Masaryk University
Faculty of Informatics

logo.pdf

C++ Support for Stanse

Master's Thesis

Martin Vejnár

Brno, May 2011

Declaration

I hereby declare that this thesis is original work, which I have written on my own. All sources, references, and literature used or excerpted during its creation are properly cited and listed in the bibliography section.

Advisor: Mgr. Jan Obdržálek, PhD.

Acknowledgement

TODO

Abstract

TODO

Keywords: TODO

Contents

Introduction **3**

1 Intermediate representation **5**
1.1 Syntax 6
1.2 Semantics 10
1.3 Multithreaded programs 19
1.4 JSON encoding of SIR program units 19
1.5 Tagging 21
1.6 Merging of SIR units 22

2 Modelling of C++ features **23**
2.1 Naming of program entities 23
2.2 Fundamental types 25
2.3 References 26
2.4 String literals 26
2.5 Unions 27
2.6 Raw memory 27
2.7 Argument passing 28
2.8 Variadic functions 29
2.9 Virtual dispatch 29
2.10 Dynamic allocation 30
2.11 Exceptions 32
2.12 Subroutine layout 35

3 Parsing C++ programs **36**
3.1 Sentinel nodes 38
3.2 Extended operands 39
3.3 Execution context 40
3.4 Exception paths 42

4 Automaton checker **43**
4.1 Pattern matching 44
4.2 Checker description 46

4.3 Propagation and error checking 47

5 Tools 48

5.1 Translator 48

5.2 Testing 49

5.3 Pretty printing 49

Bibliography 51

Introduction

Stanse is a tool that was designed to perform static analysis on computer programs written in a variety of programming languages. While it currently supports only the C99 language[2] together with several GNU extensions,[1] its internal structure allows for inclusion of additional languages. It is the primary goal of this thesis to extend the tool with the support for the C++ programming language in its ISO/IEC 14882:2003 revision[12].

Stanse achieves language independence by separating the source code parsing from the actual analysis. The parsing of source files is performed by a set of front-ends, each of which is designed to handle a specific programming language (or a family thereof). Front-ends output a language-independent model of the source code behavior, which is then used by various back-ends to perform analyses and output error traces. Splitting a tool to front-end and a back-end parts is a common technique employed by major compiler suites, including gcc[6] and LLVM. In the context of Stanse, we refer to front-ends as *parsers* and to back-ends as *checkers*.

The design of the interface between parsers and checkers—the language-independent *intermediate representation* (IR)—is of particular concern as it affects the ability of parsers to model the features of their respective languages in a way that is convenient to checkers. We will refer to the interface as the intermediate representation or intermediate language, even though the representation is in no way intermediate (it is in fact the representation that checkers perform analyses on and might therefore be considered final).

Stanse was originally written with only a single parser (for the language C), whose output naturaly defined the intermediate representation. The C parser outputs an XML serialization of the abstract syntax tree (AST) that represents the translation unit being parsed. The parser also constructs, for each function, a control-flow graph whose nodes refer to elements in the AST. One node is constructed for each expression statement (i.e. for each full expression).

Unfortunately, the AST of the C language is not an ideal language-independent representation of programs for several reasons. First, it is difficult to represent certain programming constructs—notably exception handling—in a manner that would directly communicate the set of control-flow paths to checkers. Furthermore, as the control-flow graphs are constructed with granularity of a full-expression, the control-flow within these

[1] Stanse is being periodically used to scan Linux kernel sources.

expressions is not explicitely captured. Note that a very complex behavior may result from expressions containing the conditional operator or short-circuiting logical operators, especially when automatic variables or exception handling is involved. Lastly, the C language AST is unnecessarily redundant, making it difficult to develop both parsers and checkers.

We have therefore decided to discard the AST-based intermediate language and developed a new, simpler and extensible one, which we have named the *Stanse intermediate representation* or *SIR* for short. Unfortunately, this change severely breaks all of the existing checkers. We have therefore modified Stanse in such a way, that program units represented in the old and the new representation can coexist side by side. All of the checkers can therefore still be used for the checking of C programs.

We have adapted one of the existing checkers to work with the new representation in order to ensure that the C++ parser works correctly (the amount of work involved in fixing all of the existing checkers is considerable). The checker of choice was the automaton checker, as it relies on the intermediate representation merely to search for patterns. Its reliance on the actual form of the representation therefore quite low.

In Chapter 1 we discuss the new SIR language in detail. We develop the syntax and semantics of elementary instructions and discuss the control-flow capabilities that the language provides. We also provide the reader with formal semantics for SIR, thus setting up a framework in which checkers can be reasoned about. In Chapter 2, we show how various features of the C++ language can be modelled in SIR, thus delivering a general idea of how a translated C++ program looks like. The motivation for some aspects of SIR semantics also becomes clear in this chapter. In Chapter 3, we reveal some of the key concepts involved in a the actual process of translating the C++ program (or more precisely a C++ translation unit) to SIR and describe some of the technical challenges involved. In Chapter 4 we then give an overview of how a translated program can be checked using the automaton checker. Finally, in Chapter 5, we discuss the tools that were created as a part of the thesis and how they can be invoked and used.

TODO: related work?

Chapter 1

Intermediate representation

In this chapter, we describe a new intermediate language that is to be used in Stanse as the interface between parsers and checkers. This new language replaces the original C-centric one.

Let us first note that from an engineering perspective, using an existing language—one that is well-recognized by the compiler community—would be a far better solution than developing a new one. In fact, the the LLVM assemly language[15] is specifically designed to serve as a language-independent program representation, yet it simultaneously delivers information about the program at a very high level[16]. Utilizing LLVM, we would immediately be granted support for a variety of languages, including C, C++, Objective C (all of which are handle by LLVM front-end called known as Clang), Fortran, Ada and D.

We have, however, found that the LLVM language, while providing all the information necessary for program optimization and code generation, and in fact providing nearly all information that we might find useful for static analysis, lacks the ability to represent some of the aspects of the C++ language, notably nondeterminism,[1] and presents some information in form that is difficult to deal with (virtual calls are performed through virtual tables; matching a call site to a set of potential callers is in this case non-trivial). In addition, the LLVM assembly is rather awkward to handle, especially since Stanse is a student-developed project.

We have therefore developed our own intermediate representation, which we refer to as the *Stanse intermediate representation* (SIR). SIR is transported in a JSON-encoded form—nearly any scripting language can manipulate JSON objects directly.[8] We have written several scripts in Python that perform tasks ranging from pretty-printing to merging of SIR program units. In Stanse (which is written in Java), we use a small Java library to parse the JSON-encoded SIR files.

One of the primary design goals for the new language was that it should be minimal. The complexity of the intermediate representation directly reflects on the complexity of checkers and we consider it important to make the process of creating new checkers as straightforward as possible. We hope that having a simple, yet powerful intermediate

[1] In C-like languages, the order of evaluation of subexpressions is unspecified.

language will encourage the development of both parsers and checkers. However, we do not believe that flexibility should be sacrificed for the sake of simplicity—it should still be possible to represent features of arbitrary languages accurately.

In light of the goals outlined above, we designed SIR with both simplicity and flexibility in mind. In our representation, programs are broken into smaller units (corresponding to procedures and functions in the original programs) called SIR subroutines, each of which consists of a set of nodes labelled by elementary instructions. The nodes are interconnected by conditional edges, forming a control-flow graph. A set of SIR subroutines then forms a SIR program unit; a set of SIR program units forms a SIR program.

The set of elementary instructions is not fixed and is easily extensible, making it possible for parsers to adapt SIR to new source languages. This would not be possible with an intermediate representation targeted at code generation, as the act of extending the instruction set would cause all existing back-ends to cease functioning. On the other hand, in the context of static analysis, checkers can often ignore unknown constructs and still produce useful results.

In this chapter, we first define the abstract syntax of SIR instructions and introduce the notion of a SIR program. We then give partial semantics to SIR programs. We put emphasis on control-flow aspects of the representation, while we deliberately stay vague with respect to data manipulation, defining only a few basic instructions to perform it. Note that the existing checkers make no use of data values and as such trying to define a precise data model would be an exercise in futility, and might in fact hinder attpempts to integrate some more esoteric languages if the model were later found inadequate.

At the end of the chapter, we define a JSON-based concrete encoding of SIR units. We also show how useful information unrelated to program behavior (e.g. source positions) can be communicated to checkers.

1.1 Syntax

Syntactically, a SIR subroutine is a control-flow graph, i.e. a graph, whose nodes are labelled by program instructions. Instructions modify the state of the program or interact with the external environment. The edges of the control-flow graph are labelled by conditions, which—based on the state of the program—determine whether a particular edge is enabled, and therefore restrict and direct the control flow. SIR utilizes control-flow graphs as they immediately lend themselves to certain types of static analysis.[7]

SIR modifies the notion of a control-flow graph from [7] to include a few necessary concepts, notably subroutine calls and multiple exit points. Furthermore, SIR allows edges that lead away from subroutine call nodes to be conditioned on the exact exit point taken to return from the call. We will later use this particular feature to model C++ exception paths.

Note that in other publications (with [7] being a notable exception), a node in a control-flow graph represents a sequence of instructions—called a basic block—as opposed to a single one. We have chosen to label each node with a single instruction, so as to ease the transition from the Stanse's original intermediate representation.

$$\langle \textit{inst} \rangle ::= \ [\ \langle \textit{nodelabel} \rangle\ \textbf{:}\ ]\ \langle \textit{opcode} \rangle\ [\ \langle \textit{operand} \rangle\ (\ \textbf{,}\ \langle \textit{operand} \rangle)^*\ ]$$

$$\langle \textit{operand} \rangle ::= \ \langle \textit{nodelabel} \rangle \mid \langle \textit{subroutine} \rangle \mid \langle \textit{var} \rangle \mid \textbf{\&}\ \langle \textit{var} \rangle \mid \langle \textit{const} \rangle$$

$$\langle \textit{const} \rangle ::= \ \textbf{null} \mid \langle \textit{number} \rangle \mid \langle \textit{string} \rangle \mid \langle \textit{array} \rangle \mid \langle \textit{object} \rangle$$

$$\langle \textit{array} \rangle ::= \ \textbf{[}\ [\ \langle \textit{const} \rangle\ (\ \textbf{,}\ \langle \textit{const} \rangle)^*\ ]\ \textbf{]}$$

$$\langle \textit{object} \rangle ::= \ \boldsymbol{\{}\ [\ \langle \textit{objectentry} \rangle\ (\ \textbf{,}\ \langle \textit{objectentry} \rangle)^*\ ]\ \boldsymbol{\}}$$

$$\langle \textit{objectentry} \rangle ::= \ \langle \textit{string} \rangle\ \textbf{:}\ \langle \textit{const} \rangle$$

Figure 1.1: The EBNF syntax of elementary SIR instructions.

The actual format used to transport SIR units between parsers and checkers is based on JSON. While JSON is well-suited for machine processing, it is quite difficult to read by humans, therefore we write instruction down using an alternative syntax, which—while distinct from the syntax of JSON-encoded instructions—captures their structure adequately.

1.1.1 Instructions

The Stanse intermediate representation uses the single static assignment form.[9] When a node of the control-flow graph is executed, the instruction associated with it modifies the state of the program and yields a value; the value is bound to the node and can be retrieved (but not modified) by subsequent instructions.

The grammar shown in Figure 1.1 gives the abstract syntax of instructions. An instruction starts with an optional label (a node identifier) which uniquely identifies the node in the context of its containing SIR subroutine, and which additionally defines a handle that can be used to access the value bound to it. In the JSON-encoded form, instructions are stored in an array and an index into this array serves as the node's unique label.

The optional label is followed by an opcode—a name that determines the type of the instruction—and a sequence of an arbitrary number of operands. We will write opcodes in bold font. When an instruction is executed, the values of its operands are computed and passed to the instruction for processing.

An operand is either a name of a subroutine, a value of a variable, a pointer to a variable, a constant value, or an identifier of a node. As mentioned above, the latter serves to retrieve the value bound to that node. (Node identifiers are treated specially by the **phi** instruction, which is used to select among values when two control-flow branches merge.) Beyond nodes, SIR also supports the concept of variables—objects whose address may be taken (lvalues) and whose value may be modified indirectly.

Constant values follow the JSON[8] data model, which can represent four primitive types—strings, numbers, booleans and a special value **null**—and two structured types—objects and arrays. The words object and array come from the convention of JavaScript. We removed booleans from the set of allowed values—having a separate types for booleans and numbers is rarely useful at this level of abstraction.

For the purposes of this text, we differentiate between the various types of operands using either prefixes or typography as follows.[2]

- Node labels are non-negative numbers prefixed by the dollar sign (e.g. \$1).
- Subroutine names are written in monospace font without any prefixes.
- Variable names are written in italics.
- Constant numbers, arrays, dicts and the **null** constant are all easily recognized. Constant strings are enclosed in double quotes and are written using a monospaced font (e.g. `"text"`).

Figure 1.2 shows an example of four instructions. The first causes the values of variables x and y to be added together. The result of the addition is bound to the node \$1. The second instruction is passed a pointer to the variable x and a label to the first node. As the name suggests, the instruction assigns the value of the second operand to the variable pointed to by the first. In this case, the sum of x and y is stored back to x.

The third instruction takes the new value of x and executes the subroutine named `foo`. The program then exits through exit point 0 (exit points are explained below), forwarding the value returned by `foo`.

1.1.2 Conditional branches

Every edge of a SIR control-flow graph is labelled by two values—an exit index and a condition. An exit index is a non-negative integer, while a condition is a constant (i.e. a product of ⟨*const*⟩ nonterminal). Each instruction also yields two values, which are then matched against the outgoing edge labels. In order for an edge to be enabled, its exit index and the exit index returned by the instruction must match precisely.

If the condition associated with an edge does not have the value of **null**, the condition value and the return value of the instruction must match as well. Furthermore, if any edge with a non-null condition is enabled, all edges with a null condition are disabled. The null condition therefore serves as the `else` part in `if-else` statements, or the `default` label in `switch` statements.

The notation we use in this text to write SIR programs assumes that there is an edge between each two successive instructions labelled with the exit index 0 and the

[2] In JSON-encoded units, operand types are stored explicitely and do not follow any of these conventions.

\$1:	**add** x, y	
	assign &x, \$1	`x += y;`
\$3:	**call** `foo`, x	`return foo(x);`
	exit 0, \$3	

Figure 1.2: An example of a SIR subroutine and the C++ code used to generate it.

```
def  fact(x):
     $1:  value x | 0 → $4
     $2:  phi $1, $6
          exit 0, $2

     $4:  sub x, 1
     $5:  call fact, $4
     $6:  mul x, $5 |→ $2
```

```
int fact(int x) {
    if (x)
        return x * fact(x - 1);
    else
        return 1;
}
```

Figure 1.3: An example of conditional branches in SIR.

null condition. If there is no such edge, we insert vertical space in between the two instructions.

If there are any additional edges leading from a node, we represent them in the notation by suffixing the instruction with the list of these additional edges. For each edge, we write $c \xrightarrow{i} n$, where c is the condition, i is the exit index and n is the target node. We omit c if its value is **null** and we also omit i if $i = 0$.

Figure 1.3 shows the representation of a SIR subroutine named `fact`, which computes a factorial of its only parameter x. The **value** instruction, which labels the node \$1, merely returns the value of its only argument, so that it can be matched against edge conditions. Two edges lead from the first node: the implicit $(0, \textbf{null})$ edge leading to \$2, and an explicit $(0, 0)$ edge leading to \$4. The **phi** instruction is used to gather results after the two branches join; it determines which of the nodes passed as arguments were executed last and returns its value.

1.1.3 Subroutines and program units

A SIR subroutine consists of a SIR graph, a concept described in the previous section. One of the nodes of the graph is chosen to be the entry node (in our notation, we write this node first and name it \$1). Furthermore, a SIR subroutine has a name, which can be passed to instructions as an operand. A subroutine also carries along a set of variable names. Variables in this set are considered local, all other variables are global. Using a local variable name as an operand to an instruction references an instance of the variable which is unique to the current subroutine invocation. Finally, a sequence of names from the local variable set forms the subroutine's parameter list.

We include the name and the parameter list in the notation (see for example Figure 1.3). We however omit the set of local variable names and assume that whether a variable is local or global is clear from the context. Additional information is attached to SIR subroutines when they are JSON-encoded (source code positions for example), but we do not include it in the human-readable notation.

Formally we define a *SIR subroutine* f to be a tuple $f = (N, \to, \iota, n_0, L, p)$, where $N \subseteq \langle \mathit{node} \rangle$ is an arbitrary finite set of *nodes*, $\to \in N \times \mathbb{N}_0 \times \langle \mathit{const} \rangle \times N$ is the set of labelled *edges*, $\iota \colon N \to \langle \mathit{inst} \rangle$ is the labelling of nodes with instructions, $n_0 \in N$ is the entry node, $L \subseteq \langle \mathit{var} \rangle$ is the set of local variable names, and $p \in L^*$ is the sequence of parameter variable names. We denote the set of all subroutines as $\mathcal{F}$.

A *SIR program unit* U is then represented as a partial mapping from subroutine names to subroutines, $U\colon \langle \mathit{subroutine} \rangle \to \mathcal{F}$. Again, more information is attached with SIR units in its JSON-encoded form (notably the initial values of global variables).

A *SIR program* consists of a (possibly empty) set of SIR program units.

1.2 Semantics

In this section we describe the semantics of SIR program units by demonstrating how a labelled transition system can be constructed from them. We make use of small-step semantics[22] as this type of behavioral description reflects in a direct manner the implementation of a potential simulator.[3] Furthermore, there are known techniques to generate abstract interpretations from small-step semantics.[18].

Note that we define the semantics in order to communicate the desired meaning of instructions and their operands. It should be stressed that certain aspects of our semantics, in particular the data model, are specified here merely to accomplish the aforementioned goal. Checkers and simulators need not adhere strictly to this specification. In fact, checkers will most likely operate with a different, more abstracted data model. Simulators, on the other hand, will find it necessary to extend the semantics to support additional features like dynamic memory or system objects (e.g. files).

We start the description by defining the domain of values that SIR programs can manipulate (i.e. values that can be passed as operands to instruction, values that instructions may yield, and the values that can be stored in variables). We then precisely define the state of execution, i.e. the set of states of the small-step transition system. Finally, we specify how the execution state evolves.

1.2.1 Value domain

We denote the set of all data values, or simply the *value domain*, as $\mathcal{D}$. The domain includes the special value $\bot$, all real numbers, and the set of all string $\mathcal{S}$. We are content with defining $\mathcal{S}$ informally as the set of objects corresponding to JSON strings. We do not in any way exploit the internal structure of strings.

In addition to these values, we add subroutine identities to the domain. We define the set of subroutine identities as

$$\Lambda = \{\lambda_f \mid f \in \langle \mathit{subroutine} \rangle\},$$

where λ_f is a unique symbol representing the identity of the subroutine f.

The domain also contains the structured array and object values. Arrays and objects in $\mathcal{D}$ main contain $\bot$, reals, strings from $\mathcal{S}$, subroutine identities, variable identities (described below) and other arrays and objects.

[3] Although the ability to simulate SIR programs is not the goal, having that ability strenthens our belief that the SIR language is in a certain sense complete.

For variable identities (i.e. pointers to variables), the situation is slightly more complex. Variables are identified by their name and the context in which they were instantiated. Global variables are naturally instantiated in the global context, whereas local variables are associated with the execution frame in which they were created. We will discuss execution frames later. Variables cease to exist when their associated context is destroyed.

We denote global variables simply by their name. On the other hand, local variables are tuples $(i, x) \in \mathbb{N}_0 \times \langle var \rangle$, where i is the identifier of the associated execution frame and x is the variable identifier. We define the set of basic variable identities as

$$\Omega = \{\omega_x \mid x \in \langle var \rangle\} \cup \{\omega_{i,x} \mid i \in \mathbb{N}_0, x \in \langle var \rangle\},$$

where the symbol ω_x represents the identity of the global variable x, and $\omega_{i,x}$ refers to the identity of the local variable (i, x). We will write members of Ω simply as ω or ω_i.

For variables that hold arrays or object values, we can construct an identity of subobject of the variable. To specify a pointer, one therefore must provide the basic identity of the variable, and the sequence, possibly empty, of member identities.

The set of member identities (pointers to members), is defined as

$$M = \{\mu_z \mid z \in \langle string \rangle\} \cup \{\mu_i \mid i \in \mathbb{N}_0\}.$$

The values of the form μ_i refer to members of arrays, whereas μ_z are used to refer to members of objects. We will write the members of M simply as m or m_i.

Finally, we define the set of qualified variable identities as

$$\Psi = \{\omega m \mid \omega \in \Omega, m \in M^*\},$$

where the sequence $\omega m_1 m_2 \cdots m_n$ represents the identity of the subobject $m_1 m_2 \cdots m_n$, of the variable with the basic identity ω. For instance, consider the local variable x in the execution frame i, which is assigned the value {"a" : 42}. The qualified identity $\omega_{i,x}\mu_a$ is a pointer to the subobject a of the variable x. Dereferencing such a pointer yields the value 42. We use the notation ψ or ψ_i for the members of Ψ.

The domain $\mathcal{D}$ is then the smallest set containing

- the special value $\perp$, corresponding to the constant **null**,
- all real numbers $x \in \mathbb{R}$,
- all string $s \in \mathcal{S}$,
- all qualified variable pointers $\psi \in \Psi$,
- all subroutine identities $\lambda \in \Lambda$,
- all arrays $[x_0, x_1, \ldots, x_{n-1}]$, where $x_1, x_2, \ldots x_n \in \mathcal{D}$, and
- all objects $\{s_1 : x_1, s_2 : x_2, \ldots, s_n : x_n\}$, where $x_1, \ldots x_n \in \mathcal{D}$, and $s_1, \ldots s_n \in \mathcal{S}$.

Note that the terminal values derived from the $\langle \mathit{const} \rangle$ nonterminal all have direct counterparts in $\mathcal{D}$. For a syntactic constant $c \in \langle \mathit{const} \rangle$ we denote $\mathcal{M}(c) \in \mathcal{D}$ the semantic value that is associated with c.

If $x \in \mathcal{D}$ is an array $[x_0, x_1, \ldots, x_{n-1}]$, then we use the notation $\mu_i(x)$ to access the i-th element of the array, i.e. $\mu_i(x) = x_i$, if $0 \leq i < n$, $\mu_i(x) = \perp$ otherwise. Notice that arrays are indexed from zero.

Similarly, if x is a dictionary, $x = \{s_1 : x_1, s_2 : x_2, \ldots, s_n : x_n\}$, then $\mu_s(x) = x_i$ if $s = s_i$ for some $1 \leq i \leq n$, and $\perp$ otherwise.

1.2.2 Execution state

The semantics of a SIR program are given by a labelled transition system, a tuple $(\Sigma, \mapsto, L)$, where Σ is the (possibly infinite) set of states, L is the set of labels, and $\mapsto \in \Sigma \times L \times \Sigma$ is the transition relation. In small-step semantics, all edges are labelled with the same label, τ. We therefore set $L = \{\tau\}$ and we write $\alpha \mapsto \beta$ instead of $(\alpha, \tau, \beta) \in \mapsto$.

The state of execution of a SIR program consists of a stack of execution frames and a binding of values to variables and node instances. Execution frames estabilish a context for local variables and node instances and are always associated with a subroutine. They are pushed to the stack whenever a subroutine is called; they are popped when the execution reaches an exit node. Each execution frame is given a unique number from $\mathbb{N}_0$. Once a frame identifier is used, it is never used again.

As mentioned above, besides global and local variables, values can also be bound to node instances. Akin to a local variable, a node instance is tuple (i, n), where $i \in \mathbb{N}_0$ is the identifier of the execution frame with which the node instance is associated, and $n \in \langle \mathit{node} \rangle$ is the identifier of the node. A SIR program cannot form pointers to node instances.

The binding of a value to a node instance is a tuple $b = (i, n, x)$, where (i, n) is a node instance and $x \in \mathcal{D}$ is a value. We denote the set of all bindings as $\mathcal{B}$.

The state of node instances is described by a binding sequence, $B = b_1 \cdots b_k \in \mathcal{B}^*$. The value bound to a node instance (i, n) according to the binding sequence $B = b_1 \cdots b_k$ is the value assigned by the rightmost binding corresponding to (i, n). We denote the value as $\mathcal{M}(B, i, n)$,

$$\mathcal{M}(B, i, n) = \begin{cases} \perp, & \text{if } B = \varepsilon, \\ x, & \text{if } b_k = (i, n, x), \\ \mathcal{M}(b_1 \cdots b_{k-1}), & \text{otherwise.} \end{cases}$$

Formally, we define an execution frame to be a tuple (i, f, n), where $i \in \mathbb{N}_0$ is the frame identifier, $f \in \mathcal{F}$ is a SIR subroutine with the set of nodes N, and $n \in N$ is an identifier of the node of the subroutine that is currently being executed. We denote the set of all execution frames as $\chi = \mathbb{N}_0 \times \langle \mathit{subroutine} \rangle \times \langle \mathit{node} \rangle$.

We then define the state of the execution $\sigma \in \Sigma$ as a tuple $\sigma = (c, v, B, i)$, where $c \in \chi^*$ is a stack of execution frames representing the the current *control state* of the execution, $v\colon \Omega \to \mathcal{D}$ is a partial mapping from basic variable identities to their assigned values, $B \in \mathcal{B}^*$ is a sequence of node instance bindings, and $i \in \mathbb{N}_0$ is the next available execution frame number.

We extend the function v to qualified identities in a natural manner; the function $v\colon \Psi \to \mathcal{D}$ is defined as

$$v(\psi) = \begin{cases} v(\omega), & \text{if } \psi = \omega \text{ for some } \omega \in \Omega, \\ m_n(v(\omega m_1 \cdots m_{n-1})), & \text{if } \psi = \omega m_1 m_2 \cdots m_n \text{ for some } \omega \in \Omega,\ m_i \in M. \end{cases}$$

Note that we maintain the values of node instances as a sequence of bindings instead of a simple partial map from node instances to their values, as the order in which the values were bound is significant. The **phi** instruction uses the order to choose a node that was bound last. This way, the correct value can be selected after two or more branches of execution join.

We define the following two operators to simplify the manipulation with node binding sequences. The operators will be used later to define the semantics of instructions. The operator unbind: $\mathcal{B}^* \times \mathbb{N}_0 \times \langle \mathit{node} \rangle \to \mathcal{B}^*$ removes all bindings to a given node instance. Formally, let $B = b_1 b_2 \cdots b_k$. The operator is defined recursively as

$$\operatorname{unbind}(B, i, n) = \begin{cases} \varepsilon, & \text{if } B = \varepsilon, \\ \operatorname{unbind}(b_2 \cdots b_k), & \text{if } b_1 = (i, n, x) \text{ for some } x \in \mathcal{D}, \\ b_1 \operatorname{unbind}(b_2 \cdots b_k), & \text{otherwise.} \end{cases}$$

Furthermore, we define the operator unbind′: $\mathcal{B}^* \times \mathbb{N}_0 \to \mathcal{B}^*$, which removes all bindings relating to a specific execution frame, as follows.

$$\operatorname{unbind}'(B, i) = \begin{cases} \varepsilon, & \text{if } B = \varepsilon, \\ \operatorname{unbind}'(b_2 \cdots b_k), & \text{if } b_1 = (i, v, x) \text{ for some } v \in \langle \mathit{var} \rangle \text{ and } x \in \mathcal{D}, \\ b_1 \operatorname{unbind}'(b_2 \cdots b_k), & \text{otherwise.} \end{cases}$$

Lastly, the operator bind: $\mathcal{B}^* \times (\mathcal{B} \cup \{\bot\}) \to \mathcal{B}^*$ is defined by

$$\operatorname{bind}(B, (i, n, x)) = \begin{cases} \operatorname{unbind}(B, i, n)(i, n, x), & \text{if } x \neq \bot, \\ \operatorname{unbind}(B, i, n), & \text{otherwise.} \end{cases}$$

1.2.3 Operands

Most instructions (in fact, the only exception to the rule is the **phi** instruction) when they are executed, resolve their operands into values they represent in the context of the current state.

Let $\sigma \in \Sigma$ be a state, $\sigma = (c, v, B, i)$ and $c = (i_0, f_0, n_0)(i_1, f_1, n_1) \cdots (i_k, f_k, n_k)$. The active subroutine in the state σ is the subroutine $f_k = (N, \to, \iota, n_0, L, p)$. While in state σ the variable name $x \in \langle \mathit{var} \rangle$ refers to the local variable (i_k, x), if $x \in L$, otherwise it refers to the global variable x. We denote $G = \langle \mathit{var} \rangle \setminus L$ the set of variable names which refer to global variables in the state σ.

We define the *meaning* of operands $\mathcal{M}\colon \langle \mathit{operand} \rangle \times \Sigma \to \mathcal{D}$ based on the structure of the first argument as follows:

- for $c \in \langle const \rangle$, $\mathcal{M}[\![c]\!]\sigma = \mathcal{M}(c)$,
- for $f \in \langle subroutine \rangle$, $\mathcal{M}[\![f]\!]\sigma = \lambda_f$,
- for $n \in \langle node \rangle$, $\mathcal{M}[\![n]\!]\sigma = \mathcal{M}(B, i_k, n)$,
- for $x \in L$, $\mathcal{M}[\![x]\!]\sigma = v(\omega_{i_k,x})$,
- for $x \in L$, $\mathcal{M}[\![\&x]\!]\sigma = \omega_{i_k,x}$,
- for $x \in G$, $\mathcal{M}[\![x]\!]\sigma = v(\omega_x)$, and
- for $x \in G$, $\mathcal{M}[\![\&x]\!]\sigma = \omega_x$.

1.2.4 Instructions

Recall that the semantics of a SIR program are given by a labelled transition system $(\Sigma, \mapsto, \{\tau\})$. We now define the transition relation $\mapsto$, which in other words requires us to describe all tuples of states $\sigma, \sigma' \in \Sigma$ such that $\sigma \mapsto \sigma'$. Let therefore $\sigma = (c, v, B, s)$ and $\sigma' = (c', v', B', s')$ be two states of the labelled transition system. Furthermore let $c = c_1 c_2 \cdots c_k$, with $c_i = (i_i, f_i, n_i)$. Similarly $c' = c'_1 c'_2 \cdots c'_{k'}$, with $c'_i = (i'_i, f'_i, n'_i)$. In the state σ, we say that c_k is the active execution frame, and the subroutine f_k the active subroutine. Let $f_k = (N, \rightarrow, \iota, L, p)$.

In state σ, we call the node n_k the current node. We say that in the state σ, $\iota(n_k)$ is the *current instruction*. We overload the notation and denote the current instruction in state σ also as $\iota(\sigma)$, or simply ι, if the state is obvious from the context.

We now describe the set of possible transitions from the state σ based on the type of the current instructions. Instructions **call**, **exit**, and **assign** manipulate the execution state in a complex manner, and are therefore called *complex*. All other instructions are called *simple*. For every simple instruction ι, we define its meaning $\mathcal{M}[\![\iota]\!]\sigma \in \mathcal{D}$, which corresponds to the value the instruction yields.

Simple instructions cause the execution stack to remain unchanged, save for the node indentifier of the rightmost frame. For such instructions, the current node changes along an edge in the active subroutine. We define the successor function succ: $\chi^* \times \mathbb{N}_0 \times \langle const \rangle \rightarrow 2^{\chi^*}$ by $c' \in \mathrm{succ}(c, j, \varphi)$ if and only if $k' = k$, $c_i = c'_i$ for all $1 \leq i < k$, $i_k = i'_k$, $f_k = f'_k$, and n'_i is reachable from n_i via an edge labelled with (j, φ), i.e. $(n_i, j, \varphi, n'_i) \in \rightarrow$.

If ι is a simple instruction, then $\sigma \mapsto \sigma'$ for all $\sigma' \in \Sigma$ such that $\sigma' = (c', v, B', i)$, $B' = \mathrm{bind}(B, (i_k, n_k, \mathcal{M}[\![\iota]\!]\sigma))$, and $c' \in \mathrm{succ}(c, 0, \mathcal{M}[\![\iota]\!]\sigma)$.

Note that if an instruction is malformed (i.e. the operands or their values are incorrect for that particular instruction) or its opcode in unknown, we threat the instruction as a simple instruction with meaning $\mathcal{M}[\![\iota]\!]\sigma = \bot$.

Arithmetic instructions

Arithmetic instructions are simple instructions which perform basic arithmetic calculations. The **value** instruction resolves the value of its only operand and returns it. The instruction is used to bind a value to the current node in order to branch the execution according to it.

$$\mathcal{M}[\![\textbf{value } \alpha]\!]\sigma = \mathcal{M}[\![\alpha]\!]\sigma,$$

The addition, subtraction, multiplication, divisition, remainder and negation instructions have obvious meaning and use. All of them require that the meaning of their operands is a real number.

$$\begin{aligned}
\mathcal{M}[\![\textbf{add } \alpha,\beta]\!]\sigma &= \mathcal{M}[\![\alpha]\!]\sigma + \mathcal{M}[\![\beta]\!]\sigma,\\
\mathcal{M}[\![\textbf{sub } \alpha,\beta]\!]\sigma &= \mathcal{M}[\![\alpha]\!]\sigma - \mathcal{M}[\![\beta]\!]\sigma,\\
\mathcal{M}[\![\textbf{mul } \alpha,\beta]\!]\sigma &= \mathcal{M}[\![\alpha]\!]\sigma \cdot \mathcal{M}[\![\beta]\!]\sigma,\\
\mathcal{M}[\![\textbf{div } \alpha,\beta]\!]\sigma &= \lfloor \mathcal{M}[\![\alpha]\!]\sigma / \mathcal{M}[\![\beta]\!]\sigma \rfloor,\\
\mathcal{M}[\![\textbf{rem } \alpha,\beta]\!]\sigma &= \mathcal{M}[\![\alpha]\!]\sigma - \mathcal{M}[\![\beta]\!]\sigma(\lfloor \mathcal{M}[\![\alpha]\!]\sigma / \mathcal{M}[\![\beta]\!]\sigma \rfloor),\\
\mathcal{M}[\![\textbf{neg } \alpha]\!]\sigma &= -\mathcal{M}[\![\alpha]\!]\sigma,
\end{aligned}$$

The following two instructions are arithmetic shift left and arithmetic shift right. As the value domain contains arbitrary real numbers, rotate and logical shift instructions would have no meaning.

$$\begin{aligned}
\mathcal{M}[\![\textbf{shl } \alpha,\beta]\!]\sigma &= \mathcal{M}[\![\alpha]\!]\sigma \cdot 2^{\mathcal{M}[\![\beta]\!]\sigma},\\
\mathcal{M}[\![\textbf{shr } \alpha,\beta]\!]\sigma &= \mathcal{M}[\![\alpha]\!]\sigma \cdot 2^{-\mathcal{M}[\![\beta]\!]\sigma},
\end{aligned}$$

The equality comparison instruction is a simple instruction which accepts a pair of operands and yields 1 if their values match and 0 otherwise. Similarly, the less-than and less-than-or-equal instructions accept a pair of real numbers, compare them and return 0 or 1 accordingly. We do not explicitely support greater-than and greater-than-or-equal operators, as they can be constructed from other comparison operators and the logical not operator—which also requires its only argument to be a real number.

$$\begin{aligned}
\mathcal{M}[\![\textbf{eq } \alpha,\beta]\!]\sigma &= \begin{cases} 1, & \text{if } \mathcal{M}[\![\alpha]\!]\sigma = \mathcal{M}[\![\beta]\!]\sigma,\\ 0, & \text{otherwise,} \end{cases}\\
\mathcal{M}[\![\textbf{less } \alpha,\beta]\!]\sigma &= \begin{cases} 1, & \text{if } \mathcal{M}[\![\alpha]\!]\sigma < \mathcal{M}[\![\beta]\!]\sigma,\\ 0, & \text{otherwise,} \end{cases}\\
\mathcal{M}[\![\textbf{leq } \alpha,\beta]\!]\sigma &= \begin{cases} 1, & \text{if } \mathcal{M}[\![\alpha]\!]\sigma \le \mathcal{M}[\![\beta]\!]\sigma,\\ 0, & \text{otherwise,} \end{cases}\\
\mathcal{M}[\![\textbf{not } \alpha]\!]\sigma &= \begin{cases} 1, & \text{if } \mathcal{M}[\![\alpha]\!]\sigma = 0,\\ 0, & \text{otherwise,} \end{cases}
\end{aligned}$$

Lastly, the following logical operators perform the standard bitwise and, or and exclusive or operations; they operate only on integers.

$$\begin{aligned}
\mathcal{M}[\![\textbf{and } \alpha,\beta]\!]\sigma &= \mathcal{M}[\![\alpha]\!]\sigma \textbf{ and } \mathcal{M}[\![\beta]\!]\sigma,\\
\mathcal{M}[\![\textbf{xor } \alpha,\beta]\!]\sigma &= \mathcal{M}[\![\alpha]\!]\sigma \textbf{ xor } \mathcal{M}[\![\beta]\!]\sigma,\\
\mathcal{M}[\![\textbf{or } \alpha,\beta]\!]\sigma &= \mathcal{M}[\![\alpha]\!]\sigma \textbf{ or } \mathcal{M}[\![\beta]\!]\sigma,
\end{aligned}$$

Pointer dereference

The **deref** instruction is similar to arithmetic instructions in that it merely computes a value which is then bound to a node (i.e. it is a simple instruction).

Let $\iota(\sigma) = [\![\textbf{deref}\ \alpha]\!]$ and let $\psi = \mathcal{M}[\![\alpha]\!]\sigma$ be the value of the only operand. If $\psi \in \Psi$, then we can write $\psi = \omega m_1 \cdots m_n$ for some $\omega \in \Omega$, and $m_1 \cdots m_n \in M$. If either $\omega \notin \Omega$ or $v(\omega) = \bot$ then the instruction is malformed.

We introduce a function $m(x, m_1 \cdots m_n)$, which retrieves a member value of an array or a dictionary x given a sequence of member identifiers. The function is defined recursively as follows:

$$m(x, m_1 \cdots m_n) = \begin{cases} x, & \text{if } n = 0, \\ m(m_1(x), m_2 \cdots m_n), & \text{if } m_1(x) \neq \bot, \\ \bot, & \text{otherwise.} \end{cases}$$

The meaning of the instruction is then given as $\mathcal{M}[\![\textbf{deref}\ \alpha]\!]\sigma = m(v(\omega), m_1 \cdots m_n)$. In other words, the value of the topmost variable is retrieved and the member qualifiers are then applied.

Member access

An element of an array or a member of an object is accessed using the **member** instruction. The instruction receives two arguments—a pointer to the array or the object whose member is to be accessed, and either an index into the array or a name of a member. The instruction yields the pointer to the requested member (the instruction is a simple instruction).

Let $\iota = [\![\textbf{member}\ \alpha, \beta]\!]$ and $\mathcal{M}[\![\alpha]\!]\sigma = \psi$, where $\psi \in \Psi$. The formal meaning of the instruction is then defined as

$$\mathcal{M}[\![\textbf{member}\ \alpha, \beta]\!]\sigma = \begin{cases} \psi\mu_i, & \text{if } \mathcal{M}[\![\beta]\!]\sigma = i \in \mathbb{N}_0, \\ \psi\mu_z, & \text{if } \mathcal{M}[\![\beta]\!]\sigma = z \in \mathcal{S}, \\ \bot, & \text{otherwise.} \end{cases}$$

Pointer arithmetics

The adjustment operation occurs in languages of the C language family whenever an addition or subtraction operator is applied to a pointer and an integer. Such a pointer, if pointing to an element of an array, is then redirected to point to an element with an index appropriately adjusted.

In SIR, such an operation can be performed using the **adjust** instruction. Let $\iota = [\![\textbf{adjust}\ \alpha, \beta]\!]$, where $\mathcal{M}[\![\alpha]\!]\sigma = \psi\mu_i$, with $\psi \in \Psi$, and $\mathcal{M}[\![\beta]\!]\sigma = j$ for some $j \in \mathbb{Z}$. Then $\mathcal{M}[\![\textbf{adjust}\ \alpha, \beta]\!]\sigma = \psi\mu_{i+j}$.

The other form of pointer arithmetics—the retrieval of the distance between two pointers into the same array— which occurs in C when two pointers are subtracted from each other, is in SIR performed using the **dist** instruction.

If $\iota = [\![\textbf{dist}\ \alpha, \beta]\!]$, $\mathcal{M}[\![\alpha]\!]\sigma = \psi\mu_i$, and $\mathcal{M}[\![\beta]\!]\sigma = \psi\mu_j$, then $\mathcal{M}[\![\textbf{dist}\ \alpha, \beta]\!]\sigma = i - j$. Note that if the two pointers point into different arrays, the instruction is malformed.

Branch selection instruction

The branch selection instruction, the phony function, or simply the ϕ function often pops up in the context of a single static assigment form.[21] The instruction is used to select a value of one of several nodes, depending on which node was bound the last. Typically, the instruction is used after two control-flow branches join.

For instance, the program in Figure 1.3 branches based on the value of the parameter x. Depending on the branch taken, the function returns either the constant 1, or the value returned by the recursive call multiplied by x. The instruction choses one of the two nodes based on which one was executed the last. The **phi** instruction is special in that it requires all its arguements to be nodes identifiers, and only resolves the meaning of the selected one.

We define the meaning of the **phi** instruction $[\![\textbf{phi}\ \alpha_1, \alpha_2, \ldots, \alpha_n]\!]$, where $\alpha_i \in \langle \mathit{node} \rangle$ for all i as $\mathcal{M}[\![\textbf{phi}\ \alpha_1, \alpha_2, \ldots, \alpha_k]\!]\sigma = \phi(B, i, \{\alpha_1, \alpha_2, \ldots, \alpha_n\})$, where the function ϕ is defined as follows.

$$\phi(B, i, A) = \begin{cases} \bot, & \text{if } B = \varepsilon, \\ x, & \text{if } b_n = (i, \alpha, x), \text{ for some } \alpha \in A, \text{ and} \\ \phi(b_1 \cdots b_{n-1}, i, A), & \text{otherwise.} \end{cases}$$

As with other instructions, the selected value is then bound to the current node and the execution continues along one of the enabled edges. Note that the definition of the instruction is slightly different than what is typically encountered in literature, as SIR doesn't have a concept of a basic block, on which the definition is generally based.

Assignment instruction

The assignment instruction **assign** is the first non-simple instruction—we therefore define its semantics in full. It takes two operands, a pointer to a variable and the value to be assigned to that variable. The instruction causes a change in the v mapping; no change to node value binding is performed.

In formal terms, assume $\iota(\sigma) = [\![\textbf{assign}\ \alpha, \beta]\!]$. If $\mathcal{M}[\![\alpha]\!]\sigma \notin \Psi$, then the instruction is malformed. Otherwise, denote $\mathcal{M}[\![\alpha]\!]\sigma = \omega m_1 m_2 \cdots m_n$. Let $x = v(\omega)$ be the value of the variable pointed to by ω (again, if $v(\omega) = \bot$, the state is deadlocked). A new value x' will be assigned to the variable.

Formally, $\sigma \mapsto \sigma'$ for all $\sigma' = (c', v', B, i)$ such that $c' \in \operatorname{succ}(c, 0, \bot)$ and

$$v'(\omega') = \begin{cases} v(\omega'), & \text{if } \omega' \neq \omega, \\ x', \text{otherwise.} \end{cases}$$

To calculate the value of x', we define the function r recursively as follows.

$$r(x, m_1 m_2 \cdots m_n, y) = \begin{cases} y, & \text{if } n = 0, \\ x[m_1 / r(m_1(x), m_2 \cdots m_n, y)], & \text{otherwise.} \end{cases}$$

We then set $x' = r(x, m_1 m_2 \cdots m_n, \mathcal{M}[\![\beta]\!]\sigma)$.

Control flow instructions

Since branching in SIR is performed using conditional edges, there are no intraprocedural branching instructions. The two instructions that affect the execution flow of the program are the subroutine call and subroutine exit instructions.

The **call** instruction is used to trasfer control to the entry point of the specified subroutine. Let $\iota(\sigma) = [\![\textbf{call}\ \alpha, \alpha_1, \alpha_2, \ldots, \alpha_n]\!]$. Furthermore let $\lambda_f = \mathcal{M}[\![\alpha]\!]\sigma$ be the value of the first operand of the instruction and f the subroutine referred to by this value (if the meaning of the operand is not a subroutine, the instruction is malformed). Note that the first operand need not be a $\langle \textit{subroutine} \rangle$, it may also be a $\langle \textit{var} \rangle$ or $\langle \textit{node} \rangle$ whose value was assigned a subroutine value. Then $\sigma \mapsto \sigma'$ for all $\sigma' = (c', v', B, i+1)$ with the following properties.

Firstly, a new execution frame is created with the identifier i. The execution frame is pushed onto the execution stack, $c' = c(i, f, n_0)$, where n_0 is the entry node of the subroutine f.

Secondly, the values of the rest of the operands are copied to the parameters of the subroutine. Let $x = x_1 x_2 \cdots x_n \in \mathcal{D}^*$ be the sequence values of the operands, i.e. $x_i = \mathcal{M}[\![\alpha_i]\!]\sigma$ for all i. Let $p = p_1 \cdots p_n \in \langle \textit{var} \rangle^*$ be the sequence of parameters of the function f. If the two sequences are of a different lengths, the instruction is malformed. We set $v' = \mathrm{copy}(v, i, p, x)$, where the function copy returns a variable mapping such, that all local variables names in p are assigned with the corresponding values in x in the execution frame i. The function is defined as follows.

$$\mathrm{copy}(v, i, p, x)(\omega) = \begin{cases} v(\omega), & \text{if } p = \varepsilon, \\ x_0, & \text{if } \omega = \omega_{i,p_0}, \text{ and} \\ \mathrm{copy}(v, i, p_2 \cdots p_n, x_2 \cdots p_n), & \text{otherwise.} \end{cases}$$

The **exit** instruction removes an execution frame from the execution stack and transfers the two values passed as operands back to the caller—the exit index and optionally a return value. Let $\iota(\sigma) = [\![\textbf{exit}\ \alpha, \beta]\!]$. Let $e = \mathcal{M}[\![\alpha]\!]\sigma$ and $r = \mathcal{M}[\![\beta]\!]\sigma$. If β is missing, let $r = \bot$.

Then $\sigma \mapsto \sigma'$ for all $\sigma' = (c', v', B', i)$ such that $c' = c_1 c_2 \cdots c'_{k-1}$, where $c'_{k-1} \in \mathrm{succ}(c_{k-1}, e, r)$.

All values stored in node instances and local variables associated with the removed frame are removed from variable mapping and the node binding sequence. We therefore set v' to be the mapping such that

$$v'(\omega) = \begin{cases} \bot, & \text{if } \omega = \omega_{i_k,n} \text{ for some } n \in \langle \textit{var} \rangle, \\ v(\omega), & \text{otherwise.} \end{cases}$$

All node instances in the the removed frame are unbound and the return value is bound to the calling node, i.e. $B' = \mathrm{bind}(\mathrm{unbind}'(B, i), n_{k-1}, r)$.

Note that the next frame number i remains intact—frame numbers are not reused. This allows us to detect attempts at accessing dangling pointers.

1.3 Multithreaded programs

We do not consider multithreaded programs in this thesis. However, the formal semantics of SIR can be extended to allow multithreading in a straightforward manner.

Let there be a countable set of thread identifiers $\mathcal{T}$. Then we define a set of multithreaded variable identities Ω^T as containing the symbol ω_x for each $x \in \langle globalvar \rangle$ and $\omega_{t,i,x}$ for each $t \in \mathcal{T}$, $i \in \mathbb{N}_0$ and $x \in \langle localvar \rangle$.

Then we define a multithreaded program state to be a tuple $\sigma = (c, v, n, i)$, where $c \colon \mathcal{T} \to (\mathbb{N}_0 \times N)^*$ is a partial function mapping thread identifiers to the corresponding sequence of execution frames, $v \colon \mathcal{T} \times \Omega^T \to \mathcal{D}$ is a partial mapping of variable identites to their corresponding values, $n \colon \mathcal{T} \times \mathbb{N}_0 \times N \to \mathcal{D}$ a partial mapping of node instances to their bound value, and $i \in \mathbb{N}_0$ to be the lowest unused frame identifier.

The multithreaded semantics would be constructed from standard semantics in an obvious way. The resulting semantics would allow for thread interleaving. A new operators can be defined to allow for thread creation and joining. Synchronization can be achieved similarly—by defining lock and unlock operators.

Bear in mind that, syntactically, program representation would not need to change.

1.4 JSON encoding of SIR program units

To represent SIR program units, we chose to take advantage of *JavaScrupt Object Notation* (JSON, [8]), which is a simple data-transfer format based on JavaScript (correctly referred to as EcmaScript) syntax. We say that SIR program units are encoded or serialized to JSON.

Note that originally, Stanse used XML as the data interchange format. While XML schema is easier to amend with new extensions, the nodes forming its data hierarchy are complex objects and are difficult to manipulate. On the other hand, JSON offers very simple data nodes—in fact, in many interpreted programming languages, JSON-encoded objects often map directly to native objects (for example, JSON arrays map directly to Python lists and JSON objects map to Python dictionaries).

The JSON specification defines the syntax of primitive values (null value, logical values, numbers and strings), and the recursive syntax of arrays—ordered sequences of simpler values—and objects—unordered string-value pairs. The null value is represented by the keyword `null`. The two logical values are represented by keywords `false` and `true`. Numbers are encoded as a sequence of decimal digits, possibly containing a decimal point and a minus prefix. A string is an arbitrary sequence of characters except the character `"` and `\`, enclosed in quotes. Special characters can be included in the string using escape sequences. Arrays are formed by enclosing a comma-delimited list of values in square brackets. Key-value pairs are constructed by delimiting the key and the corresponding value by a colon; JSON objects (which we will sometimes refer to as dictionaries) are then produced by enclosing a comma-delimited list of key-value pairs in braces.

We start our description of the JSON encoding of SIR program units from the bottom up, describing the encoding of operands, control-flow nodes, subroutines and finally program units. It should be noted that syntactic constants in SIR (productions of the ⟨*const*⟩ nonterminal), are directly modelled after the JSON data model (only logical values were left out). Names of subroutines and variables are stored as strings. As we will later see, control-flow nodes are stored in an array; each node can therefore be referred to by its position in that array. As such, we serialize node identifiers as simple integers.

Operands are serialized as two-element arrays. The first element describes the type of the operand; it is a string whose value is either `"const"`, `"func"`, `"var"`, `"varptr"`, or `"node"`. The second element contains the value of the operand. For `"const"`, it is an arbitrary JSON value that contains neither `true` nor `false`. For `"node"`, the value must be an integer. For the rest of the node types, the value is a string.

Edges in SIR subroutines are labelled by an exit index (an integer) and a condition (an arbitrary constant). The serialization of an outgoing edge therefore takes form of an array containing in sequence the identifier of the target node, the exit index, and the condition.

As each SIR node is labelled by an instruction and has a set of outgoing edges, SIR nodes are serialized into four-element arrays. The first element is a string containing the opcode of the instruction. The opcode is followed by a array of outgoing edges, and an array of operands.

Optionally, the fourth element containans an array of tag identifiers. Tags are subroutine-local values which carry additional metadata, e.g. source code positions. A control-flow node may have one or more of these tags attached. We will discuss tags in detain in the next section—for now let us note that all tags for the given subroutine are stored in an array; the nodes refer to them by their position in that array.

An encoding of a SIR subroutine is then a JSON object consisting primarily of the `"nodes"` key, which contains an array of encoded SIR nodes, and the `"entry"` key, containing an integer—the index of the entry node in the `"nodes"` array. Additionally, the list of local variable names is stored as an array of strings (the order is not important) in the `"locals"` key. The parameter list is stored in the `"params"` key. The names stored in the latter array must all be also contained in the `"locals"` array. Note that for the parameter list, the order matters. Lastly, the encoding of a subroutine contains the array of tags, stored in a key of the same name (`"tags"`).

A SIR program unit is serialized as a JSON object containing the following keys. Most importantly, the key `cfgs` contains the dictionary mapping SIR subroutine names to their corresponding SIR subroutines. The subroutine names are simple strings, the SIR subroutines are serialized as described above. The `globals` key contains a mappings from the names of global variables to their initial values.

The SIR program unit also carries along a list of name aliases—mappings from the machine-readable names of subroutines and variables to human-readable ones. Typically, a source language with a support for namespaces or function name overloading will produce what is called a mangled name for a subroutines (or a variable). The list of aliases allows the user to use the friendly names instead. Aliases are represented as a

JSON object; keys of the object are the machine-readable strings and map to JSON arrays of human-readable strings. The object is stored a the `"aliases"` key of the program unit object.

The `filenames` key contains and array of source file names encoded as JSON strings. The array serves as a file name repository and is referenced by source range tags.

Finally, to support dynamic binding, the C++ to SIR translator produces two keys that are stored in the program unit object—`"_vfn_map"` and `"_vfn_params_counts"`. For more information about virtual dispatch, see Section 2.9. The object `"_vfn_map"` maps the names of the dispatch subroutines to the list of possible main subroutines. The other object, `"_vfn_params_counts"`, maps the names of the dispatch subroutines to the number of parameters that these receive. The two arrays are used to generate the dispatch subroutines as described in the aforementioned Section 2.9. Storing the dynamic binding information in this manner allows it to be easily merged when two SIR program units are linked together.

TODO: an example

1.5 Tagging

While SIR units, as described above, offer sufficient functionality to perform static analysis (and simulation), the result of such analyses would not be very useful, if there was no way to map SIR nodes back to statements in the original program. While we could simply add an additional labelling to SIR nodes, we decided to add a more generic mechanism for adding metadata to SIR units.

To every SIR node, an arbitrary number of *tags* can be attached. We currently define only one type of tag, however, in the future there could be several types of tags, each associated with different kind of data. In JSON, tags are stored as JSON arrays, where the first element is always a string determining the type of the tag. The format of the rest of the sequence is dependent on the type.

A *source range* tag associates a SIR node with a character range in the source code whose behavior the node models. We represent a location in a file as a pair (l, c), where l is the line number and c is the column number. The line number of the first line is one. Similarly, the column number of the first column is one. A source code range is a pair of locations (s, e), where s represents the start location of the range and accordingly e represents the end location. The start location always precedes the end location, i.e. $s \leq e$, where the ordering is lexicographic. In case $s = e$, the range encloses no characters.

The names of files are not stored in tags directly. Instead, there a single list of filenames. The source range tags merely refer to the appropriate index in the repository.

In the JSON encoding of source range tags, five numbers are attached to the array following the `source_range` tag type. The numbers are in order (f, s_l, s_c, e_l, e_c), where f is the index into the filename repository and $((s_l, s_c), (e_l, e_c))$ is the source range. For example, the JSON-encoded tag `["source_range", 1, 64, 5, 64, 12]` encodes a range spanning seven characters, all on a single line.

1.6 Merging of SIR units

Many programming languages are designed to produce programs in fragments called program units, objects files or the like. Accordingly, a SIR program can be fragmented into several SIR program units. Before static analysis can occur, the fragments must be merged—or linked—together so that all references to subroutines and global variables are satisfied. As a part of this thesis we wrote a tool which perfoms unit merging; it is described in detail in Chapter 5.

When two units are merged, one of the units is called the source and the other is called the target. The various sections of the JSON object representing the SIR program unit are merged in the following fashion.

- All subroutines from the source program unit are copied into the target program unit. It is not an error if the subroutine already exist in the target unit—the subroutine is simply replaced. In C and especially in C++, functions tend to be generated multiple times across multiple units. This includes inline functions in C and additionally function template instantiations in C++.

- The list of global variables and their initial values are merged in the same manner—in the case of a conflict the old initial value is discarded.

- The alias mapping in the target unit is updated with the alias mapping in the source. If aliases are assigned to a machine-readable name in both the source and the target, the resulting set attached to that name is the union of the two original sets.

- The union of the file name repository is created. Note that source range tags must be updated to reflect the new positions in the repository.

- Finally, the `"_vfn_map"` and `"_vfn_params_counts"` are merged. The latter is merged in a straight-forward manner. The former is merged in the same way as the alias mapping—to each dispatch subroutine name the union of associated main subroutine names from the source and the target units is assigned.

Note that while merging is associative operation, it is not commutative in general. For most SIR programs however, multiply defined subroutines and global variables will be the same accross all program units, making the merging commutative at least in this case.

Chapter 2

Modelling of C++ features

Before we begin the description of the translation process of C++ statements (and therefore C++ function bodies) to SIR subroutines, several key considerations must be made. C++ is a complicated language, designed to be written by humans. On the other hand, SIR is a machine-generated and machine-consumed language—it does not offer the same range of features, which make C++ both concise and easy to reason about. As such, some C++ features do not map to SIR in a straight-forward manner.

In this chapter, we look at the multitude of C++ language specifics and show how various features of the language can be modelled in SIR.

2.1 Naming of program entities

In SIR, program entities (subroutines and global and static local variables) have unique names so that instructions can unambiguously refer to them. However, the C++ language source code objects (which include variables, namespaces and classes) need not have unique names. Two distinct objects may have the same name if, for example,

- they are declared in different declaration contexts (e.g. they are members of different namespaces or classes),
- they are defined in different translation units and at least one of them does not have external linkage (e.g. it is declared as static or it is local to a function), or
- they are functions and differ in the number or types of their parameters (in such a situation, the functions are said to be overloaded).

Therefore, a unique name must be generated for each C++ object in order for it to be represented in SIR. This section describes how this unique name is formed.

The first matter to consider is one of compatibility. It is quite common for C++ programs to call functions that are written in programming languages other than C++. Most often this language will be C—in fact, a large part of the standard library consists of functions with C linkage. As such, linking translation units written in C and in C++

is quite common. Ensuring that SIR units translated from C and from C++ languages can be merged as easily as they are linked together by object file linkers requires that the naming scheme for subroutines of our C++ programs be compatible with a naming scheme that a potential C to SIR translator would use.

As C supports neither namespaces, nor classes, nor function overloading, it is reasonable to conclude that the plain name of the function would serve as the name for the corresponding SIR subroutine. This conclusion is supported by the fact that popular C compilers produce unmangled, plain name of the function as the name of the corresponding symbol in the binary object files. In order to be compatible with the name-mangling schemes employed by popular C++ compilers, our C++ to SIR translation tool therefore yields the plain function name for functions with C linkage.

As for the functions with C++ linkage, a mangling scheme similar to the one defined by Itanium ABI[19], sometime referred to as gcc3 mangling, is used. The use of a standardized scheme enables the use of existing tools to decode the function name.

Functions with internal or no linkage generally produce no names in object files, since after the compilation step they are no longer necessary. However, for the purposes of static analysis, the names are in fact required. Note, that these functions can be defined with exactly the same signature, yet different body, in multiple translation units. Therefore, names of these functions must include a part specific to a translation unit to which they belong. We call this unique name the *unit identifier*. By default, the name used as the unit identifier is the string `"__unique"`, but it can be changed through the parser's command line. In practice, the identifier may be derived for example from the hash of the path to the main file of the translation unit,.

While the method of mangling names of entities with external linkage (i.e. those whose name may not contain the unit identifier) is well documented, there is—to our knowledge—no standad manner to mangle names of other types of entities. We therefore extend the Itanium mangling scheme to support them.

The rules governing the name mangling are rather complex—we will refrain from explaining them here in detail. Refer to the Itanium ABI specification[19] for thorough explanation. We will however introduce basic principles of this mangling scheme so as to provide the basis for our extension. A mangled name consists of three parts: the prefix `_Z`, the encoded name of the entity, and for functions the type of their parameters. The prefix is used to distinguish mangled identifiers from unmangled ones. Note that the C++ standard reserves all names beginning with an underscore followed by an uppercase letter for use by the implementation[13], therefore, no unmangled name may legally begin with the `_Z` prefix.

The encoded name of an entity is a sequence of names which together form the fully qualified name of the entity. Each name in the sequence is encoded as a decimal number—the length of the name—followed by the name. For example, the name `ns::foo` would be encoded as `2ns3foo`.

We will use the same method to encode the unit identifier. For entities that do not have external linkage, or those that have external linkage but are enclosed in an

```
namespace ns1 {
    int f() { // _ZN3ns11fEv
        static int var; // _S8__unique_ZZN3ns11fEvE3var
    }
    void f(int i); // _ZN3ns11fEv

    extern "C" g(int); // g
    class c {
        static int f(); // _ZN3ns11c1fEv
    };
}
static void f() {} // _S8__unique_Z1fv
namespace { void f() {} } // _S8__unique_ZN12_GLOBAL__N_11fEv
```

Figure 2.1: A C++ program containing several named entities, with their mangled names noted in comments.

anonymous namespace,[1] the string derived by mangling the name as if the entity had external linkage is prefixed with the string `_S` followed by the encoded unit identifier. For example, a variable named `foo` would normally be encoded as `_Z3foo`. If such a variable were made static in a translation unit with the identifier `unit`, its mangled name would be changed to `_S4unit_Z3foo`. This simple scheme allows the prefix to be manually stripped by the user and the rest of the name passed to a demangling tool.

Recently, clang developers added an external interface enabling their mangler to be used outside their code generator. We have opted to use it, as it ensures that our translator makes immediate use of any bug fixes applied to the mangler. However, authors of clang do not guarantee that the mangling scheme will remain fixed, in fact they actively warn about the possibility of it changing. As such, our mangling scheme may change unexpectedly; in such a case the entire code base would have to be reparsed with our translator.

Figure 2.1 gives an example of how mangled names of C++ entities are constructed. Note in particular that the static prefix on member functions does not affect the mangled name (the function `ns1::c::f` has external linkage).

2.2 Fundamental types

In C++, the types `char`, `short`, `int` and `long`, together with their signed and unsigned variants, the `wchar_t` type, the floating point types and the special type `bool` are called *fundamental types*. Fundamental types can be used to form more complex types (pointers, references, structures, arrays, etc.).

The fundamental types differ from each other by the values they can hold. Furthermore, the behavior of basic arithmetic operators differ based on the type of its operands. In particular, the value of a given type is bounded by the minimum and maximum values

[1] Anonymous namespaces behave as named namespaces with a unique name.

of the type. If an expression yields a value that exceeds these bound, a condition known as overflow occurs and a value from within the bounds is chosen instead. For unsigned types, all basic mathematical operators are required to work as if performing modulo arithmetic. For signed types, the value of an expression in undefined in the case of an overflow.

The fundamental type of the SIR execution model is currently an arbitrary precision real number. As such, we naturally model all the integral types as such. The behavior of arithmetic operators, however, does not match that of C++ operators. We currently left this imprecision unsolved and do not generate overflow conditions. If we were to add overflow handling later, we would follow each call to a SIR arithmetic operator with a call to either the modulo operator (for unsigned types), or some sort of a clamp operator (for signed types).

The values of `bool`—`false` and `true`—are treated as integers 0 and 1 respectively. For floating point types, no special treatment is necessary; the C++ language standard does not define the precise arithmetic rules for floats.

2.3 References

The C++ reference types have no direct counterpart in SIR. In many aspects, references behave as pointers which are automatically dereferenced whenever used.[2] They must be bound to an object when declared and cannot be rebound later.

In some places references exhibit a somewhat perculiar behavior. First, there are interactions between references and template deduction algorithms—fortunately, Clang already provides us with instantiated templates, allowing us to ignore this difficulty. Second, binding a temporary object to a local variable of reference type will cause the object's lifetime to be extended to match the lifetime of the reference (whereas storing an address of a temporary in a pointer variable will result in a dangling pointer).

We take this into consideration during the generation of a SIR program unit. In all other cases, we treat references as pointers. In particular, whenever an argument is passed by reference, a pointer to the object is passed by value in the resulting SIR program.

2.4 String literals

C++ string literals are arrays of characters. They do not, however, behave as constants in that it is not only possible to form a pointer to its elements, such use of string literals is quite common. In fact, most string literals are used in contexts where they immediately decay to pointers to their first element.

The above observation precludes us from treating string literals as simple SIR string or SIR array constants. Instead, for each literal a new static variable is created and

[2] Note that this statement is better not repeated in front of a C++ language lawyer.

statically initialized to have the value of the character array. This way all string literals are transformed to variables and are treated as such by the translator.

The aforementioned global variables must be assigned a unique name. While it would be possible to mangle the contents of the string to create this unique name, we failed to find any standard mangling scheme. As such, we decided to use a simple scheme, in which the strings are assigned a name of the form `_Y<n>`, where `n` is a number, which is incremented on every occurence of a string literal. The unit indentifier is then mangled into the name as described in Section 2.1.

2.5 Unions

Unions are currently treated as structs. While the behavior of correct programs will not be affected—a C++ program may only access the member of the union which was written to last[3]—no checker will be able to detect incorrect use of unions. We believe that the best way to convey the nature of a complex object is to pass the information in the object's type. We expect that typing information will be added to SIR later, but it is currently out of the scope of this thesis.

2.6 Raw memory

The C++ language (and the C language from which it was derived) is very low-level language, in that it allows one to directly access and modify the memory underlying its otherwise well-abstracted high-level objects. For example, it is quite common to initialize arrays of scalar objects (i.e. arrays of integers) not with a loop, but with a call to `memset`, a C library function which sets the value of each byte in the given chunk of memory to a given value.

Unfortunately, without knowledge of object's layout (an extremely platform-dependant property), it is in most cases impossible to model the behavior of the program precisely. Even with that knowledge, the (untyped) SIR model of a C++ program would have to be very low-level, to a point where it would only contain a single global array of bytes. Though such a model would allow precise simulation, it would hinder any reasonable attempts at static analysis.

As such, we have decided to sacrifice accuracy in favor of retaining the high level of abstraction in the model. This means that all casts between unrelated pointer types are ignored—cast expressions have the value of the corresponding castee. Furthermore, the arguments passed to `memcpy`, `memcmp`, `memmove`, and `memset` will not convey enough information for them to be simulated correctly. The problem also affects reading and writing files. While non-portable, it is a standard practice to pass pointers to whole structures as arguments to `fread` and `fwrite`. Without knowledge of the layout of the

[3] In particular, the behavior of programs which use unions to reinterpret memory locations is undefined.

objects passed to these functions, correctly simulating the behavior of programs which call these function is not possible.

Note that adding typing information to SIR and modifying our C++ translator to emit it would open up the possibility of simulating and statically checking code that performs calls to afformentioned functions. With some work, even code which performs access to objects through pointers to type other than the dynamic type of the object might be simulated. (Recall that some forms of such access are explicitely allowed by the C++ standard and do not invoke undefined behavior. For example, access through `char` or `unsigned char` pointer, through signed or unsigned version of the pointer, and a few other types of access are valid. Refer to Section 3.10 paragraph 15 of the standard[14]).

2.7 Argument passing

When the SIR **call** instruction is used, it is provided with a sequence of operands whose values are to be passed to the callee. The values are simply copied into variables local to the called subroutine.

This mode of argument passing is often called “by-value” and it is a mode native to C++ (assuming that all instances of references are treated as pointers as described in Section 2.3). The C++ argument passing is therefore directly supported in SIR. However, in C++, passing objects of a class type may cause a non-trivial copy constructor to be called. The copy construtor in question receives a reference to the original object (i.e. to the object being passed).

There are two ways to deal with this kind of copy-passing in SIR. Either a pointer to the structure gets passed to the callee, which then constructs a its local copy, or the caller copies the object and passes a pointer to the copy. In both cases only a pointer to the structure gets passed; this is an inevitable consequence of C++ objects having an immutable identity—once a C++ object gets constructed, its address will never change.

In case of the former alternative, the caller would have no knowledge of the new object. Therefore, the callee would also have to be responsible for the object’s destruction. On the other hand, either of the two functions can be responsible for argument destruction if the latter alternative was used. Since C++ allows functions to have variable number of arguments, we are forced to relegate the responsibility for argument destruction to the caller, and as such the caller must also be made responsible for the copying.

Note that in order to make the passing of structures consistent, we have decided that even structures with trivial copy-constructors will be passed by pointer, even though they could easily be passed by value without any side effects.

Returning structures from functions must also be considered. While scalar types can be returned directly (using the standard SIR mechanism), structures cannot (as the returned value must either retain its identity, or a copy must be performed). We have settled on the solution also employed by compilers—a pointer to the variable where the returned object is to be instantiated is passed as an argument to the callee. An object is constructed in that variable by the callee. After the function returns, the caller is responsible for the destruction of the object. Note that the number of return values is always known in advance (either none or one)—the caller knows if and how the return value is to be destroyed.

SIR does not offer any object-oriented abstractions. In C++, non-static member functions, when called, carry along a special value, the so-called `this` pointer. The pointer allows access to the object, in whose context the function is executed. On the other hand, all SIR subroutines behave more akin to C++ free functions. They do not nest, and they can only access global variables and their (explicit) parameters. Our translator therefore transforms the signature of non-static member functions by adding an extra parameter (named `this`) to the beginning of the parameter list.

2.8 Variadic functions

All SIR subroutines have a fixed number of parameters, whereas C++ functions can be passed variable number of parameters (this is indicated in the function's prototype by the ellipsis at the end of the parameter list; such functions are called variadic and the arguments without an associated parameter are called optional). Optional arguments are then retrieved through special library calls (`va_start`, `va_arg` and `va_end`). Our translator currently does not handle variadic functions, as we haven't yet settled on an appropriate way to deal with them. For now, the translator emits the standard non-variadic SIR subroutine, but all call instructions targeting this subroutine are provided with all of the arguments. Having a call with optional arguments in a C++ program will therefore make the SIR program malformed. (Although Stanse will ignore the problem.)

We have considered several approaches to modelling variadic functions. The straightforward solution would be to extend SIR to support them directly. In that case, the number of operands to a call instruction would not be required to match the number of parameters of the called subroutine. Either an additional instruction or a call to a library function would be used to retrieve the optional areguments.

Alternatively, we can consider variadic functions as having one additional parameter. This parameter would be explicitly indicated in the SIR subroutine signature and would be passed a "magic" list of optional arguments. Elements of the list would be retrieved with a call to an operator.

We personally incline most to the former solution—the behavior would be consistent with the current implementation. The translator would only have to recognize calls to `va_start` and others, and emit the appropriate instructions (or calls to the special subroutines).

2.9 Virtual dispatch

Virtual dispatch (sometimes called dynamic dispatch) is the ability of programs to call functions based on a dynamic type of one or more of their arguments (in the case of C++ the dispatch is always made based on the dynamic type of the hidden `this` argument). The feature is sometimes called *late binding* as the association between the caller and the callee occurs only immediately before the call is executed. There is no direct equivalent of this feature in SIR.

In practice, C++ compilers achieve late binding by constructing a so-called *virtual table* for each dynamic class (the precise definition is rather involved; it is sufficient to know that all classes containing virtual functions are dynamic). The table contains pointers to all virtual functions of that class in the order in which they were defined. Each object of the class carries a pointer to this table. In a way, the dynamic type of the object is encoded in that pointer. In order to call a virtual function, the program retrieves the pointer to the virtual table and performs a call through the appropriate entry.

There are certain complexities associated with virtual table dispatch. For example, if a class has multiple base classes, any call to a virtual function inherited from the second or later base involves pointer fixup—`this` pointer must be adjusted to point to the correct base subobject. This can be done by thunking—the virtual table entry doesn't point directly to the function to be executed, but rather to a *thunk*, which adjusts `this` pointer and forwards the call (thunking is used for example by g++ compiler).

While we could definitely model virtual dispatch in this manner, we feel that such a data-driven approach would not be handled well by static checkers. It would be difficult to determine the set of functions that can be called from a given call site (or that set would consist of all functions in the program, filtered only by function's signature). We therefore instead consider a more control-driven approach in which the possible execution paths are more reasonably exposed.

We translate each virtual function into two SIR subroutines, called the *main* and the *dispatch* subroutines. The main subroutine is generated in the same fashion as a subroutine for any non-virtual function. The dispatch logic is contained in the dispatch subroutine. The dispatch subroutine examines the `this` parameter, and based on its dynamic type it then calls the appropriate main subroutine.

Of course, the set of functions which override the given function cannot be determined by looking at a single translation unit—different execution paths of the dispatch subroutine can be contributed from different units. We therefore generate the dispatch subroutines only after all program units have been merged. The C++ to SIR translator only emits the mapping between the dispatch subroutine names and the names of possible callees. See Sections 1.4 and 1.6 for more detains on how the mapping is represented and how the program units are merged.

TODO: an example—graphical if possible—of how both mergin and the dispatch are done in practice.

2.10 Dynamic allocation

The functions `malloc` and `free` are used to allocate memory in the C language. Any calls to these functions can easily be modelled by calls to the appropriate SIR subroutines.

However, the C++ language brings additional—exception-safe and type-safe—operators `new` and `delete`. The former operator performs a call to an allocation function (called `operator new`) and in the returned memory it then constructs the new object (i.e. calls the type's constructor). If the construction fails, the memory is safely released through the call to `operator delete`.

Operator `delete` is conversly used to dispose of an object previously allocated using the `new` operator. The execution of the operator involves the call to the object's destructor (possibly with virtual dispatch) and a call to the deallocation function (whose identity may depend on the dynamic type of the object).

Note that the `new` operator may optionally receive arguments that are passed to the allocation function. For example, the C++ code `new(p) int`, where `p` is a pointer, calls the allocation function with signature `void * operator new(size_t size, void * arg)`. The operator `new` with arguments is sometimes referred to as *placement new*.

In addition, the C++ language supports array forms of the two operators, `new[]` and `delete[]`. After allocating memory, the `new[]` operator constructs all of the objects in the array, ensuring that if an exception is thrown, all objects that were already constructed (and only those objects) are released. Again, the `new[]` operator may receive optional arguments.

We currently only model the non-array `new` operator directly. We emit calls to the special subroutines called `__sir__cpp__new_array`, `__sir__cpp__delete`, and its array counterpart `__sir__cpp__delete_array` to represent operators `new[]`, `delete`, and `delete[]` respectively.

The `__sir__cpp__new_array` subroutine receives the following arguments:

1. the identity of the subroutine representing the allocation function,
2. the size of the type to be allocated,
3. the identity of the subroutine representing the constructor of the constructed type,
4. the identity of the subroutine representing the deallocation function (which is used in case the construction fails),
5. the integer 1 or 0, indicating whether the operator is to value-initialize the contents (i.e. whether the invocation of the operator ended with a pair of empty parentheses), and
6. zero or more arguments to the allocation function.

The two special subroutines that are used to destroy and release the memory receive a pointer to the object to be destroyed and the identity of the deallocation subroutine.

Figure 2.2 shows how a simple use of `new` and `delete` gets translated to SIR. Again, note that while the former is modelled directly, the latter is modelled by a subroutine call. It is our goal to eventually represent all allocation scenarios directly, removing the need for the three special functions.

```
$1:  call operator new
     call cls::cls, $1 | \xrightarrow{1} $5
     assign p, $1
     call __sir__cpp__delete, p

$5:  call operator delete, $1
```

```
p = new cls;
delete p;
```

Figure 2.2: The SIR instructions generated for the `new` and `delete` operators.

2.11 Exceptions

As many other modern languages do, C++ supports the concept of exceptions. Exceptions allow the developer to free their code of error handling issues and concentrate on the gist of the algorithm. Only once an error codition occurs, an exception object is created and the execution stack is unwound in a search for the appropriate exception handler. During this unwinding, objects with automatic storage duration are destroyed (notably their destructors are called), thus allowing the necessary cleanup (i.e. freeing of memory and other resources) to be performed. Once an execution frame with the exception handler—one that matches the type of the exception object—is found, the execution continues from that handler.

As SIR strives to be as uncomplicated as possible, it does not natively support exceptions. We therefore model exceptions and exception handling indirectly. There are several interesting points to note. One, there is a special exception object created for the purposes of communicating the error data. The object must exist during unwinding and during the execution of the exception handler—only then it can be freed. Furthermore, it must be possible to query the object about its type, in order to determine whether any given handler is able to process the exception.

Traditionally, compilers create the exception object on the stack and then call registered handler in the context of the function throwing the exception. The runtime then crawls execution frames in search of a compatible handler. Once the appropriate handler is found and executed, the exception object is destroyed and the execution is abruptly transferred to the statement following the exception handler. Handlers are usually registered at runtime at the beginning of execution of their containing functions (a method employed in 32-bit Windows systems), or statically by the compiler. In the latter case, the generated tables are walked only after an exception is thrown. Note that exception handlers are created not only by explicit catch statements, but are also generated for functions that contain automatic objects with non-trivial destructors.

Unfortunately, relying on registration records would make it very difficult for any static checker to analyze the exception control flow. We therefore chose to model exceptions in a more checker-friendly manner. As SIR allows subroutines to have multiple exit points, and allows the caller to detect the exact exit point the callee has taken, we decided to separate exit points for normal and exception flow. Normal control thus exits a subroutine through exit point zero; exception paths are directed to exit point one.

Whenever a caller detects that a call returned through exit point one, it then proceeds to destroy all objects in its execution frame and immediately exits afterwards—also through exit point one. This way, the information about an existence of a thrown exception object is propagated through the execution stack. The propagation is stopped either when the execution stack is empty, or when an exception handler willing to process the exception is found. Once the handler successfully finishes and releases the exception object, the handler transfers control to the statement immediately following the enclosing try block. The execution then continues on a non-exception path.

Originally, we wanted the exception object to be propagated via return values. Unfortunately, we found it difficult to model all the features of C++ exception handling accurately. Notably, a C++ code is allowed to issue an empty `throw` statement, indicating that a caught exception is to be rethrown. This rethrow statement, however, need not be inside a catch block, it can be executed from a subroutine. At that point, the exception object is unavailable and therefore cannot be returned.

We have therefore decided to model the exception object as residing in a special thread-local storage, which we hence-forth call an *exception object store*. Having a store where exception objects can be freely constructed mirrors the ability of compilers to store exception object on a store that survives stack unwinding. Note that multiple exception objects may be active at the same time; an exception may be thrown in the context of a catch statement while the original exception object is still live. As such, in our model, exception objects are allocated from the store and freed when they're no longer useful.

The exception object store additionally maintains a pointer to the exception object that was allocated last. We call this particular exception object the *current exception object*. The exception object store maintains an additional flag for the current exception object—it remembers whether the object is in a thrown state.

We define the following special functions which maintain the exception object store.

- New objects are allocated using the `__sir_cpp_exc_alloc` function. The function expects a single argument—a pointer to the typeinfo object representing the dynamic type of the exception. This typeinfo object is associated with newly-created object and is later used by `__sir_cpp_exc_catch` to decide whether a handler is entitled to handling the exception. The operator returns the pointer to the new storage for the exception object. The allocation never fails (the stack-based allocation of exception objects performed by compilers may fail due to stack overflow; that however rarely happens and cannot be reliably detected).

- Exception objects are freed using the `__sir_cpp_exc_free` function. A pointer to an exception object is expected as an argument. The state of the object determines the action performed. New objects are simply removed from the store. Thrown objects are not acted upon—they must be preserved until they are caught. Caught objects are first destroyed (which possibly involves calling the object's destructor), then removed from the store. We leave the details of retrieving the correct destructor deliberately vague. Trying to model the retrieval of the destructor directly would introduce unnecessary complexity while bringing little gain.

```
try {
    foo();
}
catch (cls const &) {
    bar();
}
```

```
$1:  call foo |1 → $0

$2:  call __sir_cpp_exc_current
$3:  call __sir_cpp_exc_catch, $2, ti_cls |0 →
     call bar
     call __sir_cpp_exc_free, $2
```

Figure 2.3: The SIR code layout for throw and try/catch blocks.

```
throw cls(42);
```

```
$1:  call __sir_cpp_exc_alloc, ti_cls
     call cls::cls, $1, 42 |1 → $4
     call __sir_cpp_exc_throw, $1

$4:  call __sir_cpp_exc_free, $1
```

Figure 2.4: The SIR code layout for throw and try/catch blocks.

- `__sir_cpp_exc_current` retrieves a pointer to the current exception object from the store.
- An exception object in a new or caught state may be thrown (i.e. its state can be changed to thrown) using the `__sir_cpp_exc_throw` function. Only one object may be thrown at one time. The operator kills the execution if another object is already thrown.
- A thrown exception object can be matched against a type and marked a caught using the `__sir_cpp_exc_catch` function, which accepts as an argument the pointer to the exception object. Optionally, a pointer to a typeinfo object may be passed as a second argument. In the latter case, the object is caught only if it matches the type of the exception object. The operator returns either zero, if the matching fails, or a pointer to the subobject of the exception object matching the type of the handler.

Figures 2.3 and 2.4 show the translated throw and try/catch statements. We model a throw statement that specifies an exception object as follows. First, a new exception object is allocated using `__sir_cpp_exc_alloc`. An object is then constructed into the returned storage. The construction may involve a call to a constructor and may fail (i.e. another exception can be thrown in the process). In that case, the exception object is freed using `__sir_cpp_exc_free`. As the exception object was in the new state, no destructors are called. If on the other hand the construction succeeds, the `__sir_cpp_exc_throw` operator is called, which transitions the exception object to a thrown state.

Note that if the above process is performed while another exception is already thrown, the last call (the one to `__sir_cpp_exc_throw`) kills the execution.

When the exception path leads the execution out of a try block, the handler retrieves the exception object using `__sir_cpp_exc_current`. The object is then matched against all catch statements corresponding to the handler. For each catch statement, a call to

```
int f2() {
    s a;
    return f1();
}
```

$1: **call** s::s, &a | 1 → $6
$2: **call** f1 | 1 → $5
$3: **call** s::˜s, &a | 1 → $6
$4: **exit** 0, $2

$5: **call** s::˜s, &a
$6: **exit** 1

Figure 2.5: A C++ program and a corresponding SIR subroutine. Note normal and exception (exit index one) paths.

`__sir_cpp_exc_catch` is made. If the call succeeds, the exception object is automatically transitioned to the caught state, allowing subsequent exceptions to be thrown. The catch statement body is then normally executed. At the end, the object is freed using `__sir_cpp_exc_free`. As it is in the caught state, the object's destructor is called (assuming the object has a destructor).

A rethrow statement (i.e. a throw statement that does not specify the object to be thrown) is translated into a call to `__sir_cpp_exc_current`, which retrieves the last object to be thrown, followed by a call to `__sir_cpp_exc_throw`.

2.12 Subroutine layout

Each SIR subroutine that was constructed from a C++ function has two exit points. Exit point zero is taken if the function returns normally (i.e. it the execution reaches a return statement or the end of the outmost compound statement). If the function throws an exception, exit point one is used to indicate that condition.

Figure 2.5 demonstrates the usage of various exit points. From the figure, we see that the normal execution path would traverse nodes $1, $2, $3, and $4 in that order. If an exception occurs in node $1 (during a call to the constructor), the subroutine immediately exits through exit point one (node $6). If an exception is thrown from function `f1`, the object `a` is first destroyed before the subroutine is exited through node $6. If an exception is thrown while another exception is causing stack unwinding, the C++ standard[13] requires that `std::terminate` be called. We do not model the call to `std::terminate`, we assume that the execution is aborted during the call to `__sir_cpp_exc_throw`.

Chapter 3

Parsing C++ programs

In this chapter we briefly describe the process of tranlating the C++ source code to the Stanse intermediate representation (which is defined in detail in Chapter 1).

The translation process is performed in two phases. The first one consists of preprocessing and parsing of the C++ source code and turning it into the abstract syntax tree (AST) form. Preprocessing the source code is a relatively simple task—the C++ preprocessor was inherited from the easy-to-parse C language.

Parsing of C++ language, on the other hand, is a complex process, requiring semantic analysis to be performed in parallel with the syntactic parsing. Contrast this to other, simpler programming languages (including D, Java or C#), whose grammars are commonly designed to be context-free, allowing the parser to perform name resolution and typing analysis in a separate step.

The difficulties arise from the fact that the C++ grammar is not context-free and classical parsing techniques yield ambigous parse trees. For example, the token sequence `a * b` may yield either

- a statement declaring a new variable `b` to be of type pointer to `a`, or
- a binary expression multiplying objects `a` and `b`.

A C++ parser must differentiate between these two possibilities depending of whether `a` is a type name or not. This determination is much more complicated than in the C language (which shares this particular kind of ambiguity), due to the presence of template classes and template functions.[1]

While there are C++ frontends which return the whole parse forest (as they use parsing methods like GLR[10]), which is only pruned after the parsing completes, most C++ parsers are hand-written recursive descent parsers[1] and are generally percieved as difficult to write.

In order to avoid having to write a C++ parser on our own—a task worthy of several master's theses—we chose to make use of Clang libraries.[4] Clang is a parser and LLVM

[1] Template specialization selection and template instantiation are governed by a significant number of complex language rules and require a full semantic analyzer in order to be performed correctly.

intermediate code generator used by the LLVM project as a C, C++ and Objective C front-end. We decided to use Clang, as it is written very cleanly and its parser is very well documented.[2] Thanks to Clang being open source and written in C++—a language we are most familiar with—we were able to identify and fix several critical issues that would otherwise prevent us from using Clang as our parser. We have of course provided the patches back to the Clang community.

The root of the AST produced by Clang represents the whole translation unit, which contains nested declarations (such AST nodes are referred to as declaration contexts). Typically, a translation unit would contain declarations of global variables, functions, function templates, class templates, classes and namespaces. The last four declaration types are also declaration contexts and therefore contain futher declarations (e.g. classes may contain declarations of nested classes, member functions, etc.). Those can recursively contain more declarations.

Apart from providing the AST, Clang also performs template instantiations for every template specialization that was used in the translation unit. The instantiations of class and function templates (which themselves are classes and functions respectively) do not appear explicitely in the AST as their declarations do not appear explicitely in the source code. Instead they are associated with the AST nodes that correspond to the template declarations from which they were instantiated.

To generate the SIR representation of the translation unit, we scan the AST for both ordinary functions and instantiations of function templates. Only function declarations that are also definitions (i.e. they have an associated body) are considered. For each such function definition we then translate its body into a single SIR subroutine. Note that virtual functions are also translated in this manner. (However, they are treated differently when they are called. The handling of virtual dispatch was described earlier in Chapter 2.) Each SIR subroutine that was constructed in the process is assigned a unique name and added to the resulting SIR unit.

Apart from functions, all variables whose lifetime is not limited by their scope are also of concern. This includes global variables, but also static local variables. These variables are initialized by the execution environment before the execution of the program begins. The initial values must be captured in the SIR unit, otherwise the description of the program would be incomplete. Therefore, during the AST scan, the list is constructed of all the global and local static variables together with their initial values.

Global and local static variables that require dynamic initialization (i.e. those, whose constructors must be called) are not handled in the current version of the code generator. Traditionally, this initialization is performed by generating an initialization function for each translation unit. A list of these functions is then included in the resulting program and traversed by the program runtime before the main program function is invoked. Similar solution could be employed to add support for dynamic initialization to SIR.

In the rest of this chapter, we will concentrate on the translation of C++ functions to SIR subroutines, starting with the translation of elementary expressions and statements.

[2] We encourage the reader to look over the AST documentation so as to have the necessary context for the rest of this chapter.

We will describe how to connect pieces of a SIR subroutine together, and how to perform backpatching. We will also describe the main ideas behind exception path generation.

As we will reference classes and functions that form the `cpp2sir` translation tool, we strongly suggest that the reader browse the source code while reading this chapter.

3.1 Sentinel nodes

The function responsible for translating a single C++ function to the corresponding SIR subroutine is called `detail::build_cfg` (it is called from the `build_program` function, which represents the core of the code generator—it turns a translation unit into the corresponding SIR unit). The function internally constructs a context object[3] and indirectly executes the `context::build_stmt` function on the body of the C++ function to be translated (i.e. on the compound statement attached to the function).

The `context::build_stmt` function is responsible for translating a single statement to a SIR subgraph. For statements which internally contain other statements (the already mentioned compound statement, all control-flow statements, the `try` statement, etc.), the function calls itself recursively.

Instead of returning a standalone graph which would then be potentially embedded by the caller into a larger graph, the function grows new subgraphs into the final graph. This design allows all data pertaining to the translation process to be stored in one place, instead of constantly moving it from temporary objects. Note that the data consist not only of SIR nodes, but of other important structures including execution contexts and backpatching sentinels (described below).

In order to mark the position where new subgraphs should be grown, the graph may during translation contain so-called *sentinel nodes*, which are empty nodes that do not contain a valid SIR instruction. New node are always added to the graph by filling a sentinel nodes with data and appending a new sentinel node to it.

The sentinel nodes that mark locations onto which new statements are to be grown are called *heads*. The first head is inserted into the (empty) graph before the translation process is begun on the body of a function. The head is passed as a parameter to the `context::build_stmt` function. When the statement translation is complete and the `context::build_stmt` function returns, the excess head is removed.

Sentinel nodes may be split in two (`context::duplicate_vertex`) or joined together (`context::join_nodes`). This happens for example during the translation of the `if` statement, during which the head is split, two branches are grown (corresponding to the `then` and `else` statements), and eventually are joined together.

Sentinel nodes are also used to mark locations in the graph that require further processing. Break and continue statements, for instance, generate and register sentinel nodes, which are later joined into the head after the containing loop statement is finished translating.[4] Similarly, label statemets and statements that throw exceptions leave sentinel nodes behind.

[3] Perhaps `context` is not the most expressive name for the class.

[4] In other words, sentinel nodes are used to mark locations for back-patching.

\$1: **none**

(a)

\$1: **value** c
\$2: **none**

(b)

\$1: **value** $c \mid 0 \to \$3$
\$2: **none**

\$3: **none**

(c)

\$1: **value** $c \mid 0 \to \$3$
\$2: **call** `f1`
\$4: **none**

\$3: **call** `f2`
\$5: **none**

(d)

\$1: **value** $c \mid 0 \to \$3$
\$2: **call** `f1`
\$4: **none**

\$3: **call** `f2` $\mid\to \$4$

(e)

Figure 3.1: A simple example of a SIR subroutine and the C code used to generate it.

Figure 3.1 shows the process of translation of the C++ statement `if (c) f1(); else f2();`. The construction starts with the `context::build_stmt` function receiving the sentinel node \$1 and the AST node for the `if` statement. We depict the graph as containing a single node, since all nodes but the sentinel are irrelevant (3.1a). The condition expression is then evaluated, appending a new node into the graph (3.1b). Note that the evaluation of the expression resulted in the graph having again only a single sentinel node. The sentinel node is then split in two and the condition on the edge connected to the new sentinel is labelled with 0, indicating that the new sentinel will be used to grow the `else` statement (3.1c). The two nested statements are then grown (3.1d). Finally, the two heads are joined together, yielding the final version of the subgraph (3.1e).

3.2 Extended operands

While translation of statements merely causes new subgraphs to be included in the SIR subroutine, the translation of expressions additionally yields a value that may be used as an operand to an instruction. Recall from Chapter 1 that SIR allows for five types of operands—a subroutine name, a constant, the value of a variable, the pointer to a variable and the value of a node.

Consider for example the statement `f1(a);`, which calls the function `f1`, passing to it the value of the variable a. The translation of the subexpression `a` does not yield any new nodes, it does however return the SIR operand of the variable value type referencing the variable a. Similarly, the evaluation of the subexpression `f1` yields no new nodes in the graph, but returns the SIR operand of the subroutine name type, referring to the subroutine `f1`. Finally, the evaluation of the function call operator then emits a new node, $[\![\textbf{call}\ \texttt{f1}, a]\!]$. The result of this expression is then the operand of node value type referencing this newly created node. The operand is then discarded.

Unfortuntely, this simple scheme fails to work in C++, as certain expression values may behave as lvalues (i.e. refer to the object itself) or as rvalues (referring to the value of the object). The behavior is context dependent; direct translation of a C++ expression to a SIR operand is therefore not possible in general.

To solve the abiguity, we always treat expressions as rvalues and promote them to lvalues only when necessary (when applied to the address-of operator, on the left side of the assignment or when binding to a reference). Additionally, we extend the set of SIR operands by extended operand types of *target of a variable* and *target of a node* in order to keep track of the corresponding lvalues. In the sources of the `cpp2sir` tool, these operands are represented by the `eop` type. This is also the return type of the `context::build_expr` function responsible for the translation of expressions to their subgraphs and operands.

Three functions are used to deal with extended operands. The `context::make_addr` promotes operand types from the value of a variable to the address of a variable, the target of a variable to the value of the variable, and the target of a node to the value of a node. Similarly, the `context::make_deref` demotes the operands to a lower type. Additionally, for the target of a variable or the target of a node, it emits the **deref** instruction and returns an operand of the node target type referring to its result. Note that the two aforementioned functions correspond directly to the address-of and the dereference operators of C++.

The third function, `context::make_rvalue` converts an extended operand to the standard SIR operand by emitting the **deref** instruction for node target and variable target operands, and returning the node value operand referring to the new instruction. The function is used when it becomes clear that the expression is going to be used as an rvalue.

Consider for instance the `&*p` expression. The `p` subexpression translates to variable value operand p. The `*p` yields variable target operand p. If the expression were used as an rvalue, the **deref** instruction would be emitted and the value of that node would be used as an operand. The expression is however used as an lvalue and due to the address-of operator, the operand is promoted back to the variable value type.

3.3 Execution context

During back-patching, the appropriate sentinel nodes are not merely joined with the current head. All automatic variables (i.e. the local variables) that cease to exist due to a `return`, `break`, `continue` or `goto` statement must have their destructors called. As sentinel nodes marking the locations to be back-patched are generated long before the back-patching occurs, additional information (notably the ordered list of variables that were in scope at the time) must be maintained.

The *execution context* associated with a SIR node is the list of automatic variables intermingled with the list of `try` statements that were in scope during the generation of that node. The context evolves as the source statements are processed. Whenever an automatic variable definition is encountered, a new variable registration record is pushed to the top of the current context. When the declaration scope of the variable is left, the registration is removed. Similarly, an exception handler registration record is pushed when the scope of the try statement is entered and removed when the processing of that statement has finished. At the start of the translation, the context is empty.

Execution contexts are maintained in a single context registry, represented by the class context_registry. The class allows new contexts to be created by appending and removing registration records from existing contexts. The old contexts remain in the registry. Contexts can be referred to by a descriptor (context_registry::context_type), allowing them to be associated with sentinel nodes and used during back-patching.

The two registration records that can be a part of a context are var_regrec and except_regdecl, which correspond to automatic variables and try statements, respectively. (There is a third registration record, exc_object_regrec, which is appended to the execution context during the translation of the throw statement. This record ensures that the magic exception memory allocated for the exception object using __sir_cpp_exc_alloc is freed in case the construction of the exception object throws.)

The registry ensures that for each context it contains—with the exception of the empty context—there is also a *parent context* consisting of the same set of registration records save for the topmost one. As such, the contexts stored in the registry form a tree, with the empty context serving as the root and other contexts being children of their parent context.

All of return, break and continue statements always cause the execution to jump into an outer scope. We will call the node to which these statements ultimately transfer control the *target node*. The execution context associated with the target node will be referred to as the *target context*.

Note that the target node does not exist at the time either of the transfer statements is processed. We deal with these statements by creating a sentinel node that represents the source point of the control flow transfer. The node and the descriptor of its associated context are then registered for further processing. Once the target node is created, the back-patching is performed—a path is created from the sentined node to the target node, consisting of nodes which cause calls to the destrustors of automatic variables.

Notice that the target context is always an ancestor of the registered context. We therefore peform the back-patching by traversing the subtree of the context registry rooted at the target context. During the traversal, each encountered context is associated with a graph node (either an existing one or a newly created one). The target context is associated with the target node. Contexts that add variable registration record are associated with a newly created node that calls the destructor of the registered variable; an edge is created leading from the new node to the node associated with the parent context. Contexts that add exception registrations inherit the node of their parents (i.e. exception registrations are ignored during backpatching).

This way, a path is created for each context that is a descendant of the target context, including the contexts of the registered sentinel nodes. Therefore, for each sentinel node, its associated context is used to lookup a node into which the sentinel should be joined.

While goto statements do not necessarily transfer control to an outer scope, it is guaranteed that the tranfer does not cause a variable with a non-trivial constructor to be introduced in the new scope. Our translator currently does not generate destructor chains for `goto` statements, however, that a technique similar to the one described above can be used for these statements as well.

3.4 Exception paths

Besides back-patching of transfer statements, context registry is also used to generate exception paths after the body of the function has finished generating. Exception paths are not generated during the function body translation, instead, sentinel nodes are left at places where exception paths begin (which can occur either through an explicit `throw` statement, or through a function call). After the translation of the function body is complete, the registered exception sentinels are back-patched to a newly created exit node (the exit node 1).

The procedure is similar to the one described in the previous section—the context tree is traversed from the target context (the empty context in this case) and destructor call nodes are created whenever a variable registration record is encountered. However, instead of inheriting their parent's node, contexts that add exception registrations are associated with the entry node to the appropriate catch handler.

Chapter 4

Automaton checker

In Chapter 1 an intermediate representation of programs called SIR was introduced. In this chapter, we show how Stanse—or to be more precise, the automaton checker—perfoms analyses of these SIR programs. We will refrain from going into too much detail, as our work on Stanse itself and on its automaton checker is limited to getting the checker working with SIR.

Ideally, a static checker would compute all states (i.e. pairs of control location and mapping from variable names to their values) that the program can reach and verify that none of the states violate any safety properties. Unfortunately, it can be shown that for any Turing-complete language, determining the set of reachable states is an intractable problem. (TODO: cite) While in practice the domain of all states of the program is often finite, it generally remains very large; model checking—as this kind of analysis on finite domain is called—is therefore mostly performed in a distributed and parallel manner.[3]

In order to decrease the number of states that the analysis tool must process, Stanse forgoes the tracking of concrete states and instead performs the analysis in the domain of *abstract states*. Consider, for example, the function on Figure 4.1, which performs an action consisting of two steps (prepare and finish), which must be executed atomically. To ensure that two parallel invocations of the function do not interleave, the function acquires a lock before performing the preparation, and releases it when the action finishes. Notice that the function fails to release the lock if the preparation fails.

```
void perform_action()
{
    lock(m);
    if (prepare() == -1)
        return;
    finish();
    unlock(m);
}
```

(a) The C++ source code

```
$1:  call lock, &m              {U[&m]}
$2:  call prepare | −1 → $5     {L[&m]}
$3:  call finish                {L[&m]}
$4:  call unlock, &m            {L[&m]}
$5:  exit 0                     {U[&m], L[&m]}
```

(b) Equivalent SIR program labelled with state sets

Figure 4.1: A faulty program, which fails to release a mutex on some execution paths.

$\langle\textit{pattern}\rangle$ ::= $\langle\textit{opcode}\rangle$ [$\langle\textit{pattern-operand}\rangle$ (**,** $\langle\textit{pattern-operand}\rangle$)*]

$\langle\textit{pattern-op}\rangle$ ::= $\langle\textit{subroutine}\rangle$ | $\langle\textit{var}\rangle$ | **&** $\langle\textit{var}\rangle$ | $\langle\textit{const}\rangle$ | $\langle\textit{placeholder}\rangle$ | **(** $\langle\textit{pattern}\rangle$ **)**

$\langle\textit{placeholder}\rangle$ ::= $\%n$

Figure 4.2: The EBNF grammar of SIR patterns

To discover this defect, it is sufficient to consider the program's state at any point during computation to be in the abstract domain $S = \{\mathtt{U}[\&m], \mathtt{L}[\&m]\}$, where the abstract state $\mathtt{U}[\&m]$ signifies that the mutex m is unlocked, whereas the state $\mathtt{L}[\&m]$ infers otherwise.

Each node in the control-flow graph of the program (which we write down using SIR, see Figure 4.1b) is then associated with a context $c \in \mathcal{C}$, where $\mathcal{C} = 2^S$, which represents the set of states that the program can be in immediately before executing the node. The context vector C, $C\colon N \to \mathcal{C}$, where N is the set of nodes in the control-flow graph, then assigns to each control location the set of reachable states. The context vector is used directly to detect the error; in this case the defect manifests itself through the state $\mathtt{L}[\&m]$ associated with the exit node.

The context vector is computed as described by Cousot et al.[7]. The computation starts with the least context vector (i.e. the context vector C_0 such, that for all $n \in N$, $C_0(n) = \emptyset$). The context is then refined by repeatedly applying a checker-specific propagation function, until a fixpoint is reached.

It can be show that such a fixpoint can always be found, as long as the set of all context vectors forms a complete lattice and the propagation function is order-preserving.[20] Context vectors with pointwise ordering ($C_1 \leq C_2$ if and only if for all $n \in N$, $C_1(n) \subseteq C_2(n)$) indeed form a complete lattice. Later we will show that Stanse always induces a propagation function that is order-preserving with respect to the pointwise ordering.

4.1 Pattern matching

Stanse uses pattern matching to identify nodes in the control-flow graph that are important for the analysis, in particular the nodes that cause a change in the abstract state of the program.

The lock checker, for instance, uses the pattern $[\![\textbf{call}\ \texttt{lock}, \%1]\!]$ to identify all nodes that cause a mutex to be locked. The pattern contains a placeholder, which identifies the subexpression that is to be returned if the matching is successful. When applied to the program in Figure 4.1, the pattern matches the node \$2, with $\%1 = [\![\&m]\!]$.

The grammar of SIR patterns, given in Figure 4.2, builds on the grammar of SIR instructions as given in Figure 1.1. Note that patterns follow the grammar of SIR instructions, but they allow node labels neither as an instruction label nor as an operand. The pattern may contain special placeholders $\%n$, with $n \in \mathbb{N}$, which match arbitrary operands and can be used to retrieve the corresponding subexpression after the matching finishes. Moreover, operands can be matched against nested patterns, allowing the matching of complex expressions.

We say that a partial function $I\colon \mathbb{N} \to \langle\mathit{operand}\rangle$ is a *variable assignment* or *interpretation.* A pattern operand $p \in \langle\mathit{pattern\text{-}op}\rangle$ matches a SIR operand $o \in \langle\mathit{operand}\rangle$ with interpretation I if and only if

- $p = o$,
- $p = \llbracket\%n\rrbracket$, for some $n \geq 1$ and $I(n) = o$, or
- $p \in \langle\mathit{pattern}\rangle$, $o = \llbracket\$k\rrbracket$, and p matches the node $\$k$ with interpretation I.

The pattern $p = \llbracket c\ o_1, o_2, \ldots o_n\rrbracket$ matches the node $n = \llbracket\$k : c'\ o'_1, o'_2, \ldots o'_n\rrbracket$ with interpretation I if and only if $c = c'$ and for all $1 \leq i \leq n$, the pattern operand o_i matches the operand o'_i with interpretation I. We say that an interpretation of a pattern p on a node n, written as $\mathcal{I}(p, n)$, is the least interpretation I such that the pattern p matches n with I.

The ability of patterns to match across multiple program nodes can be used to locate complex expressions. Consider, for example, the statement `m = create_mutex();`, which is translated to the following two-node SIR program.

$1: **call** `create_mutex`
$2: **assign** $\&m$, $1

The pattern $\llbracket\textbf{assign}\ \%1, (\textbf{call}\ \texttt{create\_mutex})\rrbracket$ can be used to locate all nodes in which a newly created mutex is assigned to a variable, with the placeholder %1 matching the target variable. In the case of the above SIR program, the pattern matches the node $2 with $\%1 = \llbracket\&m\rrbracket$.

Note that we have developed the above notation for the purposes of this thesis. The user specifies patterns in the automaton checker definition files using XML syntax. Figure 4.3 shows an example of such an XML fragment.

```
<pattern name="create-mutex">
    <node type="assign">
        <var name="P1"/>
        <node type="call">
            <function>create_mutex</function>
        </node>
    </node>
</pattern>
```

Figure 4.3: XML-serialized pattern $\llbracket\textbf{assign}\ \%1, (\textbf{call}\ \texttt{create\_mutex})\rrbracket$

4.2 Checker description

Stanse borrows from Hallem et al.[11] the approach of using finite automata to simplify for the user the description of the abstract domain. Stanse uses this automata-based description to automatically induce the propagation function, least fixpoint of which we seek.

In the automaton-based abstract state representation, the state of the program is given by a set of bound automaton states. A bound automaton state s is a tuple $s = (q, o_1 o_2 \cdots o_n)$, where $q \in Q$ is an (unbound) automaton state and $o_1 \cdots o_n \in \langle \mathit{operand} \rangle^*$ is the sequence of program objects to which s is bound. We use the symbol $\mathcal{S}$ to denote the set of all bound states. We write states using the notation $q[o_1 \cdots o_n]$. For example, in Figure 4.1, we set $Q = \{\mathtt{U}, \mathtt{L}\}$ and operated on bound states $\mathtt{U}[\&m]$ and $\mathtt{L}[\&m]$.

The user defines sets of bound states using state templates. A state template is a tuple $t \in Q \times (\langle \mathit{operand} \rangle \cup \langle \mathit{placeholder} \rangle)^*$, i.e. a bound state in which zero or more operands are replaced by placeholders. The state template $q[p_1 \cdots p_n]$ matches a bound state $q'[o_1 \cdots o_m]$ with interpretation I (a partial function $I \colon \mathbb{N} \to \langle \mathit{operand} \rangle$) if and only if $q = q'$, $n = m$, and for all $i \leq n$, either $p_i = o_i$ or $p_i = \%k$ and $I(k) = o_i$. Again, the state template t matches a bound state s if t matches s with some interpretation I. We denote the least such interpretation $\mathcal{I}(t, s)$.

An interpretation I can be used to make a state template more specific, by replacing all placeholders for which the interpretation is defined. We denote the result of such replacement by $t[I]$. For example, the state template $t = \mathtt{U}[\%1]$, when applied to the interpretation I such that $I(1) = [\![\&m]\!]$, becomes the state template $t[I] = \mathtt{U}[\&m]$. Keep in mind that the resulting state template may still contain placeholders.

The user specifies the abstract behavior of program objects by providing a set of transition rules of the form $t_1 \xrightarrow{p} t_2$, where $t_1, t_2 \in Q \times (\langle \mathit{operand} \rangle \cup \langle \mathit{placeholder} \rangle)^*$ are state templates and $p \in \langle \mathit{pattern} \rangle$ is a pattern. The rule indicates that if the program in the state that matches t_1 encounters a node that matches p, the state of the program changes to an instantiation of the state template t_2. To illustrate, recall the program in Figure 4.1. The rule decribing the transition from the state U to the state L, would be written as

$$\mathtt{U}[\%1] \xrightarrow{\textbf{call}\ \mathtt{lock},\%1} \mathtt{L}[\%1] \tag{4.1}$$

We will denote the set of all rules as $\mathcal{R}$.

We define the transformation function $T_0 \colon \mathcal{S} \times \mathcal{R} \times N \to \mathcal{S}$ by $T_0(s, r, n) = t_2[I_r][I_s]$, where r is the rule $t_1 \xrightarrow{p} t_2$, $I_r = \mathcal{I}(r, n)$, and $I_s = \mathcal{I}(t_1, s)$. If either r does not match n, s does not match t_1, t_1 is not a state (i.e. it contains placeholder), or the two interpretations I_r and I_s are incompatible, we set $T_0(s, r, n) = \perp$. We say that two interpretations I_r and I_s are incompatible if and only if there is a point $k \in \mathbb{N}$ in the domain of both I_r and I_s such that $I_r(k) \neq I_s(k)$, In other words, the function T_0 translates a state into a new state according to the given rule in the context of the specified node.

We define the function $T_1 \colon 2^{\mathcal{S}} \times 2^{\mathcal{R}} \times N \to 2^{\mathcal{S}}$ as $T_1(S, R, n) = \{s \in S \mid T_0(s, r, n) \neq \perp \text{ for some } r \in R\}$, i.e. the set of states that are acted upon by at least one rule. We calculate the set of transformed states using the function $T_2\colon 2^{\mathcal{S}} \times 2^{\mathcal{R}} \times N \to 2^{\mathcal{S}}$, defined as $T_2(S, R, n) = \{T_0(s, r, n) \mid s \in S, r \in R, \text{ and } T_0(s, r, n) \neq \bot\}$.

Finally, we define the transformation function $T\colon 2^{\mathcal{S}} \times 2^{\mathcal{R}} \times N \to 2^{\mathcal{S}}$ as $T(S, R, n) = T_2(S, R, n) \cup (S \setminus T_1(S, R, n))$. In other words, the transition rules cause the states on the left-hand side to be removed from the state set, while the states of the right-hand side are added. The states that are not acted upon remain in the set. A special case worth mentioning involves nodes that are matches by none of the transtion rules in R. For such nodes n, the transition function is an identity, $T(S, R, n) = S$.

In order to initiate the state propagation, special rules apply to start nodes of subroutines. Therefore, in addition to the transition rules, the user also specifies the initial (unbound) state $q_0 \in Q$. For each node n, and every rule r that matches the node, the interpretation $I = \mathcal{I}(r, n)$ is used to create a bound state $s = q_0[o_1 \cdots o_k]$, where $o_i = I(\alpha_i)$ and $\alpha_1 \cdots \alpha_k$ is the maximal sorted sequence of numbers in the domain of I. We denote the set of initial states as S_0.

4.3 Propagation and error checking

We now construct the propagation function $F\colon \mathcal{C}^N \to \mathcal{C}^N$ and show that it is indeed order-preserving and thus has a least fixpoint. Using the transformation function T, we define $F(C_0) = C$, where

$$C(n) = C_0(n) \cup \bigcup_{m \in \operatorname{pred}(n)} T(C_0(m), R, m),$$

for all $n \in N$ except the start node n_0, where $\operatorname{pred}(n)$ denotes the set of all predecessors of the node n, and R the set of all user-supplied transition rules. For the node n_0, the context additionally includes the set S_0,

$$C(n_0) = C_0(n_0) \cup \bigcup_{m \in \operatorname{pred}(n_0)} T(C_0(m), R, m) \cup S_0.$$

It can be seen that the transformation function T is order-preserving in its first parameter, making F order-preserving as well. Therefore, the function F has a least fixpoint, which can be found (due to the finite nature of the control-flow graph and the state set) by repeatedly applying the function, starting on a an empty context vector, until the vector stabilizes. We denote the final context vector as $\overline{C}$.

Once the propagation finishes, the user-supplied error rules are checked. Error rules are tuples (t, p), where $t \in \mathcal{T}$ is a state template and $p \in \langle \textit{pattern} \rangle$ is a pattern. Any node n that matches p, whose context $\overline{C}(n)$ contains a state s matching t, and the interpretations $\mathcal{I}(p, n)$ and $\mathcal{I}(t, s)$ are compatible, introduces an error.

The error rules additionally contain human-readable description of an error, which is then shown to the user. An error trace is generated by backtracking from the error node to either the initial node or to the node where the error state s was produced.

Chapter 5

Tools

In this chapter we shortly discuss how the tools developed in the course of this thesis can be used. All of the tools are release under the permissive MIT license(TODO: cite?), which allows others to reuse our source code. First we discuss the main tool, the `cpp2sir` translator, then we move to the test framework and SIR pretty-printing tools.

All of the tools mentioned are included on the attached CD, in the source and Windows binary form. The tools are also available in an online repository, the link to which can be found in the `README` file in the source code directory on the CD.

5.1 Translator

The main contribution of this thesis is the `cpp2sir` tool, which translates C++ programs to the Stanse intermediate representation, allowing its consumption by Stanse static checking tool.

The tool, as it is based on Clang libraries, accepts the same set of command line options as the clang parser.[5] Since clang attempts to be a drop-in replacement for the gcc compiler, the command line options are mostly compatible. Notable options include `-I<dir>` to specify directories in which headers files should be searched for, `-D<symbol>=<value>` to define the values for preprocessor symbols. Of course, free arguments are treated as names of source files to be compiled.

Our tool additionally accepts several custom options, which influence the translation process or change the format of the output. The unit identifier for the source file being compiled can be set using option `--unitid <id>`. The mode of output returned by the tool can be controlled by one of the flags `-Jjuac`, where

- `-J` is the default mode in which the JSON-encoded SIR unit is printed to the standard output. There is no unnecessary whitespace produced in this mode and as such it is quite unreadable. However, it is well-suited for machine-processing and it is therefore the mode Stanse uses during checking.
- `-j` is similar to `-J`, except that the JSON document is formatted to be human-readable.

- -u prints the AST of the translation unit.
- -a prints the AST for each function (including instantiated function templates) that would be translated to a SIR subroutine.
- Finally, -c prints nothing—it can be used to silently check whether the source code can be correctly parsed.

Note that if the parsing fails, the mode is ignored and only error messages are produced (to the standard error output).

5.2 Testing

In order to ensure that the output of the cpp2sir tool remains correct even in the face of modifications, we have included a batch of tests that can be used verify the output's correctness. The tests are located in the tests subdirectory of the cpp2sir project. Each test case consists of two files, a source code file with the .cpp extension and a pattern file (which is a JSON-encoded SIR unit) with .cfg extension.

In order for a test case to pass, the cpp2sir tool must parse the source code file without reporting any errors, and produce a valid JSON-encoded SIR unit. For each SIR subroutine in the pattern file, there must be a subroutine with the same name in the parser's output. Furthermore, the pattern subroutine must be isomorphic to a subgraph of the corresponding output subroutine.[1]

5.3 Pretty printing

The tools directory of the cpp2sir project contains several useful tools designed to print or visualize SIR graphs. All of the tools are written in the Python scripting language, an interepreter must be installed on the host system in order for the tools to work.

The tool pretty_print.py is a filter which accepts a JSON-encoded SIR unit on input and produces its pretty-printed representation on output. The format of the tool's output will be familiar to readers of this thesis, as it is exactly the format we describe in Chapter 1. The tool accepts no command line arguments.

For graphical visualization of the unit, the tool cfg2dot.py transforms a JSON-encoded unit to a .dot file, which can be passed to the dot program (part of the graphviz package). The tool produces the file onto the standard output. If dot is on the system search path, the tool cfg2pdf.py can be used to transorm a SIR unit directly to a PDF document.

Throughout the development of cpp2sir tool, it became clear that PDFs are indispensable for diagnosting test fails. Therefore, the tool tests2pdf.py was developed,

[1] The subgraph isomorphism is an NP-complete problem. However, SIR nodes can often be differentiated by the attached SIR instruction, and therefore the tests run reasonably fast even on large graphs.

which traverses all SIR unit files passed to it on the command line (standard UNIX wildcards, also known as globs, are allowed) and for each such file it

1. creates a PDF version of that file, storing it under a name constructed by appending `.pdf` extension to the unit file, and

2. looks for a file with the same name as the unit file, but with `.cpp` extension, runs the `cpp2sir` parser on it, and generates a PDF file from the output; again the name of the file is constructed by appending `.pdf` to the name of the source file.

Using this tool allows one to generate PDFs for all test files (both pattern and source files) in a single step.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0-201-10088-6.

[2] American National Standards Institute. *ANSI/ISO/IEC 9899-1999: Programming Languages — C.* 1430 Broadway, New York, NY 10018, USA: American National Standards Institute, 1999, ???? ISBN: ???? URL: `http://webstore.ansi.org/ansidocstore/product.asp?sku=ANSI%2FISO%2FIEC+9899%2D1999`.

[3] Jiří Barnat, Luboš Brim, and Petr Ročkai. "DiVinE 2.0: High-Performance Model Checking". In: *2009 International Workshop on High Performance Computational Systems Biology (HiBi 2009).* IEEE Computer Society Press, 2009, pp. 31–32.

[4] Clang contributors. *Clang: a C language family frontend for LLVM.* Feb. 2011. URL: `http://clang.llvm.org/`.

[5] Clang contributors. *Clang Compiler User's Manual.* May 2011. URL: `http://clang.llvm.org/docs/UsersManual.html`.

[6] GCC Wiki Contributors. *Structure of GCC.* Feb. 2011. URL: `http://gcc.gnu.org/wiki/StructureOfGCC`.

[7] P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixed Points". In: *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles.* New York, NY: ACM, 1977, pp. 238–252.

[8] D. Crockford. *RFC 4627: The application/json Media Type for JavaScript Object Notation (JSON).* Feb. 2011. URL: `http://www.ietf.org/rfc/rfc4627.txt`.

[9] Ron Cytron et al. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *ACM Transactions on Programming Languages and Systems* 13.4 (1991), pp. 451–490. URL: `http://doi.acm.org/10.1145/115372.115320`.

[10] Dick Grune and Ceriel J. H. Jacobs. *Parsing techniques: a practical guide.* Upper Saddle River, NJ, USA: Ellis Horwood, 1990. ISBN: 0-13-651431-6.

[11] Seth Hallem et al. “A system and language for building system-specific, static analyses”. In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation.* PLDI ’02. Berlin, Germany: ACM, 2002, pp. 69–82. ISBN: 1-58113-463-0. DOI: `http://doi.acm.org/10.1145/512529.512539`. URL: `http://doi.acm.org/10.1145/512529.512539`.

[12] ISO. *ISO/IEC 14882:1998: Programming languages — C++.* Available in electronic form for online purchase at `http://webstore.ansi.org/` and `http://www.cssinfo.com/`. Geneva, Switzerland: International Organization for Standardization, Sept. 1998, p. 732. ISBN: ???? URL: `http://www.iso.ch/cate/d25845.html;https://webstore.ansi.org/;http://webstore.ansi.org/ansidocstore/product.asp?sku=ISO%2FIEC+14882%2D1998;http://webstore.ansi.org/ansidocstore/product.asp?sku=ISO%2FIEC+14882%3A1998`.

[13] ISO. *ISO/IEC 14882:2003: Programming languages — C++.* Geneva, Switzerland: International Organization for Standardization, 2003, p. 757. ISBN: ???? URL: `http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=38110`.

[14] *ISO/IEC 14882:2003: Programming languages: C++.* 2003. URL: `http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=38110`.

[15] Chris Lattner. *LLVM Language Reference Manual.* Feb. 2011. URL: `http://llvm.org/docs/LangRef.html`.

[16] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04).* Palo Alto, California, 2004.

[17] Steven S. Muchnick. *Advanced compiler design and implementation.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN: 1-55860-320-4.

[18] David Schmidt. “Abstract interpretation of small-step semantics”. In: *Analysis and Verification of Multiple-Agent Languages.* Ed. by Mads Dam. Vol. 1192. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1997, pp. 76–99.

[19] Code Sourcery. *Itanium C++ ABI.* Jan. 2011. URL: `http://www.codesourcery.com/public/cxx-abi/abi.html`.

[20] Alfred Tarski. “A lattice-theoretical fixpoint theorem and its applications.” In: (1955).

[21] Mark N. Wegman and F. Kenneth Zadeck. “Constant propagation with conditional branches”. In: *ACM Transactions on Programming Languages and Systems* 13 (1991), pp. 291–299.

[22] Glynn Winskel. *The formal semantics of programming languages: an introduction.* Cambridge, MA, USA: MIT Press, 1993. ISBN: 0-262-23169-7.