Justus Matthiesen

Mathematizing the C Programming Language

Computer Science Tripos, Part II

Gonville & Caius College

May 20, 2011

Proforma

Name:	**Justus Matthiesen**
College:	**Gonville & Caius College**
Project Title:	**Mathematizing the C Programming Language**
Examination:	**Computer Science Tripos, Part II, June 2011**
Word-count:	**11,991 (as counted by TeXcount)**
Project Originator:	**Dr Peter Sewell**
Project Supervisor:	**Dr Peter Sewell**

Original aims of the project

The goal of the project was to formalise a small fragment of the C programming language, including at least integer arithmetic, and to produce a tool that could be used as an oracle: Given a C program within the covered fragment, it should output its meaning.

Work completed

The formalisation of the language covers far more than the original proposal suggested. It includes integer arithmetic, arrays, pointers, pointer arithmetic, and more. The tool implementing the developed semantics produces and solves logical statements describing the meaning of a given C program and, additionally, it includes a type checking phase. Comparison of the tool output with results produced by C compilers did not reveal any discrepancy for any of the tested C programs.

Special difficulties

None.

Declaration of Originality

I, Justus Matthiesen of Gonville & Caius College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date: May 20, 2011

Contents

1. Introduction **7**
- 1.1. A brief history of C . . . 7
- 1.2. Underspecification and portability . . . 7
- 1.3. Aim of the project . . . 8

2. Preparation **11**
- 2.1. Scope of the project . . . 11
- 2.2. Semantic engineering . . . 11
  - 2.2.1. Problematic language features . . . 12
  - 2.2.2. Choice of semantic style . . . 14
- 2.3. Development methodology . . . 18

3. Implementation **21**
- 3.1. Syntax . . . 21
  - 3.1.1. Parsing . . . 21
  - 3.1.2. From concrete to abstract syntax . . . 23
- 3.2. Static semantics . . . 25
  - 3.2.1. Type checking . . . 25
- 3.3. Dynamic semantics . . . 28
  - 3.3.1. General form . . . 28
  - 3.3.2. Expression semantics . . . 30
  - 3.3.3. Statement semantics . . . 35
  - 3.3.4. Interpreting the standard . . . 36
  - 3.3.5. Memory . . . 36
- 3.4. Constraint solving . . . 39
- 3.5. Software design . . . 40

4. Evaluation **45**
- 4.1. Success criteria . . . 45
- 4.2. Capabilities . . . 46
- 4.3. Limitations . . . 51

5. Conclusions **55**

A. An overview of semantic styles **61**
- A.1. Big-step semantics . . . 61
- A.2. Small-step semantics . . . 62

A.3. Felleisen-style semantics 63
A.4. Continuation semantics 64

B. Proposal **67**

1. Introduction

We present a novel formalisation of a subset of C and describe the implementation of a symbolic evaluator based on the newly developed semantics capable of computing all behaviours of a (small) C program. The fundamental motivation behind the project is the belief that the foundation of existing real world computer systems should be provably solid; our semantics makes large parts of the upcoming C1X standard precise, and our evaluator could be used as an oracle for C compiler testing.

1.1. A brief history of C

The C programming language was created in the early 1970s by Dennis M. Ritchie at the Bell Telephone Laboratories to fill the need of the new operating system Unix for a system programming language [Rit93]. The language evolved into a general-purpose programming language over the next two decades and its use spread so widely that a standardisation became necessary to avoid the different dialects of C to diverge even further. Today, C is still an evolving language: Since 2007 the ISO standards committee for the programming language C has been preparing the newest revision of the C standard, called C1X. It is expected to be finalised in the coming year.

1.2. Underspecification and portability

The first C implementation was written for the DEC PDP-11 architecture but it has since been ported to a multitude of different architectures. To allow programmers to target multiple architectures at the same time and to prevent C from degenerating into a "high-level assembly language", the committee has declared that C as defined by the standard must make it possible to write programs whose behaviour is guaranteed to be independent of any particular implementation and their underlying architecture ('C can be portable' [Ben07]).

Unfortunately, portability is in conflict with C's original purpose as a systems programming language: If the standard required, for example, consistent behaviour in the case of integer overflows across different implementations, we could no longer expect a compiler to be able to map a single C integer operation to a single machine instruction. The compiler would have to account for the difference between C standard and the specifics of the target architecture. However, constant factors matter to the systems programmer and hence hidden costs involved in the translation from C to machine code should be kept at a bare minimum. The standard's solution to the conflict is to under-

specify behaviour when different architectures are likely to be incompatible ('make it fast, even if it is not guaranteed to be portable' [Ben07]).

To get a taste for the sometimes subtle consequences of underspecification, consider the following program.

```
int main(void) {
  signed long a = -1;
  unsigned int b = 1;
  return a < b;
}
```

If we tried to determine the outcome of the program without any knowledge of the standard, we would expect the comparison always to be true. Indeed, if we compile the program with any common C compiler on an x86-64 machine, the result will be true. If we try the same program on a 32-bit x86 machine, however, the program will return false. The discrepancy in behaviour is a consequence of implicit type conversion, which we will discuss in detail at a later point.

1.3. Aim of the project

The previous discussion of underspecification should give an indication that even very simple C programs have non-trivial semantics. Throughout the following chapters we will examine a number of such programs that demonstrate how the C language diverges from the programmer's intuition. Even those who have intricate knowledge of the C standard often struggle with the subtleties of C, as witnessed by the committee's internal mailing[1] discussing what the correct interpretation of the standard should be, and by the number of requests for clarification on the more ambiguous details of the standard made in the form of defect reports[2].

Like most standards, the C standard is written in natural language which, despite the committee's great efforts to produce a precise description, is part of the problem: A specification in natural language will unavoidably leave room for vagueness and ambiguity. In contrast, a formal description of the language, where we take "formal" to mean 'admits logical reasoning' [Nor98, p. 2], would resolve the issue of ambiguity. Underspecification, in particular, makes the behaviour of many programs very difficult to predict without a formal model. Furthermore, a formal understanding of C is especially desirable since the language is very widespread in safety-critical applications.

The aim of this project is therefore to produce a formal description for part of the C programming language. Unfortunately, to find such a description, we will, like any other implementer, have to interpret the C standard first. If we are not careful, misinterpretation of the standard will introduce inaccuracies into the formalisation. As there is no way of proving that our derived semantics captures the intent of the standard, it is also important that the semantics is "recognisably" a description of C. Hence, we consider it

[1] c.f. C – Document register, http://www.open-std.org/JTC1/SC22/WG14/www/documents

[2] c.f. WG14 Defect Report Summary for ISO/IEC 9899:1999, http://www.open-std.org/jtc1/sc22/wg14/www/docs/summary.htm

necessary to formulate the semantics in a style that allows us to follow the definitions of the standard as closely as possible.

Although it resolves ambiguity, a formal semantics alone cannot help us to tackle the inherent complexity of C, since, even for fairly small programs, it would be infeasible to determine their meaning by manual interpretation according to the rules of the semantics. Instead, if we design the semantics to be executable, we can write a program, namely a symbolic evaluator, based on the pen-and-paper semantics that computes the meaning of a given C program. Since C programs often exhibit non-deterministic behaviour, we could either design an executable semantics that, when run, outputs just *one* behaviour or a semantics that considers *all* behaviours at once. The potential applications for a tool of the latter kind are plentiful: It could be used to check whether a program contains undefined behaviour, it could be used as an oracle in automatic compiler testing, or it could be used to empirically explore how a change to the standard would affect the overall language.

2. Preparation

In this chapter, we discuss the scope of the project, explore what a formal semantics for C must achieve, analyse the risks of particular implementation strategies and examine the choice of development methodology.

2.1. Scope of the project

Our project covers a sizable fragment of C: In terms of general concepts, we type check C programs and disambiguate overloaded operators. We faithfully model the non-deterministic evaluation order including sequence points and the sequenced-before relation. We also take into account the limited width of integer types and express type conversions and integer overflows precisely. Moreover, our semantics handles pointer and array types including the alignment of memory addresses. Sadly, including a formal treatment of floating point types would have gone beyond what can be done within the scope of this project. We also excluded union, structure, and enumeration types as they come with a large number of special cases in the language specification. However, our semantics is expressive enough that they could be supported in principle.

We have defined and implemented rules for most of the expression language, including recursive function calls and pointer arithmetic. We have no support, however, for the syntactically heavy compound literals, generic selections, and compound initializers. We opted not to model any of the bitwise operators apart from the two shift operators since their behaviour is mostly implementation-defined. Out of the statement language, we have support for all of the iteration statements and, although our semantics does not give meaning to the goto statement, our treatment of the continue, break and return statements demonstrates that we are capable of expressing non-structural control-flow.

2.2. Semantic engineering

As outlined in the previous chapter, it is important that the formal description closely resembles that of the standard. The standard defines the meaning of C programs in a largely structural style, i.e., the definition is structurally recursive in the grammar. As an example, consider how the behaviour of the logical-and operator is specified.

The behaviour of the logical-and operator is recursively defined in terms of the values of its operands, where *logical-AND-expression* and *inclusive-OR-expressions* are non-terminal symbols of the expression grammar. Fortunately, the different grammatical symbols only encode associativity and operator precedence but otherwise do not carry

> **6.5.13 Logical AND operator**
>
> **Syntax**
>
> 1 *logical-AND-expression:*
> *inclusive-OR-expression*
> *logical-AND-expression* **&&** *inclusive-OR-expression*
>
> **Constraints**
>
> 2 Each of the operands shall have scalar type.
>
> **Semantics**
>
> 3 The **&&** operator shall yield 1 if both of its operands compare unequal to 0; otherwise, it yields 0. The result has type **int**.
>
> 4 Unlike the bitwise binary **&** operator, the **&&** operator guarantees left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. If the first operand compares equal to 0, the second operand is not evaluated.

Fig. 2.1.: Definition of the logical-and operator [Jon10, p. 99]

any operational meaning. As a consequence, we can work on an abstracted syntax instead.

2.2.1. Problematic language features

Although C is defined in a largely structural style, it has a number of properties and features that require us to use an unconventional semantic style. We examine the most important of these in the following starting with underspecification. Underspecification comes in three distinct flavours: Implementation-defined, unspecified and undefined behaviour.

Implementation-defined behaviour

The behaviour of an operation is implementation-defined if the standard allows a range of options but requires each implementation to make a consistent and documented choice. For instance, the standard leaves it open whether the type **char** is a signed or an unsigned integer type. As a consequence, the result of the program

```
int main(void) {
  char c = -1;
  return c < 0;
}
```

depends on which choice an implementation makes. Since our aim is a semantics that is in direct correspondence to the standard, we need to be careful that dependence on implementation-defined behaviour is made explicit. After all, our semantics should describe C and not just a specific implementation of C.

Unspecified behaviour

Behaviour is unspecified if the standard presents several options without imposing any further requirements on which is chosen in any circumstance. Unspecified behaviour may seem very similar to implementation-defined behaviour, but the key difference is that the former need not be the same on every run of a program. The following program is an example of unspecified behaviour.

```
int global = 0;

int f(void) {
  return global;
}

int g(void) {
  global += 1;
  return global;
}

int main(void) {
  return f() + g();
}
```

Evaluation order in C is largely unspecified with the result that either function call in expression `f() + g()` might be evaluated first. Thus, `main` returns either 1 or 2, which is unspecified.

As we will see shortly, the unspecified evaluation order of subexpressions is especially problematic for many semantic styles since it causes sequential programs like the above to behave non-deterministically.

Undefined behaviour

When a program containing undefined behaviour is executed, absolutely any outcome is permitted: The program is semantically meaningless. It is the programmers responsibility to keep programs free of undefined behaviour.

In many cases, undefined behaviour allows compilers to perform optimisation that otherwise would require expensive analyses to guarantee safety. Division-by-zero is such an example.

```
int main(void) {
  return 1 / 0;
}
```

A compiler is allowed to assume that a program is free of division-by-zero errors since, if otherwise, behaviour is undefined making any interpretation valid. Consequently, compilers can perform code motion optimisation (even on architectures where division by zero causes an exception) without having to perform the data-flow analysis that ensures that all divisors are non-zero.

When dealing with undefined behaviour in a reduction semantics, we are faced with two fundamental options: We could either introduce a special configuration **undef**, say,

and reduce all programs that exhibit undefined behaviour to **undef** or we represent undefined behaviour as a stuck configuration, i.e. we ensure that no program with undefined behaviour can be (fully) reduced to a value. We believe that a semantics that implements the former option and gives an explicit set of rules for undefined behaviour rather than defining it 'by omission of any explicit [...] behaviour' [Jon10, chapter 4, §2] is of greater value as it could be used to find the source of undefined behaviour.

Non-structural constructs

Even though most of the language can be defined structurally, C has a range of features whose meaning is context-dependent making them fundamentally non-structural. Examples are the **`break`** and **`continue`** statements inside loops but most strikingly the **`goto`** statement. Newer languages, like C#[1], only allow **`goto`** statements that jump to labels that are lexically in scope. In contrast, C only requires the label to be within the same function, which, for instance, allows programs to execute both branches of the same **`if`**-statement.

```
int main(void) {
  int x;
  if (0) {
    x = 0;
    goto label;
  } else {
    label: x = 1;
  }
  return x;
}
```

Memory model

The standard committee has decided to support multi-threaded program execution in the next C standard. It will be based on the C++ memory model. Although we will not include the memory model in our formal description, it is beneficial to pick a semantic representation that allows it to be added in later without any major modification of the semantics. The reason is that even sequential C programs, as we have seen previously, can have non-deterministic behaviour. If we view non-determinism as the general case rather than treating it as a special case, we can hope to find a more uniform and concise specification of C.

2.2.2. Choice of semantic style

We now return to the question of semantic style. We summarise our reasons to reject conventional semantic styles and give a contrasting, high-level overview of our own semantics. For a full discussion we refer the reader to Appendix A.

[1] c.f. C# Reference, `http://msdn.microsoft.com/en-us/library/13940fs2.aspx`.

Big-step semantics

Since, in general, evaluation order is unspecified the order of evaluation of subexpression e_1, e_2 and e_3 in

$$(e_1 \texttt{ + } e_2) \texttt{ + } e_3$$

can be any permutation of the three subexpressions. If e_1, e_2, e_3 are side-effecting, then the order of evaluation can affect the result of the entire expression. Hence, our semantics must be able to produce all of the (at least) six evaluations of the above expression.

Big-step semantics forces us to make a choice of whether to fully evaluate the left-hand or the right-hand operand of a binary operator first. The rules

$$[\textsc{add-l}]\ \frac{\langle e_1, \sigma\rangle \Downarrow \langle m, \sigma'\rangle \qquad \langle e_2, \sigma'\rangle \Downarrow \langle n, \sigma''\rangle}{\langle e_1 \texttt{ + } e_2, \sigma\rangle \Downarrow \langle k, \sigma''\rangle} \quad \text{if } k = m + n$$

and

$$[\textsc{add-r}]\ \frac{\langle e_1, \sigma'\rangle \Downarrow \langle m, \sigma''\rangle \qquad \langle e_2, \sigma\rangle \Downarrow \langle n, \sigma'\rangle}{\langle e_1 \texttt{ + } e_2, \sigma\rangle \Downarrow \langle k, \sigma''\rangle} \quad \text{if } k = m + n\ .$$

express the two choices.

As a result, we cannot obtain the order e_1, e_3, e_2. If we reduce expression e_1 first, then we must evaluate e_2 before we can evaluate e_3:

$$\cfrac{\cfrac{\langle e_1, \sigma\rangle \Downarrow \langle m, \sigma_1\rangle \qquad \langle e_2, \sigma_1\rangle \Downarrow \langle m + n, \sigma_2\rangle}{\langle e_1 \texttt{ + } e_2, \sigma\rangle \Downarrow \langle m + n, \sigma_2\rangle}[\textsc{add-l}] \qquad \langle e_3, \sigma_2\rangle \Downarrow \langle k, \sigma_3\rangle}{\langle (e_1 \texttt{ + } e_2) \texttt{ + } e_3, \sigma\rangle \Downarrow \langle m + n + k, \sigma_3\rangle}[\textsc{add-l}]\ .$$

It should be clear that this limitation makes big-step semantics a very poor choice.

Small-step semantics

Whereas big-step semantics do not permit fine-grained control over the flow of evaluation, small-step semantics give us the notion of a single computational step. Although it resolves the issue of non-deterministic evaluation order in a straightforward manner, the same cannot be said about non-structural language features.

Continuation semantics

Continuation semantics, however, provide a neat framework for defining rules that depend on their context. We have rules that produce evaluation contexts, e.g.

$$[\textsc{add-l}]\ \langle e_1 \texttt{ + } e_2 \cdot \kappa, \sigma\rangle \to \langle e_1 \cdot [\_ \texttt{ + } e_2] \cdot \kappa, \sigma\rangle\ ,$$

and also rules that destroy the context again:

$$[\textsc{add-l-v}]\ \langle v_1 \cdot [\_ \texttt{ + } v_2] \cdot \kappa, \sigma\rangle \to \langle v \cdot \kappa, \sigma\rangle \quad \text{if } v = v_1 + v_2\ .$$

Again, we have to ensure that all possible evaluation orders of expression $(e_1 \texttt{ + } e_2) \texttt{ + } e_3$ are considered. Unfortunately, the linear way in which we construct continuations prohibits certain evaluation orders. We cannot, for example, evaluate e_1 first followed by e_3 since after the first transition

$$\langle (e_1 \texttt{ + } e_2) \texttt{ + } e_3 \cdot \mathbf{stop}, \sigma \rangle \rightarrow \langle e_1 \texttt{ + } e_2 \cdot [\_ \texttt{ + } e_3] \cdot \mathbf{stop}, \sigma \rangle \,,$$

expression e_3 is hidden inside the continuation and we cannot access it again until $e_1 \texttt{ + } e_2$ has been reduced to a value.

Trace semantics

Unfortunately, there is no obvious solution to the above problem that does not violate executability other than choosing a fundamentally different semantic style.

In order to overcome the limitations of each of the choices above, we could use a combination of several reduction relations, one for each semantic style, and apply each to the expressions and statements they handle best. Whilst, as Norrish [Nor98] demonstrates, it is possible to express the whole of C by using a range of different techniques, we would prefer a more uniform description. Moreover, we still have to tackle two outstanding issues: implementation-defined behaviour and support for a memory model.

Implementation-defined behaviour still stands in the way of executability. As an illustration, consider the **`sizeof`** operator. It tells us the size of a given type in bytes[2] but the standard only enforces a minimum size. However, if we are to evaluate the expression **`sizeof(int)`**, then the semantic styles we have discussed so far require us to either provide a concrete number or sacrifice executability. Both are in direct conflict with our initial goals.

It should be clear at this point that we will only be able to retain executability if underspecification is mirrored in the language that we use to describe the meaning of programs, e.g. **`sizeof(int)`** must not be reduced to a concrete number. Instead, it should be a symbolic value subject to logical constraints encoding the requirements of the standard. Norrish's work is a good demonstration. Since Cholera is implemented in the logic of the HOL proof assistant, Norrish [Nor98, p. 30] can, for example, express the fact that the number of bits in a byte is implementation-defined by simply asserting

$$\vdash \texttt{CHAR\_BIT} \geqslant 8 \,.$$

Describing the meaning of programs within a logical framework gives us the advantage that we can express underspecification naturally and still reason about programs with implementation-defined behaviour.

Even though multi-threading will not be part of our semantics, we want our model to be expressive enough to support it at least in principle. Multi-threaded execution gives rise to weak ordering of accesses to shared memory. The memory model specifies

[2] Even the number of bits in a byte is implementation-defined. According to the standard, it can be any number but must be at least eight.

which orderings of memory actions are valid and which result in undefined behaviour. In concrete terms, we want to be able to apply our semantics for single-threaded programs to several threads in isolation and specify the order of memory actions at a later point. The implications for our semantics are twofold.

Firstly, when reading a shared memory object, we cannot provide a concrete value since another thread might have altered the memory location since the last write. As with implementation-defined behaviour, the issue can be resolved by describing the meaning of programs within a logical framework: We associate a symbolic variable with the result of the memory read giving us the option of assigning a concrete value to it at a later stage with the knowledge of the specific memory model in use.

Secondly, we have to expose shared memory accesses and the order in which they occur to the memory model. Since even sequential C programs can have non-deterministic behaviour, we can simplify our description of non-determinism due to unspecified evaluation order by abandoning the notion of a global memory completely and treating accesses to local memory in the same way as accesses to shared memory. The key observation is that different choices in evaluation order can only have an observable effect in the presence of side-effects: If both operands of an addition are free of side-effects, then it does not matter which operand is evaluated first. Hence, we could imagine an alternative semantics that does not model the choice in evaluation order by a corresponding choice in the next transition step (i.e. by defining a non-functional reduction relation) but instead expresses it as an ordering relation on side-effects. Unlike before, the expression e_1 `+` e_2 will no longer give rise to two distinct evaluations. The unrestricted choice in evaluation order will instead be expressed by a lack of ordering between the side-effects of the expression.

We illustrate the general idea using the following program.

```
int main(void) {
  int x = 2;
  int y = 0;
  y = (x == x);
  return 0;
}
```

In Figure 2.2, we present the ordering relation, or rather its transitive reduction, on the program's actions as produced by our actual semantics. Actions include all side-effects but also other events, e.g. $\mathsf{Id}_8(v_{48})$, for reasons that we will explore in Section 3.3.

Function `main` declares two block scope identifiers `x` and `y` of type `int`. They become live at the start of the block and, accordingly, action $\mathsf{Create}_2(\texttt{int}, v_{48})$ and $\mathsf{Create}_1(\texttt{int}, v_{47})$ appear at the top of the graph. The arguments v_{48} and v_{47} are symbolic names for the new memory locations of `x` and `y`, respectively. Each action has an arbitrary but unique label, here 2 and 1, so that they can always be distinguished. The type information is necessary to inform the memory model of the size of the object and to ensure that later reads and writes to the location adhere to the relevant typing restrictions. Similarly, the following two memory writes correspond to the initialisation of `x` and `y`. More interestingly, in expression `y = (x == x)`, the two loads from `x`, i.e $\mathsf{Load}_{11}(\texttt{int}, v_{48}, v_{53})$ and $\mathsf{Load}_9(\texttt{int}, v_{48}, v_{51})$, can be executed in either order: they are unrelated. Variables

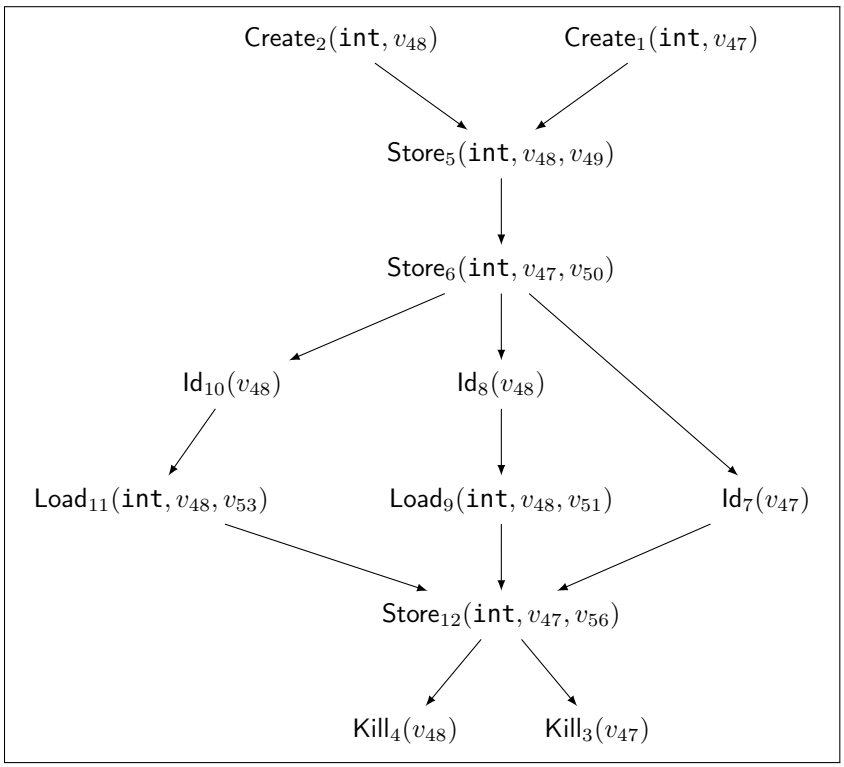


Fig. 2.2.: Ordering relation on actions as produced by our semantics.

v_{53} and v_{51} are symbolic placeholders for the values read from memory since we cannot provide concrete values until after a total order on memory actions has been fixed. Action $\mathsf{Store}_{12}(\texttt{int}, v_{47}, v_{56})$ writes the result of the comparison, v_{56}, to memory, whose value the semantics will further constrain by a logical statement. Finally, $\mathsf{Kill}_4(v_{48})$ and $\mathsf{Kill}_3(v_{47})$ announce the end of lifetime of identifiers `x` and `y` to the memory model.

A semantics of this kind can resolve all the complications that are inherent to C reasonably well as we will see in Chapter 3, where we present our final semantics in full detail.

2.3. Development methodology

Since the project proposal left a number of decisions open and since the requirements imposed by C's unique set of features on the semantics still had to be examined, we decided to adopt an iterative development process to allow an explorative phase at the start of each cycle to refine the original development targets. Formal development processes like Boehm's spiral model [Boe88] have been mainly devised to guide large-scale software development including interaction with a paying customer. The overhead involved in fully adopting such a model would be overwhelming for a one-developer project and a comparatively small time budget. Instead, we apply the principles of the '*risk-driven*

approach' [Boe88, p. 61] of the spiral model in less formal procedure. After a first prototype that contained common functionality, we implemented one feature at a time going through a development cycle consisting of the following stages: initial experiments and investigations to aid the requirement analysis, implementation, refactoring and finally testing.

In addition, we make use of a range of tools to ensure development to go smoothly. Most importantly, we use a version control system[3] to facilitate documentation and tracking of changes to the code base, to allow effortless roll-backs and to enforce a strict separation of work-in-progress and production code. To prevent disruption of development by data loss, automatic backups of the local copy (containing unfinished and experimental work) and hosted repository were performed[4] at regular intervals.

[3]Namely, Mercurial (`http://mercurial.selenic.com/`).

[4]Using rsync (`http://rsync.samba.org/`) for incremental backups and the cron replacement cronie (`http://fedorahosted.org/cronie/`) for scheduling.

3. Implementation

Recall that the aim of our project is a tool that computes a logical statement describing the meaning of a given source program. Implementing such a tool for C is not an easy task: Parsing must not alter the semantics of the C program but at the same time the result must be abstract enough to make the subsequent type checking and semantic analysis feasible, the type checker has to cope with a huge number of special cases, our dynamic semantics must be expressive enough to model the subtle and often ambiguously defined behaviour of C operators, and constraint solving has to be powerful enough to produce results but must be fast enough to be practical. In this chapter we give an overview of the implementation of this tool.

3.1. Syntax

Before we can analyse the semantics of a program, we must convert its plain text description to an intermediate form that reveals the structure of the program.

The standard describes what constitutes a syntactically correct program and how to build the concrete syntax tree of a program in form of a BNF grammar. As usual much of the information encoded in the grammar, e.g. associativity and precedence of operators, has the purpose to make parsing unambiguous. As we do not want to complicate the subsequent semantic analysis unnecessarily, we would like to work on an abstract syntax that ignores information that is only relevant for parsing.

Our tool breaks the transformation into two distinct stages: The first parses C programs to yield an intermediate representation that is still very close to the original grammar. The second stage checks various syntactic constraints on this intermediate representation and outputs the abstract syntax we use for semantic analysis.

3.1.1. Parsing

Since the BNF is already a formal description of a language grammar, parsing was not a priority and our project proposal planned to use the popular C front-end CIL [NMRW02]. Despite that, we ended up writing our own parser.

Why our own parser?

Inspection of both the documentation and the source code revealed that CIL was not suitable for our purposes: It not only parses input programs but also performs a number of transformations that can alter the semantics of a program. For example, CIL splits

complex expressions into several smaller pieces by use of temporary variables. Our example program demonstrating unspecified behaviour (see Section 2.2.1), for instance, is rewritten into one that is no longer non-deterministic. The new program

```
[...]

int main(void) {
  int tmp0;
  int tmp1;

  tmp0 = f();
  tmp1 = g();
  return (tmp0 + tmp1);
}
```

enforces left-to-right evaluation order and thus eliminates the behaviour that arises from evaluating the function call to `g` first.

Another issue we observed resulted from a transformation that turns all implicit type conversions into explicit type casts. The transformation requires the user to provide CIL with the implementation-defined size of each integer type and, depending on this information, CIL will produce different results.

These observations lead us to the conclusion that writing our own parser was necessary.

Parsing to an intermediate representation

Apart from avoiding the deficiencies of existing front-ends, creating our own parser gives us another advantage, as we will see below.

A grammatical C program is not necessarily a syntactically correct program. The grammar allows, for example, functions to have storage class **`register`** but an additional syntactic constraint rules such programs to be invalid. In our effort to stay close to the standard, we want our parser to accept precisely those programs that are recognised by the BNF grammar. Hence, in a first stage we parse to a low-level representation that is still very close to the original grammar and then check syntactic constraints on the intermediate representation, and translate it to an abstract syntax appropriate for further analysis, in a second stage. Figure 3.1 demonstrates that our parser indeed faithfully reproduces the grammatical production of the standard.

(6.5.4) *cast-expression:*
unary-expression
`(` *type-name* `)` *cast-expression*

```
/* 6.5.4#1 Cast operators, Syntax */
cast_expression:
| unary_expression
    {$1}
| LPAREN type_name RPAREN cast_expression
    {C.CAST ($2, $4)}
;
```

Fig. 3.1.: Comparison of C grammar (left) and our parser (right).

The strict separation of parsing and checking of other syntactic constraints that mirrors the structure of the standard is only possible because we wrote our own parser and did not choose a third-party front-end like CIL.

3.1.2. From concrete to abstract syntax

The second stage of the tool front-end generates abstract syntax from the intermediate representation of the first stage. Many aspects of the transformation are straightforward but, for example, the syntax of declarations needs to be rewritten substantially before we can give a reasonable description of their meaning.

In the following we will illustrate only some of the intricacies of transforming the declaration syntax by use of a simple example. Figure 3.2 gives the full grammatical

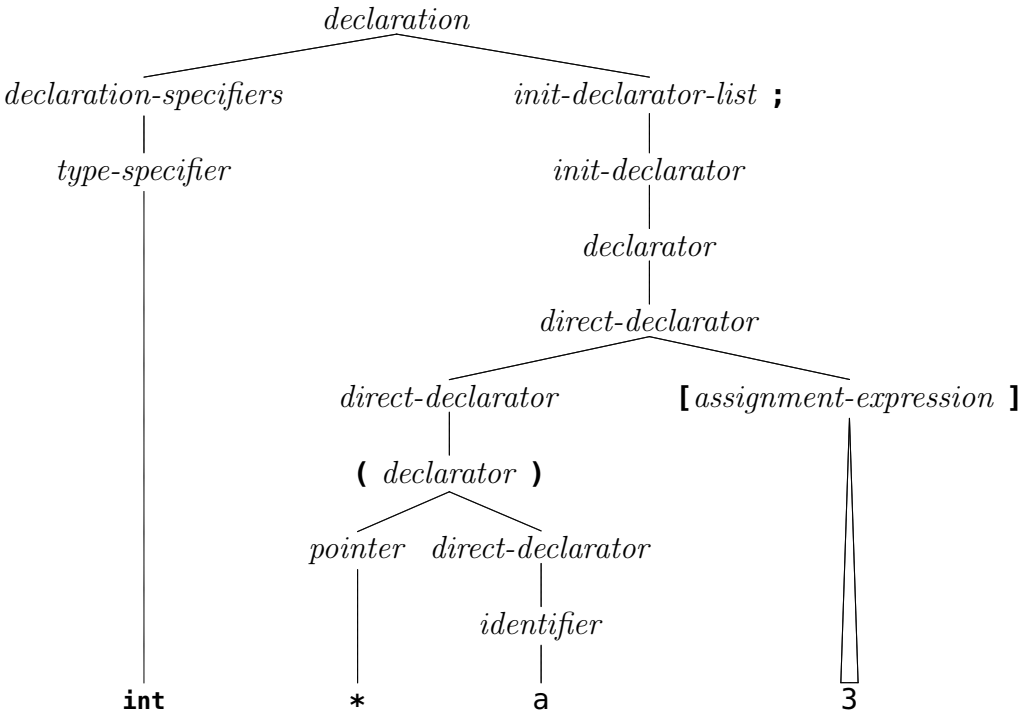


Fig. 3.2.: Derivation of `int (* a)[3];`.

derivation of the declaration `int (* a)[3];`. The declaration introduces variable `a` and defines it to be a pointer to an integer array of size three. From Figure 3.2 it should be apparent that the structure of the derivations does not map very well to our intuitive understanding: A declaration specifies the type of an identifier but the identifier is deeply buried inside the grammatical structure of the type. And `int` is conceptually part of the array but it is grammatically completely unrelated. Hence, our transformation needs to separate identifier and type information and needs to bring the type information into a form that mirrors its hierarchical structure. After transformation, our example will

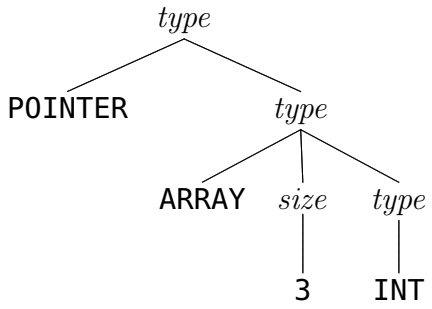


Fig. 3.3.: Example 3.2 after transformation.

have the abstract syntax tree given in Figure 3.3. Note how close the abstract syntax is to our prose description of the type.

Programmatically, we make extensive use of higher-order functions to rearrange the parse tree. In particular, when faced with the grammar production[1] for array types, i.e.

$$\textit{direct-declarator} \to \textit{direct-declarator}\ \texttt{[}\ \textit{assignment-expression}\ \texttt{]},$$

we translate the *direct-declarator* non-terminal to a function of type

$$[\![\textit{direct-declarator}]\!] : \texttt{AST.type} \to \texttt{AST.type}\ ,$$

where `AST.type` is our abstract syntax representation of C types. Then we can represent the entire production as the function composition of $[\![\textit{direct-declarator}]\!]$ and a function that, given a term of type `AST.type`, produces an array type of size $[\![\textit{assignment-expression}]\!]$:

$$[\![\textit{direct-declarator}\ \texttt{[}\ \textit{assignment-expression}\ \texttt{]}]\!] : \texttt{AST.type} \to \texttt{AST.type} \triangleq$$
$$[\![\textit{direct-declarator}]\!] \circ (\lambda t : \texttt{AST.type}\ .\ \texttt{ARRAY}\,([\![\textit{assignment-expression}]\!]\ ,\ t))\,.$$

Finally, the rules for transforming pointer types,

$$[\![\texttt{*}]\!] : \texttt{AST.type} \to \texttt{AST.type} \triangleq \lambda t : \texttt{AST.type}\ .\ \texttt{POINTER}\,(t)\ \ ,$$

and the type specifier **`int`**,

$$[\![\texttt{int}]\!] : \texttt{AST.type} \triangleq \texttt{INT}\ ,$$

are straightforward again. Putting it all together, we can now see how the AST for the

[1] To simplify matters, we will leave out the identifier `a` from here on.

type declaration `int (*)[3]` is generated[2]:

$$
\begin{aligned}
&[\![\texttt{int (*)[3]}]\!] \\
&= [\![\texttt{(*)[3]}]\!]\,[\![\texttt{int}]\!] \\
&= ([\![\texttt{(*)}]\!] \circ (\lambda t : \texttt{AST.type} \,.\, \texttt{ARRAY}\,(\texttt{3},\ t)))\,\texttt{INT} \\
&= (\lambda t : \texttt{AST.type} \,.\, \texttt{POINTER}\,(\texttt{ARRAY}\,(\texttt{3},\ t)))\,\texttt{INT} \\
&= \texttt{POINTER}\,(\texttt{ARRAY}\,(\texttt{3},\ \texttt{INT}))\ .
\end{aligned}
$$

To summarise, our implementation is a very concise description of the non-trivial tree transformation due to the use of higher-order functions. We leave it to the reader to imagine the stack-based implementation that would be an appropriate choice in an imperative programming language like C without first-class functions.

In this section we have looked at how to translate C programs to an intermediate language suitable for semantic analysis. Most issues we faced originated from the size and the unusual structure of C's grammar. Indeed, the syntax of C and especially the declaration syntax has been a primary cause of criticism: As Peter van der Linden puts it in his 'Expert C Programming' book: 'the syntax of C declarations is a truly horrible mess that permeates the use of the entire language. It's no exaggeration to say that C is significantly and needlessly complicated because of the awkward manner of combining types.' [vdL94, p. 65].

3.2. Static semantics

The static semantics of C is not particularly complex in comparison to modern typed languages like ML or Haskell: It lacks advanced features like implicit polymorphism or type inference and the guarantees given by the type system are very weak and not well understood. Because of this, our proposal did not anticipate the need to implement the static semantics, but it was still necessary: C operators are highly overloaded and hence, we need the type information to disambiguate their use.

3.2.1. Type checking

We will now examine the type system as used by our tool to annotate each node of the AST with a type. We restrict the overview of our type system, which consists of over 70 rules, to a handful of typeset and simplified rules that highlight properties of the type system that are unique to C, rather then presenting the actual OCaml implementation. The full type system requires a large number of auxiliary functions and case distinctions to account for overloading, special cases, and differently qualified types.

The type system of C can be described in the usual style: Our typing relation has the general form $\Gamma, \Phi \vdash e : \tau$, where e is an expression and τ is a C type. Moreover, the

[2] We cheat slightly in the first derivation step: To make the example presentable, we omitted describing the corresponding rule.

typing environment, Γ, is a partial map from identifiers to types. Finally, Φ is the set of all function names.

We begin with the base cases: An integer constant n has type `SIGNED INT` provided it is within the range of values that can be represented by the type `SIGNED INT`.

$$[\textsc{const}]\ \frac{\texttt{INT\_MIN} \leqslant n \leqslant \texttt{INT\_MAX}}{\Gamma, \Phi \vdash n : \texttt{SIGNED INT}}$$

The standard only specifies a minimum range but the exact range is implementation-defined. When n is outside the minimum range, our implementation enumerates all possible typings and remembers what assumptions, e.g. $n \leqslant$ `INT_MAX`, have been made for each.

Next, we look at identifiers. If an identifier id has type τ according to the type environment and is not a function name, then the type of the expression is $\texttt{lvalue}[\tau]$.

$$[\textsc{id}]\ \frac{\Gamma(id) = \tau \qquad id \notin \Phi}{\Gamma, \Phi \vdash id : \texttt{lvalue}[\tau]}$$

According to the standard an '*lvalue* is an expression that can potentially designate an object', where an object is a region of memory that in most cases contains a value. If id is a function name, then the result is not an lvalue but will be a function type.

$$[\textsc{id-fun}]\ \frac{\Gamma(id) = \tau \qquad id \in \Phi}{\Gamma, \Phi \vdash id : \tau}$$

Unless an expression of function type τ is operand of the `&` or **`sizeof`** operator, however, the expression will be coerced to type pointer to τ, i.e. $\texttt{POINTER}(\tau)$. The fact that C does not have first-class functions is reflected in the irregular behaviour of function types. The rules do not allow function to be lvalues: The special identifier rule for function names produces an expression type rather than lvalue type and, as we will explore later, function pointers unlike other pointers cannot be turned into lvalues.

The standard distinguishes between lvalue and expression types to account for the asymmetric meaning of expressions in different contexts. An expression to the left of an assignment operator is meant to represent the memory location that is written to (hence the name *l*value).

$$[\textsc{assign-arith}]\ \frac{\begin{array}{c}\Gamma, \Phi \vdash e_1 : \texttt{lvalue}[\tau_1] \qquad \Gamma, \Phi \vdash e_2 : \tau_2 \\ \textit{is-modifiable}(\texttt{lvalue}[\tau_1]) \qquad \textit{is-arithmetic}(\tau_1) \qquad \textit{is-arithmetic}(\tau_2)\end{array}}{\Gamma, \Phi \vdash e_1 \texttt{ = } e_2 : \textit{unqualify}(\tau_1)}$$

In most other contexts, we are interested in the value and not the memory location of an object stored inside memory. We therefore must be able to convert an lvalue type to a regular expression type when it occurs, for example, to the right of an assignment

operator:

$$[\textsc{lvalue-conv}]\ \frac{\Gamma, \Phi \vdash e : \texttt{lvalue}[\tau]}{\Gamma, \Phi \vdash e : \mathit{unqualify}(\tau)}\ .$$

To ensure that our rules coincide with an implementation, the choice of the next rule in a type derivation must be strictly directed by the syntax of expressions. Observe that the above rule breaks with the syntax-directed style of the previous rules. It does not pose any restrictions on the form of expressions e. In practice, we avoid the issue by inlining the rule wherever necessary.

So far we have not mentioned the various auxiliary functions and predicates that occur in the previous two rules. We restrict our presentation to a very brief and informal description: Predicate *is-arithmetic*$(\cdot)$ is true for all integer types[3] and *is-modifiable*$(\cdot)$ is true for lvalue types that are neither **`const`**-qualified nor array types. Finally, *unqualify*(τ) is type τ but without any qualifiers.

The next topic of our overview is pointer types. Using the address operator `&`, we can turn any expression of lvalue type into a pointer, as witnessed by the following typing rule:

$$[\textsc{addr-lvalue}]\ \frac{\Gamma, \Phi \vdash e : \texttt{lvalue}[\tau] \qquad \mathit{is\text{-}object}(\tau)}{\Gamma, \Phi \vdash \texttt{\&}e : \texttt{POINTER}(\tau)}\ ,$$

where predicate *is-object*$(\cdot)$ is valid for all but function types. The standard does, however, allow function pointers but since we cannot have lvalues of function type, an extra rule is necessary, i.e.

$$[\textsc{addr-fun}]\ \frac{\Gamma, \Phi \vdash e : \tau \qquad \mathit{is\text{-}function}(\tau)}{\Gamma, \Phi \vdash \texttt{\&}e : \texttt{POINTER}(\tau)}\ .$$

The indirection operator $*$ is the inverse of taking the address of an object.

$$[\textsc{indir-pointer}]\ \frac{\Gamma, \Phi \vdash e : \texttt{POINTER}(\tau) \qquad \mathit{is\text{-}object}(\tau)}{\Gamma, \Phi \vdash *e : \texttt{lvalue}[\tau]}$$

Again, function types need a special case that disallows the creation of function objects.

$$[\textsc{indir-fun}]\ \frac{\Gamma, \Phi \vdash e : \texttt{POINTER}(\tau) \qquad \mathit{is\text{-}function}(\tau)}{\Gamma, \Phi \vdash *e : \tau}$$

We can now explain why speaking of "function pointers" is rather misleading. Indirection on expressions of type pointer to function is an idempotent operation: If the operand e of the indirection operator has function type τ, the operand is, as we noted above, implicitly converted to type $\texttt{POINTER}(\tau)$. Hence, expression e and, for example, $***e$ both have type τ. Thus, Norrish [Nor98, pp. 16f. and 23] distinguishes between normal and function pointers at a type level reducing the number of side-conditions and special cases. We decided against Norrish's approach in order to stay close to the standard.

Finally, we examine typing rules for integer arithmetic. We start with the unary `+`

[3] And floating types, but we do not model those.

operator.

$$[\text{UNARY-PLUS}]\ \frac{\Gamma, \Phi \vdash e : \tau \qquad \textit{is-arithmetic}(\tau)}{\Gamma, \Phi \vdash \texttt{+}e : \textit{promote}(\tau)}$$

Operationally, the unary `+` operator has no effect on the operand other than *promoting* its type. Integer promotion coerces an expression to **`signed int`**, if it has integer type whose range is a subset of the **`signed int`** type. The rationale is the following: An implementation is meant to make the range of **`signed int`** correspond to the register size of the target architecture such that arithmetic operations on expressions of type **`signed int`** map directly to machine instructions. Hence, integer promotions allow an implementation to use single machine instructions for smaller integer types, too, without having to worry about overflows.

Similar type coercions are part of the binary arithmetic operations. The typing rule for addition is for example:

$$[\text{ADD-ARITH}]\ \frac{\Gamma, \Phi \vdash e_1 : \tau_1 \qquad \Gamma, \Phi \vdash e_2 : \tau_2 \qquad \textit{is-arithmetic}(\tau_1) \qquad \textit{is-arithmetic}(\tau_2)}{\Gamma, \Phi \vdash e_1 \ \texttt{+}\ e_2 : \textit{usual-arithmetic-conversion}(\tau_1, \tau_2)}\ .$$

The resulting type, $\textit{usual-arithmetic-conversion}(\tau_1, \tau_2)$, is the *common* type of $\textit{promote}(\tau_1)$ and $\textit{promote}(\tau_2)$. The common type of two arithmetic types is formed according to a list of rules that have the purpose of selecting a type whose range (in most cases) subsumes that of both types. Unfortunately, the rules for determining a common type can have unexpected consequences when signed and unsigned integer types are used within the same operation, as we have already seen in Chapter 1.

3.3. Dynamic semantics

We have seen so far how our tool turns a given C program into a type-annotated abstract syntax tree. The next phase in the tool pipeline consists of analysing the dynamic semantics of a program: It transforms the type-annotated AST into a logical description of the program's runtime behaviour.

The development of our dynamic semantics involved by far the greatest number of design decisions. The result, unlike the previous stages, uses very little in the way of standard techniques and style, required most creativity, and is without a doubt the most novel part of this project.

3.3.1. General form

As we have outlined in Section 2.2.2, in order to cope with underspecification and non-deterministic behaviour, our dynamic semantics does not compute concrete values. Instead, it accumulates logical constraints and actions, including their ordering. More

formally, we define a record[4] type

$$\begin{aligned}\mathsf{meaning} \triangleq \big\{&\mathsf{actions} : \mathcal{P}(\mathsf{action});\\ &\mathsf{function\text{-}actions} : \mathsf{action} \to \mathcal{P}(\mathsf{action});\\ &\mathsf{sequenced\text{-}before} : \mathcal{P}(\mathsf{action} \times \mathsf{action});\\ &\mathsf{constraints} : \mathsf{constraint}\big\},\end{aligned}$$

where the label constraints is the crucial component. It is the logical representation of the meaning of an expression e. All other components have the purpose to make up for the lack of a global memory and to account for weakly ordered memory actions (c.f. 2.2.1).

The component actions is the set of all actions that occur in a particular execution of e, where an action is either a memory read, a write, an introduction of an object, an elimination of an object at the end of its lifetime, or a function call[5]:

$$\begin{aligned}\mathsf{action} ::= &\ \mathsf{Store}_{uid}(\tau, addr, \alpha) \mid \mathsf{Load}_{uid}(\tau, addr, id) \mid \mathsf{Modify}_{uid}(\tau, addr, \alpha, \alpha)\\ &\mid \mathsf{Create}_{uid}(\tau, id) \mid \mathsf{Kill}_{uid}(id)\\ &\mid \mathsf{Id}_{uid}(addr) \mid \mathsf{Same}_{uid}(addr, addr)\\ &\mid \mathsf{Call}_{uid}.\end{aligned}$$

Each action is given a unique label *uid* to make them distinct.

We have already seen an example of the component sequenced-before in Fig. 2.2: It is a non-reflexive, transitive order on the elements of actions (c.f. 2.2.1) encoding the evaluation order. The last component of the record, function-actions, is an auxiliary ordering constraint. It is a map from call actions to the set of actions that occur in the execution of the corresponding function call.

With the definition of meaning in place, we can describe the form of the relation that defines the dynamic semantics: Given a typing $\Gamma, \Phi \vdash e : \tau$ and a partial map, Σ, from program identifiers to symbolic memory locations, the meaning of e is a pair

$$[\![\Gamma, \Phi \vdash e : \tau]\!]_{\Sigma} : \mathsf{constant} \times \mathcal{P}(\mathsf{meaning}).$$

The first component of the pair allows us to refer to the value produced by expression e, where type constant is the set of constants in our logical framework (e.g. symbolic names, addresses or integers). Since a program often has more than just one behaviour, the second component is a set of meaning records. We will sometimes speak of m-sets to refer to sets of type $\mathcal{P}(\mathsf{meaning})$. Furthermore, see Table 3.1 for a list of metavariables and their corresponding types.

Above we have consciously avoided mentioning statements. Unlike expressions, statements do not produce values but otherwise there is little conceptual difference between

[4] We borrow the notion of records from programming languages, using $m.\mathsf{sequenced\text{-}before}$, for example, for projection.

[5] Modify corresponds to infix operations and id actions determine the identity of an lvalue.

$a, a_1, \ldots$	action
$\alpha, \alpha_1, \ldots$	constant
$c, c_1, \ldots$	constraint
$m, m_1, \ldots$	meaning
$M, M_1, \ldots$	$\mathcal{P}$(meaning)
$u, u_1, \ldots$	*uid*

Table 3.1.: List of metavariables

the two. Hence, the semantics of statements can be seen as a special case of the expression semantics:

$$[\![\Gamma, \Phi \vdash s : \texttt{void}]\!]_\Sigma : \mathcal{P}(\mathsf{meaning}).$$

3.3.2. Expression semantics

In this section, we present our dynamic semantics for expressions using a selection of rules that demonstrate how we resolve the issues related to evaluation order, undefined behaviour, implementation-defined behaviour, implicit type conversions, and the interaction with a memory model. We begin with the base cases for integer constants and identifiers.

Base cases

An integer constant n has, of course, no side-effects, no control-flow, and its value is unconditionally just n[6]:

$$[\textsc{const}]\ [\![\Gamma, \Phi \vdash n : \tau]\!]_\Sigma = (n, \mathsf{empty})\ ,$$

where empty is the singleton set

$$\begin{aligned}\mathsf{empty} \triangleq \Big\{\big\{&\mathsf{actions} = \{\}\,;\\ &\mathsf{compound\text{-}actions} = \{\}\,;\\ &\mathsf{function\text{-}actions} = \{\}\,;\\ &\mathsf{sequenced\text{-}before} = \{\}\,;\\ &\mathsf{constraints} = \mathsf{true}\big\}\Big\}.\end{aligned}$$

For other rules, we need to be able to add further constraints and actions to each element in a set M. Again, we borrow notation from programming languages: $\{m\ \mathit{with}\ \mathsf{constraints} = c\}$ is record m with component constraints replaced by c. Using this notation, adding a constraint is simply

$$m \wedge c \triangleq \{m\ \mathit{with}\ \mathsf{constraints} = m.\mathsf{constraints} \wedge c\}\,.$$

[6]For notational convenience, we will often omit the parentheses around pairs in future.

Similarly,

$$m \mathbin{\barwedge} a \triangleq \{m \ \mathit{with}\ \mathsf{actions} = m.\mathsf{actions} \cup a\}\,.$$

We lift both operations to the set level in the obvious way, i.e.

$$M \wedge c \triangleq \{m \wedge c | m \in M\}$$

and

$$M \mathbin{\barwedge} a \triangleq \{m \mathbin{\barwedge} a | m \in M\}\,.$$

In order to reduce the number of parentheses necessary, we will assume that operators $\otimes$ and $\overrightarrow{\otimes}$ (to be introduced later) are more tightly binding than $\wedge$ and $\mathbin{\barwedge}$ and that $\wedge$ and $\mathbin{\barwedge}$ associate to the left.

Now, we can specify the second base case. An identifier id, that does not designate a function, represents a memory object. In a non-lvalue context, it denotes the contents of the object, i.e. it loads the value residing at location $\Sigma(id)$ from memory:

$$[\textsc{id}]\ \frac{id \notin \Phi}{[\![\Gamma, \Phi \vdash id : \tau]\!]_\Sigma = \alpha, \mathsf{empty} \mathbin{\barwedge} \mathsf{Load}_u(\tau, \Sigma(id), \alpha)} \quad \alpha,\ u \text{ fresh}\ .$$

The exact value for the fresh name α will be provided later by the memory model in form of an additional constraint (e.g. $\alpha = 3$).

If, however, the identifier appears in an lvalue-context, it denotes the memory location $\Sigma(id)$. We use a separate functions, $[\![\Gamma, \Phi \vdash e : \tau]\!]_\Sigma$ and $[\![\Gamma, \Phi \vdash e : \tau]\!]_\Sigma^{lvalue}$, to represent the semantic split between value- and lvalue-producing contexts. In particular,

$$[\textsc{lvalue-id}]\ \frac{id \notin \Phi}{[\![\Gamma, \Phi \vdash id : \tau]\!]_\Sigma^{lvalue} = \Sigma(id), \mathsf{empty}}\ .$$

Integer arithmetic

Next we turn to inductive rules. Since our semantics is recursive in the structure of expressions, we often have to combine the meaning of two subexpressions. For instance, expression e_1+e_2 has exactly the actions of its subexpressions, it has at least the constraints of e_1 and e_2, and evaluation order is unrestricted, i.e. the actions of e_1 are not related to e_2 by the sequenced-before relation[7]. The following definition expresses this operation for records m_1 and m_2.

$$\begin{aligned} m_1 \otimes m_2 \triangleq \big\{ & \mathsf{actions} = m_1.\mathsf{actions} \cup m_2.\mathsf{actions}; \\ & \mathsf{compound\text{-}actions} = m_1.\mathsf{compound\text{-}actions} \cup m_2.\mathsf{compound\text{-}actions}; \\ & \mathsf{function\text{-}actions} = m_1.\mathsf{function\text{-}actions} \cup m_2.\mathsf{function\text{-}actions}; \\ & \mathsf{sequenced\text{-}before} = m_1.\mathsf{sequenced\text{-}before} \cup m_2.\mathsf{sequenced\text{-}before}; \\ & \mathsf{constraints} = m_1.\mathsf{constraints} \wedge m_2.\mathsf{constraints} \big\} \end{aligned}$$

[7] We will say *unsequenced* from here on.

As before, we lift the operation to sets of such records:

$$M_1 \otimes M_2 \triangleq \{m_1 \otimes m_2 | \, (m_1, m_2) \in M_1 \times M_2\} \, .$$

With these operators at our disposal, we can give the semantics for the (signed) addition of two integers.

$$[\text{ADD}] \quad \frac{\begin{array}{c} \mathit{is\text{-}arithmetic}(\tau_1) \qquad \mathit{is\text{-}arithmetic}(\tau_2) \qquad \mathit{is\text{-}signed\text{-}arithmetic}(\tau) \\ [\![\Gamma, \Phi \vdash e_1 : \tau_1]\!]_\Sigma = (\alpha_1, M_1) \qquad [\![\Gamma, \Phi \vdash e_2 : \tau_2]\!]_\Sigma = (\alpha_2, M_2) \end{array}}{[\![\Gamma, \Phi \vdash e_1 \texttt{ + } e_2 : \tau]\!]_\Sigma = \alpha, (M_1 \otimes M_2) \wedge (\alpha = \mathsf{conv}_\tau(\alpha_1) + \mathsf{conv}_\tau(\alpha_2) \wedge \neg \mathsf{range}_\tau(\alpha) \rightarrow \mathsf{undef})} \quad \alpha \text{ fres}$$

Since addition is overloaded (pointer arithmetic uses the same operator), we have to check first that both operands really are integers. Before the values of the two operands are added, the standard requires us to convert both operands to their common type but as we only operate on well-typed programs we know that τ is already the common type (c.f. Section 3.2.1) of τ_1 and τ_2. Here, we use the notation $\mathsf{conv}_\tau(\cdot)$ as a shorthand for the type conversion. In the case $\tau =$ `UNSIGNED INT`, for example, we would expand it to

$$\mathsf{conv}_{\texttt{UNSIGNED INT}}(\alpha) \triangleq \alpha \bmod \texttt{UINT\_MAX}.$$

Finally, if the result of the (usual mathematical) addition, is outside the range of integers that can be represented by signed type τ, then behaviour is undefined according to the standard. Again, $\mathsf{range}_\tau(\cdot)$ is a shorthand that in our actual implementation is replaced by a type-specific condition, e.g. $\texttt{INT\_MIN} \leqslant \alpha \wedge \alpha \leqslant \texttt{INT\_MAX}$. Term undef is a special nullary predicate that indicates undefined behaviour.

The corresponding rule for signed integer addition only differs in the treatment of overflows. Unsigned integer overflows do not exhibit undefined behaviour but instead the resulting value, α, is reduced modulo the largest integer representable by τ, e.g. `UINT_MAX`.

So far we have only looked at rules that do not impose any restrictions on evaluation order. In contrast, the logical-and operator uses a strict left-to-right evaluation strategy: In e_1&&e_2, the actions of e_1 are sequenced before those of e_2. Consequently, we can no longer use operator $\otimes$ to combine the m-sets of the two subexpressions as it assumes evaluation order to be unspecified. Hence, we define a non-commutative operator that inserts a sequenced-before dependency between each pair of actions (a_1, a_2) where a_1 occurs in m_1 and a_2 in m_2:

$$\begin{aligned} m_1 \overrightarrow{\otimes} m_2 \triangleq \big\{ & \mathsf{actions} = m_1.\mathsf{actions} \cup m_2.\mathsf{actions}; \\ & \mathsf{compound\text{-}actions} = m_1.\mathsf{compound\text{-}actions} \cup m_2.\mathsf{compound\text{-}actions}; \\ & \mathsf{function\text{-}actions} = m_1.\mathsf{function\text{-}actions} \cup m_2.\mathsf{function\text{-}actions}; \\ & \mathsf{sequenced\text{-}before} = m_1.\mathsf{actions} \times m_2.\mathsf{actions} \\ & \quad \cup \left(m_1.\mathsf{sequenced\text{-}before} \cup m_2.\mathsf{sequenced\text{-}before}\right); \\ & \mathsf{constraints} = m_1.\mathsf{constraints} \wedge m_2.\mathsf{constraints} \big\}. \end{aligned}$$

Lifting to m-sets, gives us

$$M_1 \overrightarrow{\otimes} M_2 \triangleq \left\{ m_1 \overrightarrow{\otimes} m_2 \,\middle|\, (m_1, m_2) \in M_1 \times M_2 \right\}.$$

Recall from the definition in Fig. 2.1 that C's logical-and operator also has non-trivial control-flow: The second expression is only executed if the first expression does not evaluate to 0. Fortunately, our choice of semantic style makes branching control-flow a non-issue: each element of an m-set represents a (potentially) distinct execution path. The m-set associated with $e_1 \&\& e_2$ can therefore be described as the union of two m-sets, where the first,

$$M_1 \wedge \alpha_1 = 0 \wedge \alpha = 0,$$

corresponds to the case when only e_1 is evaluated and the second m-set,

$$M_1 \overrightarrow{\otimes} M_2 \wedge \alpha_1 \neq 0 \wedge \alpha = (\alpha_2 \neq 0 \to 1|0),$$

represents the case when both subexpressions are evaluated:

[LAND]
$$\frac{[\![\Gamma, \Phi \vdash e_1 : \tau_1]\!]_\Sigma = (\alpha_1, M_1) \qquad [\![\Gamma, \Phi \vdash e_2 : \tau_2]\!]_\Sigma = (\alpha_2, M_2) \qquad \alpha \text{ fresh}}{[\![\Gamma, \Phi \vdash e_1 \ \texttt{\&\&}\ e_2 : \tau]\!]_\Sigma = \alpha, M_1 \wedge \alpha_1 = 0 \wedge \alpha = 0 \cup M_1 \overrightarrow{\otimes} M_2 \wedge \alpha_1 \neq 0 \wedge \alpha = (\alpha_2 \neq 0 \to 1|0)}$$

We use the notation[8] $(p \to \alpha_1 | \alpha_2)$ to say "if p holds then α_1 else α_2".

Assignments

Lvalue-contexts occur in the simple and compound assignment, pre- and postfix increment, and the address operator but, here, we only present the rule for the simple assignment operator (on arithmetic types). The expression to the left of the assignment must evaluate to an lvalue. We therefore use $[\![\Gamma, \Phi \vdash e_1 : \tau_1]\!]_\Sigma^{lvalue}$ in the premise of the assignment rule instead of the usual value-producing semantic function:

[ASSIGN]
$$\frac{\begin{array}{cc} \mathit{is\text{-}arithmetic}(\tau_1) & \mathit{is\text{-}arithmetic}(\tau_2) \\ [\![\Gamma, \Phi \vdash e_1 : \tau_1]\!]_\Sigma^{lvalue} = (\alpha_1, M_1) & [\![\Gamma, \Phi \vdash e_2 : \tau_2]\!]_\Sigma = (\alpha_2, M_2) \end{array}}{[\![\Gamma, \Phi \vdash e_1 \ \texttt{=}\ e_2 : \tau]\!]_\Sigma = \alpha, (M_1 \otimes M_2) \overrightarrow{\otimes} (\mathsf{empty} \wedge \mathsf{Store}_u(\tau, \alpha_1, \alpha)) \wedge \alpha = \mathsf{conv}_\tau(\alpha_2)} \quad \alpha,\ u \text{ fresh}$$

Hence, α_1 in the above rule denotes a symbolic memory location and can be used as the address argument of the load action.

Also note that, according to the standard, the actions of the two subexpressions are unsequenced with respect to each other, i.e.

$$M_1 \otimes M_2.$$

[8] Following Gordon's lectures notes on 'Specification and verification I': `http://www.cl.cam.ac.uk/~mjcg/Teaching/SpecVer1/Notes/Notes.pdf`.

They are both, however, sequenced before the write to memory and therefore

$$(M_1 \otimes M_2) \overrightarrow{\otimes} (\mathsf{empty} \curlywedge \mathsf{Store}_u(\tau, \alpha_1, \alpha)) .$$

Pointers and pointer arithmetic

The ability to take the address of any object is essential to the character and spirit of C. The lack of restrictions in taking addresses (e.g. returning the address of a local variable is not a compile-time error) and pointer arithmetic distinguishes C from many other programming languages and is a very common cause of security-relevant software bugs.

The rules for the two main pointer operations, the address and indirection operators, are closely related. The address operator determines the memory location of the object denoted by its operand:

$$[\text{ADDR}]\ \frac{[\![\Gamma, \Phi \vdash e : \tau]\!]^{lvalue}_{\Sigma} = \alpha, M \qquad \neg\exists e'.e \equiv *e'}{[\![\Gamma, \Phi \vdash \&e : \tau]\!]_{\Sigma} = \alpha, M} .$$

The indirection operator can be seen as the inverse of the address operator. It turns its operand which must be of pointer type into an lvalue:

$$[\text{INDIR-LVALUE}]\ \frac{[\![\Gamma, \Phi \vdash e : \tau]\!]_{\Sigma} = \alpha, M}{[\![\Gamma, \Phi \vdash *e : \tau]\!]^{lvalue}_{\Sigma} = \alpha, M \wedge (\alpha = \mathsf{null} \vee \neg\mathsf{align}_{\tau}(\alpha)) \rightarrow \mathsf{undef}} .$$

In fact, the standard states that expression `&*`e is equivalent to e even if e evaluates to a null pointer. Hence, the rule [ADDR] excludes the case when e is an indirection operator. The indirection rule also expresses that behaviour is undefined if either the operand evaluates to a null pointer or the pointer has the wrong alignment.

We conclude the section with a short discussion of pointer arithmetic.

$$[\text{ADD-PTR}]\ \frac{\begin{array}{cc}\tau = \mathsf{POINTER}(\tau_1) & \mathit{is\text{-}integer}(\tau_2) \\ [\![\Gamma, \Phi \vdash e_1 : \tau]\!]_{\Sigma} = \alpha_1, M_1 & [\![\Gamma, \Phi \vdash e : \tau_2]\!]_{\Sigma} = \alpha_2, M_2\end{array}}{[\![\Gamma, \Phi \vdash e_1\texttt{+}e_2 : \tau]\!]_{\Sigma} = \alpha, M_1 \otimes M_2 \wedge \alpha = \alpha_1 + \alpha_2 * \mathsf{sizeof}_{\tau_1} \curlywedge \mathsf{Same}_u(\alpha_1, \alpha)} \quad u \text{ fresh.}$$

The rule above should look fairly unsurprising to C programmers: The result is the sum of the address denoted by the pointer and the product of integer α_1 and the size of type τ_1 in bytes, i.e. sizeof_{τ}. However, the rule also contains a memory action $\mathsf{Same}(\alpha_1, \alpha)$ that we have not encountered before. We need the action to express a less-known detail of pointer arithmetic: If the pointer e_1 and the resulting pointer e_1`+`e_2 do not point into (or to a region starting at the end of) the same memory object, then behaviour is undefined. As a result, the following program has, perhaps unexpectedly, undefined behaviour.

```
#include <stdio.h>

int main(void) {
  int a[] = {1,2,3};
  int *pa = &a[2];
```

```
  int sum = 0;
  while (pa > a) {
    sum += *pa;
    pa = pa - 1;
  }
  printf ("%d\n", sum);
  return 0;
}
```

In the last iteration of the loop, `pa` points to the first element of the array but `pa - 1` no longer points into the array (or just past the end of it). Hence, behaviour is undefined.

In general, it is undecidable whether the property holds for two particular pointers. Our solution is to defer the decision to the memory model by introducing a special "memory action". Once a particular ordering of memory actions has been fixed, the property can be easily checked.

3.3.3. Statement semantics

The semantics of statements generally follows the form of the expression semantics. We therefore only present the rule for blocks to introduce the concept and our treatment of *lifetime* but leave out the more complicated rules for the if statement, loops, and the **`continue`** and **`break`** statements.

Blocks

Syntactically, blocks are lists of statements and declarations enclosed by curly brackets. They are operationally important as they control the lifetime of objects created by a declaration within the list: The lifetime of such an object already begins with entry to the block and not, as one might expect, when execution reaches the declaration.

Our abstract syntax for blocks includes a (typed) list of the identifiers to be introduced in the block, so we can announce the start and end of lifetime of each object to the memory system and extend Σ with a fresh symbolic memory location for each object.

$$\text{[BLOCK]}\ \frac{\begin{array}{c}\Sigma' = \Sigma \cup \{id_1 \mapsto v_1, \ldots, id_m \mapsto v_m\} \\ M = \mathsf{empty} \barwedge \mathsf{Create}_{u_1}(\tau_1, v_1) \barwedge \ldots \barwedge \mathsf{Create}_{u_m}(\tau_m, v_m) \\ M' = \mathsf{empty} \barwedge \mathsf{Kill}_{u_{m+1}}(v_1) \barwedge \ldots \barwedge \mathsf{Kill}_{u_{2m}}(v_m) \\ [\![\Gamma, \Phi \vdash s_1 : \mathtt{void}]\!]_{\Sigma'} = M_1 \qquad \ldots \qquad [\![\Gamma, \Phi \vdash s_n : \mathtt{void}]\!]_{\Sigma'} = M_n\end{array}}{[\![\Gamma, \Phi \vdash \{s_1 \ldots s_n\}_{id_1:\tau_1,\ldots,id_m:\tau_m} : \mathtt{void}]\!]_{\Sigma} = M \overrightarrow{\otimes} M_1 \overrightarrow{\otimes} \ldots \overrightarrow{\otimes} M_n \overrightarrow{\otimes} M'}\quad \begin{smallmatrix} v_1,\ldots,v_m \\ u_1,\ldots,u_{2m}\end{smallmatrix} \text{ fresh.}$$

Note that the rule also correctly expresses, by using the $\overrightarrow{\otimes}$ operator, that the list of statements is executed in order.

3.3.4. Interpreting the standard

Before we move on, we give two demonstrative examples that show how difficult it can be to extract a precise definition from the standard.

Alignment Different computer architectures have various restrictions on the addresses where data can be stored, e.g. MIPS requires 32-bit data to be stored only at word-aligned memory addresses. To account for this, the standard has a notion of address alignment. Unfortunately, it has two conflicting definitions of the term! The first defines it to be a 'requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address' [Jon10, p. 3]. But the second specifies that it 'is an [...] integer value representing the number of bytes between successive addresses at which a given object can be allocated' [Jon10, p. 48]. The first matches our intuitive understanding of alignment, whereas the second is strictly too weak to express the constraint: It only defines how much space must be between two contiguously allocated objects but not where an object is allowed to start. Thus, we interpret alignment to be the former.

Sequence points The current ISO standard does not yet use the sequenced-before relation to order side effects. It uses instead the concept of a sequence point, which gives the following program well-defined behaviour.

```
int main() {
  int i = 0;
  return i = (i++, i-1);
}
```

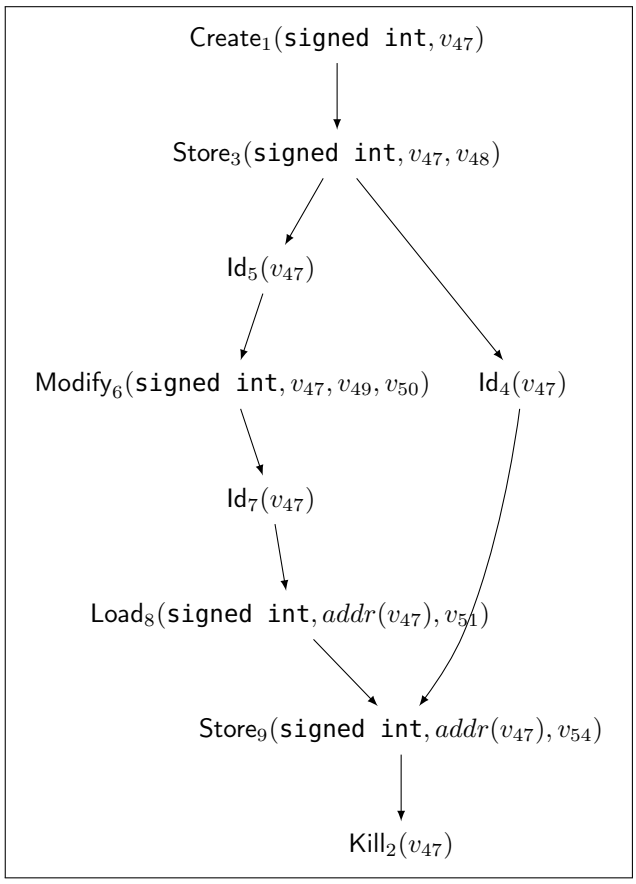
The newly introduced sequenced-before relation is meant to be a clearer but equivalent mechanism to order actions. However, examining the ordering of action of the program, we notice an unsequenced race between the Modify and the Id action. Hence, behaviour is undefined. We choose to follow the definition of the latest draft but since the standard committee intends to keep the sequencing behaviour unchanged, this change might be an error in the draft.

3.3.5. Memory

The rules of the previous sections interact with memory by accumulating actions and ordering constraints but we still have to create an explicit model of memory in order to determine, for example, the outcome of a particular execution or to determine whether a program has undefined behaviour.

Map of values or array of bytes?

The first decision we have to make concerns what kind of information is stored in memory. The standard states that each value stored in memory is represented as a sequence of



bytes, which is called *object representation*. A program can use **`unsigned char`** pointers to examine the object representation of a value byte-by-byte and therefore our semantics must take object representations into account. Thus, we could view memory as a map from addresses to bytes.

Unfortunately, the memory-as-byte-array view has serious disadvantages. According to the standard, there is no function that relates values to object representations since a value can have several distinct object representations. We can only rely on the (implementation-defined) inverse, which is a surjective function from object representations to values. The lack of a bijection between the two representations makes the memory-as-byte-array model impractical.

Our semantics takes an alternative view: memory is represented as map from symbolic location to values and we only revert to the object representation of a value when strictly necessary. In comparison, the choice has the drawback that pointer operations are not quite as straightforward to model as before. However, most memory operation, e.g. `a = 1`, are substantially simplified.

Enumerating all orderings of memory actions

Our dynamic semantics describes the meaning of C programs as a collection of m-sets. Each m-set corresponds to a distinct path through the program and is comprised of a

set of constraints, a set of memory actions, and the sequenced-before relation. However, m-sets do not directly give us the information we want. For example, we would like to check whether a program has undefined behaviour or what the set of possible return values is.

To retrieve this information, we need to symbolically execute the memory actions and, in turn, complete the constraint set with the information gained from symbolic execution. Since different orderings of actions potentially lead to different behaviour, we have to, in general, consider every permutation of actions that satisfies the sequenced-before relation.

The naive algorithm of generating every permutation and checking whether it is in accordance with the sequenced-before relation would be very wasteful. Instead, we exploit a number of observations to arrive at a more conservative algorithm.

(i) According to the standard, actions of one function call are not allowed to overlap with those of another call. Hence, we can consider each function call separately.

(ii) Only the relative ordering of load, store, modify and call actions can change the end result, so we do not have to permute any other unsequenced actions.

(iii) If two unsequenced actions cause a data races, behaviour is undefined unless they are separated by an indeterminately sequenced function call. Thus, when we have a number of load, store, or modify actions that are unsequenced with respect to each other (but no such call action), we can choose any ordering. We only need to include a constraint expressing that, if any two actions, where one is a write or a modify action, operate on the same address, then predicate undef is true.

These insights almost directly lead to an algorithm that produces the minimal set of memory orderings that potentially have different behaviour assuming that we cannot distinguish the memory addresses that the actions operate on, which often will still be unknown at this point. We omit describing the exact operation of the algorithm since it is rather technical.

Executing a trace

Once all traces that satisfy the sequenced-before relation have been found, we need to execute each trace. The general process is conceptually fairly straightforward.

Actions model the interactions of a program with memory. We represent memory as a partial map from symbolic memory locations to (typed) objects, where an object can either be a scalar, i.e. containing just one value, or an array of values.

Given a trace and a constraint set, we execute each action of the trace in order, where our memory is initially just the empty map. If the next action is, for example, $\mathsf{Create}_3(\mathtt{int}, s_2)$, we add a new mapping, $s_2 \mapsto \mathsf{Scalar}(\mathtt{int}, \mathsf{Indet})$, to memory. Note that the value of the object is initially indeterminate as indicated by the abstract value Indet. If we subsequently encounter a load from address s_2 without an intermediate write, then behaviour is undefined and, thus, we add the predicate undef to the constraint set.

Executing a load action, e.g. $\mathsf{Load}_7(\mathtt{int}, s_4, v_5)$, is more involved: First, we have to check whether s_4 is a valid memory address. If not, behaviour is undefined. Otherwise, we retrieve the corresponding object from memory and verify that type `int` is compatible with the type of the object. If not, behaviour is undefined. Otherwise, we examine the value v stored in the object. If it is Indet, behaviour is undefined. Otherwise, we add $v_5 = v$ to the constraint set.

Unfortunately, the address, e.g. s_4, will often not be a symbolic location but a variable, whose specific value has to be inferred from the constraint set. We need a constraint solving procedure that allows us to translate variables to ground terms.

3.4. Constraint solving

We have already seen that equational reasoning is necessary to execute traces. If we restrict equational logic to ground terms, congruence closure algorithms give us a complete decision procedures. Congruence closure is most readily understood by comparison with the union-find problem. A union-find algorithm maintains an equivalence relation, $\sim$, induced by a list of pairs (t_i, s_i) such that $t_i \sim s_i$. Similarly, a congruence closure algorithm maintains a congruence relation $\approx$ induced by a list of pairs (t_i, s_i) such that $t_i \approx s_i$ satisfying, as before, reflexivity, symmetry, and transitivity but also the congruence property, i.e.

$$[\textsc{congruence}]\ \frac{\Delta \vdash s_1 \approx t_1 \qquad \ldots \qquad \Delta \vdash s_n \approx t_n}{\Delta \vdash f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n)}\ .$$

For example, given the formula $a = b \wedge f(a) = e \wedge f(b) = d$, a congruence closure algorithm must be able to infer that $e = d$: By congruence, $a = b$ implies $f(a) = f(b)$ and transitivity gives us $e = d$.

We implement a modified, persistent version of Nieuwenhuis and Oliveras' 'fast congruence closure' algorithm [NO07], which has best known time complexity $O(n \log n)$. Our version works on top of a persistent variant of Tarjan's classic union-find algorithm and uses persistent arrays. Both are due to Conchon and Filliâtre [CF07]. Our modification has the benefit that it can be easily used within a backtracking solution procedure.

A congruence closure algorithm alone is only powerful enough to decide conjunctions of equation but the logic of constraint sets also includes disjunctions, implications, various integer operations, and integer comparisons. We could extend the decision procedure to disjunctions and implications by either translating input formulae into disjunctive normal form or use a satisfiability solver to perform semantic case splits. Inspection of our dynamic semantics and the trace execution shows that disjunctions and implications are only used in constraints describing implementation-defined behaviour and the occurrence of undefined behaviour. The former involves implicit universal quantification, for which far more proof power would be necessary, and the latter is irrelevant for inferring memory addresses. Thus, there is no need to extend the decision procedure.

We combine the congruence closure algorithm with simple term rewriting techniques.

Using the information collected by the congruence closure algorithm, we rewrite variables to equivalent ground terms and check equalities and disequalities. Furthermore, we evaluate constants, e.g. $5+2$, and evaluate predicates, e.g. $4 < 3$, that no longer contain any variables. We also have specialised rules that remove unnecessary comparisons due to type conversions and overflow checks, e.g. $i \leqslant$ `INT_MAX`, when i is known to be smaller than or equal to minimal value of `INT_MAX` allowed by the standard.

In practice, our approach turns out to be powerful enough to tackle all tested programs free of implementation-defined behaviour. It is not clear how one should proceed with implementation-defined programs: We choose to return a partially evaluated m-set if constraint solving fails. Another approach would be to encode trace execution in a more powerful logic but the benefits are not obvious.

3.5. Software design

Throughout the implementation chapter we have presented the workings of our tool in the form of simplified and stylised inference rules. In this last section we give an overview of the tool design and demonstrate how closely our code corresponds to the typeset rules.

The general structure is very similar to a compiler pipeline. In fact, the first few phases of our tool are commonly found in compilers: lexing, parsing, scope analysis and identifier disambiguation, removal of syntactic sugar, translation to a common intermediate language, and type checking and annotating the AST with type information. The subsequent phases, the semantic analysis, enumeration of all traces of memory action, and constraint solving, are, however, highly non-standard. The reason is that our target language of m-sets bears little resemblance to the machine languages compilers usually generate.

Above we speak of a pipeline structure because each phase is, in abstract terms, a function from some type t_i to t_{i+1} and the next a function from t_{i+1} to t_{i+2}, etc. Thus, at the highest level of code, we use the *pipeline* function [SGC+07, p. 45]

```
let (|>) x f = f x : 'a -> ('a -> 'b) -> 'b
```

to combine the different phases, i.e.

```
...
      file_name
      |> Input.from_file
      |> Lexer.make
      |> Parser.parse
      |> CabsToAil.desugar "main"
      |> AilTyping.annotate
      ...
```

Each such phase is contained in a separate module. We followed the principle that a each module should only implement a single task. To enforce this separation of concerns, we made extensive use of module signatures and abstract types to seal the module definition and to prevent implementation details of the module leaking into unrelated

code.

In the following we focus on the type-checking and semantic analysis phases. For each, we show code excerpts and put them in relation to rules presented earlier.

Function `check_exp` in Fig. 3.4 computes the type of a given expression `exp` whose subexpressions have already been annotated with types. The excerpt implements the [INDIR-POINTER] and [INDIR-FUNCTION] rules from section 3.2.1. Since the indirection operator expects a non-lvalue, the inlined lvalue conversion rule (c.f. inference rule [LVALUE-CONV]) is applied to operand e as part of function `f`: The function retrieves the type annotation (`AA.type_of`), performs, if necessary, lvalue conversion (`AT.lvalue_convert`), and turns array and function types into pointer types (`AT.pointer_convert`). If the type of e after appropriate conversions is a not a pointer type, the expression is

```
let rec check_exp env exp =
  ...
  let f e =
    let t = match AA.type_of e with
      | A.Exp    t -> t
      | A.Lvalue t -> AT.lvalue_convert t in
    AT.pointer_convert t in
  ...
  match exp with
  ...
  | A.UNARY (A.INDIRECTION, e) ->
      let msg = "Violation of constraint 6.5.3.2 #2 Address and indirection \
                 operators, Constraints: \'The operand of the unary * \
                 operator shall have pointer type.\'" in
      let t = f e in
      if AT.is_pointer t then
        if AT.is_pointer_to_object t then
          A.Lvalue (AT.base_of_pointer t)
        else A.Exp (AT.base_of_pointer t)
      else invalid msg
  ...
```

Fig. 3.4.: Type-checking expression $*e$

ill-typed and we produce an error message quoting the corresponding paragraph from the standard. Otherwise we make sure to return an expression type instead of an lvalue type if the operand is a function.

The code implementing the dynamic semantics is even closer to the typeset version. The OCaml datatype below is practically identical to the record type meaning given in Section 3.3.1.

Similarly, our code uses the same set-based operations to combine such records, e.g. Fig. 3.6 is the OCaml equivalent of the $\overrightarrow{\otimes}$ operator. The only difference lies in the representation of constraints: In the typeset version the conjunction of two logical formulae

```
type meaning = {
  actions : action Set.t;
  seq_before : (action * action) Set.t;
  fs_actions : (action, action Set.t) Map.t;
  constraints : AC.t
}
```

Fig. 3.5.: The OCaml record type corresponding to `meaning` defined in Section 3.3.1

```
let and_meanings_sb m1 m2 = {
  actions = Set.union m1.actions m2.actions;
  seq_before =
    begin
      let sb = Set.product m1.actions m2.actions in
      Set.union sb (Set.union m1.seq_before m2.seq_before)
    end;
  fs_actions = Map.union m1.fs_actions m2.fs_actions;
  constraints = AC.union m1.constraints m2.constraints
}
```

Fig. 3.6.: Implementation of $m_1 \overrightarrow{\otimes} m_2$, where `Set.product` is the Cartesian product

is a purely syntactic operation but our code must use an explicit data structure, namely sets.

Our last code sample, Fig. 3.7, implements rule ADD: `reduce_exp` is the equivalent of $[\![\Gamma, \Phi \vdash e : \tau]\!]_\Sigma$, where the function argument `env` corresponds to Σ. The typed-related premises of rule [ADD] are translated to pattern matching guards (e.g. `when AT.is_arithmetic ...`). And the inductive premises on expressions `e1` and `e2` are expressed by function `conv`, which also implements the necessary type conversions. The main thing to notice is, however, how close the resulting value of the function, `a, m1 <&> m2 <&- sum <&- overflow` (where `<&>` stands for $\otimes$ and `<&-` for $\wedge$), is to that of the inference rule.

To give an idea of the size of the implementation, we conclude the chapter with a few rough source line counts (excluding signature files): Parsing required about 1000, translation into abstract syntax 500, type checking 1000, semantic analysis 1500, and constraint solving 750 lines of OCaml code. Additionally, around 500 lines of source code are shared between the individual modules.

```
and reduce_exp env file exp =
  ...
  let conv e = ATC.conv (AA.exp_type_of exp) (reduce_exp env file e) in
  ...
  match AA.exp_of exp with
  ...
  | A.BINARY (Cabs.ARITHMETIC Cabs.ADD, e1, e2)
      when AT.is_arithmetic (AA.exp_type_of e1)
        && AT.is_arithmetic (AA.exp_type_of e2)
        && AT.is_signed_integer(AA.exp_type_of exp) ->
      let a1, m1 = conv e1 in
      let a2, m2 = conv e2 in
      let a = AC.fresh_name () in
      let sum = AC.eq a (AC.plus a1 a2) in
      let overflow =
        AC.implies
          (AC.neg (ATC.in_range (AA.exp_type_of exp) a))
          AC.undef in
      a, m1 <&> m2 <&- sum <&- overflow
```

Fig. 3.7.: The dynamic semantics of e_1+e_2 (c.f. rule [ADD] in Section 3.3.2)

4. Evaluation

In this chapter, we examine how the tool stacks up against the success criteria of our project proposal and analyse its capabilities and limitations and also take a look at its performance using a range of selected programs.

The fundamental problem with evaluating a project whose principal goal it is to demonstrate that C can be fully formalised, is lack of a litmus test: We obtain a formal definition from interpreting the standard which sometimes, as we have seen in the previous chapter, involves making unverifiable choices that will not be universally accepted. The best we can do is to compare our interpretation with those of others. Specifically, we compare against the results produced by compilers for programs that exhibit interesting behaviour.

4.1. Success criteria

The original proposal declared that, to be deemed successful, our project would have to satisfy the following two requirements.

(i) The tool accepts C programs restricted to integer expressions and generates graphs presenting the standard-conforming traces.

(ii) For relevant programs from GCC's C test suite, the set of results produced by the tool subsumes the behaviour of the compiled program. Any discrepancy must be explained.

As we will see later, our tool goes far beyond what the first half of Criterion 1 demands: It parses C programs, detects ill-typed programs and is able to produce meaningful result for not just integer expressions but also for programs with pointers, pointer arithmetic, fixed-length arrays, (recursive) function calls, and loops containing continue and break statements. All of these features were implemented on top of what the proposal required giving a far more complete picture of C than originally planned.

The second half of Criterion 1 is also met: The various graphs in the following sections visualising the sequenced-before relation have been automatically generated by our tool.

Criterion 2 demands that we test our tool on C programs selected from GCC's C language test suite. The test suite is comprised of a large number of regression tests. Most of them test GCC-specific features that are not part of the standard, compiler flags, or platform-specific miscompilations. As the test suite contains very few relevant as well as interesting programs, we mainly used test programs due to John Regehr, whose work on automated testing of C compilers has revealed hundreds of compiler bugs [YCER11]. These programs are particularly interesting since some compilers do get them wrong.

4.2. Capabilities

We begin with the following program.

```
int foo (int *p1, int *p2) {
  return (*p1)++ % (*p2)++;
}

int main (void) {
  int a = 1;
  return foo (&a, &a);
}
```

It is taken straight from a blog entry by John Regehr[1] on the potential benefits of an executable semantics for C. The program has a whole range of possible sources of undefined behaviour: The pointers `p1` and `p2` must both be valid and correctly aligned, the result of `*p2` must not be zero, the increment operators must not produce overflows, and `p1` and `p2` must point to disjoint memory regions. Otherwise behaviour is undefined.

The last condition in the list is not immediately obvious and requires more explanation. Consider the graph Fig. 4.1 generated by our tool. It shows the sequenced-before relation on memory actions in function `foo`. An execution of the function is allowed to choose any order of side effects that satisfies the sequenced-before relation. In particular, the two modify actions, 15 and 19, can appear in either order, since they are unsequenced. It follows from the discussions of the previous chapter that behaviour is undefined if both actions operate on the same address, i.e. $v_{52} = v_{56}$.

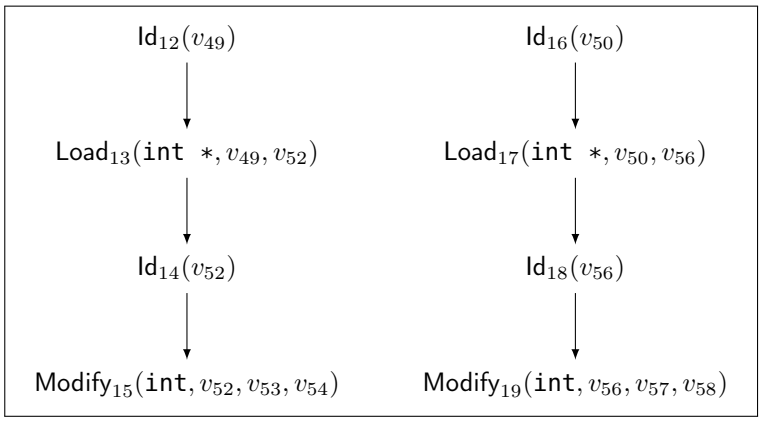


Fig. 4.1.: Ordering of actions in function `foo`

Our algorithm enumerating action traces detects that the modify actions are unsequenced and therefore adds a constraint

$$v_{52} = v_{56} \rightarrow \mathsf{undef}$$

[1] J. Regehr, *An Executable Semantics For C Is Useful*, `http://blog.regehr.org/archives/523`.

to the set of constraints. Later, when each trace is (symbolically) executed, the tool (c.f. store actions 6 and 9 in 4.2) stores the symbolic address v_{47} in the memory locations v_{49} and v_{50} for the function arguments `p1` and `p2`, respectively. Subsequently, function `foo` retrieves the value of each argument from memory (load actions 13 and 17) yielding two further constraints: $v_{52} = v_{47}$ and $v_{56} = v_{47}$. The congruence closure algorithm is then able to infer that, by transitivity, we also must have $v_{52} = v_{56}$. Finally, our constraint simplification algorithm uses the new information to rewrite the constraint

$$v_{52} = v_{56} \rightarrow \mathsf{undef}$$

to undef.

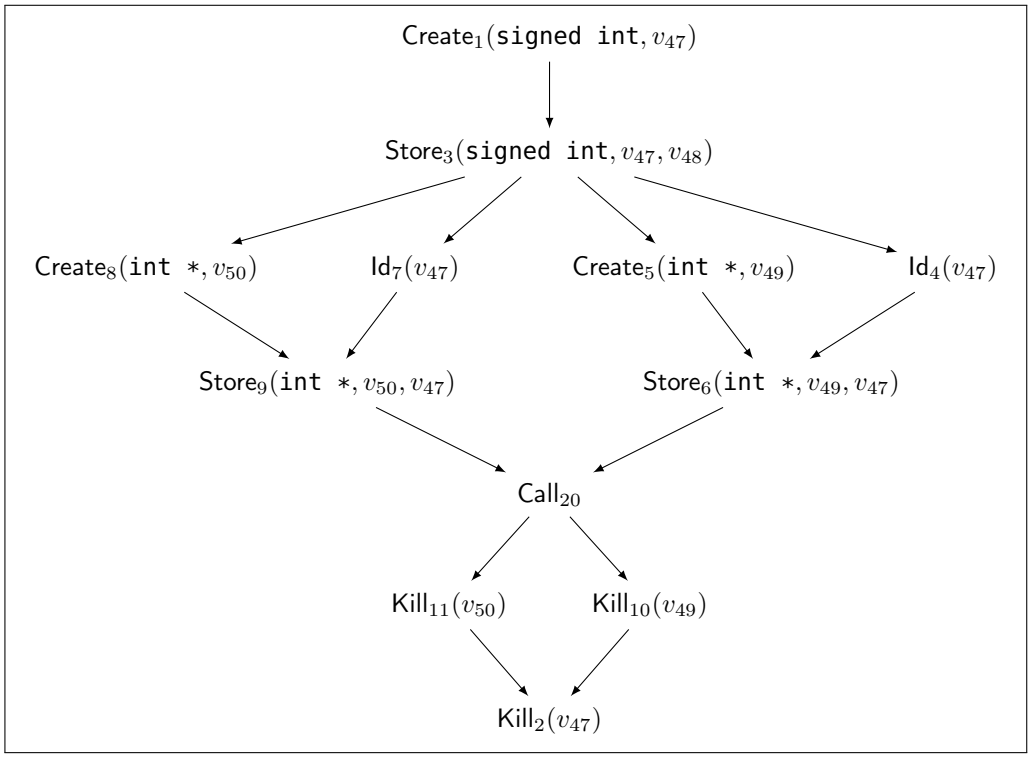


Fig. 4.2.: Ordering of actions in function `main`

Since the constraint set is still satisfiable after all actions have been executed, i.e. the set does not correspond to an infeasible path, we can conclude that the program has undefined behaviour. After the process described above, our tool therefore outputs the following fully simplified constraint set:

```
/\ {
  UNDEFINED = 1,
  v1_48 = 1,
  v1_52 = addr (v1_47),
  v1_53 = 1,
  v1_54 = 2,
```

```
  v1_55 = 1,
  v1_56 = addr (v1_47),
  v1_57 = v1_54,
  v1_58 = 3,
  v1_59 = 2,
  v1_60 = v1_51,
  v1_60 = 1,
  v1_61 = return,
  v1_61 = 1
}
```

Even though the program is very small, the analysis necessary to reliably detect the unsequenced race that results in undefined behaviour is surprisingly involved. Which may also be why 'very few tools for analyzing C code find this error' as Regehr remarks referring to his example program.

Next, we briefly look at two variations of the above program demonstrating that our tools models C's underspecified evaluation order correctly. The first programs splits the increment operations into separate function calls but otherwise preforms the same work.

```
int f(int *px) {
  return (*px)++;
}

int g(int *px) {
  return (*px)++;;
}

int main(void) {
  int x = 1;
  return f(&x) % g(&x);
}
```

Indeed, the ordering of memory events in function `main` (see Fig. 4.3) is almost exactly the same as before. The only difference is the additional call action which is unsequenced with respect to the other call action. Since each function call contains a modify action on the location of `x`, one might naively expect this program also to have undefined behaviour but, as discussed before, the standard states that the actions of a function call are indeterminately sequenced with respect to the actions of the callee[2]. In other words, the two modify actions are not unsequenced, they can be executed in any order and behaviour is still defined. Correspondingly, our tool indicates that the program is free of undefined behaviour.

```
/\ {
  ...
  v1_51 = addr (v1_47),
  ...
  v1_58 = addr (v1_47),
  ...
  v1_63 = return,
```

[2]Actions of one function cannot not overlap with those of another function.

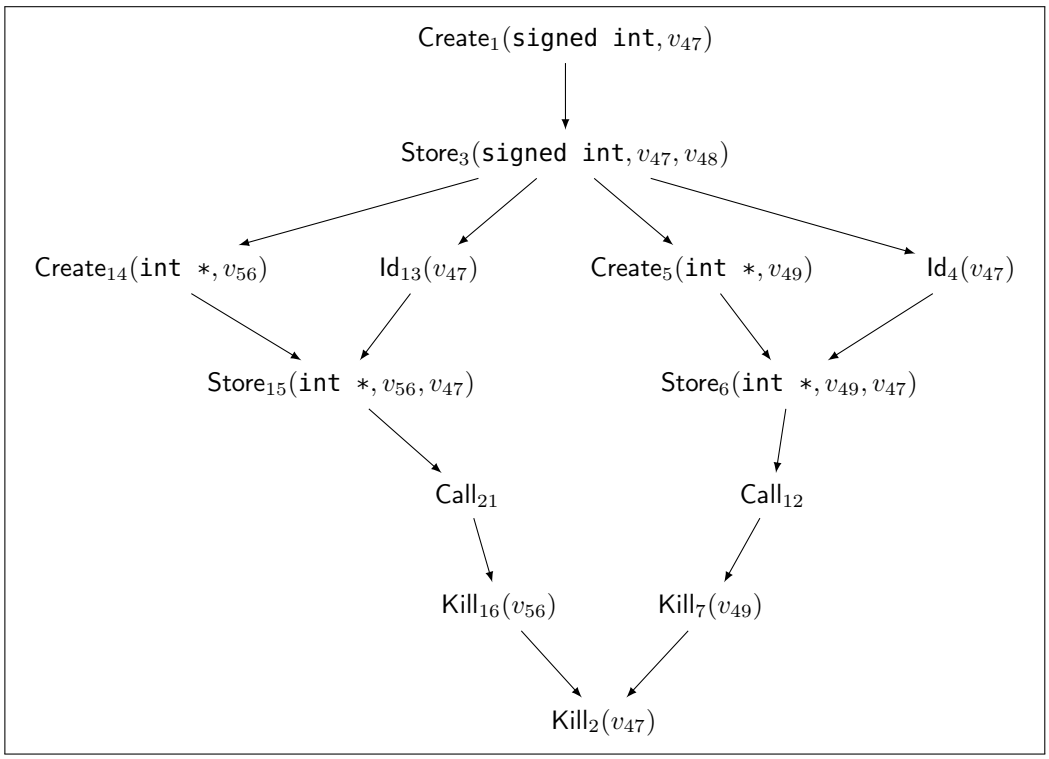


Fig. 4.3.: Sequenced-before relation for actions in function `main`

```
  v1_63 = 0
}
```

Furthermore, due to the sequenced data race the tool detects a second distinct, well-defined behaviour for the program with return value one instead zero.

```
/\ {
  ...
  v1_51 = addr (v1_47),
  ...
  v1_58 = addr (v1_47),
  ...
  v1_63 = return,
  v1_63 = 1
}
```

The second variant of Regehr's program only moves one of the increment operators into a function call.

```
int f(int *p) {
  return (*p)++;
}

int main(void) {
  int a = 1;
  int *pa = &a;
  return f(pa) % a++;
```

```
}
```

As before, the actions of the call to function `f` are indeterminately sequenced with respect to the actions occurring in the main function. However, the increment operator gets special treatment: An implementation is not allowed to split the load and store actions associated with an increment across a function call. We should observe exactly two distinct return values identical to those of the previous program. Our tool produces the simplified constraint

```
/\ {
  ...
  v1_61 = return,
  v1_61 = 0
}
```

and

```
/\ {
  ...
  v1_61 = return,
  v1_61 = 1
}
```

and, hence, fulfils the rules of the standard.

As a last demonstration, we look at a program with pointer arithmetic. Namely, the example from Section 3.3.2.

```
int main(void) {
  int a[3];
  a[0] = 1;
  a[1] = 2;
  a[2] = 3;
  int *pa = &a[2];

  int sum = 0;
  while (pa > a) {
    sum = sum + *pa;
    pa = pa - 1;
  }
  return sum;
}
```

When fed into our tool, it returns the following constraint set indicating undefined behaviour.

```
/\{
  true,
  UNDEFINED = 1
  ...
}
```

If we modify the program to iterate through the array in the other direction, behaviour is no longer undefined.

```
int main(void) {
```

```
  int a[3];
  a[0] = 1;
  a[1] = 2;
  a[2] = 3;
  int *pa = &a;

  int sum = 0;
  while (&a[3] > pa) {
    sum = sum + *pa;
    pa = pa + 1;
  }
  return sum;
}
```

Correspondingly, our tool declares that the sum of the array, six, is returned:

```
/\{
  true,
  ...,
  v1_102 = return,
  v1_102 = 6,
  ...
}
```

In summary, we have demonstrated that our tool meets our original goals and more. It faithfully models evaluation order, it can successfully detect undefined behaviour, and the constraint solving and simplification phase is powerful enough to solve the constraint sets for many programs. Moreover, despite written with clarity and correctness in mind rather than performance, the tool is fast enough for small programs. The tool takes less than a second for most of programs discussed above.

4.3. Limitations

We have already mentioned one factor that limits our tool: Most real-world programs are not written in standard-conform C. Like GCC, all major C compilers support features that are not part of the languages standard and therefore not supported by our tool.

The other issue we have to worry about is performance. Whilst our tool is demonstrably fast enough to process a typical GCC test suite program, which was our main goal, its performance does not scale very well with an increasing number of case splits. As an illustration consider the following GCC test computing Fibonacci numbers.

```
int fib (int n) {
  if (n <= 1) {
    return 1;
  } else {
    return fib (n - 2) + fib (n - 1);
  }
}

int main (void) {
  if (fib (5) != 8) {
```

```
    return 1;
  } else {
    return 0;
  }
}
```

Each if statement causes our tool to perform a case split, i.e. the number of m-sets is doubled. Hence, if n is the recursive call depth, then the number of m-sets generated by our tool will be $2a(n)$, where $a(k+1) = a(k)^2 + 1$ and $a(0) = 0$. When the above program is fed into our tool, it will correctly determine that all m-sets correspond to infeasible paths for $n \leqslant 4$ and that exactly one m-set is satisfiable for larger n. However, the time it takes to return an answer increases very quickly: For for a call depth $n \leqslant 3$ the tool responds within a few seconds but for $n = 5$ it already takes several hours.

Since the above implementation of the Fibonacci function has exponential time complexity, we should, of course, also expect to see an exponential increase in the tool run time but the situation is worsened by the strict separation of constraint generation and constraint solving: For $n = 5$, the semantic analysis phase of the tool produces $2a(5) = 1534$ distinct m-sets. Each m-set is then evaluated separately. However, we know that the execution paths from which the individual m-sets have been generated, have a large overlap. We could exploit this knowledge by memoizing and reusing partial evaluations of m-sets. We decided against this optimisation to keep the semantic analysis clean. As a compromise, we chose to use persistent data structures in the constraint solving algorithm such that a backtracking strategy can be integrated at later stage without destroying the software structure of the implementation.

However, the number of m-sets alone cannot explain the long running times. If we modify the program slightly by introducing two temporary variables `x` and `y` that store the intermediate results of the recursive calls, our tool needs less than two minutes to spit out a result even though the number of case splits due to diverging execution paths is unchanged.

```
int fib (int n) {
  if (n <= 1) {
    return 1;
  } else {
    int x = fib (n - 2);
    int y = fib (n - 1);
    return x + y;
  }
}

int main (void) {
  if (fib (5) != 8) {
    return 1;
  } else {
    return 0;
  }
}
```

The discrepancy in run times is best explained with reference to the ordering of

actions in function `fib`. The following graph shows the sequenced-before relation for the original program. Note that the two recursive calls, represented by actions Call_{21} and

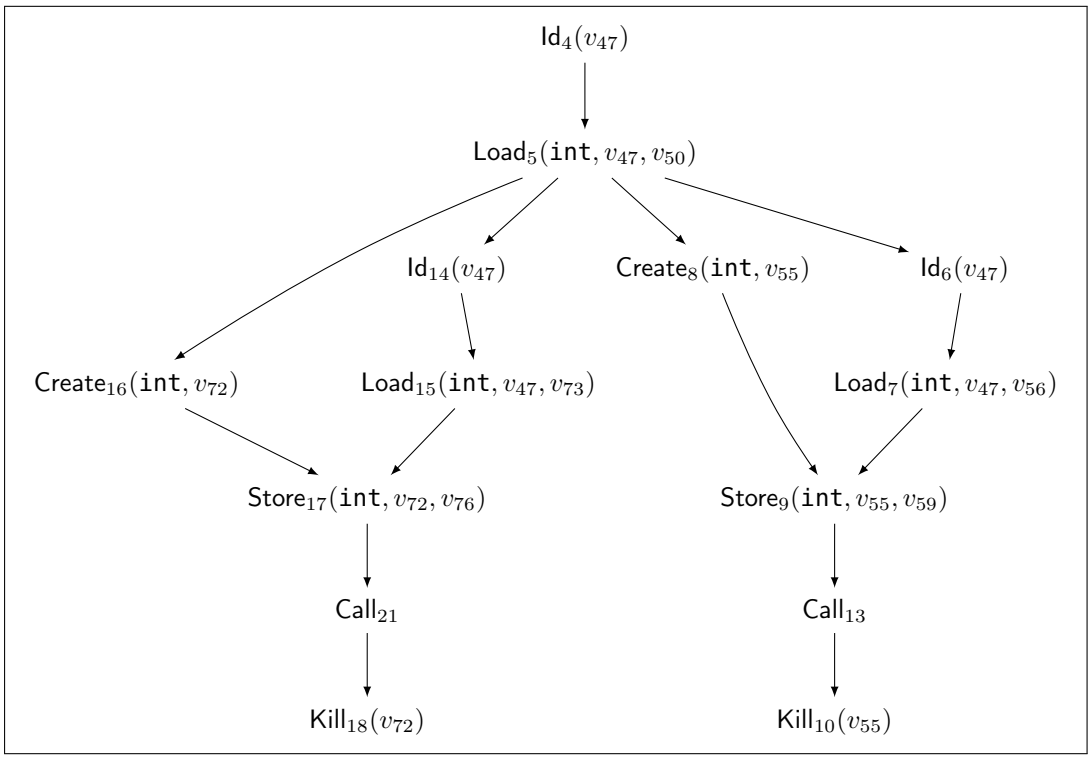


Fig. 4.4.: Sequenced-before relation for actions in function `fib`

Call_{13}, are unsequenced. Recalling the trace enumeration algorithm, our tool executes both orderings of the two actions. Hence, each recursive call doubles the number of traces.

In comparison, the call actions of the modified program, Call_{27} and Call_{46} in Fig. 4.5, are related: Call_{27} is sequenced before Call_{46}. Thus, only one trace needs to be executed. The example indicates that the trace enumeration algorithm could be enhanced by a further heuristic. For the specific cases when all addresses are statically known, as for the above program, we can avoid reordering function calls if they only operate on addresses that correspond to local variables. In general, however, the explosion in the number of traces is an intrinsic problem for any tool that computes all behaviours of a non-deterministic program, not just an implementation issue of our tool.

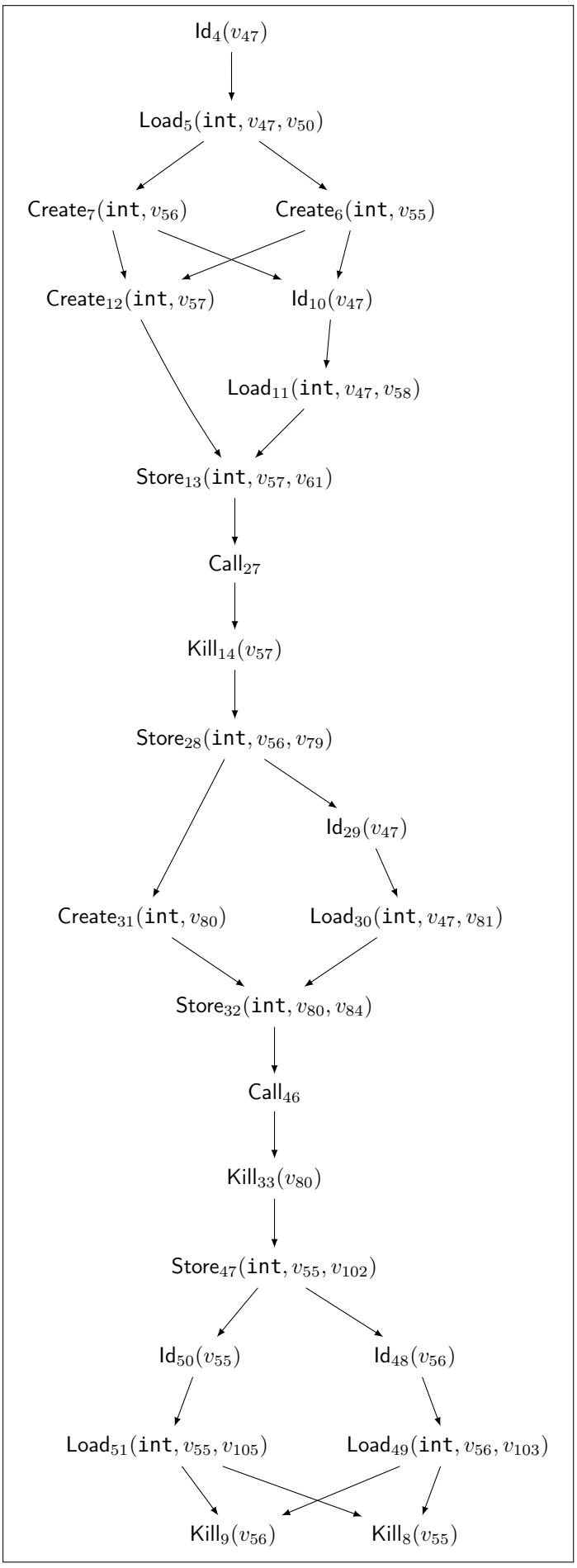


Fig. 4.5.: Sequenced-before relation for actions in function `fib`

5. Conclusions

We set out with the ambitious goal to formalise a fragment of the C programming language. It was our intention to implement the resulting semantics in a tool that could be used as a test oracle that, given a C program, outputs its meaning.

We have demonstrated throughout the dissertation that the work undertaken goes far beyond what was proposed. The implemented tool is able to give meaning to a larger subset of C than originally intended: Our static and dynamic semantics deal with integer, pointer and array types and cover most of the expression and much of the statement language. The unconventional and novel style of our dynamic semantics proved to be adequate for C. It allows us to cope with underspecification, non-determinism, undefined behaviour, and non-structural control-flow. Since we have designed the semantics with the entire standard in mind, we have reason to believe that our semantics is also expressive enough to be extended to the remaining parts of the language.

There are many areas of the project that could be extended and used in future developments.

Since clarity, rather than performance, was goal of this project, there is room to improve the performance of the tool. As we have seen, we could enhance the pruning of infeasible paths and use heuristics to cut down the number of action traces generated. An external high-performance decision procedure could be integrated to make the constraint solving process faster.

Currently, we do not cover all of the language. A truly practical tool would have to be extended to the rest of the standard, including the standard library. It would also have to cover non-standard extensions provided by the major C compilers, since they are used by many real world projects, e.g. the Linux kernel.

Another interesting direction would be to employ the semantics for real application: The tool could be modified to find undefined behaviour in large software projects or it could be used as a test oracle in automated compiler testing.

We could gain higher assurance for the model by expressing it within the framework of a proof assistant. It would also allow us to prove meta-theorems about C: We could, for example, explore what the formal guarantees of its type system are.

Even though there is, of course, much work still to be done to make the tool practical on a larger scale, it is not just an amusing toy project: C is at the foundation of many software systems and it will not vanish any time soon. And, yet, we still do not have a complete and accepted formal model of the language. It is our hope that work done within this project is a little step towards improving the situation.

Bibliography

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering – a new definition. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 2–14, New York, NY, USA, 1990. ACM.

[App97] Andrew W. Appel. *Modern compiler implementation in ML: basic techniques*. Cambridge University Press, New York, NY, USA, 1997.

[Ben07] John Benito. C – The C1X Charter. Technical Report N1250, ISO/IEC JTC1/SC22/WG14, June 2007.

[Boe88] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61 –72, may 1988.

[BOS+10] Mark Batty, Scott Owens, Peter Sewell, Susmit Sarkar, and Tjark Weber. Mathematizing C++ Concurrency: The Post-Rapperswil Model. Technical Report N3132=10-0122, Revision 5190, University of Cambridge, Computer Laboratory, September 2010.

[CF07] Sylvain Conchon and Jean-Christophe Filliâtre. A Persistent Union-Find Data Structure. In *Proceedings of the 2007 Workshop on ML*, ML '07, pages 37–46, New York, NY, USA, 2007. ACM.

[FF86] Matthias Felleisen and Daniel P. Friedman. Control Operators, the SECD-Machine, and the Lambda-Calculus. In *Proceedings of the IFIP TC 2/WG2. 2 Working Conf. on Formal Description of Programming Concepts Part III*, pages 193–219, August 1986.

[Gar00] Jacques Garrigue. Code reuse through polymorphic variants. In *Foundations of Software Engineering*, 2000.

[HB11] Florian Haftmann and Lukas Bulwahn. *Code generation from Isabelle/HOL theories*, January 2011.

[Ins89] American National Standards Institute. *American National Standard for Information Systems — Programming Languages – C: ANSI X3.159-1989.* American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1989.

[Jon10] Larry Jones. Working Draft (SC 22 N4578). Technical report, ISO/IEC JTC1/SC22/WG14, November 2010.

[Kri10] Neelakantan R. Krishnaswami. The AST Typing Problem. `http://lambda-the-ultimate.org/node/4170#comment-63836`, December 2010.

[ML85] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Technical Report 2, Universita di Siena, Dipartimento di Matematica, Scuola di Specializzazione in Logica Matematica, 1985.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

[NO07] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205(4):557 – 580, 2007. Special Issue: 16th International Conference on Rewriting Techniques and Applications.

[Nor98] Michael Norrish. C formalised in HOL. Technical Report UCAM-CL-TR-453, University of Cambridge, Computer Laboratory, December 1998.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[RD06] Norman Ramsey and João Dias. An applicative control-flow graph based on huet's zipper. *Electronic Notes in Theoretical Computer Science*, 148(2):105 – 126, 2006. Proceedings of the ACM-SIGPLAN Workshop on ML (ML 2005).

[Rey93] John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6:233–247, 1993.

[Rit93] Dennis M. Ritchie. The development of the C language. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 201–208, New York, NY, USA, 1993. ACM.

[SGC+07] Don Syme, Adam Granicz, Antonio Cisternino, Don Syme, Adam Granicz, and Antonio Cisternino. Introducing functional programming. In *Expert F#*, pages 27–68. Apress, 2007.

[SNO+10] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(01):71–122, 2010.

[vdL94] Peter van der Linden. *Expert C Programming: Deep C Secrets*. Prentice Hall, 1994.

[Wad98] Philip Wadler. The expression problem. `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`, November 1998.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, June 2011. ACM.

A. An overview of semantic styles

A.1. Big-step semantics

Viewed in isolation, big-step semantics appears to be a perfect fit to express the definition given in Figure 2.1 formally: we define a binary *evaluation relation* $\Downarrow$ on *configurations*. A configuration is a pair $\langle e, \sigma \rangle$, where e is an abstract syntax phrase and σ a state. At this point, we are mainly interested in the limitations of different semantic styles so we will skip over the definitions of abstract syntax and state for now and focus on the general form of the evaluation relation instead.

As usual, we define the evaluation relation inductively as a set of inference rules, where in the visual representation of each rule, the set of logical statements on top of the horizontal line forms the premise and the conclusion is given below the line.

We observe that the behaviour of the logical-and operator splits into three distinct cases: If the left-hand operand, e_1, evaluates to 0, then expression e_1 `&&` e_2 is reduced to 0 and the right-hand operator, e_2, is left unevaluated.

$$[\textsc{l-and-1}]\ \frac{\langle e_1, \sigma \rangle \Downarrow \langle 0, \sigma' \rangle}{\langle e_1 \texttt{ \&\& } e_2, \sigma \rangle \Downarrow \langle 0, \sigma' \rangle}$$

Otherwise, e_1 evaluates to some integer other than 0 and we have to evaluate e_2 to determine the outcome of the operation. If e_2 evaluates to 0, the outcome is 0.

$$[\textsc{l-and-2}]\ \frac{\langle e_1, \sigma \rangle \Downarrow \langle m, \sigma' \rangle \qquad \langle e_2, \sigma' \rangle \Downarrow \langle 0, \sigma'' \rangle}{\langle e_1 \texttt{ \&\& } e_2, \sigma \rangle \Downarrow \langle 0, \sigma'' \rangle} \quad \text{if } m \neq 0$$

If it instead evaluates to a non-zero integer, the result will be 1.

$$[\textsc{l-and-3}]\ \frac{\langle e_1, \sigma \rangle \Downarrow \langle m, \sigma' \rangle \qquad \langle e_2, \sigma' \rangle \Downarrow \langle n, \sigma'' \rangle}{\langle e_1 \texttt{ \&\& } e_2, \sigma \rangle \Downarrow \langle 1, \sigma'' \rangle} \quad \text{if } m, n \neq 0$$

Notice that the above rules encode, as required, a strict left-to-right evaluation order but, in general, evaluation order is unspecified in C. For instance, the order of evaluation of subexpression e_1, e_2 and e_3 in

$$(e_1 \texttt{ + } e_2) \texttt{ + } e_3$$

can be any permutation of the three subexpressions. If e_1, e_2, e_3 are side-effecting, then the order of evaluation can affect the result of the entire expression. Hence, our semantics must be able to produce all of the, at least, six distinct evaluations of the

above expression.

Big-step semantics, however, forces us to make a choice of whether to fully evaluate the left-hand operand or the right-hand operand of a binary operator first, e.g. the former choice is represented by

$$[\text{ADD-L}]\ \frac{\langle e_1, \sigma\rangle \Downarrow \langle m, \sigma'\rangle \qquad \langle e_2, \sigma'\rangle \Downarrow \langle n, \sigma''\rangle}{\langle e_1 + e_2, \sigma\rangle \Downarrow \langle k, \sigma''\rangle} \quad \text{if } k = m + n$$

and the latter by

$$[\text{ADD-R}]\ \frac{\langle e_1, \sigma'\rangle \Downarrow \langle m, \sigma''\rangle \qquad \langle e_2, \sigma\rangle \Downarrow \langle n, \sigma'\rangle}{\langle e_1 + e_2, \sigma\rangle \Downarrow \langle k, \sigma''\rangle} \quad \text{if } k = m + n\ .$$

As a result, not all evaluation orders of the above expression are possible. If we choose to reduce expression e_1 first, then we must evaluate e_2 before we can evaluate e_3:

$$\begin{array}{l}[\text{ADD-L}]\ \dfrac{\langle e_1, \sigma\rangle \Downarrow \langle m, \sigma_1\rangle \qquad \langle e_2, \sigma_1\rangle \Downarrow \langle m + n, \sigma_2\rangle}{\langle e_1 + e_2, \sigma\rangle \Downarrow \langle m + n, \sigma_2\rangle} \qquad \langle e_3, \sigma_2\rangle \Downarrow \langle k, \sigma_3\rangle \\ [\text{ADD-L}]\ \dfrac{}{\langle (e_1 + e_2) + e_3, \sigma\rangle \Downarrow \langle m + n + k, \sigma_3\rangle}\end{array}\ .$$

Hence, we cannot obtain the order e_1, e_3, e_2. It is clear that this limitation makes big-step semantics a very poor choice for the problem at hand.

A.2. Small-step semantics

Big-step semantics focuses on specifying the value of an expression but does not permit fine-grained control over the flow of evaluation. In contrast, small-step semantics give us a notion of a single computational step. A small-step semantics is defined in terms of a transition relation, $\rightarrow$, which is, again, a binary relation on pairs $\langle e, \sigma\rangle$. To model addition, we specify a rule for performing a single computational step on the left-hand operand

$$[\text{ADD-L}]\ \frac{\langle e_1, \sigma\rangle \rightarrow \langle e_1', \sigma'\rangle}{\langle e_1 + e_2, \sigma\rangle \rightarrow \langle e_1' + e_2, \sigma'\rangle}$$

and the same for the right-hand operand

$$[\text{ADD-R}]\ \frac{\langle e_2, \sigma\rangle \rightarrow \langle e_2', \sigma'\rangle}{\langle e_1 + e_2, \sigma\rangle \rightarrow \langle e_1 + e_2', \sigma'\rangle}\ .$$

Once both operands have been reduced to values (expressions that cannot be reduced any further) v_1 and v_2, respectively, the result of the next transition step will be the usual mathematical sum of the two values.

$$[\text{ADD-V}]\ \langle v_1 + v_2, \sigma\rangle \rightarrow \langle v, \sigma'\rangle \quad \text{if } v = v_1 + v_2$$

Using the above set of rules, we can take another look at the example from the previous section. Evaluating e_1 first followed by e_3 is no longer impossible:

$$\langle (e_1 \texttt{ + } e_2) \texttt{ + } e_3, \sigma \rangle \to^* \langle (m \texttt{ + } e_2) \texttt{ + } e_3, \sigma_1 \rangle \to^* \langle (m \texttt{ + } e_2) \texttt{ + } k, \sigma_2 \rangle \to^* \langle m + n + k, \sigma_3 \rangle ,$$

where $\to^*$ is the transitive closure of the reduction relation.

Although small-step semantics resolve the issue of non-deterministic evaluation order in a straightforward manner, the same cannot be said about undefined behaviour and non-structural language features. We shall start with undefined behaviour.

Conceptually, undefined behaviour is no great challenge. We just have to introduce a rule for each occurrence of undefined behaviour. The rule for division by zero would be

$$[\textsc{div-zero}]\ \langle v_1 \texttt{ / } 0, \sigma \rangle \to \textbf{undef} ,$$

for instance. However, if a program contains some subexpression with undefined behaviour that will be evaluated at some point, then the entire program has undefined behaviour. As a result, we need a rule like the following for every single context in which undefined behaviour could occur in order to propagate **undef** to the top-most level of the transition relation.

$$[\textsc{add-l-undef}]\ \frac{\langle e_1, \sigma \rangle \to \textbf{undef}}{\langle e_1 \texttt{ + } e_2, \sigma \rangle \to \textbf{undef}}$$

Such an approach would hugely increase the number of rules necessary. Since all of the additional rules to propagate undefined behaviour have operationally the same effect, it should be possible to express the same with a single rule by introducing the notion of an *evaluation context*.

Similarly, to define the semantics of, for example, the **`continue`** statement, we need to know the context in which the statement is executed, e.g. the enclosing loop. The concept of evaluation contexts could help us to give meaning to non-structural constructs and at the same time reduce the number of semantic rules.

A.3. Felleisen-style semantics

Evaluation contexts were first introduced by [FF86]. They describe the positions within an expression where the next transition can take place. Such a position is called *hole* and symbolically represented by an underscore. Using the notion of holes, we can define contexts $\mathcal{E}$ as a grammar. For example,

$$\mathcal{E} ::= \_ \mid \mathcal{E} \texttt{ + } e_2 \mid e_1 \texttt{ + } \mathcal{E} \mid \mathcal{E} \texttt{ - } e_2 \mid e_1 \texttt{ - } \mathcal{E} \mid \mathcal{E} \texttt{ \&\& } e_2 \mid v_1 \texttt{ \&\& } \mathcal{E} .$$

We then write $\mathcal{E}[e]$ to say that expression e fills the hole in the context $\mathcal{E}$.

With this new tool at our disposal, a single rule is sufficient to describe the propagation

of undefined behaviour. Namely,

$$[\textsc{undef}]\ \frac{\langle e, \sigma\rangle \to \mathbf{undef}}{\langle \mathcal{E}\,[e]\,, \sigma\rangle \to \mathbf{undef}}\ .$$

We quickly notice, that we can also employ the same technique to unify those rules whose sole purpose is to recursively reduce a subexpression ([ADD-L] and [ADD-R], for example), i.e.

$$[\textsc{frame}]\ \frac{\langle e, \sigma\rangle \to \langle e', \sigma'\rangle}{\langle \mathcal{E}\,[e]\,, \sigma\rangle \to \langle \mathcal{E}\,[e']\,, \sigma'\rangle}\ .$$

Unfortunately, the use of evaluation contexts in the previous two examples is not much more than notational trick that, on paper, leads to a neater presentation of the rules. The two inference rules are strictly-speaking no longer formal, where '[...] formal means precisely that there must be no semantic conditions involved in the rule: it may only put conditions on the forms of the premises and conclusion.' [ML85, p. 5]. The notion of formality is in close correspondence with our goal of executability. An inference rule can only be executable if it is purely of syntactic nature. Specifically, the rules above require us to know how to construct an evaluation context, e.g. e_1 + , $[e_2]$, from an arbitrary expression e_1 + e_2, say.

If we want to regain executability, we are left with two options: Either we introduce further rules that make the construction of evaluation contexts explicit or we have to expand out the definition of $\mathcal{E}$, i.e. create a separate rule for every grammar production in $\mathcal{E}$. The latter is clearly undesirable; it would just bring us back to the small-step semantics of the previous section. Explicit construction of evaluation contexts, however, seems like an acceptable compromise. Although we need two additional rules for each production of the context grammar (one to construct a context and one to destroy it again), we no longer need special rules to propagate undefined behaviour and we can model the semantics of non-structural features in the same style we use for the rest of the language.

A.4. Continuation semantics

A very common approach to explicit context construction is the use of continuation semantics. Continuations originate from the field of denotational semantics but we shall leave aside the mathematical context and look at them from a purely operational perspective. The idea is to refine the definition of configurations to pairs of the form $\langle e \cdot \kappa, \sigma\rangle$, where κ is a list of continuations and each continuation is a *singular* evaluation context. For demonstration purposes, we will use the following definition of κ,

$$\kappa ::= \mathbf{stop} \mid [\_ \texttt{ + } e_2] \cdot \kappa \mid [e_1 \texttt{ + } \_] \cdot \kappa \mid [\_ \texttt{ / } e_2] \cdot \kappa \mid [e_1 \texttt{ / } \_] \cdot \kappa\ ,$$

where **stop** is the empty list of continuations[1]. Furthermore, we consider a configuration to be *final* or fully evaluated when it has the form $\langle v \cdot \mathbf{stop}, \sigma \rangle$.

Now, we can give rules that produce evaluation contexts. The following, for example, allows us to reduce the left-hand operand of a sum before the right-hand operand[2].

$$[\text{ADD-L}]\ \langle e_1 \texttt{ + } e_2 \cdot \kappa, \sigma \rangle \to \langle e_1 \cdot [\_ \texttt{ + } e_2] \cdot \kappa, \sigma \rangle$$

We also need rules that destroy the continuation again. In particular, if the current expression is a value within the context of an addition whose other operand is also a value, we compute the sum of the two values and remove the topmost continuation from the list:

$$[\text{ADD-L-V}]\ \langle v_1 \cdot [\_ \texttt{ + } v_2] \cdot \kappa, \sigma \rangle \to \langle v \cdot \kappa, \sigma \rangle \quad \text{if } v = v_1 + v_2\ .$$

If the addition continuation still contains non-value expressions, we reduce those first.

$$[\text{ADD-L-R}]\ \langle v_1 \cdot [\_ \texttt{ + } e_2] \cdot \kappa, \sigma \rangle \to \langle e_2 \cdot [v_1 \texttt{ + } \_] \cdot \kappa, \sigma \rangle$$

With the general framework for construction and destruction of contexts in place, we can start to exploit the benefits of continuation semantics. As before, we have a rule that deems division-by-zero to be undefined.

$$[\text{DIV-ZERO}]\ \langle 0 \cdot [e_1 \texttt{ / } \_] \cdot \kappa, \sigma \rangle \to \mathbf{undef}$$

Since κ encodes the 'meaning of the rest of the program' [Rey93, p. 240], by simply discarding κ, the propagation of **undef** becomes unnecessary.

Continuations also enable us to formulate rules for statements whose meaning is context-specific in a very natural way. As we will see later, C distinguishes between statements and expression: Statements, unlike expressions, do not produce values. Hence, we need a slightly different reduction relation but, nevertheless, we can retain the semantic style by introducing a special value **skip** to which all well-defined and terminating statements are reduced. We can, for example, imagine the following definition of the `break` statement

$$[\text{BREAK}]\ \langle \texttt{break} \cdot [\texttt{while (}e\texttt{) \{\_\}}] \cdot \kappa_s, \sigma \rangle \to_s \langle \mathbf{skip} \cdot \kappa_s, \sigma \rangle\ ,$$

for appropriate choices of $\to_s$ and κ_s, in combination with a rule that discards the second statement when `break` is first in a sequence of statements, i.e.

$$[\text{SEQ-BREAK}]\ \langle \texttt{break} \cdot [\_\texttt{;}s_2] \cdot \kappa_s, \sigma \rangle \to_s \langle \texttt{break} \cdot \kappa_s, \sigma \rangle\ .$$

As the preceding discussion illustrates, continuation semantics makes it possible to give very elegant and natural semantic descriptions of statements with involved control-flow as well as expressions that require knowledge of the context. However, so far we have ignored the very first issue we tried to solve: Unspecified evaluation order. Given

[1] In a more mathematical setting, we would define **stop** to be the identity function and $\cdot$ to be function composition.

[2] We omit the symmetric rules for the right-hand operand in favour of brevity.

an expression

$$(e_1 \texttt{ + } e_2) \texttt{ + } e_3,$$

does our continuation semantics induce all possible evaluation orders? Unfortunately, the linear way in which we construct continuations prohibits certain evaluation orders. We cannot, for example, evaluate e_1 first followed by e_3 since after the first transition

$$\langle (e_1 \texttt{ + } e_2) \texttt{ + } e_3 \cdot \mathbf{stop}, \sigma \rangle \to \langle e_1 \texttt{ + } e_2 \cdot [\_ \texttt{ + } e_3] \cdot \mathbf{stop}, \sigma \rangle \,,$$

expression e_3 is hidden inside the continuation and we cannot access it again until $e_1 \texttt{ + } e_2$ has been reduced to a value.

Unfortunately, there is no obvious solution to the above problem that does not violate executability other than choosing a fundamentally different semantic style.

B. Proposal

Introduction and Description of the Work

The C programming language was created in the early 1970s by Dennis M. Ritchie at the Bell Telephone Laboratories to fill the need of the new operating system Unix for a system programming language [Rit93]. The language evolved into a general-purpose programming language over the next two decades and its use spread so widely that a standardisation became necessary to avoid the different dialects of C to diverge even further. Thus, in 1989, the first C Standard was established by the American National Standards Institute. In the rationale that accompanies the standard the authors state that their "overall goal was to develop a clear, consistent, and unambiguous Standard for the C programming language" [Ins89]. The language specification given by ANSI was later adopted as an ISO standard, which in turn was superseded by an updated and extended version of the standard which is informally known as C99. Since 2007 the ISO working group for C is discussing the next revision of the C standard called C1X.

The purpose of the standard is to specify the semantics of C programs. Like most programming language specifications the C standard is written in natural language leaving room for ambiguity and making it unsuitable as a basis for a formal treatment of the language. However, a formal understanding of C is especially desirable since the language is very widespread in safety critical applications and also since it combines features whose interaction are difficult to predict without a formal model: The order of side-effect applications, evaluation order of expressions, and even the semantics of signed integer arithmetic are underspecified in the C standard and these are just a few sources of non-determinism and implementation-dependent behaviour in C.

The aim of the project is to write a tool that aids understanding C's implicit non-determinism. For a given C program the tool will compute the set of all standard-conforming execution paths. Such a tool could be used to find flaws in the standard, to empirically explore how a change to the standard would affect the overall language, or to test whether a compiler is generating standard-compliant code. However, covering the entirety of C is beyond the reach of this project: The C programming language is too large and its semantics too involved. Thus, the project will only work with a selected fragment of C.

Substance and Structure of the Project

Given the size of the project, it is necessary for realistic planning to divide the project into manageably sized components. Fortunately, the processing done by the tool naturally

divides into a number of distinct stages where each stage only depends on the completion of the previous stage.

To further reduce the risk of the project, I will adopt an iterative development style, i.e. when building a particular stage of the tool I will implement the smallest set of features that is sufficient to meet the success criteria of the project, repeat the process for the next stage, and then later continually revisit each stage to augment the tool with one new feature at a time.

The first phase of development will look at the most crucial part of the project: Finding a suitable structural operational semantics that captures the intent of C1X as closely as possible for a small part of C, namely integer expressions including bitwise operators. The subset of C has been chosen since it does not require an involved memory model, which would add too much complexity at a very early stage, and also since it contains

- undefined behaviour, e.g. adding 1 to the largest representable integer,
- unspecified behaviour, e.g. an implementation can freely choose whether to represent signed integers in sign-and-magnitude, one's complement, or two's complement form,
- underspecified evaluation order, e.g. the expression $e_1 + e_2$ can be evaluated in any particular order[1].

To avoid duplicating already existing research on the semantics of C, a significant part of the initial phase of this project will be spent on reading and familiarisation with relevant papers on the subject. In particular, the semantics for this project will borrow ideas from Michael Norrish's Cholera project [Nor98] which gives an almost complete operational semantics for a large subset of C. It will have to be adapted, however, to accommodate for the particular needs of the tool: The semantics has to work with symbolic rather than concrete values, it needs to be executable such that it can later be used to compute the set of all legal outcomes of a program, and it will need a more sophisticated memory model at a later phase of the project when the subset of C dealt with is extended with pointers or a similar construct that depends on correct memory alignment.

Once a semantics suitable for our purposes has been specified, it will be re-written in the second phase of the project within the framework of a proof assistant, most likely Isabelle/HOL[2] [NPW02]. The Isabelle/HOL formalisation of the semantics will then be used to calculate the set of all valid execution paths of a given program which involves symbolically solving the particular constraints the C standard imposes on the standard-conforming evaluations of the program. At the core of the process an SMT solver might be employed to allow for fast constraint solving.

The previous two stages of the project implement the computational heart of the tool, the next two will build the front- and back-end. Both will be written in OCaml. The

[1] Assuming the expression does not contain any *sequence points*.

[2] The choice of Isabelle/HOL is still tentative. A final decision will be made within the first weeks of the project but for now I will only refer to Isabelle/HOL.

front-end to read in, parse, and translate C programs to an appropriate intermediate representation will rely on CIL (C Intermediate Language) [NMRW02], which simplifies this process significantly. The rôle of the back-end is to visualise the set of standard-conforming execution paths using a graph representation similar to [BOS+10]. The graphs will be drawn with the Graphviz[3] toolkit as it supports automatic graph layout and has API bindings for OCaml.

The implementation of the back-end concludes the core of the project but the tool allows for a range of possible extensions. The most obvious is to extend the subset of C the tool can process. Another would be to write a testing framework that allows automatic compiler checking: Given a test program it would run both the compiler and the tool and then determine whether the program produced by the compiler gives a result that is in the set of allowable outcomes. The framework would build on top on the constraint solving facilities of the tool.

Success Criteria

The core of the project will be deemed a success if the following criteria are met.

(i) The tool accepts C programs restricted to integer expressions, which may exhibit non-deterministic behaviour, and generates a graph presenting the full set of standard-conforming execution paths of the program.

(ii) When given relevant programs from GCC's C language test suite[4], the set of execution paths calculated by the tool includes the one chosen by the GCC compiler. If not, the discrepancy must be explained in the dissertation.

Starting Point

The tool proposed will be implemented in OCaml as well as Isabelle/HOL. Even though I have only very limited experience with OCaml, using it throughout the project should not prove too challenging as I do have a working knowledge of the two closely related languages F# and Standard ML. Unfortunately, I do not have any prior experience with Isabelle/HOL but my supervisor has agreed to give me an introduction to reduce the time I have to invest in familiarising myself with the system.

Before devising an operational semantics I will study Michael Norrish's Cholera semantics in detail. Furthermore, Susmit Sarkar[5] has already produced an executable semantics for a very small fragment of C written in OCaml that I will use to produce an initial semantics for the tool.

[3] Graphviz – Graph Visualization Software, `http://www.graphviz.org`.

[4] Chapter 7.4 C Language Testsuites. *GCC Internals Manual*, `http://gcc.gnu.org/onlinedocs/gccint/C-Tests.html`.

[5] Website: `http://www.cl.cam.ac.uk/~ss726/`, Email: `Susmit.Sarkar@cl.cam.ac.uk`.

Resources Required

The project has no special requirements. The development will take place on my personal machine and I intend to use the services provided by the PWF and SRCF for back-up purposes.

Timetable and Milestones

Officially, the project begins with the submission of the project proposal on 22nd October 2010 and ends with the submission of the dissertation on 14th May 2010. This period has been broken down into ten units of work each of which should take approximately three weeks to complete.

Slot 0 — Sat, 6th Oct – Fri, 22nd Oct

The weeks preceding the start of the project will be used to discuss the scope and feasibility with supervisor and overseers, to begin the study of relevant papers and the C programming language standard, to set up a development environment including a source code management system, and also to devise a back-up strategy.
Milestone: Submission of project proposal.

Slot 1 — Sat, 23rd Oct – Fri, 12th Nov

Starting with Susmit Sarkar's semantics, iteratively produce a more and more refined operational semantics for integer expressions in C. Study the intermediate representation of C used in CIL, familiarise with Isabelle/HOL, and begin re-writing the semantics in Isabelle/HOL. Also source C programs from compiler test suites for later evaluation of the tool.
Milestone: Operational semantics for integer expressions, selection of test programs.

Slot 2 — Sat, 13th Nov – Fri, 3rd Dec

Research the use of an SMT solver within the proof assistant. Start implementing the constraint solving necessary to compute the set of valid execution paths.
Milestone: An Isabelle/HOL formulation of the semantics.

Slot 3 — Sat, 4th Dec – Fri, 24th Dec

Finish implementation of algorithm to generate the set of execution paths.
Milestone: Working constraint solving for integer expression, a working algorithm to determine the set of allowable execution paths.

Slot 4 — Sat, 25th Dec – Fri, 14th Jan

Use remaining time to extend the fragment of C dealt with as far as time allows.

Slot 5 — Sat, 15th Jan – Fri, 4th Feb

Write front-end and back-end for the tool. Start drafting preparation and implementation chapters for the core of the project. Prepare progress report and presentation.
Milestone: A working core implementation, a skeleton dissertation with content for preparation/implementation of the core project. Progress report and slides for the presentation.

Slot 6 — Sat, 5th Feb – Fri, 25th Feb

Run selected test programs through the tool, GCC, and other compilers. Compare results. Update preparation and implementation chapters and start writing the evaluation chapter.

Slot 7 — Sat, 26th Feb – Fri, 18th Mar

Further write-up of the evaluation chapter. Write introduction and conclusion.
Milestone: Send first draft version of the dissertation to supervisor.

Slot 8 — Sat, 19th Mar – Fri, 8th Apr

Incorporate any comments received into the dissertation text, finish outstanding evaluation work, and send out a second draft to proofreaders.
Milestone: Produce a second draft.

Slot 9 — Sat, 9th Apr – Fri, 29th Apr

Address any issues brought up by proofreaders and produce a printed and bound dissertation by the end of the time slot.
Milestone: Printed dissertation.

Slot 10 — Sat, 30th Apr – Fri, 20th May

The final time slot ends with the dissertation deadline. Ideally, the slot should not be used for any further work and instead it provides a buffer in case any serious issues still remain.
Milestone: Submission of dissertation.