Dresden University of Technology
Operating Systems Group

Diploma thesis

Development of a semantics compiler for C++

Matthias Daum

September 22, 2003

Declaration

I declare that all parts of this work were autonomously written by me while using only legal resources. All resources, that were used within this work are explicitly announced. To the best of my knowledge, the content of this work is original and was not published before by me or another author.

Dresden, September 22, 2003

Matthias Daum

Contents

1 Introduction

Currently, correctness of software is usually ensured by careful design and extensive testing. Tests may reveal errors, but even the most immense testing effort can never guarantee their absence. A reliable proof of correctness can only be obtained with the help of formal methods.

In most application fields errors are undesirable, but tolerable. However, the area of security critical applications such as cryptographic software is continuously growing. Today, home banking and online shopping are very popular. Here, trust is currently based foremost on hopes and beliefs. In the best case, it is attemted to gain security by obscurity. Despite of that, correctness is of utmost importance in these applications that work in such an open environment with potentially malicious attackers.

This precarious situation results not only from unawareness or carelessness, but the lack of suitable alternatives. Security issues cannot be addressed at the application layer alone. If the operating system does not provide a trustworthy environment, any effort to ensure security of applications will fail.

Unfortunately, common operating systems are too complex for formal verification, even though the essential trusted base could be very small. The operating system kernel must be reduced to only provide protection of applications against each other. Liedtke [Lie96] proved the concept of a minimalistic kernel with his *L4* microkernel. The great advantage of the microkernel approach is the flexibility it offers. Applications with special needs like quality of service or security issues can run in parallel to a standard production system, as the port of Linux to L4 [Hoh96] shows.

Based on the innovative ideas of Liedtke, a whole family of microkernels has adopted the interface specification of his first implementation. Among them is FIASCO [Hoh98], a real-time capable microkernel operating system developed at the Dresden University of Technology. The VFIASCO [THH01] project aims to verify substantial security properties of this microkernel.

FIASCO was originally designed for IA32 processors, but has been successfully ported to other architectures. It is almost entirely written in C++ [Str00]. Assembler instructions are embedded only for operations that cannot be expressed in C++, such as the access of CPU control registers. With only about 30,000 lines of code for its core functionality, FIASCO has a reasonable size for verification.

For reasoning, it is planned to employ a state-of-the-art general-purpose theorem prover. The current development focuses on PVS [ORR$^+$96]. Prior to verification, the C++ source code is translated to a semantics representation in higher-order logic. This technique is called *shallow embedding*, in contrast to *deep embedding*, where parsed programs are directly represented in the logic and the semantics of the source-code phrases is expressed with semantic functions.

In my thesis, I began the development of the semantics compiler to prove the practicality of, until then, rather theoretical ideas and preliminary concepts for the semantics representation. Thus, the key goal in compiler design was flexibility and extensibility. It aims for architecture independence to benefit from the recent porting efforts for FIASCO. The focus of my work was on semantics translation rather than C++ parsing, which was taken care of by already existing parser components.

1.1 Structural outline

The remainder of this thesis is divided into 5 chapters. In the next chapter, I will provide some background information. This includes theoretical basics, an overview of the VFIASCO project and the software environment my translator component has to fit in, and finally, information on related work done in that field.

Chapter 3 contains a detailed description of my general design goals.

The following two chapters form the main part of this thesis. At first, I will describe the semantics specification of C++. I will explain the general approach that was taken and show how the various features of C++ fit into this model.

Afterwards, I will depict design and implementation of the translator. This includes an interface specification, an explanation of design obstacles, an overview of the implementation structure, and a description of some specialties in the implementation.

The final chapter will assess the result and give an outlook on future work.

1.2 Conventions

The program semantics is modelled based on the International C++ Standard ISO / IEC 14882 [ISO98]. I will call this document just »the standard«. The notation § $n.m$ (k) refers to section $n.m$, paragraph k in the standard.

Where appropriate, I use a mathematical notation to express logical formula. This is usually shorter than the PVS equivalent. In the formula, partial functions are denoted by a broken arrow ($\dashrightarrow$). The following definitions are used for some auxiliary sets:

$$\begin{aligned} \mathbb{B} &\stackrel{\text{def}}{=} \{\text{true}, \text{false}\} \\ \mathbf{1} &\stackrel{\text{def}}{=} \{\bot\} \\ \mathbb{N} &\stackrel{\text{def}}{=} \{0, 1, 2, \ldots\} \end{aligned}$$

Disjoint unions are denoted as

$$\overset{\kappa_1:}{M_1} \uplus \overset{\kappa_2:}{M_2}$$

for M_1, M_2 sets, with $\kappa_{1,2}$ as suggestive names for the injections. The names are very helpful with complex disjoint unions of 3 or more members. Moreover, these names correspond to the constructor names of (non-recursive) *Abstract data types* in PVS, which model disjoint unions. Similarly, the cartesian product corresponds to PVS records. It is denoted as

$$\overset{\pi_1:}{M_1} \times \overset{\pi_2:}{M_2}$$

for M_1, M_2 sets, with $\pi_{1,2}$ naming the projections.

Often I use parametrized definitions. These define a collection of similar items (sets or constants) instead of a single one. The parameters are surrounded by square brackets. Consider

$$M_1[P_1] \stackrel{\text{def}}{=} \mathbf{1} \uplus M_2[P_1]$$

Here, M_1 has one parameter, denoted as P_1. M_1 is defined as disjoint union of $\mathbf{1}$ and M_2 with the applied parameter P_1.

1.3 Acknowledgements

I wish to thank all who—directy or indirectly—supported my work. Namely, *Hendrik Tews and Michael Hohmuth* for supervising me, *Shigeru Chiba and Grzegorz Jakacki* for developing OpenC++ and making it available as free software, *Stefan Reuther* for writing the Annotator and patiently explaining to me how it works and why it behaves the way it does. Furthermore, I would like to acknowledge the various people, who asked and answered silly and intelligent questions, talked and discussed, set examples, or otherwise morally sustained me. Acting for all, I would like to mention Alex, Clemens, Harvey, Sarah, and Udo. Very special thanks go to my parents who made my studies possible and always supported me wherever they could.

2 Background

In this chapter I will introduce the context of my work. At first, I will explain the theoretical fundamentals that are needed to understand the problem. Here, I will mainly focus on aspects of program verification. Afterwards, I will give an overview of the VFiasco project. I will also describe the framework of existing compiler components. Finally, I will set my work in relation to similar projects.

2.1 Basics

The semantics compiler translates C++ source code into formal PVS specifications [OSRSC01]. C++ is a universal programming language, especially dedicated, but not limited to low-level and system programming. It is a high-level language supporting data abstraction as well as object-oriented and generic programming. C++ was standardized in 1998 (see [ISO98]). A language description is far beyond the scope of this thesis. I assume basic knowledge about C++ programming and terminology (for example, *cast operators*, *compound assignments*, the `volatile` keyword, or type `size_t`). A comprehensive reference manual [Str00] was written by Bjarne Stroustrup, the inventor of C++.

PVS is a *P*rototype *V*erification *S*ystem for development and analysis of formal specifications. It consists of a specification language and an interactive, semi-automated theorem prover. For my work, only the PVS language is of interest. It provides a higher-order logic with predicate subtypes and some other extensions.

Certainly, I cannot cover all facets of the PVS specification language. For a detailed description, the reference manual [OSRSC01] should be consulted. Here, I will only outline some rather typical concepts to give a quite general idea of a specification language.

2.1.1 The PVS language abstracted

A PVS specification consists of a collection of theories. Each theory is identified by a name, and consists of a signature and the axioms, declarations, and theorems associated with this signature. The signature specifies generic type names and constants to generalize the theory's declarations (thus, the signature acts much like a template's parameter list in C++). An example is shown below. Conceptually, type `Optional` expands a generic type (`Base_type`) by an additional value `none`. Note: This example is oversimplified and not meant for practical use. PVS provides the type constructor `lift` for this purpose.

```
Type_Optional_Theory[Base_type: NONEMPTY_TYPE]: THEORY
BEGIN
  Optional: NONEMPTY_TYPE

  none: Optional
  is_none? (x: Optional): bool = (x = none)
```

```
    wrap: [Base_type -> Optional]
    unwrap: [Optional -> Base_type]

    value_is_kept: AXIOM
      FORALL (t: Base_type): unwrap(wrap(t)) = t

    none_is_distinct: AXIOM
      FORALL (t: Base_type): wrap(t) /= none

    optional_is_minimal: AXIOM
      FORALL (x: Optional):
        is_none?(x) OR EXISTS (t: Base_type): x = wrap(t)

    transparent_equality: THEOREM
      FORALL (a, b: Base_type): wrap(a) = wrap(b) IMPLIES a = b
  END Type_Optional_Theory
```

The `Type_Optional_Theory`'s signature has just one parameter, its `Base_type`. It declares the new (non-empty) type `Optional`, the constant `none`, and three functions: `is_none?`, `wrap`, and `unwrap`.

Note that `none` is a *constant*, even though its value is not given. The actual value does just not matter in this specification. `none` serves as general placeholder for some fixed, unknown (or uninteresting) value. It is called an *uninterpreted constant*.

The function `is_none?` takes an argument of type `Optional` and returns a boolean value: true if the argument was `none`, and false otherwise. `wrap` takes a `Base_type` value and returns an `Optional`; `unwrap` does it vice versa. The particular mapping of the values is left unspecified, here.

It should be pointed out that this is *not* like a function declaration in C++. In PVS one cannot subsequently »supplement« these functions with a specific mapping. In style of the One Definition Rule in C++ one could speak of a *One Declaration Rule* in PVS. Each entity must be declared exactly once, and there is no way to subsequently change this declaration.

`wrap` and `unwrap` were declared just like the constant `none`—as (abstract) placeholders for some arbitrary functions. Indeed, constants and functions are quite similar in PVS. Practically, a constant is just a function that takes no arguments and returns a fixed value.

However, the given axioms specify some general properties of the specified functions: `value_is_kept` requires `unwrap` to be the left-inverse of `wrap`, `none_is_distinct` assures that `none` does not relate to any value of `Base_type`, and `optional_is_minimal` ensures minimality of `Optional`. Axioms are a distinguishing feature of specification languages. They describe program behavior abstracted from a particular implementation.

`transparent_equality` is a theorem. It states that the equality is transparent to `wrap`. Theorems and axioms are similar regarding that both state a logical property. However, an axiom specifies a *new requirement* for the introduced declarations, while a theorem states a *provable* property. Changes to axioms will alter the meaning of the specification. Theorems are just given to ease complex proofs.

Comments start with `%` and end with the end of the line (not shown in the example).

Theories may build on other theories. Such dependencies are expressed via an `IMPORTING` clause. A theory might be instantiated on import. For example, if another theory needs only optional values for natural numbers, it might import `Type_Optional_Theory[nat]`.

The PVS language is strongly typed. To supplement the predefined types like `nat` or `bool`, complex data types can be constructed in the different ways. Records combine several arbitrary types like a cartesian product with named members. Function types represent the usual mathematical concept of mappings between sets (types). The most powerful way of new type definitions are Abstract data types (ADTs). I will only use non-recursive ADTs in my thesis. Practically, they can be conceived as a disjoint union of several named member types.

Instead of the above example, one can simply write `Optional` as an ADT:

```
Optional[Base_type: TYPE]: DATATYPE
BEGIN
  none: is_none?
  wrap(unwrap: Base_type): is_value?
END Optional
```

In this case, PVS automatically generates a theory with various functions and axioms, which is conceptually similar to `Type_Optional_Theory`.

The notation of an ADT can be viewed as an enumeration of value constructors. Each constructor introduces a new, distinct member type. The new Abstract data type is formed by the (disjoint) union of these members. The co-domain of a constructor is a subtype (subset) of the new ADT. It is described by a predicate on the ADT. The predicate is called recognizer. An additional function is defined for each parameter of a constructor. If an ADT value was constructed with the belonging constructor, this function restores the original value of this parameter. These functions are called accessors.

In the little example, `none` represents a trivial constructor function that takes no arguments and returns a constant value. `is_none?` is a predicate that evaluates to true for this and only this constant value. The constructor `wrap` takes one argument of `Base_type` and returns values of the new ADT. The predicate `is_value?` is only true for all values returned by `wrap`. `unwrap` is not only the parameter name of the `wrap` constructor. At the same time, it denotes an accessor. This is a function that returns the original `Base_type` value for `Optional` values constructed with `wrap`.

2.1.2 Towards modelling program execution—general principles

> ≫ *The main thing that was prominent in computer science that wasn't in mathematics was an emphasis on the state of a process as it changes, where it changes in time in a discrete way. In computer science when you say n is replaced by $n + 1$, the old value disappears and the new value takes over. We know how to think about an algorithm that is half-way executed; it has a state consisting of the current values of all the variables, and the state also specifies what rule to apply next. In order to formulate this for most mathematicians, it requires putting subscripts on everything.* ≪ — *Donald Knuth.*

In this section I will give a very brief introduction on general principles in verification of imperative programs. In chapter 4, on the contrary, I will explain the semantics representation of C++ in detail.

The notion of program execution strongly corresponds with its change in state. The program state includes all (variable) information a program can access. This certainly includes, but is not limited to programming variables. Programs might read and manipulate processor registers, main memory, the system's clock, and mass storage. Taking that into account, the program state grows to a fairly complex type, like

$$State \stackrel{\text{def}}{=} Registers \times Memory \times \ldots$$

with *Registers* and *Memory* being types.

While keeping this in mind, we just require *State* to be a type, but do not further care for its internal layout. There are two ways to reflect that in a theorem prover: The type can first be defined as an arbitrary one, $State : Type$, and changed as needed. A more general approach is the introduction of a type parameter *State* on every dependent theory. This way, a specific type could later be defined and associated with the generic type *State*. This is especially useful in contexts where theories are used with different types.

Statements

In the simplest case, a program fragment consists of a list of statements, which must be subsequently executed. Each statement is executed based on the current state, usually resulting in a new one. Thus, one could represent a statement as a function of $State \to State$, expressing subsequent statement execution by function composition.

This is not generally feasible, since certain statements change the flow of control. Abrupt termination can be handled by extending the result type of the statement functions, e. g.:

$$Statement \stackrel{\text{def}}{=} State \to \overset{OK:}{State} \uplus \overset{exit:}{\mathbb{N}} \uplus \overset{abort:}{\{\bot\}}$$

The statement composition operator now has to examine the result. If it encodes a normal state (the first injection), the next statement is executed, otherwise the execution is stopped.

In practice, the result type is much more complicated, for there are several possibilities of abnormal termination: `break`, `continue`, `return`, `goto`, and many more. Note that abnormal termination does not have to be ultimate. Usually, normal operation resumes at a certain point. Consider a `while` statement. If the body results in a `break` abnormality, execution is expected to continue after the `while` statement. Thus, the `while` statement has to »heal« this abnormality and yield a normal result. However, a detailed description is beyond the scope of my thesis. Tews and Reichel provide a practical introduction into this subject in [TR03, part II]. This approach was first used by Huismann in [Hui01].

Again abstracting from internal details, a statement's result is just denoted as type *Result*. The above *Statement* type can now be written as:

$$Statement \stackrel{\text{def}}{=} State \to Result$$

Certainly, statements sometimes require parameters. An `if`-statement needs at least 2 parameters: a boolean expression, and the statement that is executed if the expression evaluates to true. An optionally third parameter could provide a statement to be executed on false. However, for statement composition as well as for substatements like the second and third parameter of `if`, we need a common type.

Therefore, parametrized statements are evaluated in two steps. At first, the parameters are applied to the statement function, which returns a *Statement*:

$$\begin{aligned} if :\ & \mathbb{B} \times Statement \times Statement \to Statement \\ & (b, s_1, s_2) \mapsto \begin{cases} s_1 & \text{if } b = \text{true}, \\ s_2 & \text{otherwise.} \end{cases} \end{aligned}$$

Now, the program state can be applied to the returned *Statement*, which is a function itself ($State \to Result$, as declared above).

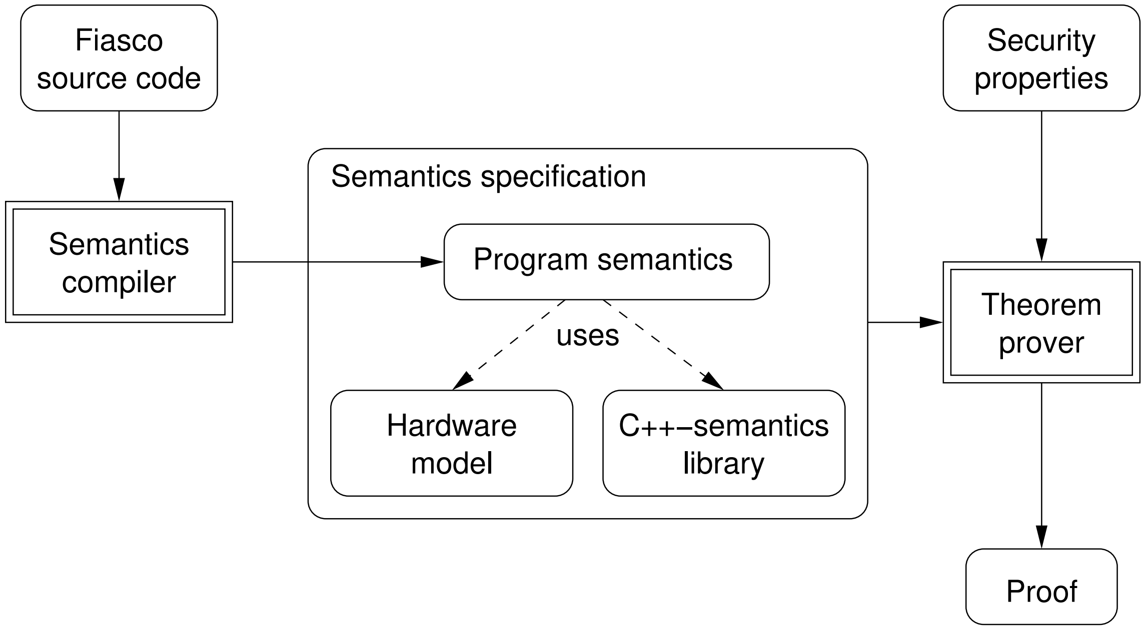

Figure 2.1: The VFiasco project

2.2 The VFiasco project at a glance

The VFiasco project aims to prove important security properties of the Fiasco-microkernel. Figure 2.1 provides an overview of the whole verification process.

At first, the semantics compiler translates the C++ source code of Fiasco into its semantics formulated in higher-order logic. The program semantics is expressed using functions provided by the hardware model [Hof03] and the C++-semantics library. All together, they form the semantics specification of Fiasco. Besides, one has to specify the security properties that should be proven. The theorem prover can now be used to verify the semantics specification against the security properties. The verification results in the proof.

2.3 Framework of existing components

> ≫ *If I have seen a little farther than others, it is because I have stood on the shoulders of giants* ≪ — *Isaac Newton.*

I could rely on a solid foundation of selected components that were tested, gathered, or entirely new developed prior to my work. As usual for most good software, all components are still under continuous development. Each component will be briefly described here.

OpenC++ [Chi99] is a free, open source C++ parser written by Shigeru Chiba and Grzegorz Jakacki. Its design focuses on the use for C++ translation and analyzing tools. The parser frees developers from worrying about tedious parsing work and provides a type system. Unfortunately, handling of types and names is quite rudimentary. For this and other reasons, some valid C++ code is misparsed [Reu03]. I started to work with version 2.5 and switched later to version 2.6.

The annotator [Reu03] enriches the output of OpenC++ with extended type information, and does overload resolution as well as some normalization for easier post-processing. Meanwhile it corrects some of the parsing errors of OpenC++. The annotator has been written by Stefan Reuther at the Dresden University of Technology.

liblogics provides an abstract view to a theorem prover's specification language. This serves two goals: On the one hand it simplifies switching between theorem provers, and on the other hand, it eases the output formatting. This programming library was mostly contributed by Michael Hohmuth. I slightly enhanced the library by adding many convenient shortcuts and some new functionality.

Based on the framework of these components and various less important utility classes, I developed the compiler's conceptual heart: the translator. This component transforms the annotator's output into an abstract specification that serves as input for liblogics.

2.4 Related work

Of course, there is a plethora of C++ compilers. However, it is a novel idea to translate C++ into a semantics specification in higher-order logic. Only a few related projects aim verification of programs in similar programming languages.

VeriOS is a project of the Saarland University, Saarbrücken.[1] It is a continuation of the *VAMP* project which formally verified a microprocessor. VeriOS is about to start and information can rarily be found. The project attempts to verify an implementation of the L4-microkernel [Lie96] that was written in C. I did not find out, which implementation this exactly is. It might be an early version of Hazelnut, developed at the University of Karlsruhe. The VeriOS project will also employ PVS as theorem prover and use shallow embedding. Up to my knowledge, the compiler development has started but is in a very early stage. Strong constraints on the input language make it impossible to compile real C code.

The *LOOP* [vdBJ01] compiler translates Java sources into a PVS representation. Due to the similarity of C++ and Java, this project was an excellent source of inspiration. However, C++ is more challenging than Java. While Java is a well-typed language with a fully specified semantics, C++ is not type-safe. So, for instance, the LOOP project uses a typed memory model, which is not feasible for C++.

[1] Public information on that project is quite rare. I could only find a small note at http://www-wjp.cs.uni-sb.de/forschung/forschung.php. Some additional facts I gained from a talk of Mark Hillebrandt on July 11, 2003 at the Dresden University of Technology.

3 Design goals

> ≫*If you want to build a ship, don't drum up the men to gather wood, divide the work and give orders. Instead, teach them to yearn for the vast and endless sea.*≪ — *Antoine de Saint-Exupery.*

Setting the goals of a project is of utmost importance. They will further characterize the problem, and set the fundamental course at the same time. In this chapter I will define the central goals and guidelines for my work.

3.1 Flexibility and extensibility

Verification on real software is a yet emerging trend in computer science, and reasoning about low-level C++ programs is a complete novelty. Consequently, the semantics representation cannot be fully established, but will be changed and further developed together with the compiler.

Many problems are expected to arise during the development process, and might lead to substantial changes. Thus, flexibility and extensibility of the compiler are vital to the whole verification project. Considering additionally the limited application area of the compiler, it becomes evident that a comprehensible structure is much more important than runtime efficiency.

3.2 High coverage

Right from beginning it was clear that I would not be able to finish the compiler development in the scope of my work. Some known problems, like those regarding evaluation order (see section 5.2.1), were explicitly excluded. In contrast, the aim was a high coverage, revealing as many hidden problems as possible.

To gain first results sooner, the kernel page-fault handler is prioritized in the VFiasco project. This module was designated as a practical touchstone for the compiler. Attention is turned especially to C++ constructs used there.

3.3 Architecture independent semantics representation

The standard is most vague about nearly all C++ constructs. It is evidently impossible to write, and yet verify an operating system without knowledge of the underlying architecture. On the other hand, most parts of the Fiasco kernel are machine independent. It would be tedious and resource wasting to manually reason twice about the same code and prove nearly identical properties over and over again.

The aim is to concentrate all architecture dependencies and hide them in a lower abstraction level. If the hardware is replaced, only these machine dependent levels should be exchanged, while the source code representation could remain static. If this can be achieved, proofs would either remain valid, or could be rerun automatically, but do not require costly human interaction.

3.4 Independence from the theorem prover

A rapid development is still ongoing in the quite novel area of theorem provers. First studies in the VFIASCO project started with Isabelle, a free, open source theorem prover. However, it did not fully satisfy the project's needs. PVS has a wider variety of features, but of course it is as well not free of bugs and problems. As a main disadvantage, this theorem prover is not open source, so one relies on support from the software developer.

The theorem prover should carefully be chosen, since a later switch is always a very costly operation. Yet, verification projects are long term projects, and changes might be inevitable in some situations. Therefore, it is desirable to abstract from the particular details of the theorem prover in charge.

3.5 Support for separate compilation

It is aimed to reason only about parts of the kernel. Evidently, this requires compiling of only these parts of the sources. Most reasonable, such a part will consist of a number of C++ translation units.[1]

It is conceivable to combine former separate parts, or add new components. It would be helpful if this would not require a recompilation of all parts. Instead, a linker is needed with similar requirements to the C++ linker.

[1] In C++, a translation unit is essentially a precompiled source file although the standard does not formally require that the sources must be stored in files.

4 Representing C++ semantics in higher-order logic

This chapter focuses on the semantics specification of C++. The various features of C++ are presented, and examined in detail regarding their semantics. However, this is primarily an introduction to the general approach, not a bare reference of supported features. On the contrary, possible options of representation are demonstrated, advantages and disadvantages explained, and the final decision substantiated. Open problems are discussed, and often an outlook is given to show how currently unsupported constructs can be rendered. On the other hand, I neglect tedious details where the general idea seems evident, and further specification is a matter of labor.

The development of the C++-semantics representation is the joint work of the people involved in the VFIASCO project. I will present here the current state of our collective development. My part of the work on this subject addressed the representation of some compound types (namely, pointer to members, references, arrays, and enumerations), the C++ memory model, expressions, and functions. Additionally, I refined the class representation and could reveal some problems in our current approach.

If I use the mathematical notation for functions, an omitted definition does not necessarily denote an uninterpreted function. Instead, functions are always defined in PVS unless it is explicitly stated otherwise. Declarations are marked with a prime ($'$) if they do not have a counterpart in the current semantics representation. Longer sections of PVS code presented in figures are illustrative only. They are not relevant for the understanding of my thesis.

4.1 Basic structure

4.1.1 Modelling statements and expressions—state transformers

The *program state* is modelled as a fixed type for the C++ semantics layer. This was done for simplicity even though the actual state representation may vary. However, it is not expected to reason about the same piece of code with different state representations in parallel. For the time being, the program state was declared as an uninterpreted type named *State*. This declaration will be replaced with the respective *State* definition tailored for the particular verification environment.

Expressions are a fairly complex construct in C++: On the one hand, expressions like assignments cause side effects. Hence expression evaluation might lead to a new state. On the other hand, the evaluation of expressions does not always yield a value. Certain expressions, like division by zero, are not specified by the standard and could cause a program crash. Moreover, the evaluation of an expression may not even terminate at all. A function call, for instance, could result in an endless loop, causing the program to hang.

These facts dictate the layout of expression functions. Obviously, it must take the current *State* as argument. On faultless evaluation, the result of an expression must encode the new program state and its value. In case of an abnormal termination, the result has to provide some error specific information. An oversimplified formal definition for illustration:

$$\mathit{Expression}'[\mathit{Data}] \stackrel{\text{def}}{=} \mathit{State} \to \overset{OK:}{(\mathit{State} \times \mathit{Data})} \;\uplus\; \overset{\mathit{fail}:}{\{\bot\}} \;\uplus\; \overset{\mathit{hang}:}{\{\bot\}}$$

with *Data* being a generic type parameter for the expression's type. OK names normal termination, *fail* and *hang* are examples for abnormal termination respectively for a program crash and an endless loop.

Statements, in contrast, do not usually[1] yield a value. However, they might cause a program to hang or fail as well. Certainly, there are many other abnormalities for statements as noted in section 2.1.2. Consequently, one could define two distinctive types for expressions and statements. This would require frequent conversions between result types. Hence, statements and expressions use one universal state transformer type.

The *state transformer type* is defined as:

$$\mathit{ST}[\mathit{Data}] \stackrel{\text{def}}{=} \mathit{State} \to \mathit{Result}[\mathit{State}, \mathit{Data}]$$

whereas *Data* is a generic type parameter for an expression's type, and *Result* denotes a complex type formed by a disjoint union of $^{OK:}(\mathit{State} \times \mathit{Data})$ and various abnormalities, error states, and hardware-conditioned faults. For statements, *Data* is always instantiated with the one-element type **1** (written `Unit` in PVS).

4.1.2 Data manipulation—objects and memory

The key concept of data manipulation in C++ is the (memory) object. An *object* is simply defined as a region of storage, that can be created, destroyed, refered to, accessed, or manipulated. An object has a *type* and a *storage duration.* The type of an object determines its size (memory consumption) and meaning (the storage's interpretation). Storage durations relate to the object's lifetime.

In C++ the available *memory* is conceived as one or more sequences of contiguous bytes. A byte is the fundamental storage unit. Every byte has a unique address, and consists of a bit-sequence with an implementation-defined length. Our formal semantics adopts the notions of *Byte* and *Address* (for a detailed explanation of the memory model see section 4.3).

Access to memory objects is usually gained through variables. A *variable* is introduced by an object declaration and provides a name for the declared object. The object's type and storage duration are precisely specified by the declaration, while its exact address will be arbitrarily chosen by the linker. Even for indirect access to memory objects, information on the expected type is always provided statically, whereas the address is determined at runtime. Moreover, memory objects may be reinterpreted with other types. Therefore, objects are identified only by their starting address.

4.2 Data types

There are two kinds of types within C++: fundamental types and compound types. The fundamental data types are basically formalized as specified in [HT03]. I will only summarize some general principles, here, and expand foremost upon the challenges of compound data types.

[1] According to the standard, a statement *never* yields a value. On the contrary, there is a GNU C extension called statement expressions, which allows the embedding of a compound statement as an expression. Although valid, this feature is strongly discouraged in C++.

4.2.1 Basic type semantics

The key design goal of data types is the abstraction from a bytewise organized storage to problem-oriented value sets. A type is characterized by its memory consumption and the interpretation of the used storage. Following from this, a formal semantics will count the memory consumption in bytes, and define the type's interpretation by functions mapping the value set ($Data$) to a list of bytes and vice versa:[2]

$$Data\_type\_structure[Data] \stackrel{\text{def}}{=} \begin{array}{l} {}^{size:}\,\mathbb{N} \;\times \\ {}^{to\_byte:}\,(Data \longrightarrow list[Byte]) \;\times \\ {}^{from\_byte:}\,(list[Byte] \dashrightarrow Data) \end{array}$$

Certainly, this definition neglects some basic restrictions on a type semantics specification, since size and interpretation functions are closely related:

1. For all $Data$, to_byte must return a list of $size$ bytes.
2. $from\_byte$ is only defined on byte lists that have a length of $size$.
3. $from\_byte$ must be a left-inverse of to_byte (i. e. $from\_byte$ is defined on all byte lists returned by to_byte, and yields the original value.)

These general properties can be best formalized by a predicate $data\_type?[Data]$, applicable to the above type. The predicate describes the set of possible data types. Every data type is represented by a constant of type ($data\_type?[Data]$). Whenever a $Data\_type\_structure$ constant is defined, one must show that it complies with the predicate. This fact is usually formulated in a theorem. An uninterpreted data-type constant can directly be declared of type ($data\_type?[Data]$), but one has to assure by axiom that this type is not empty (i. e. $data\_type?[Data]$ evaluates to true for at least one instance of $Data\_type\_structure[Data]$).

Note that data-type specifications usually involve uninterpreted constants constrained by axioms. If so, the soundness of the axioms must be ensured. This is currently established with the help of a *type model*. Such a model is a theory similar to the type specification. Instead of the uninterpreted constants in the type specification, specific definitions are provided in the model. Additionally, the axioms are replaced by theorems stating the same logical property. If all theorems can be proven, one has shown that the type specification is sound.

4.2.2 Fundamental types

Every fundamental type is described by a fixed value set, a constant of type $Data\_type\_structure$, and a finite number of axioms stating some additional properties. It should be mentioned, that the standard usually is quite vague about a type's details. Thus, a semantics specification relies on a certain compiler implementation in order to allow sensible reasoning.

Formalization of signed and unsigned integers is rather straightforward. Some additional care must be taken on character types. The standard postulates that arbitrary chunks of memory can be copied back and forth as character arrays without loss. Consequently, all bits of their object representation must be significant to character values. The floating point types are not specified since they are not needed in our project.

Type `void` is special. In the first place, it is used as the return type of functions that only produce side effects. The `void` type is specified to have an empty set of values. However, in certain situations expressions can have type `void` (see § 3.9.1). From a set-theoretic point of view this is rather confusing. Therefore we decided to represent type `void` by a one-element set requiring no storage.

[2] Note, that this is a slightly simplified version compared to [HT03]. However, this version is currently used. I will propose an alternative definition in section 4.7.10.

The standard defines the following conversions between the value sets of fundamental types: integral promotions, integral conversions, boolean conversions, and explicit conversions to type `void`. All these conversions can easily be encoded as simple state transformer functions. If fully specified by the standard, this is done by a function definition, otherwise the conversion function is declared and restricted by axioms.

4.2.3 Compound types

C++ allows the construction of compound types in several ways. This includes arrays, functions, various kinds of pointers, references, class types, and enumerations.

Although functions perfectly fit into the type model of C++, they must be treated completely differently regarding specification. They are not intended to be altered, created, nor destroyed during runtime. Functions are handled in section 4.6.

Class types are all varieties of composite types, including unions, C-like structures, and complex user defined data types with constructors and inheritance. They are fairly complex and often correlate with other concepts. For instance, class types may contain functions to manipulate their objects. Therefore, I will first describe more essential features and expand on classes later in their own section (4.7).

The rendering of the remaining compound types, I will explain in the following subsections. The C++ type conversions are covered together with each type.

Pointers

According to their targets, three groups of pointers can be differentiated:

- pointers to objects or `void`,
- pointers to ordinary[3] functions, and
- pointers to nonstatic class members.

Pointers to objects or `void` designate the starting address of a memory object. More precisely, every object pointer can only refer to objects of a particular type. The semantics model can either reflect the belonging type information as part of the pointer's value type, or provide it explicitly to the operands working with pointers.

In the former case, the value type is defined as:

$$Pointer'[Target\_type] \stackrel{\mathrm{def}}{=} Address$$

The generic type parameter might surprise, here, since it is not applied in the type definition. However, it introduces some type information from C++ into the representation, and thus allows PVS a stricter type checking. This is very useful since the standard does not require the same representation for all object pointers. Only the reinterpret cast of a null pointer must yield a null pointer, again. This condition could be expressed as $rcast\_p2p[T_1, T_2](ptr0[T_1]) = ptr0[T_2]$ for $rcast\_p2p$ performing a reinterpret cast between pointer types, and $ptr0[T]$ representing the null pointer of type `T*`.

However, it should be noted, that this type check does not automatically ensure correct use of pointers. In C++ two types can be distinct, even though identical by layout. This is not the case in PVS. Moreover, this approach breaks on user defined types. In certain situations,

[3] Nonstatic member functions are treated specially. All other functions I will collectively call ordinary functions.

C++ types can be used prior to their definition. A translation unit does not have to be aware of the internals of `struct X` just to encode a pointer to it:

```
struct X;
X* px;
```

Moreover, structure definitions may depend on each other, and lead to circular dependencies in PVS:

```
struct X;
struct Y { X* px; };
struct X { Y* py; };
```

The other alternative is that the compiler provides the type information on demand. For instance, the addition of an integer and a pointer could be performed by a function with the following signature:[4]

$$\overset{data\_type:}{(data\_type?[Data])} \times \overset{pointer:}{ST[Address]} \times \overset{offset:}{ST[int]} \longrightarrow ST[Address]$$

This approach is consistent with the usual translation of C++ into assembler. Certainly, addresses are not type-tagged in assembler. Wherever type information is needed, it is encoded explicitly. Therefore, we represent a pointer's value set by the plain *Address* type.

Note that this indirectly has an impact on reinterpret casts. They are currently modelled by translating the value of the source expression into its byte representation, and subsequently convert this into the destination's value representation (see section 4.4). If all pointer types share exactly the same value representation, their *to_byte* and *from_byte* functions are identical. This leads to the conclusion, that all pointer types must be represented in the same way. Since this is the case in practice, anyway, this restriction is not expected to be relevant.

Modelling of standard pointer conversions is straightforward. Null-pointer-constant conversion and conversion from arbitrary pointers to `void` pointers can easily be encoded as state-transformer functions. The derived-to-base conversion just corrects the pointer offset that is specified with the concerned derived class (see section 4.7).

Function pointers are traditionally treated fundamentally differently than pointers to objects. On some implementations they do not even share the same size. Moreover, the use of function pointers is quite limited: There is no pointer arithmetic and no dynamic creation or destruction of functions, for instance.

Functions remain at fixed locations during the whole execution time. Thus, a function pointer will *reliably* identify the function assigned to the pointer until reassignment. Consequently, the value set of function pointers must consist of some sort of function identifiers. This can be real addresses in a code segment of the memory, or just the (mangled) function names. Since the C++ model abstracts from the fact that functions are stored in memory, it will preferably use function names. However, function pointers are currently not translated by the semantics compiler.

The rendering of function calls on function pointers is described together with functions in section 4.6.

[4]Note that the function operates on expressions. Hence arguments and result are state transformers ($ST[Address]$), not plain value sets ($Address$), as explained in section 4.1.1.

Pointers to members can either point to subobjects within a user defined type, or to a member function. The latter can be modelled in analogy to pointers to ordinary functions. Pointer to data members can also be easily modelled. Essentially, they encode the offset of a particular subobject within a user defined type. The offset is always positive and could be represented by a plain *Address*. As we will see in section 4.7, member access usually operates on offsets.

Care must be taken with the null-pointer-constant conversion. A pointer-to-member null value must not have the value 0 because it must be distinct of any other value and 0 is a possible offset. The base-to-derived member-pointer conversion trivially works just like the derived-to-base conversion known from ordinary pointers.

The semantics compiler does not support pointers to members at the moment.

References

References have a semantics quite similar to constant pointers that are implicitly dereferenced upon each use. Therefore, we do not explicitly model references, but internally encode them as usual pointers, and add a dereference operation when used.

Under certain circumstances a compiler can optimize away a reference and spare its storage. This case is neither detected nor specially treated, since it is possibly rare, and should not have a considerable impact on the verification's result. Note that compilers might optimize away other non-`volatile` variables as well.

Arrays

Most of the time, arrays can be treated just like pointers. Array subscripting, for instance, is explicitly defined by pointer arithmetic (see § 5.2.1 (1)). The substantial difference between pointers and arrays is that only the latter have a size assigned to their type.

According to § 5 (8), array expressions are usually converted to pointers when used as rvalues. This conversion is only suppressed if the value is of no interest (e. g. for an expression statement—see § 6.2 (1)), or if array expressions are used as arguments for `sizeof` and `typeid` operators (see § 5.3.3 (4) and § 5.2.8 (3), respectively). The former case is just an optimization, only the latter case needs special treatment of arrays.

Lvalues of array types are only required in conjunction with references (e. g. a function returning an array reference—see § 3.10 (4), or a reference initialization—see § 8.5.3 (5)), and the address-of operator (see § 5.3.1 (2)). Increment, decrement, and assignment operations are not allowed with arrays. The use of the address-of operator basically just denotes the array-to-pointer conversion explicitly, and does not need further special care.

Additionally, array types may occur in declarations, casts, or wherever types are named directly. There, array dimensions are particularily important, if storage has to be allocated, or an array type acts as base type of a compound type (pointer, reference, class, or multidimensional array).

In short, arrays must be treated specially with the operators `sizeof` and `typeid`, when storage is allocated, and if part of a compound type. Notably, the value set is not used in all these situations. However, for the *Data_type_structure* a value set is formally required. I will denote it here as the uninterpreted type *Semantics_array* : *Type*. The particularily needed *Data_type_structure* can now be provided by a generic function:

$$array[Data] : \overset{dt:}{(data\_type?[Data])} \times \overset{size:}{\mathbb{N}} \longrightarrow (data\_type?[Semantics\_array])$$

Arrays are currently not handled by the semantics compiler.

Enumerations

Enumerations are distinct types associated with named constants. However, they are closely related to integral types. The underlying type of an enumeration is an integral type, and an enumeration's object must be able to hold all values in the range of the enumeration values (see § 7.2). When used in arithmetic contexts, enumerations are promoted to integral types.

Therefore, enumeration types are currently modelled as `int`, while the enumeration values are treated as constants. In some situations, this can lead to an inconsistency of formal specification and actual behavior. If a compiler decides to represent a certain enumeration in a smaller type, explicit conversion of integral values to the enumeration type might result in a different value. However, this is quite unlikely since »plain `int`s have the natural size suggested by the architecture« (§ 3.9.1 (2)), and other types would require frequent conversions. Problematic is an enumeration if its values do not fit into `int`. For the time being, `int` is the largest type supported by the compiler, and the annotator assumes that enumerations fit into `int`s, anyway.

Exact models of enumerations depend mostly on implementation details. However, the value set will always be an integral range including at least all enumeration values, and the interpretation will closely follow one of the fundamental type's interpretations (i. e. sizes are equal, and the conversion functions map common values to the same byte representation). If one restricts an enumeration's value set to this minimum, extra conversion functions must be defined for explicit conversions from integral types to the enumeration type.

4.3 Memory model

The hardware model [Hof03] provides the *plain memory* abstraction tailored to match the C++ memory model. According to the IA32 achitecture, addresses range from 0 to $2^{32} - 1$, and bytes consist of 8 bits:

$$\begin{aligned} \mathit{Address} &\stackrel{\text{def}}{=} \{n \mid n \in \mathbb{N} \wedge n < 2^{32}\} \\ \mathit{Byte} &\stackrel{\text{def}}{=} \{n \mid n \in \mathbb{N} \wedge n < 2^{8}\} \end{aligned}$$

The memory state is part of the program state; the exact details do not matter in this context. C++ only needs two operations on memory: read and write. The hardware model provides both for byte lists:

$$\begin{array}{rrcl} \mathit{memory\_read\_list}: & \mathbb{N} \times \mathit{Address} & \longrightarrow & \mathit{ST}[\mathit{list}[\mathit{Byte}]] \\ \mathit{memory\_write\_list}: & \mathit{Address} \times \mathit{list}[\mathit{Byte}] & \longrightarrow & \mathit{ST}[\mathbf{1}] \end{array}$$

The C++ layer, however, usually operates on typed values, not on plain byte lists. Therefore, read and write operations were provided with an additional data-type parameter, and the byte-list argument was replaced by a value set. Moreover, expressions are represented as state transformers for they might be composed from subexpressions:

$$\begin{array}{rlclcl} \mathit{read\_data}[\mathit{Data}]: & (\mathit{data\_type?}[\mathit{Data}]) & \longrightarrow & \mathit{ST}[\mathit{Address}] & \longrightarrow & \mathit{ST}[\mathit{Data}] \\ \mathit{write\_data}[\mathit{Data}]: & (\mathit{data\_type?}[\mathit{Data}]) & \longrightarrow & \mathit{ST}[\mathit{Address}] \times \mathit{ST}[\mathit{Data}] & \longrightarrow & \mathit{ST}[\mathit{Address}] \end{array}$$

4.3.1 Memory management and variables

As depicted in section 4.1.2, in C++, the memory is manipulated with the help of memory objects. To each object, a storage duration is assigned that determines the object's lifetime. C++ knows three kinds of storage duration:

- Global objects usually have *static storage duration.* For these objects, memory is allocated at program start and shall last for the duration of the program.
- Local objects have *automatic storage duration* by default. The storage for these objects lasts until the block, in which they are created, exits.
- Objects can also be created *dynamically* during program execution.

Static and automatic objects are named by variables, dynamic objects not. The latter are accessed with pointers (see section 4.2.3). *Variables* and pointer values are quite similar regarding the fact that both refer to memory objects. In contrast to pointers, only complete types can be used with variable definitions. Thus, one could annotate the variable representation with type information (as discussed with pointers) without causing circular dependencies. However, this was not done for the sake of consistency with pointers. Instead, variables are modelled as constant state transformers with an *Address* value, and thus, they share a common interface with pointer expressions.

The address for objects of static storage duration is arbitrarily chosen by the linker. Its exact value is not intended to be of interest during program execution, though it can be detected. Assuming it is irrelevant, static variables are represented as uninterpreted constants.

In analogy to C++, automatic variable names are introduced as local names in the semantics representation. Higher-order logic allows this via λ-terms. Automatic memory allocation is modelled with a special function. It needs two arguments: the data type of the requested object and a representation of the remaining block which forms the variable's scope. The second argument will usually consist of the λ-term that binds the new address to the desired variable name and contains the usual state transformer for its scope. This version is slightly simplified and will be extended for `goto` (see section 4.5.2):

$$\begin{aligned}&\mathit{with\_new\_stackvar}'[\mathit{Data}_1, \mathit{Data}_2]: \\ &\quad (\mathit{data\_type}?[\mathit{Data}_1]) \times (\mathit{ST}[\mathit{Address}] \to \mathit{ST}[\mathit{Data}_2]) \;\longrightarrow\; \mathit{ST}[\mathit{Data}_2]\end{aligned}$$

Local variables might also have static storage duration. In this case they are initialized when program control reaches their declaration statement for the first time. Therefore, an additional `bool`-variable is introduced internally to remember whether the variable was initialized already. The declaration statement is translated into a function call of:

$$\mathit{init\_staticvar}[\mathit{Data}]: \; \mathit{ST}[\mathit{Address}] \times \mathit{ST}[\mathit{Data}] \;\longrightarrow\; \mathit{ST}[\mathbf{1}]$$

The location of the internal boolean variable should be given as the first argument and an initializer expression as the second one. The function will read the boolean variable; if it is false, *init_staticvar* evaluates the initializer expression and sets the boolean variable to true. Note that the concerned local variable is not known to *init_staticvar*. The initializer expression must be an assignment expression or a constructor call. For illustration, the definition of *init_staticvar* in PVS is shown in figure 4.1.

Dynamic memory allocation is usually hidden in a library function, though FIASCO organizes it for itself. However, dynamic memory allocation is in general based on pointer arithmetic on character arrays and type-conversion mechanisms, so no additional language support is needed.

```
Static_Init[Data: TYPE]: THEORY
BEGIN
  IMPORTING Read_Write,          % provides read_data and write_data
            Fundamental_Types,   % provides dt_bool
            State_Transformer,   % provides State and ST[TYPE]
            Skip                 % provides skip

  init_staticvar (
    st_init?: ST[Address],       % address of the internal bool variable
    st_do_init: ST[Data]         % initializer expression
  ): ST[Unit] =
    eval_if_ok(read_data(dt_bool)(st_init?),
      LAMBDA(already_initialized: bool):
        IF already_initialized THEN
          skip
        ELSE
          to_unit(st_do_init ##
                  write_data(dt_bool)(st_init?, const(true))
                  )
        ENDIF
    )
END Static_Init
```

The auxiliary functions used within this code, are defined as follows:

$$eval\_if\_ok[Data_1, Data_2] :$$
$$ST[Data_1] \times (Data_1 \to ST[Data_2]) \longrightarrow ST[Data_2]$$

$$(expr, f) \mapsto \lambda\ s \in State. \begin{cases} f(value)(next) & \text{if } \exists\ next \in State, value \in Data_1. \\ & \qquad expr(s) = OK(next, value) \\ expr(s) & \text{otherwise} \end{cases}$$

$$\begin{array}{rrcl} skip: & State & \to & Result[State, \mathbf{1}] \\ & s & \mapsto & OK(s, \perp) \end{array}$$

$$to\_unit[Data] :$$
$$ST[Data] \longrightarrow ST[\mathbf{1}]$$
$$expr \mapsto \lambda\ s \in State. \begin{cases} OK(next, \perp) & \text{if } \exists\ next \in State, value \in Data. \\ & \qquad expr(s) = OK(next, value) \\ expr(s) & \text{otherwise} \end{cases}$$

$$\begin{array}{rrcl} const[Data]: & Data & \to & ST[Data] \\ & d & \mapsto & \lambda\ s \in State.\ OK(s, d) \end{array}$$

Figure 4.1: The complete definition of `init_staticvar`

4.4 Expressions

Expressions are most diverse in C++, and some of them are closely related to other concepts. Two expressions I already presented in the previous section: *write_data* encodes a simple assignment and *read_data* is technically an lvalue-to-rvalue conversion. Another example is the function call which is explained in section 4.6 together with functions. Translation of the comma expression is very similar to compound statements and depicted in section 4.5.3. Member access, `this` pointer, and dynamic casts are class-specific features, see section 4.7.

The vast majority of expressions can easily be modelled. Thus I will not expand on the obvious details, but shortly summarize general principles by example. The remainder is devoted to interesting aspects and challenging expressions.

Primary expressions are foremost variable names and literals, though formally they include function names and `this` pointers as well. However, function names may only be used in conjunction with function calls, since function pointers are not supported. The `this` pointer is very similar to a variable name.

Variable names can be used as they are for they denote constant state transformers. Literals are generally written as decimal numbers, wrapped in a state transformer via the *const* function:

$$\begin{aligned} const[Data]: \; Data &\rightarrow ST[Data] \\ d &\mapsto \lambda\, s \in State.\; OK(s, d) \end{aligned}$$

Most operators can be represented by state transformer functions, though the number of required functions and the layout of their signatures will vary greatly. As an example, the addition for arithmetic types is modelled by a function for each particular type, e. g. the addition of two `int` expressions is handled by:

$$st\_add\_i: \; ST[Semantics\_int] \times ST[Semantics\_int] \longrightarrow ST[Semantics\_int]$$

Pointer arithmetic works very similar except for the additional data-type parameter (as I already explained in section 4.2.3):

$$\begin{aligned} &st\_add\_P\_i: \\ &\quad (data\_type?[Data]) \longrightarrow ST[Address] \times ST[Semantics\_int] \longrightarrow ST[Address] \end{aligned}$$

The `sizeof` operator takes only a data-type argument:

$$dt\_sizeof[Data]: (data\_type?[Data]) \longrightarrow ST[Semantics\_int]$$

However, expression translation is not always that obvious. Consider an additive assignment (`+=`). Essentially, the additive assignment yields the same result as if the sum of both subexpressions were computed and assigned to the left-hand side. A problem arises if the subexpressions do not share a common type.

On binary arithmetic operators, both subexpressions are converted to the (wider) type, and the operation is performed on that type. This is not feasible for a compound assignment. Since the left-hand side is an lvalue, it cannot freely be converted to another type. The naïve solution is the definition of compound assignment operators for each type combination, but this leads to an explosion of operator functions.

Therefore, compound assignments are decomposed upon translation. For instance:

```
i += 5    // i is an int.
```

is translated into

```
% first: evaluate the left-hand side
st_evaluate(lvar_i, LAMBDA (lhs : ST[Address]) :

  % assigning the sum to the left-hand side
  write_data(dt_int)(lhs, st_add_i(read_data(dt_int)(lhs), const(5)))
)
```

Type conversions can be written in many different ways. However, from the semantics perspective, all conversions can be expressed by the four conversion operators:

- `static_cast`
- `reinterpret_cast`
- `const_cast`
- `dynamic_cast`

Static casts are implemented with usual state transformer functions, as explained with data types. All *reinterpret casts* are handled by a single function: *ri_cast* translates the given value via *to_byte* into its byte representation. The resulting byte list is subsequently converted via *from_byte* into the destination type's value. Const casts and dynamic casts are currently not supported.

Modelling *const casts* is technically trivial for constness is simply not reflected in the semantic representation. Normally this is not necessary. An object declared as constant is intended to remain unchanged for its whole lifetime. If no const casts are involved, the attempt to change a constant object can be detected statically. Therefore, the semantics compiler rejects programs that cast away constness.

Dynamic casts are only useful in conjunction with polymorphic classes which are not yet translated. A possible implementation for dynamic casts is presented in section 4.7.

Dynamic memory allocation. Operator `new` hides the call to an allocation function (with possible placement arguments), and the initialization of the newly created object. If initialization fails, the corresponding deallocation function must implicitly be called with the address of the created object and the placement arguments.

Due to possible placement arguments, a general mechanism for `new` operator is somewhat elaborate. In principle, `new` can be represented as a function taking

1. the type of the object to allocate,
2. the C++ allocation function `operator new`,
3. the C++ deallocation function `operator delete`, and
4. the initializer expression.

The problem is that the signatures of the C++ functions depend on the placement arguments. If they are not present, the `new` operator semantics can be rendered as:

$$\begin{array}{rl} st\_new[Data_1, Data_2] : & {}^{obj\_type:}\,(data\_type?[Data_1]) \;\longrightarrow \\ & {}^{op\_new:}\,(ST[Semantics\_int] \to ST[Address]) \;\times \\ & {}^{op\_delete:}\,(ST[Address] \to ST[Semantics\_void]) \;\times \\ & {}^{init:}\,ST[Data_2] \\ \longrightarrow & ST[Address] \end{array}$$

Operator delete works quite similar, it is yet easier because of the lack of placement arguments. Both operators are not supported at the moment due to limited support by the annotator.

4.5 Statements

It is a complex task to render statements like loops or jumps of an imperative programming language in higher-order logic. Here, the divergence of the design principles emerges most impressively. However, many statements in C++ are conceptually similar. Essentially, one can classify 5 groups:

- expression statements,
- selection statements,
- iteration statements (i. e. loops),
- jump statements, and
- declaration statements.

The most challenging statements are loops and jumps. It is far beyond the scope of this thesis to discuss them here in detail. The general approach was already depicted in section 2.1.2. A practical introduction can also be found in [TR03, part II].

From the compiler's perspective, the rendering is surprisingly simple since the actual semantics is hidden in wrapper functions. While statements, for example, are translated as application of the following function:

$$stm\_while : \overset{condition:}{ST[\mathbb{B}]} \times \overset{body:}{ST[\mathbf{1}]} \longrightarrow ST[\mathbf{1}]$$

Thus I will not elaborate on the various problems, but confine my descriptions to a few interesting aspects of selected statements which relate to my work.

4.5.1 Expression statements

Earlier in this project, statements were represented just like expressions as arbitrarily typed state transformers although the value is not needed. This was expected to increase readability of generated PVS code because of the following observation: Expression statements are used most frequently and consist just of a single expression. If statements can have any state transformer type, the expression can be left as is. If, in contrast, all statements are required to be of type $ST[\mathbf{1}]$, expression statements must explicitely converted. However, due to ambiguities in connection with labeled statements, conversion is performed now.

4.5.2 Labelled statements and `goto`

A `goto` statement yields a *goto* abnormality, initiating a jump to the corresponding label. Consequently, this abnormality is characterized by the current program state and the desired target:

$$Result[State, Data] \stackrel{\text{def}}{=} \ldots \uplus \overset{goto:}{(State \times Labels)} \uplus \ldots$$

The jump target is defined by a labelled statement. A labelled statement must »catch« a *goto* abnormality and resume normal operation, i. e. yield a normal *Result*. Catching an abnormality requires an argument of type *Result*, but usually statements take just a program state as argument and only yield a *Result*. To differentiate both statement types, I will call statements simple if their argument is a *State*, and complex if it is a *Result*.

While jumping to a label, it is possible to transfer into a block, but not in a way that bypasses declarations of automatic variables with initialization. Therefore, the *with_new_stackvar* function introduced in section 4.3 has to be adapted in two ways: 1. An additional argument memorizes whether the new stack variable has an initializer. 2. A version for complex statements will be able to catch a `goto` jump trying to bypass this declaration. The changed stack allocation functions are:

$$\begin{aligned}&\mathit{with\_new\_stackvar}_{\mathit{simple}}[\mathit{Data}_1, \mathit{Data}_2] :\\ &\quad (\mathit{data\_type?}[\mathit{Data}_1]) \times \mathbb{B} \times (\mathit{ST}[\mathit{Address}] \to \mathit{ST}[\mathit{Data}_2]) \;\longrightarrow\; \mathit{ST}[\mathit{Data}_2]\end{aligned}$$

$$\begin{aligned}&\mathit{with\_new\_stackvar}_{\mathit{complex}}[\mathit{Data}_1] :\\ &\quad (\mathit{data\_type?}[\mathit{Data}_1]) \times \mathbb{B} \times (\mathit{ST}[\mathit{Address}] \longrightarrow (\mathit{Result}[\mathit{State}, \mathbf{1}] \to \mathit{Result}[\mathit{State}, \mathbf{1}]))\\ &\qquad \longrightarrow \quad (\mathit{Result}[\mathit{State}, \mathbf{1}] \to \mathit{Result}[\mathit{State}, \mathbf{1}])\end{aligned}$$

The selection of the appropriate function is done via overload resolution in PVS.

`goto` can also be used to construct loops, i. e. the target is a label prior to the corresponding `goto` statement. In analogy to the usual loop statements, this is handled by *goto_block*, a function catching and »healing« the (possibly abnormal) result. The scope of a label is the whole body of the corresponding function. Therefore the complete function body is wrapped into *goto_block* if a label is found in it. To ensure consistency, the catch function takes a list of defined labels as argument:

$$\mathit{goto\_block} : \overset{\mathit{labels}:}{\mathit{list}[\mathit{string}]} \times \overset{\mathit{function\_body}:}{(\mathit{Result}[\mathit{State}, \mathbf{1}] \to \mathit{Result}[\mathit{State}, \mathbf{1}])} \;\longrightarrow\; \mathit{ST}[\mathbf{1}]$$

If it catches a *goto* abnormality with a label not in this list, the program is ill-formed, and *goto_block* will terminate with *fail*. Otherwise, the C++ function's body is evaluated again with the result applied to it.

4.5.3 Compound statements

For readability, statement composition is expressed by a special right associative operator[5] (denoted as $\odot$ in this thesis). The composition operator is overloaded for the two kinds of statements. Simple statements take a *State* as argument, complex statements a *Result*. This requires 4 forms of statement composition:

$$\begin{aligned}\odot_1 :\ & \mathit{State} \to \mathit{Result}[\mathit{State}, \mathit{Data}_1] \times\\ & \mathit{State} \to \mathit{Result}[\mathit{State}, \mathit{Data}_2] && \longrightarrow \quad \mathit{State} \to \mathit{Result}[\mathit{State}, \mathit{Data}_2]\\[1ex] \odot_2 :\ & \mathit{State} \to \mathit{Result}[\mathit{State}, \mathbf{1}] \times\\ & \mathit{Result}[\mathit{State}, \mathbf{1}] \to \mathit{Result}[\mathit{State}, \mathbf{1}] && \longrightarrow \quad \mathit{Result}[\mathit{State}, \mathbf{1}] \to \mathit{Result}[\mathit{State}, \mathbf{1}]\\[1ex] \odot_3 :\ & \mathit{Result}[\mathit{State}, \mathbf{1}] \to \mathit{Result}[\mathit{State}, \mathbf{1}] \times\\ & \mathit{State} \to \mathit{Result}[\mathit{State}, \mathbf{1}] && \longrightarrow \quad \mathit{Result}[\mathit{State}, \mathbf{1}] \to \mathit{Result}[\mathit{State}, \mathbf{1}]\\[1ex] \odot_4 :\ & \mathit{Result}[\mathit{State}, \mathbf{1}] \to \mathit{Result}[\mathit{State}, \mathbf{1}] \times\\ & \mathit{Result}[\mathit{State}, \mathbf{1}] \to \mathit{Result}[\mathit{State}, \mathbf{1}] && \longrightarrow \quad \mathit{Result}[\mathit{State}, \mathbf{1}] \to \mathit{Result}[\mathit{State}, \mathbf{1}])\end{aligned}$$

The simple composition operator ($\odot_1$) is also used to model the C++ comma operator and thus has a *Result* type for arbitrary $\mathit{Data}_{1,2}$. The other versions operate only on statements that do not yield a value.

[5] To be precise, the left associative `##` is used in PVS, and right evaluation order is enforced by parentheses.

4.6 Functions

This section will describe ordinary functions only. By *ordinary* I refer to all but nonstatic member functions. The latter have to be treated special and will be discussed in section 4.7 together with classes.

A C++ function body is just a compound statement. When translated into higher-order logic, its semantics is wrapped into the *catch_return* function that »heals« an abnormality caused by a `return` statement, and possibly the *goto_block* function as described in section 4.5.2. The result is defined as a PVS function that I will refer to as the *body semantics function*.

In C++, a function call requires some effort: The caller must provide storage for the parameters, and initialize them with the corresponding arguments. Depending on architecture and calling conventions, some registers might have to be saved on the stack. The called function may save some registers as well, and change the stack frame. Upon return, the caller has to destruct the parameters and will possibly free their storage.

While this is usually encoded in assembly language each time the function is called, it is worthwile to wrap this procedure in a *call semantics function* when translating the C++ function into higher-order logic. The architecture dependent calling conventions are concealed in the PVS functions `caller_prepare_stack` and `callee_prepare_stack`. Both functions take two arguments: the C++ function name and a state transformer containing the translated function body. This allows a very flexible modelling of the internal calling conventions, even if they differ from C++ function to C++ function.

As example, consider a trivial C++ function:

```
void f() { return; }
```

In PVS, the above function would be represented as:

```
% body semantics function
fun_f : ST[Unit] =
  catch_return(
    stm_return  % semantics of the return statement
  )

% call semantics function
call_f : ST[Semantics_void] =
  % hide stack changes
  caller_prepare_stack("f",
    callee_prepare_stack("f",
      fun_f  % call body semantics function
  ) )
```

This simple example, however, neglects parameters and return values. Apparently, parameters are ordinary automatic variables whose lifetimes span the duration of the function call. About the value return mechanism of a function, on the contrary, the standard is quite vague. In practice, the return value is treated in analogy to an automatic variable that is initialized upon `return`.

The call semantics function will allocate and initialize the automatic variables and present them as arguments to the body semantics function. Afterwards, the return value is read and variables are deallocated (implicitly upon return from *with_new_stackvar*). For example,

```
int f(int i) { return i; }
```

would be translated into:

```
% body semantics function, takes variables as arguments
fun_f(lvar_i : ST[Address], retval : ST[Address]) : ST[Unit] =
  catch_return(

    % initialize the return value
    write_data(dt_int)(retval, read_data(dt_int)(lvar_i)) ##

    stm_return
  )

% call semantics function
call_f(arg_i : ST[Semantics_int]) : ST[Semantics_int] =
  caller_prepare_stack("f",

    % allocate and initialize the automatic variables
    with_new_stackvar(dt_int, false, LAMBDA (retval : ST[Address]) :
      with_new_stackvar(dt_int, true, LAMBDA (lvar_i : ST[Address]) :
        write_data(dt_int)(lvar_i, arg_i) ##

        callee_prepare_stack("f",

          % call body semantics function
          fun_f(lvar_i, retval) ##

          % deliver the return value
          read_data(dt_int)(retval)
  ) ) ) )
```

A function call is translated into PVS by applying the (translated) arguments to the corresponding call semantics function. If a C++ function call is applied to a function pointer, the situation is slightly more complicated because the call semantics function must be first figured out. This can be done by an uninterpreted PVS function that maps C++ function names to their call semantics. The function has to be specified by axioms: For each newly introduced C++ function, an axiom must be defined, stating that the function name maps to the corresponding call semantics function. Note that function pointers are not supported at the moment.

Unfortunately, the straightforward translation described here is not always possible. In contrast to C++, recursive function calls are strictly limited in PVS. The next section will explain, why.

4.6.1 Recursive function calls

≫ *To iterate is human, to recurse divine.* ≪ — *L. Peter Deutsch.*

Recursive function calls are a very convenient way to express certain algorithms in C++. However, the free and unrestricted use of recursion stands in conflict with the type system of PVS. In C++ it is perfectly legal to define a function that causes an infinite loop for certain arguments (and thus, does not produce a result). On the contrary, all functions in PVS must be total.[6]

Therefore, PVS permits only a restricted form of recursive function definition: mutual recursion is prohibited, and a so called *measure function* must be specified on a recursive definition. This measure function allows the proof of totality. Its signature must match the signature of the defined function but has a range type of `nat` or `ordinal`.[7] If the measure function is strictly monotonic decreasing for recursive function calls, the recursion must finally terminate.

Obviously, one cannot translate a recursive C++ function into a PVS function the same way as this is done for non-recursive functions. Fortunately, recursive function calls were completely avoided in the design of FIASCO. Thus, the semantics compiler simply always assumes non-recursive functions; if this does not hold, it generates invalid code which would result in a typecheck error when loaded into PVS.

However, correct handling of recursion is generally possible, even though it clashes with some design issues. Since recursive functions are treated substantially differently, they must be known as such *in advance*. C++ itself does not give any hints about whether a function is used recursively. Hence, all functions must be examined to reveal cyclic call dependencies. At first, this means the compiler needs to pass the code one more time, before the actual translation starts. Currently, translation occurs in just one pass. Second, an examination of all function bodies obviously implies the availability of the complete definitions for *all* used functions. In that, it opposes the C++ concept of separate compilation, even though recursion occuring in a single translation unit could be handled correctly.

One might argue that recursion involving multiple translation units will usually lead to circular dependencies and thus, should be avoided anyway. This might be true for simple, global functions, but it does not hold for virtual member functions. Consider the following example. There is a tree of nodes, held in a linked list. An abstract base class provides a visitor for this tree:

```
template <typename NodeInfo>
struct NodeList {
  NodeInfo  data;   // this node's application data
  NodeList* next;   // next node of the same level in tree
  NodeList* down;   // first node of the next lower level in tree
}

template <typename NodeInfo>
class TreeVisitor {

public:
  virtual void processNode(NodeList<NodeInfo>* n) = 0;
  void recurse(NodeList<NodeInfo>* n);
}

inline void TreeVisitor::recurse(NodeList<NodeInfo>* n)
{
  do processNode(n) while (n = n->next);
}
```

[6]A function is total if and only if it is defined for every value of its domain. In other words: the function yields a result for any given argument.

[7]Technically, a measure function can have any range type as long as a well-founded binary order relation is assigned to that type.

In a second translation unit, the above framework is used to define a visitor for a specific tree:

```
class MyVisitor : public TreeVisitor<int> {
  int level;

public:
  void processNode(NodeList<int>* n)
  {
    cout << std::string(' ', level*2) << n->data <<std::endl;
    level++;  recurse(n);  level--;
  }
}
```

Here, the recursive function call is encapsulated in the virtual function call mechanism. If the `recurse` function is not inlined, as shown in this little example, the translation of the second translation unit cannot detect the recursion, even though there is no circular dependency between the translation units.

4.7 Class types

A class type combines several members of possibly different types, forming a new data type. Since the FIASCO source code is shallow embedded into PVS, each single class has to be modelled individually.

The generally simple concept of class types shows fairly diverse varieties, ranging from plain C-like structures over unions to complex user defined data types. The standard distinguishes POD (plain old data) classes and non-POD classes. Essentially, POD classes are layout compatible with the C data types. They may be copied in raw bytes between memory locations, while objects of non-POD classes must be fully controlled by constructors and operator functions.

POD classes are very similar to C-like structures and unions. However, they may also have static members and member functions (even though no constructors, destructors or assignment operator). The distinguishing features of non-POD classes are constructors, destructors, inheritance, and virtual functions. Non-POD classes are not supported at the moment.

In the following sections I will develop the full range of class types step-by-step.

4.7.1 Plain C-like structures

In C, a structure provides the possibility to collectively manage a group of related memory objects as a whole. The new type is constituted by several named data members that have arbitrary object types, for example:

```
struct Sample { char c;  int i; };
```

Intuitively, the value set of a structure is simply the cartesian product of all data members' types, e. g.:

$$\mathit{Values\_Sample} \stackrel{\text{def}}{=} \overset{\mathit{mem\_c}:}{\mathit{Values\_char}} \times \overset{\mathit{mem\_i}:}{\mathit{Values\_int}}$$

The memory layout of the new data type depends in the first place on the members. Certain constraints are defined by the standard, and other decisions are implementation-defined. Since all members must be stored separately, the structure will consume at least as much memory as all the data members together, but might be larger due to alignment requirements.

Note that the access to a data member is independent from the remaining structure. As long as the structure is not accessed as a whole, it is perfectly legitimate to work with partly initialized structures. To reflect this in the data-type model, an offset into the structure's object representation is declared for each data member. When a member is accessed, only the bytes belonging to it will be determined and manipulated with the interpretation functions of the member's type. For convenience, the offsets are grouped in a cartesian product, and a member access operator is defined:

$$\mathit{offsets\_Sample} : {}^{\mathit{offs\_c}:}\mathbb{N} \times {}^{\mathit{offs\_i}:}\mathbb{N}$$

$$\mathit{st\_addr\_offs} : ST[\mathit{Address}] \times \mathbb{N} \longrightarrow ST[\mathit{Address}]$$

Offsets and size are generally uninterpreted to allow implementation-defined padding. However, some constraints are postulated by the standard and will be reflected in axioms:

- All member objects completely fit into the structure's object.
- Member objects do not overlap.
- The offset of the first member is 0 (for all POD-classes).
- The members are ordered in memory as declared in the source code (unless divided by an access-specifier label).

The interpretation of a structure's object representation is now mostly determined. Basically, it is the combination of the member access functions. The only part that is left to the implementation, is the setting of padding bytes when object representations are constructed from a value. As usual, the *Data_type_structure* is declared uninterpreted and refined by axioms, that ensure:

- If the structure can be read as a whole, each member value can be read individually and yields the same value.

- If all members can be read, the structure can be read as a whole and its value will be the composition of the read member values.

4.7.2 Unions

In contrast to structures, unions store the value of at most one member at a time. The concered member is said to be *active*. This might suggest a value representation as disjoint union of the members' value sets.

However, unions can be used to implicitly reinterpret values. This practice relies on the fact that all members of a union are stored in it with offset zero. If a value is assigned to the union via one member, it can be read via another one, and the result is the same as if a reinterpret cast were used.

Taking a closer look on the actual usage of unions, one discovers that only three operations are allowed on them: read, write, and member access. Read and write rely basically on the conversion from a byte list to the internal value representation and vice versa. Here, the actual value representation does not matter as long as all values can be represented. The member access operator works directly on lvalues (memory objects).

Consequently, the actual value representation does not matter at all. For simplicity, it could just be its object representation (a byte list). The semantics compiler does not support unions at the moment.

4.7.3 Bit-fields

Bit-fields are not distinct data types, but special members within class types. Unfortunately, the standard lets the implementation define allocation and alignment issues. Since FIASCO usually requires a particular memory layout, bit-manipulation operators are used on fundamental data types instead of bit-fields. Therefore, they do not need to be modelled within the VFIASCO project.

4.7.4 Static class members

Essentially, static class members can be treated analogous to global variables or functions, respectively. Differences relate only to the name resolution and the visibility. Both can statically be checked by the compiler[8] and do not have to be reflected in higher-order logic.

4.7.5 Nonstatic member functions

Nonstatic member functions differ from ordinary functions only by the new `this` keyword. It denotes a non-lvalue expression whose value is the address of the object for which the function is called. The standard does not oblige a particular implementation for `this`. The semantics compiler passes the object's address via an implicit parameter into the function.

The parameter is special since `this` is not an lvalue. The call semantics function takes as first argument a reference to the class object, which will be implicitly provided when the member function is called. In contrast to explicit parameters, this one is not copied but internally evaluated and provided to the function body. The function body uses the value like a pointer's rvalue.

The call semantics for a member function `f(int i)` would look like:

```
call_f(arg_this: ST[Address], arg_i: ST[Semantics_int])
: ST[Semantics_void]
= caller_prepare_stack("f",
    with_new_stackvar(dt_int, true, LAMBDA (lvar_i: ST[Address]):
      write_data(dt_int)(lvar_i, arg_i) ##
      st_evaluate(arg_this, LAMBDA (this: ST[Address]):
        callee_prepare_stack("f", fun_f(this, lvar_i))
  ) ) )
```

4.7.6 Access specifiers

The idea of access specifiers is the declaration of access rules for class members. They can be checked by the compiler and need not be reflected in higher-order logic.

However, they are also relevant in determining the ordering constraints of data members. Successive data members must be successively stored in memory unless divided by an access specifier. This subtleness is not reflected at the moment since the annotator does not provide information on access specifiers. Instead, it is postulated that all data members are ordered as they appear in the sources.

[8] The semantics compiler does not perform these checks. It relies on the C++ compiler and just assumes correct code.

4.7.7 Constructors and destructors

Constructors and destructors can be handled very similarly to usual member functions. The characteristic difference of constructors are the member initializers. They can just implicitly be added to the front of the function body. The only specialty of destructors is that they have implicitly to be called prior to deallocation of a memory object. Appropriate function calls can be statically inserted wherever necessary since variable scopes and exit paths are known at compile time.

4.7.8 Inheritance

Base classes can be reflected in a subclass hierarchy in the semantics. For member access, the offsets of the base class members can either be supplemented to the offset's record or be computed on a member access operation.

Care must be taken if base classes have no nonstatic members. While complete memory objects require storage by definition, a base class might have size zero. This could be expressed generically by an additional field in the data type structure. For virtual base classes, axioms have to ensure that the values for the shared subclasses are always identical.

With the `static_cast` operator, pointers to a derived class can be converted to pointers to a base class. This might require an adjustment of the pointer to the offset of the base class, and can be handled in analogy to the member access. The base class offsets can just be stored with the member offsets, especially since similar rules apply for them (fitting into the structure, no overlaps).

4.7.9 Virtual functions

Virtual functions require late binding. Therefore, the value set of classes with virtual member functions will be equipped with an extra type identification field. A binding semantics function is declared for each virtual function. The binding semantics is specified by axioms. This allows a later refinement when new subclasses are introduced. However, this approach does not work with recursion.

Dynamic casts can easily be modelled: If expressing a derived to base conversion, they can just be treated like a static cast. For downcasts, the polymorphic type identification field is used to determine the validity of the cast. Pointers might have to be readjusted. The offset can be found in the offset record of the derived class.

Dynamic casts are not used within FIASCO since it is currently not prepared for runtime library support. Instead, compile-time polymorphism is used where applicable (see [War03]).

4.7.10 Remaining problems

The value representation of class types is still open. The problem refers to member-function calls on rvalues. When calling a member function, the object's address must be known for the this pointer. However, the current rendering does not preserve the object's address in the value representation.

This is especially problematic if the class is of non-POD type. A non-POD class cannot be copied in raw bytes between memory locations but the copy constructor and the assignment operator cannot be called either since these are member functions. Consequently, an rvalue of non-POD class type can effectively neither be copied nor used.[9]

The key observation is that the value is not independent from the address. If the address became an additional part of the value's representation, it would be somewhat redundant since the value can just be determined with the help of the address. As observed with unions, a class type can only be read, written, or its members accessed. Thus, there is no need to encode the value representations of the members and base classes within the class's value; it is sufficient to hold only a reference to the memory object.

However, the current interpretation functions for data types do not have access to an object's address. A new definition of the data-type structure can change this:

$$\begin{aligned} \mathit{Data\_type\_structure}'[\mathit{Data}] \overset{\text{def}}{=} & {}^{\mathit{size}:}\,\mathbb{N} \;\times \\ & {}^{\mathit{to\_byte}:}\,(\mathit{Data} \longrightarrow \mathit{list}[\mathit{Byte}]) \;\times \\ & {}^{\mathit{from\_byte}:}\,(\mathit{list}[\mathit{Byte}] \times \mathit{Address} \longrightarrow \mathit{Data}) \end{aligned}$$

Note that the *from_byte* function should still fail unless all members can be read from the byte list. This can be assured by axioms.

[9] Only the data member access operation could be encoded for an rvalue of a non-POD class

5 Translating C++ source code into PVS

This chapter describes design and implementation of the translator component. At first, I will explain how the translator interacts with other components. In the troublemakers section, I tell about conceptual obstacles encountered during the design phase. Afterwards, I give a general overview of the translator's internal structure. I conclude with selected implementation details and pecularities.

5.1 Interaction of components

The OpenC++ parser represents source code as a list of parse trees. The trees consist of linked `Ptree` objects. `Ptree` is an abstract base class. Its various subclasses correspond to C++ constructs, including all kinds of statements and the great many varieties of expressions. The annotator processes these trees, constructing similar ones. The resulting trees are compatible to the input's parse tree structure, but expressions are annotated. Annotations can be revealed by a dynamic cast of expression `Ptree`s to the `Annotation` class.

The annotator replaces the linear code representation of OpenC++ by a symbol table. It collects all information on the items declared in the translation unit. Variables, types, and functions can be looked up here, and are linked to the corresponding parse trees. The symbols are not intended to be ordered in a particular way. Since order of global variable declarations is significant for program semantics, they are additionally stored in a list.

During translation, the symbol table and the list of global variable declarations are scanned. Variable declarations, functions, classes and enumerations are processed and translated into higher-order logic, expressed in the terms of liblogics. The result is a hierarchy of theories with possible dependencies. They are ordered according to their dependencies, formulated in PVS using liblogics, and written out into a file.

For processing of parse trees, the annotator uses a `Paranoid_visitor` class template. It is based on the visitor design pattern [Ale01]. The only template parameter defines the return type of the `visit` method. Reasonably, this class template is re-used by the translator.

5.2 Troublemakers in design

> ≫... *[the nations] shall hear all these statutes, and say, Surely this great nation is a wise and understanding people.*≪ — *Deuteronomy 4:6b*

5.2.1 Implementation-defined behavior, or: The art of omitting

Originally, C++ was designed mainly for low-level system programming in a convenient and machine-independent manner. It is the attempt to realize a most runtime efficient, highly compiler-optimizable language without committing to a certain, underlying machine. Combining such contrary design goals naturally yields some drawbacks. As a trade-off, certain constructs are not fully specified. Those specifications must be substantiated by the implementation in charge. Implementation-defined details have to be known in various situations in order to compile—and sometimes even to write—C++ programs. Some I will explain.

The type system—machine dependencies

For efficiency, the type system depends on the machine's data types. This, in turn, even effects the calculation of some constant expressions (namely those containing a `sizeof`-expression), as well as type determination of constant expressions (often, the type is specified in dependency on whether its value fits into a certain type), and the representation of a type itself (calculation of array bounds, or the underlying type of an enumeration, a `size_t`, or similar higher-level types).

The semantics compiler, however, tries to abstract from these machine-dependencies wherever possible. Unpleasantly, most stages of compilation need information on the underlying machine at some point. As mentioned above, the annotator has to make some assumtions to correctly determine the type of certain constructs. The translator needs type information for correct rendering of enumerations, and to perform some conversions. Last but not least, the semantics representation does fully specify the types in PVS.

Evaluation order—room for optimizations

For optimization purposes the standard does not specify a particular evaluation order for all expressions. However, this might lead to different program behavior since side effects might take place during evaluation. In that case the program is ill-formed, but it is hard to determine this automatically.

The intuitive solution, to make equality of results a proof obligation, is not feasible because it results in enormous complexity. Therefore a particular strict order must be picked. For the time being, this was carefully chosen to match the order of the used C++ compiler in most cases. Further development has to address order constraints and enforce a fixed, strict order for both compilers.

It should be noted that the concept of separate compilation introduces a subtle source of indetermination: There is no guaranteed order of initialization of global variables in different translation units. If initializers are mutually dependent, the result relies on the evaluation order.

5.2.2 Numerous theories—structuring the output

A specification consists of theories that group declarations, axioms and theories that belong together. At the beginning it was thought to group all global definitions in only two theories per translation unit: one for global variable declarations and another for all global function definitions. User defined types require their own set of theories.

Soon it turned out that this would be impractical in conjunction with classes and multiple translation units. Of course, class methods should be defined in one of the class theories, but a method can call a global function and vice versa. If all global functions live in one theory and methods in another, this might lead to circular dependencies between theories. The situation is depicted in figure 5.1. A similar problem arises if two translation units are mutual dependent.

The solution is whether to put *all* function definitions of all considered translation units into one giant theory, or create a new theory for every function. Taking into account that a theory is an administrational unit for PVS, the latter option was chosen.

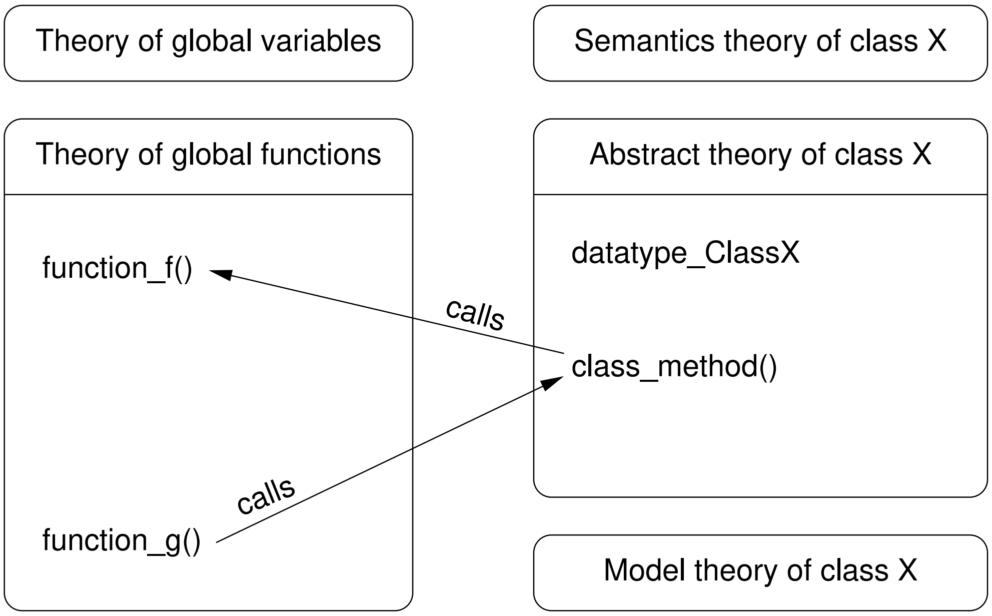

Figure 5.1: Example for a circular dependency if all global functions live in one theory

5.3 Structural overview

This section outlines general design and structure of the translation component. Obviously, some interfaces and design decisions were self-suggesting or yet already determined by the used framework. Furthermore, many design issues arose late during the implementation process due to the semantics representation was developed together with the compiler's implementation. This certainly sometimes influenced specific decisions. However, all decisions were made based foremost on the general design criteria discussed in chapter 3.

5.3.1 Processing statements and expressions

Like the annotator, the translator has to examine parse trees in various situations. Evidently, both should share the same mechanisms for similar tasks. Therefore, the translator re-uses the `Paranoid_visitor` class template from the annotator. The return type of the `visit` method is the translation's result. Usually, this will simply be a theorem prover's formula.

However, under certain circumstances it might be useful to return some additional information. For instance, the expression translator could carry around a type annotation. This is not done in the current implementation, but was kept in mind for flexibility and extensibility. Since statements and expressions might have quite different needs on additional information, two distinct visitors were implemented to allow different return types. `Sc_stmt_translator` and `Sc_expr_translator` translate respectively statements and expressions into their semantics representation.

One-pass translation

The parse tree structures are passed exactly once. This is just done for simplicity. A one-pass system is a clean and simple approach with a long tradition. C and C++ were actually designed to allow a single-pass translation. All entities (such as types, functions and variables) must be declared before they can be used; and declarations always provide just enough information to encode the use of the declared entity.

Nonetheless, the compiler—regarded as a whole—already passes the code multiple times. At first, it is completely parsed by OpenC++; subsequently, the parse trees are processed by the annotator, and finally, the symbol table is translated. If needed, even more passes could be easily introduced. This might become necessary if possibly recursive functions will be translated (see section 4.6.1).

Top-down analysis

The translator analyzes the code top down. This was considered as the easiest and most obvious translation scheme. However, it requires that all needed information about subtrees is available in advance.

For statements, this is easy to accomplish since virtually no information on substatements or subexpressions must be known. For processing of expressions, in contrast, plenty of type information is needed in various situations. This is provided by the annotator to allow a straightforward top down analysis.

5.3.2 Context information and theory management

Many translator objects are involved in a single translation process. Usually, a new object is created for every level descended in a parse tree. Yet some context information is shared with multiple levels, and must be passed down upon object creation.

This information is collected in `Sc_scope`. As the name tells, it was originally intended to reflect variable scopes, but today it provides information about theories, goto labels, and more.

The most important task of `Sc_scope` is currently the theory management. All theories needed for translation are created and registered here together with possible dependencies. The theory database is stored in static data structures of `Sc_scope`. When translation is finished, the theory implementations can be retrieved. The theories are returned as a list, which is ordered with respect to possible dependencies. The table below summarizes the theory names and functions in relation with the different C++ declarations:

Related item	**Theory name**	**Function**
translation unit	`Var_`*file*	collects the constants defined for global variables
	`Init_`*file*	contains initializer functions for global variables
class	`C_U`*class*	home of the data type's value representation and the constants for static members
	`Abstract_U`*class*	hosts the abstract definition of the data-type structure
	`Model_U`*class*	contains a sample model for the abstract data-type structure
any function	`Fun_`*function*	collects the definitions that belong to the corresponding function

5.3.3 Initial ignition: the `Sc_prog_translator`

The external interface of the translator subsystem is provided by the `Sc_prog_translator`. To start a translation, the static `translate` function must be called. It will set up the translation system and create a program-translator object. The latter scans the annotator's symbol table and examines the global variable declarations, translating user defined types, functions and variable declarations. During this process, statement and expression translators are frequently

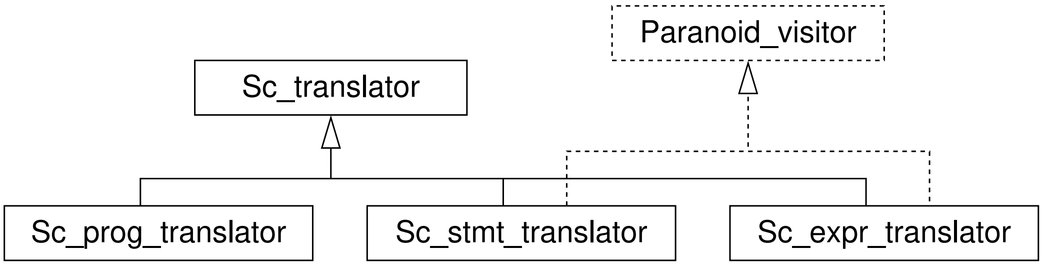

Figure 5.2: Inheritance diagram of the translator hierarchy

called to process function bodies or variable initializers. At the end, the initial startup code for the initialization of global variables is generated, and the theories built during translation are gained from `Sc_scope` and returned. The main function will finally write them out via the liblogics' `pvs_print` function.

5.3.4 Translator hierarchy

`Sc_stmt_translator`, `Sc_expr_translator`, and `Sc_prog_translator` inherit from `Sc_translator`. The latter serves as an abstract base class, providing functionality common to all translator classes. This includes a remarkable amount of utility functions, an immense variety of constants, and some type definitions, as well as mechanisms to manage the context information. Figure 5.2 shows the inheritance diagram.

5.4 Implementation pecularities

5.4.1 Standard conversions

The standard obliges to perform lots of implicit conversions[1] during compilation. It is more than reasonable to make those explicit by the annotator. However, the standard's fashion to define certain constructs does not allow this in every case. Namely, I refer here to compound assignments and increment operators.

Compound assignments are defined as follows (§ 5.17 (7)):

> *The behavior of an expression of the form* ***E1*** op= ***E2*** *is equivalent to* ***E1*** = ***E1*** op ***E2*** *except that* ***E1*** *is evaluated only once.*

To perform a binary operation, usually both operands are first converted into a common type which is determined by both operands and represents the result's type as well. Practically, this type is usually the wider one of the operand's types. Thus, a compound assignment might involve two type conversions: At first, ***E1*** is converted to the (wider) type of ***E2***. Then, the operation is performed and afterwards, the result is converted into ***E1***'s type.

Here, the design of C++ makes it hard to keep type annotation and translation cleanly separated. Therefore, the translator partly re-uses the same implementation, that the annotator uses for conversion of binary operators. This was foremost done to avoid inconsistency by design since some conversion decisions are based on assumptions on the machine.

To make things worse, the prefix increment operator[2] is defined in § 5.3.2 (1) to be equivalent to ***E1*** += `1`. Since `1` is of type `int`, this involves two implicit type conversions if ***E1*** is of a

[1] see § 4: Standard conversions

[2] Respectively, the same holds for the prefix decrement operator.

narrower type. Contrary to the compound assignments, the increment operator is not encoded like ***E1*** = ***E1*** + `1` for simplicity and readability. Instead, it is represented by special operator functions (`st_incr_`<type>). They have to be defined with the specification of the C++ semantics layer.

The functions' definitions can either faithfully model the whole set of conversions as defined in the standard, or optimize there. I prefer the latter since prefix increment is simple enough to overlook the correctness of optimizations, and is in practice never implemented the way it was defined.

5.4.2 Determining the valueness of expressions

As *valueness* of an expression I denote whether an expression is an lvalue or an rvalue. In short, lvalues refer to an object in memory (think of a variable), rvalues conventionally not.[3] Lvalues can automatically be converted into rvalues (i.e. the value of the memory object is read), and under certain circumstances rvalues can be turned into lvalues (by creating a temporary variable).

In earlier versions of the annotator, these valueness conversions were not always denoted explicitly. Most times this is not necessary since the valueness can determined by the context. For example, on an assignment operator, the left-hand side must always be an lvalue, and the right-hand side an rvalue. Thus, both subexpressions are just *processed as* lvalue and rvalue, respectively. When an expression is processed as rvalue—for instance as right-hand side of an assignment—and an lvalue is encountered, it is automatically converted into an rvalue.

However, there are expressions whose valueness is determined by their subexpressions: the comma operation and the conditional operation (`?:`). I will only elaborate on the comma operator here since both operators work analogous regarding the valueness. The comma operator just inherits type and valueness from its last subexpression. Consider the following example:

```
int i, j = (1, i);
```

Here, two variables `i` and `j` are defined, the former is implicitly initialized with zero, the latter explicitly with the comma expression `(1, i)`. This comma expression itself is an lvalue as `i` is, and is later converted to an rvalue as initializer of `j`. In a C++ like pseudo-notation this could be written as:

```
int i = 0; int j = rvalue_cast<int>((1, i));
```

For the `rvalue_cast` is indicated, the comma expression itself will be processed as an lvalue. Otherwise, the annotated comma expression `(1, i)` would be processed expecting an rvalue. To determine the comma expression's valueness (and generate the `rvalue_cast` if necessary), the last subexpression has to be examined. However, for top-down processing of the subexpressions as described above, the expected valueness has to be provided. An easy way out of this problem is to just pass the valueness expected from the comma expression down when processing its last expression although this would slightly change the result:

```
int i = 0;
int j = (1, rvalue_cast<int>(i));
```

This might not be important to a particular implementation, but it clearly contradicts the standard's specification. Therefore, all lvalue-to-rvalue conversions are made explicit by the annotator.

[3]For a detailed description and exceptions see § 3.10, Lvalues and rvalues.

5.4.3 Verification on program parts

A translation unit forms the smallest, self-contained part one can separately compile in C++ but their translation into object code is just a first step. All parts must be linked in order to form an executable program. Verification, on the contrary, is aimed on parts of a program.

Thus, a C++ compiler will only generate code for completely defined entities while the semantics compiler has to declare all entities to generate a valid specification. Especially in case of an external function the declaration in logic will be quite generic for the unknown function body.

If different parts of the program semantics are combined later, the duplicated declarations have to be eliminated. This is a quite costly operation. Therefore it should be avoided if not necessary. Instead, multiple translation units could be compiled at the same time. Moreover, recursive function calls involving multiple translation units are a serious problem since recursion cannot be detected upon separate compilation. If the translation units are compiled at the same time, this would not be a problem. However, the annotator does currently not support annotation of multiple translation units at the same time.

The compiler does not yet fully support partial compilation. Undefined symbols are currently just neglected. In case of an undefined function, the necessary call semantics function is missing, and a warning is issued. If variables are not defined, the compiler generates incorrect importings. It wrongly assumes the variable would be defined in the translation unit's theory for global variables. Additionally, names of global variables with internal linkage clash if the same name is used in different translation units.

5.4.4 Treatment of constant expressions

In certain contexts, C++ requires expressions that evaluate at compile time to an integral or enumeration constant. They are called *constant expressions* and may be used, for instance, as array bounds, as enumerator initializers, and as static member initializers. However, since constant expressions may involve the `sizeof` operator, evaluation in general can be architecture dependent. Annotator and translator, on the contrary, aim to be architecture independent where possible. Consequently, both do not evaluate constant expressions, but defer it completely to the theorem prover.

For the time being, constant expressions are not even detected as such. This yields some subtle changes in the semantics. The standard distinguishes two phases of initialization. Objects with static storage duration are statically initialized if the initializer is a constant expression. Otherwise they are dynamically initialized in the order of their declaration. In particular, static initialization takes place before dynamic initialization.

In contrast, the translator does not recognize constant expressions and performes all initializations in the declaration's order. This will lead to a different result if a dynamic initializer will use a variable, that is later defined and initialized with a constant expression:

```
extern int i1;
int i2 = (0, i1);  // comma expression ==> dynamic initialization
int i1 = 5;
```

Here, `i1` should be statically initialized with 5, and `i2` dynamically with the value of `i1` (5). Instead, the translator would first zero-initialize `i1` and `i2` (this must be done prior to any other initialization, see § 3.6.2), and then dynamically intialize `i2` with `i1` (0) before `i1` is initialized with 5.

If the `sizeof` operator is not involved, constant expressions could be evaluated. However, currently evaluation is never performed because it is unclear whether the full expression might help to understand the program specification. On the one hand, the original expression might document its meaning; on the other hand, a long, complex expression is usually rather confusing.

5.4.5 Miscellaneous open issues

Certainly, the semantics compiler is not ready yet. Besides the missing features mentioned earlier, the following issues have to be addressed in the further development (a complete list of missing features can be found on table 6.1):

Conversions are currently handled quite undifferentiated by the annotator. As a result, static casts cannot be cleanly differentiated from reinterpret casts. Thus, the translator initially treats every conversion[4] as a static cast. If the attempt of a static conversion fails, a reinterpret cast is encoded as fallback. There is no special handling for null pointer constants.

Enumerators are currently treated as constant variables. Such an approach must surprise since enumerators do usually not have an address. This is just a quick interim solution as long as constant expressions are not supported, and must change in the near future.

Character literals are interpreted as numbers at the moment. This is a generally avoidable machine dependency. However, it is expected not to be important. A machine-independent encoding would be quite elaborating.

References are rendered as auto-dereferencing pointers. At the moment, the dereference operation is implicitly supplemented in the `visit_name` function. Evidently, this cannot change the `Ptree`'s annotation. Thus, the reference type must still be adjusted to its basis type at all places. This problem could be solved with a type annotated formula as return type for the expression translator.

Temporaries are not yet supported.

[4] Up to now, only the semantics of static casts and reinterpreted casts are rendered in the model while const casts and dynamic casts are not supported.

6 Quo vadis?—Assessment and future work

≫ *Only he is poor who has no dreams.* ≪ — *Stefanie Zweig.*

Certainly, the semantics compiler in its current state is by far not finished. However, it greatly advanced the development of the semantics representation. Many concepts could be proved in practice, and even more new ideas arose during the development process. Problems, especially regarding the value representation of classes types, could be revealed and can now be addressed.

Further work on the semantics representation has now to focus on two issues: On the one hand, the C++ semantics library must be set up. For the time being, it consists foremost of uninterpreted declarations. On the other hand, general rendering of non-POD classes has to be cleared up. Though it should be noted here, that other issues might restrict the use of non-POD classes. This concerns especially the evaluation order (see section 6.1.1).

Independently from this rather theoretical work, the compiler can be extended to support arrays. This has not been done yet for former limitations of the annotator. Still, additional support is needed from the annotator for constant expressions, for casts, and for access specifiers.

Reviewing the design goals, one can draw the following conclusions:

- The compiler has a flexible structure, and future extensions can easily be realized. Though the goals of flexibility and coverage were competing each other in a few cases, this did not affect the overall structure.
- It must be confessed that the current coverage of the compiler does not meet the high expectations aimed at the beginning. It is not possible to compile the kernel page-fault handler. However, the compiler can handle a substantial part of the C++ programming language, and missing features are mainly the result of external open issues. Table 6.1 lists the missing features with their respective reasons.
- The semantics compiler is mostly architecture independent. A few dependencies were inherited from the annotator. This regards the standard conversions carried out in conjunction with compound assignments and the underlying type of enumerations. Additionally, the compiler depends on an architecture's character encoding, but this is not considered to be important.
- Moreover, the compiler is independent from the theorem prover as long as it supports all currently used features of PVS. Attention was paid to keep this set of used features as small as possible.
- Finally, it must be stated that support for separate compilation is not yet implemented. The compiler can easily be extended to generate declarations for declared but not yet defined items. Processing of several translation units at the same time needs support by the annotator. To join separately compiled program parts, an extra linker program must be built.

Missing C++ feature	Reason / remarks
Data types (section 4.2)	
function pointers	not needed
pointers to members	not needed
arrays	former limited support of the annotator (now possible)
enumerations[1]	the annotator does not support constant expressions
Expressions (section 4.4)	
`const_cast`	not wanted—for full support, the memory model had to be extended to support read-only memory
`dynamic_cast`	relies on virtual functions
new & delete	the annotator has only limited support for delete
Statements (section 4.5)	
switch-case	rendering yet open
Functions (section 4.6)	
recursive function calls	not needed
Class types (section 4.7)	
unions	not needed (rare, usually avoidable)
bit-fields	not needed (rare, usually avoided)
access specifiers[2]	need support from the annotator
non-POD classes	rendering yet open
Implementation pecularities (section 5.4)	
partial compilation	could be done
constant expressions	need support from the annotator
temporaries	partially relies on constructors (non-POD classes)
member access[1]	relates to rendering of classes
conversions	need support from the annotator

[1] This feature is supported but support is incomplete.
[2] Access specifiers are not directly rendered but are relevant for the ordering of data members.

Table 6.1: Currently not supported C++ features

6.1 Auxiliary tools

Besides the semantics compiler itself, some auxiliary tools are needed to form a complete compiler suite. This includes a pre-compiler to determine a strict order of evaluation and prevent some compiler optimizations that could lead to an observably different behavior. The pre-compiler is called Determinator. The work on it has just started.

Furthermore, a semantics linker is desirable to combine several autonomous compiled program parts. The linker will be interesting once the compiler can handle partial compilation. Up to now, only a requirement definition exists.

This section gives a short introduction to both tools.

6.1.1 Determining the evaluation order—the Determinator

As depicted in section 5.2.1, the order, in which expressions are evaluated, is not specified in the standard although it is sometimes significant to the program behavior. Therefore, one single, fixed, strict order must be enforced for both, the semantics compiler and the C++ compiler. This is best done in a pre-compilation step, transforming arbitrary C++ code to a restricted subset. This subset will limit the use of expressions with potential side effects in subsequent expressions.[1]

The Determinator is aimed to be a small tool, performing only minimal changes. However, it already turned out that sometimes substantial changes are necessary to ensure a particular evaluation order. Thus, the Determinator might completely eliminate some C++ constructs that are currently not rendered. For instance, the Determinator has to prevent the in-place construction of class types. Therefore, all constructors are transformed into usual functions.

6.1.2 The linker

As mentioned in chapter 3, it would be helpful to link separately compiled program parts together. This is clearly beyond the scope of this work. However, a requirement definition does also help to extend the compiler for partial compilation of several translation units (see section 5.4.3).

When the semantics compiler encounters a C++ declaration of an externally defined item, it must declare it in higher-order logic in order to use it. These declarations are generated for each translated part and duplicates have to be removed when different parts of the program semantics are later combined.

Here, three different kinds of external declarations must be regarded: classes, functions, and variables. If a class is only declared but not defined in a translation unit, no declaration is needed—it can only be used with pointers or external variable declarations. However, a class might autonomously be defined[2] in two translation units. In this case the class specification is duplicated, but both definitions are exactly identical. The linker can remove either one.

In contrast, if a function is only declared, the compiler will produce a generic declaration. For the program part containing the function definition, a detailed declaration is generated. The linker has to preserve the latter when joining these parts. This might lead to additional dependencies between theories. Though PVS does not formally demand theories to appear in a particular order in files, especially file-recursive dependencies are error-prone. Therefore,

[1] Note that it is neither possible nor desirable to completely avoid side effects in subsequent expressions. Consider an assignment of a function call's result to a variable: Certainly, a function call might have side effects and the assignment itself is an expression.

[2] Note that a class definition itself does not include the member-function definitions.

theories should be reordered regarding their dependencies, and files splitted or joined to solve recursive dependencies between files.

Finally, for each external variable, the compiler does generate an uninterpreted declaration for its name. These declarations should be collected in an extra theory with a universal name. When two program parts are linked, both instances of this theory must be merged, eliminating duplicates. Additionally, declarations for variables defined in one of the program parts must be removed. Instead, all theories containing global variable declarations (`Var_`*file* – see section 5.3.2) must be imported. This way, importings on theories that reference externally defined variables need not to be adjusted.

Of course, the semantics linker should perform the same consistency checks as the C++ linker, and detect:

- inconsistent declarations (e. g. `extern double b;` vs. `int b;`),
- double definitions (e. g. `int i;` vs. `int i;`),
- different implementations of an inline function, and
- different definitions of classes or templates.

Another problem arises with the ongoing compiler development. In software development it is feasible to recompile the complete source code when switching to a new compiler release. The object files are usually not compatible with different compiler versions.

In the VFiasco project, on the contrary, frequent changes to the semantics compiler are common while the sources will remain quite static. Here, it is very desirable that a new compiler release does not require a complete recompilation. Especially slightly changed naming conventions could trash valuable and time-consuming proofs.

References

[Ale01] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley, Boston, MA, 2001.

[Chi99] Shigeru Chiba. *OpenC++ 2.5 Reference Manual.* Institute of Information Science and Electronics, University of Tsukuba, 1999. Available via http://opencxx.sourceforge.net/.

[Hof03] Sarah Hoffmann. *Formalising PC Hardware: A Model of the x86 Achitecture.* Diploma thesis, Dresden University of Technology, September 2003.

[Hoh96] Michael Hohmuth. *Linux-Emulation auf einem Mikrokern.* Diploma thesis, Dresden University of Technology, August 1996. In German; with English slides. Available at http://os.inf.tu-dresden.de/~hohmuth/prj/linux-on-l4/.

[Hoh98] Michael Hohmuth. The FIASCO kernel: Requirements definition. Technical Report TUD–FI–12, Dresden University of Technology, December 1998. Available at http://os.inf.tu-dresden.de/papers_ps/fiasco-spec.ps.gz.

[HT03] Michael Hohmuth and Hendrik Tews. The semantics of C++ data types: Towards verifying low-level system components. In *Proceedings of Theorem Proving in Higher-Order Logics (TPHOLs), Emerging Trends*, Rom, Italy, September 2003. Available at http://vfiasco.org/types-tphols-2003-a4.ps.

[HTS02] M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel — the VFIASCO project (extended abstract). In *Proceedings of the Tenth ACM SIGOPS European Workshop*, September 2002.

[Hui01] M. Huisman. *Reasoning about Java programs in Higher-order logic using PVS and Isabelle.* PhD thesis, University of Nijmegen, The Netherlands, 2001.

[ISO98] International Organization for Standardization, Geneva, Switzerland. *ISO/IEC 14882:1998: Programming languages — C++*, September 1998.

[Lie96] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer, Berlin, 1996.

[OSRSC01] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference, Version 2.4.* SRI International, Menlo Park, CA, December 2001.

[Reu03] Stefan Reuther. *Annotator Documentation*, 2003. Available via http://os.inf.tu-dresden.de/~sr21/c++/.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Co., Boston, MA, Special edition, 2000.

[THH01] H. Tews, H. Härtig, and M. Hohmuth. VFiasco — towards a provably correct μ–kernel. Technical Report TUD-FI01-1 – January 2001, Dresden University of Technology, Department of Computer Science, 2001. Available via http://wwwtcs.inf.tu-dresden.de/~tews/science.html.

[TR03] Hendrik Tews and Horst Reichel. *Programmverifikation und -spezifikation mit Coalgebren*, 2003. Available via http://wwwtcs.inf.tu-dresden.de/~tews/SpecLecture.

[vdBJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer Verlag, 2001. Available at http://www.cs.kun.nl/~bart/PAPERS/tacas01.ps.Z.

[War03] Alexander Warg. *Software Structure and Portability of the Fiasco Microkernel*. Diploma thesis, Dresden University of Technology, July 2003.