, was not fast enough. The aim was to be 10 to 100 times faster than *casmintr*, which was written in Python. *C++* was chosen as the programming language for *casmi*, because of the strong focus on performance.

6.1 Benchmarks

To compare the performance of the two interpreters, the same CASM program is executed with both *casmi* and *casmintr*. Each interpreter is invoked 10 times with the same file to receive an average execution time. Only the average execution times of the two interpreters are compared.

The next table shows execution times of 15 different real-world programs. These are all programs, which are run in the test suite. The second column shows the execution time in seconds for *casmi*, while column 3 shows the time for *casmintr*. Column 4 shows the time for *casmintr* divided by the time needed for *casmi*. All tests were executed on a desktop computer with 64-bit Fedora 18 using GCC 4.7.2 and Python 2.7.3, the CPU is an Intel Core i5-3570K @3.4 GHz with 16 GiB RAM.

Since *casmintr* is written in Python, the interpreter has a startup overhead compared to *casmi*. Therefore *tests/basic/hello-world.casm* was used as a micro benchmark for measuring startup time of the interpreter, which is about 0.090 seconds for *casmintr*. The fifth column shows the factor between *casmi* and *casmintr* without the startup time of *casmintr*.

file name	casmi [s]	casmintr [s]	Factor	Factor*
tests/examples2/0.casm	0.035	6.879	194.688	192.141
tests/examples2/1.casm	0.029	6.871	236.931	233.827
tests/examples2/2.casm	0.036	8.393	235.318	232.794
tests/examples2/3.casm	0.032	10.203	318.844	316.031
tests/examples2/4.casm	0.031	6.356	205.022	202.118
tests/examples2/5.casm	0.032	8.781	274.406	271.594
tests/examples2/6.casm	0.021	6.900	333.855	329.500
tests/examples2/7.casm	0.028	8.917	314.707	311.530
tests/examples2/8.casm	0.037	8.365	226.072	223.640
tests/examples2/9.casm	0.037	10.304	276.009	273.598
tests/examples2/10.casm	0.026	8.608	326.899	323.481
tests/examples2/11.casm	0.022	6.817	309.879	305.788
tests/examples2/12.casm	0.019	6.572	345.894	341.157
examples/sad.casm	3.076	1077.123	350.208	350.179
tests/examples/proof35.mir.casm	0.026	0.123	4.658	1.241

All files except *sad.casm* run in symbolic mode, so potential performance problems in both modes should be detected through these benchmarks. *sad.casm* is an arithmetic-intensive program, which runs much longer than most other programs for translation validation. *0.casm* to *12.casm* are used in instruction set simulation and contain typical CASM code fragments. *proof35.mir.casm* was used as benchmark too, to also have a very short-running file.

6.2 Performance Analysis

The tests reveal big performance differences between *casmi* and *casmintr*. All files except *proof35.mir.casm* are more than 190 times faster than *casmi*, 7 of the 15 used benchmarks are more than 300 times faster in *casmi*. *sad.casm* is a big file with more than 18,500 lines of code, which lasts for about 18 minutes in *casmintr* and 3.076 seconds in *casmi*. *sad.casm* indicates that the performance advantage of *casmi* is also given for long running programs.

The benchmarks illustrate that startup time does not have a big impact on the comparison, however the very short program *proof35.mir.casm* with only 60 lines of code shows a big difference between the two factors. This is also by far the file with the smallest performance difference. The reason for this is that the execution time is so small that *casmi* cannot be much faster.

The interpreters spend most of the execution time interpreting, when further analyzing the execution time of *sad.casm* in *casmintr* and *casmi*. Although *casmintr* spends 2 seconds for parsing and 25 seconds for type annotation, this is only a small portion compared to the overall execution time of 1077 seconds. The same is still true for *casmi*, which spends 0.01 seconds parsing and 0.03 for type annotation, so the rest of the 3.076 seconds are spent on interpreting. One should focus on interpreting for further improving the execution speed.

The big difference for parsing is a bit unexpected, since *casmi* and *casmintr* both use lex/yacc-like tools for lexing and parsing and therefore the code looks similar. The performance differ-

ence seems to be mostly caused by the used library and programming language. Type inference uses a completely different algorithm in *casmi*, which explains the big difference here. *casmintr* traverses the AST and annotates nodes until all nodes in the tree are stable, while *casmi* only annotates specific parts of the tree again.

CHAPTER 7

Conclusion

As demonstrated in this thesis, *casmi* is a better performing and a better tested successor of *casmintr*, the first prototype of CASM. *casmi* is in all benchmarks, except the very short *proof35.mir.casm*, at least 190 times faster than *casmintr*. Therefore the aim, being 10 to 100 times faster than *casmintr* could be fulfilled.

CASM matured as a language with superior type annotation and became incompatible with *CoreASM*. *casmi* also serves as a good starting point for further improvements of CASM, the test suite is easily extensible and already introduced features are not so likely to break because of existing tests and their good code coverage. It is also easy to add new functions or statements to *casmi*.

Due to the implementation of type inference in *casmi*, most of the work is done that the compiler can easily be rewritten in C++ using the annotated AST of *casmi*. The type inference unit is not needed in its full extent for the interpreter alone. The compiler can reuse the lexer, parser and type inference by using the AST of the interpreter. All existing test cases in the test suite can also be used to check the correctness of the CASM compiler.

7.1 Further Work

For further performance improvements in *casmi*, a benchmark suite could be useful to compare run times with older versions. Since *casmi* is a simple AST interpreter, there is still room for improvements. The interpreter could use faster architectures such as threaded code [1] or a JIT-compiler.

Since the missing syntactic differentiation between a list and a tuple constant was a tough part while implementing type inference, these constants could be separated in a later version of CASM. At the moment it is also possible to assign arbitrary integer values to an enumeration, due to the automatic conversion of integers to an enumeration and vice versa. Enumeration types and integer could be completely separated, therefore new built-in functions could be needed to convert between these types.

The test suite does not allow to check the error message in test cases. This could be implemented in future versions of the test suite and to make this check more robust, error codes could be introduced in *casmi* to uniquely identify a message. Therefore changing error messages would not break tests. However for the demands of CASM it was so far enough to check exit code and the line for errors.

```
1 // error annotation @4 EC0024
```

Listing 7.1: Defining error code for a test case

Bibliography

[1] James R. Bell. Threaded code. *Commun. ACM*, 16(6):370–372, June 1973.

[2] Egon Börger and Joachim Schmid. Composition and Submachine Concepts for Sequential ASMs. In *Proceedings of the 14th Annual Conference of the EACSL on Computer Science Logic*, pages 41–60, London, UK, UK, 2000. Springer-Verlag.

[3] Roozbeh Farahbod. CoreASM Language User Manual. `http://www.coreasm.org/downloads/CoreASM-UserManual-DRAFT.pdf`, 2006. Accessed: 2013-04-10.

[4] Roozbeh Farahbod and Vincenzo Gervasi. Design and specification of the coreasm execution engine. `http://coreasm.svn.sourceforge.net/viewvc/coreasm/engine-carma/trunk/doc/CoreASM-DesignDocumentation.pdf`, 2005. Accessed: 2013-04-10.

[5] Yuri Gurevich. Specification and validation methods. chapter Evolving Algebras 1993: Lipari Guide, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995.

[6] Roland Lezuo and Andreas Krall. A unified processor model for compiler verification and simulation using asm. In *Proceedings of the Third international conference on Abstract State Machines, Alloy, B, VDM, and Z*, ABZ'12, pages 327–330, Berlin, Heidelberg, 2012. Springer-Verlag.

[7] 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.

[8] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Commun.*, 15(2,3):91–110, August 2002.

[9] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.