Nova Micro-Hypervisor Verification

Formal, machine-checked verification of one module of the kernel source code (Robin deliverable D.13)

Hendrik Tews* Tjark Weber* Marcus Völp†
Erik Poll* Marko van Eekelen* Peter van Rossum*

* Radboud Universiteit Nijmegen, The Netherlands
http://www.sos.cs.ru.nl/

† Technische Universität Dresden, Germany
http://www.tudos.org/

May 30, 2008

Revision 74

This work has been supported by the European Union through PASR grant 104600.

1 Executive Summary

This document describes our achievements in work package 4 (kernel specification and verification) of the Robin project towards the verification of selected parts of the Nova micro-hypervisor. Despite organizational difficulties that were beyond our control (see below) we were able to finish our task successfully. In this line of work we achieved the following results.

1. A precise formalization of the IA32 hardware in the interactive theorem prover PVS, see Sections 4.1–4.6. The formalization faithfully models all the peculiarities of the real hardware that can lead to subtle errors in kernel programming. It includes a novel approach for formally describing memory-mapped devices. Well-behaved and well-typed memory (which are necessary to reason efficiently about the Nova source code) are established as an invariant and a set of theorems on top of the low-level formalization. Well-behaved and well-typed memory therefore rest only on properties that the Nova hypervisor itself ensures and not on additional assumptions.

2. A formal semantics in PVS of a sufficiently rich subset of C++, see Section 4.7. The semantics follows the C++ standard [Int07b], Compiler specifics can be added as additional assumptions in an orthogonal way. The semantics covers all C++ statements, expressions and data types with a few exceptions (such as `goto` and `float/double`) that are not needed for Nova.

3. A semantics-compiler prototype capable of translating C++ into its semantics in PVS, see Chapter 3. The semantics compiler is based on Olmar, our novel OCaml framework for parsing C++.

4. The whole verification environment has been successfully used in several case studies, see Section 4.8. As proof of the expressive power of our IA32 hardware formalization we modelled a memory mapped device. Additionally we verified a small search algorithm in C++. The verification of about 100 lines of Nova source code has been started, the proofs are however not finished yet.

During the project we were confronted with two major difficulties. Firstly, because of administrative difficulties our second postdoc in work package 4, was not able to timely renew his working permit for the Netherlands and had to quit his contract in the project in the sequel. Secondly, substantial parts of the Nova hypervisor source code became only available very late in the project.

Because of these difficulties the original goal of verifying selected properties of one module of the Nova hypervisor was not sensible any more. Instead we focused on a deeper and more exact formalization of the IA32 hardware and attempted smaller case studies. As a result we achieved a new goal that was not planned originally: We developed a novel approach for the formalization of memory mapped devices and proved its applicability in a small case study (see Section 4.8.4).

Contents

2 The Robin Verification environment

This document describes the work performed and the achievements reached in work package 4 (kernel specification and verification) of the Robin project towards the goal of verifying selected properties of the Nova micro-hypervisor source code (which was developed in parallel in work package 1). In order to reach our goals we built a verification environment for kernel-level C++ code, *the Robin verification environment*. The Robin verification environment is based on (1) a formalization of the execution context of kernel-level C++ code in the interactive theorem prover PVS [ORR+96] and (2) a semantics compiler that translates C++ code into its semantics in PVS.

In this introduction we first analyze the challenges in kernel verification and list in Section 2.2 (on page 9) the requirements the Robin verification environment should fulfill. Section 2.3 (on page 9) describes our verification approach and, finally, Section 2.4 (on page 11) contains the results that we achieved.

In the remaining document Chapter 3 describes the semantics compiler in detail and Chapter 4 gives a detailed introduction into the formalization of the execution environment in PVS. For completeness Appendix B contains the complete PVS source code and Appendix C contains the proofs for all the results in the PVS source code. These two appendices exceed 3,000 pages, they are therefore only contained in the electronic version, available at `http://www.cs.ru.nl/~tews/Robin`. The PVS source code can also be downloaded from that web-site.

2.1 Challenges for low-level systems-code verification

The Nova micro-hypervisor is an operating-system kernel written in C++. Its low-level code deviates sufficiently from examples in textbooks about verification or semantics. Therefore the development of a suitable semantics and the verification were challenging in several ways.

C++ source code Currently, there seems to be no convincing alternative to C/C++ for kernel programming. Consequently the Nova micro-hypervisor is written in C++. C++ programs are more difficult to formalize than, say, Haskell or Modula3 programs, for a number of reasons:

- The C++ standard [Int07b] is relatively vague in order to permit conforming C++ implementations on the weirdest platforms. For instance, the signed integral types are not required to contain negative numbers. Further, casts between different pointer types might change the pointer (to satisfy alignment requirements), except for the case where one casts to `void *` and back to the original pointer.

Because of the vagueness of the C++ standard almost every program relies in some way on platform or compiler specific properties. Consequently, a formalisation of C++ program must incorporate some properties of the specific C++ implementation that is used to compile the program.

- The template mechanism of C++ alone is Turing complete [Vel]. This means, the compiler can be forced to do arbitrary computations *at compile time*. A formalisations of C++ templates is accordingly difficult.

  We were never aiming at a semantics of C++ templates. During the project we never dealt with source code containing templates. If we would have encountered templates, we would have used the template instantiation facilities of our semantics compiler to generate a semantics for instantiated template code.

- Type casts and `goto`-jumps are features that are traditionally not handled in textbooks on program semantics. However, it is impossible to write a micro hypervisor without typecasts. Moreover, in order to avoid unduly performance penalties one needs some kind of unstructured jump such as `setjmp/longjmp` [lon] or continuations at a few, selected places.

Embedded assembly code and direct hardware manipulations For operations that are not supported in C++ (mostly direct hardware manipulations) the hypervisor sources contain some assembly code, mostly in the form of inlined assembly. Assembly code is needed at least for the following operations:

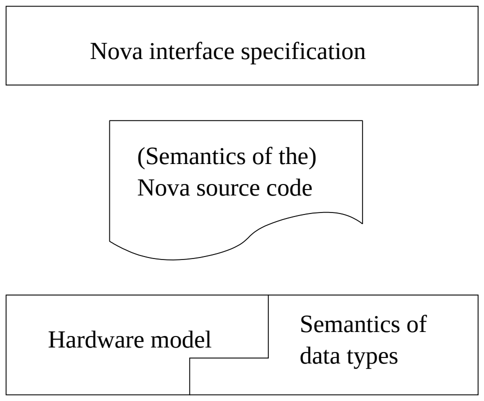
- Access to hardware registers, such as those from the APIC (Advanced Programmable Interrupt Controller), but also special processor registers, such as CR3 (page-directory base register), EFLAGS (the flags register), the global descriptor table, the interrupt descriptor table, the task-segment register and the feature control registers CR0 and CR4.

- Embedding special instructions in the code, such as IRET (return from interrupt), INVLPG (invalidate a TLB entry).

- Manipulating the stack frame to access and modify parameters of system calls or for programming non-local exists such as `longjmp` of continuations.

Nonstandard program environment The hypervisor runs like usual programs in virtual memory. However, the hypervisor manipulates the virtual memory mapping itself. Some parts of the memory are visible multiple times at different virtual addresses. One can therefore have very subtle aliasing: A variable x at address a_1 can be changed by writing to the totally different address a_2.

The hardware manipulations that the hypervisor must perform bear the possibility of very subtle errors. Some reserved bits in hardware data structures, such as the page directory entries, must be zero. Other reserved bits have an unknown value and must not be changed.



The hardware model and the semantics of data types provide the basic operations and properties for the verification of the hypervisor. For hardware data types the hardware model relies on the semantics of data types. Technically

$$\Phi_{data\_types}, \quad \Phi_{hardware} \quad \vdash \quad \varphi(\text{hypervisor})$$

where φ is one property from the specification, such as *termination without runtime type errors.*

Figure 2.1: Robin verification approach

2.2 Requirements for Kernel Verification

Because of the special nature of kernel-level code we aimed at a verification environment that meets the following requirements.

Correct execution environment The hardware features used in Nova must of course be modelled correctly. Hardware features not or only partially used must be modelled such that any wrong use of them definitely triggers an verification error.

Kernel programming errors Apart from common programming errors the verification environment must also catch as many as possible kernel specific programming errors. Such errors include reserved bit violations, casting errors and errors resulting from a broken virtual address translations

C++ Standard The verification environment should follow as closely as possible the C++ standard [Int07b] (even though the standard is ambiguous). Compiler specific assumptions are added in an orthogonal way only where needed.

2.3 The Robin Verification Approach

We give now an overview about our verification approach and the design of the Robin verification environment. Our approach is depicted in Figure 2.1, it has already been worked out in the VFiasco project [HT05, HT]. We rely heavily on the interactive theorem prover PVS [ORR+96], which is therefore a key component of the Robin verification environment. The input language of PVS is higher-order logic enriched with predicate subtyping and some other forms of dependent types. Higher-order logic contains a complete lambda calculus. For the verification one therefore models the system at hand in a functional way inside PVS and later uses the prover component of PVS to establish theorems about it.

We use source-code verification, that is we translate the C++ source code into a set of specific functions that are defined in the PVS input language. With source-code verification one can benefit from the relatively high abstraction level present in the source code (which is lost in object code). However, source-code verification also means that we do not directly verify the object code that will really be running.

In our approach the translation of the C++ code into PVS depends on a formalization of the runtime environment of the kernel in PVS. This runtime environment consists of a number of parts that can roughly be split into a hardware model and a semantics of data types, as depicted in Figure 2.1. The runtime environment provides a formalization for

- relevant IA32 hardware details
- state transformers as semantic domain for C++ statements and expressions
- physical memory, virtual memory and an abstract memory interface for kernel programming (called plain memory)
- memory allocation
- C++ statements and expressions

For a detailed description see Chapter 4. The formalization does not blindly model the reality. Instead the modelling is done in such a way that certain subtle programming errors yield a specific error state instead of doing nonsense. For instance the attempt to interpret a string as a page-directory entry yields an abnormal result value. This kind of error checking works even for the hardware initiated page directory traversals done during address translation.

The Robin verification environment consists of

- the interactive theorem prover PVS,
- the formalization of the kernel runtime environment in PVS
- the semantics compiler implementing a denotational semantics for (a subset of) C++ in the higher-order logic of PVS

Figure 2.2 depicts the data flow in the Robin verification environment. A semantics compiler generates the semantics of the C++ sources as PVS source code. This is loaded into PVS, together with the formalization of the runtime environment and the specifications that one wants to prove.

Verification proceeds by reasoning in PVS over a nontrivial state transformer that represents the semantics of some source code. This is mostly done by applying a start state to the state transformer and proving properties of the result. Such a verification could equivalently be performed by computing the weakest precondition of the verification goal with respect to the program.

A slightly different view on the verification environment is as follows: The runtime environment defines an abstract state machine. The basic state transformers of the

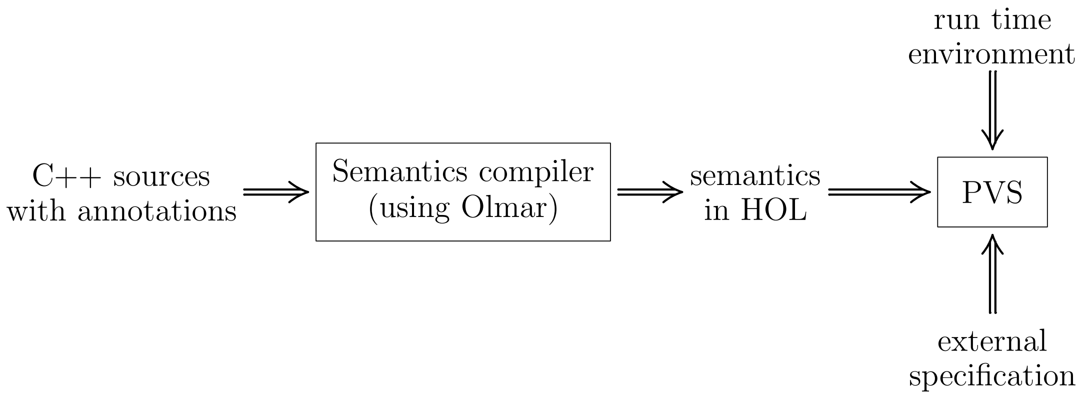


Figure 2.2: Approach for source code verification

C++ semantics describe the actions of this state machine. The program is symbolically executed on top of the state machine. Properties are derived from the state changes that one observes.

In our design the ingredients of our verification environment are relatively independent from each other. It is therefore possible

- To add additional axioms for certain properties of certain data types. For instance, to model a compiler specific assumption about the size of `unsigned int` or the precise behaviour of some type casts.
- to add new operations to the runtime environment
- to use different versions of the runtime environment for different pieces of the Nova hypervisor. The boot code of the hypervisor can be verified against physical memory and the hardware independent parts can be verified against a traditional, untyped memory model.
- to adopt the semantics for new C++ features or compiler specific C++ constructs.

2.4 Consistency and Completeness

Our runtime environment in PVS and the C++ semantics are necessarily incomplete (both in the technical and the logical sense). However, many of the omissions do not lead to global assumptions on the validity of our verification. The hardware model, for instance, does not contain virtual 8086 mode, but the validity of our verification does not hinge on the absence of instructions that enable virtual 8086 mode. Instead the `VM` flag, which controls this mode, is protected with a suitable side effect. Any attempt to enable virtual 8086 mode will result in a verification error. Hence a proof of normal termination suffices to show that virtual 8086 mode is never enabled. Similarly, the use of missing features in the C++ semantics will trigger an assertion in the semantics compiler.

A number of features are currently completely absent in our verification environment, because their formalization is considered future work. These features are (1) the Translation Lookaside Buffer (TLB)[1], (2) cache policy checking for devices, (3) segment offsets and segment size checking, (4) linking object code and instruction fetch to the abstract C++ semantics. Because of their absence we are currently unable to detect certain kinds of errors, namely

- TLB errors, e.g. inconsistencies between the TLB and the page tables, or implicit assumptions about the TLB size and structure,
- segment violations (the Nova micro-hypervisor uses a flat memory model where no segment violations can occur, however, currently we do not check that the segment descriptors are filled with the proper values),
- cache policy errors for memory-mapped devices, and delayed side effects for cachable memory-mapped devices,[2]
- discrepancies between our C++ semantics and the compiled object code, which (apart from compiler bugs) could occur for the following reasons: `volatile`-related errors in the source code[3], certain compiler optimizations (e.g. delayed write-back to memory), or self-modifying code (however, no self-modifying code is contained in our current verification target).

Ignoring the missing features, our verification will build on the following general assumptions:

- The software to be verified will be executed on a single-processor system.
- Caches for real memory are working completely transparent and can be ignored. This should be guaranteed by the hardware on single-processor systems.
- The involved software tools—the C++ compiler used to compile the Nova micro-hypervisor, our semantics compiler (including the Elsa C++ parser and type checker), and PVS—produce correct results.

[1] The TLB is a special CPU-internal cache for virtual-to-physical address translations.

[2] The source code that we currently target does not involve any devices. In general, cache policy checking for memory-mapped devices is trivial to add with our mechanism for side effects. To model cache effects on cachable devices, the model of the device should include the relevant cache effects.

[3] Any C++ compiler is permitted to perform arbitrary optimizations with respect to non-`volatile` data. Memory accesses to such data are not part of the *observable behavior* of a C++ program, which makes a correct semantics difficult. At the moment our C++ semantics treats all data as `volatile`. A verification based on the current semantics will therefore not catch missing `volatile` annotations or missing memory fences.

3 Translating C++ into PVS

For reasoning about C++ programs one needs a semantics of (at least some subset of) C++. For doing so in a theorem prover one needs a tool that translates C++ into its semantics in the input language of the theorem prover. We call this translation tool the *semantics compiler*. This chapter is about the semantics compiler.

The semantics compiler does not do very complicated things. It mostly performs some kind of syntactic translation, for instance connecting statements with the ##-operator for state-transformer composition instead of with semicolons. Most complicated parts of the semantics are inside the semantic combinators, see Section 4.7. The semantics compiler only performs the translation from statements to semantic combinators.

The most complicated part of the semantics compiler is therefore the C++ parser and type checker. The requirements here are lower than for a normal compiler, because the semantics compiler will only be run on type-correct programs and does not need to generate good diagnostics on error. Nevertheless, a C++ parser and type checker are still far too complex to rewrite them from scratch within the scope of the Robin project.

The choice of freely available C++ parsers is not particularly big. In the predecessor project VFiasco we collected some experience with Open C++ [Chi93]. The meta programming model of Open C++ is not really suited for a semantics compiler and the enforced programming style using the visitor pattern with several template instantiations is far from easy. Open C++ lacks a type checker and there is not even a symbol table. Moreover, according to its SourceForge web page [Chi], Open C++ is not actively maintained anymore since August 2004. For the Robin project we were therefore forced to chose a new C++ front-end.

An obvious choice seems to be reusing the GNU Compiler Collection (GCC) front-end. For semantic analysis however, GCC is far from optimal. The nodes of the internal abstract syntax tree are dynamically typed. Further GCC tends to change the syntax tree *in place*, possibly loosing valuable information. Because of difficulties with GCC the Goanna project [FHJ+07] abandoned GCC and chose a proprietary C++ front-end.

In Robin we decided to reuse the Elsa C++ front-end developed by Scott McPeak [McP]. Elsa contains a fairly complete C++ parser and type checker. It has been developed for semantic analysis within the Oink collection of C++ static analysis tools [CW]. Its internal abstract syntax tree is fairly detailed and relatively well suited for semantic analysis. Elsa is used in a number of different projects as well, most notably in the semantic analysis of the Mozilla source code performed by the Mozilla foundation itself.

The next section describes our adaptations to Elsa and Section 3.2 the design of the semantics compiler.

3.1 Olmar – An OCaml backend for Elsa

Elsa is based on the generalized LR parser generator Elkhound and on Astgen, a special tool for creating class structures that mimic variant types from ML. A big portion of the types used in the abstract syntax tree (abbreviated as Ast in the following) of Elsa are automatically generated by Astgen from a description of the variant types and the contained data. The types for the Ast nodes follow the standard encoding of variant types in C++: Every variant type is encoded with a class hierarchy that has one subclass for each variant. Such structures are usually traversed using the visitor programming pattern. In addition to the C++ class hierarchies, Astgen generates several visitors for the Ast. The approach using Astgen opens the possibility to extend Elsa by meta programming: Instead of adding the feature to Elsa itself, one extends Astgen and lets it generate the source code for the feature.

In comparison with ML style pattern matching, the visitor pattern is tremendously verbose. Further, the Astgen-generated visitors are very limited. They cannot pass computed values upwards and downwards during the Ast traversal. Even writing a simple depth function is impossible. For those reasons and because for symbolic processing of abstract syntax trees C++ is much weaker than ML-style languages we decided to develop an OCaml backend for Elsa, called Olmar, that outputs its internal Ast as an OCaml data structure that can be processed by independent OCaml programs.

In the first version of Olmar we modified and extended Astgen to let it generate the OCaml reflection code, which rebuilds the Elsa Ast as an OCaml object. The OCaml reflection code is linked with a modified version of Elsa, which is then capable of storing its internal Ast as an OCaml object on disk (using OCaml's capability to marshal arbitrary data structures to disk). These marshalled data structures can then be read by separate Olmar tools, which are pure OCaml programs. The first Olmar tools we developed were memcheck, a type consistency checker for OCaml Ast objects, and ast_graph, an Ast visualizer generating dot files for the graphviz package [Gra]. The semantics compiler was also developed as such an Olmar tool, see Section 3.2.

A second version of Olmar is currently underway, which fixes a number of shortcomings of our first approach. Most notably, Olmar version 1 is too tightly integrated with Elsa. It requires therefore a specifically patched Elsa version and cannot be combined with other versions of Elsa, especially not with the one the Mozilla foundation is using and developing in its *pork* repository.

In Olmar version 2 we exploit the fact that the Astgen internal Ast for Astgen source files is generated by astgen itself. Therefore one can easily add OCaml reflection support to Astgen itself, and manipulate the Ast's from Astgen in OCaml programs. In Olmar version 2 Astgen is run on Elsa's Ast description, generating an OCaml object of the Astgen Ast of Elsa's Ast description. Then a bunch of Olmar tools read that Ast description and generate the reflection code for Elsa as well as all the structurally determined utility code. Especially, all the Ast traversal code of the visualization tool ast_graph is generated this way. Then Elsa is compiled and one can feed it with C++ sources whose Ast is then saved as an OCaml object to be read from (other) Olmar tools.

There is clearly a bootstrap problem in Olmar version 2, because the Olmar tools generate the OCaml reflection code for Astgen. For that they need the OCaml object of Astgen's own Ast, which can only be created from Astgen *with OCaml reflection code compiled in.* To break this bootstrap cycle the generated OCaml reflection code for Astgen is contained in the distribution.

In Olmar version 2 the OCaml reflection code is more or less independent from Elsa or Astgen. Additionally, the Olmar tools that generate the reflection and utility code are highly customizable to deal with exceptions from the general code-generation rules. In the future it will therefore be possible to distribute Olmar for several versions of Elsa.

Olmar is freely available at `http://www.cs.ru.nl/~tews/olmar/`. Olmar was mainly developed to ease the development of the semantics compiler. However, after the first release it attracted attention as the only OCaml solution for C++ parsing. The Mozilla foundation plans to use Olmar for their analysis tools for the sources of the Mozilla web browser [Gle07].

3.2 Semantics compiler

The semantics compiler translates the C++ kernel sources into their semantics in higher-order logic, as defined in PVS. It takes a pre-processed C++ source file (created e.g. with the GCC `-E` option) and yields a PVS theory file that contains the corresponding semantics in our shallow C++ embedding (see Section 4.7).

The semantics compiler is implemented in approximately 3,200 lines of OCaml code, not including the Elsa parser and the Olmar front-end. Elsa is used to parse the C++ input file, and Olmar generates an abstract syntax tree of the C++ sources as an OCaml data structure. The translation of this syntax tree into PVS definitions is mostly straightforward. Therefore the semantics compiler is relatively simple, which increases our confidence in its correctness. This is important because the correctness of the semantics compiler is one of the top-level assumptions on our verification results.

Not surprisingly, the translation performed by the semantics compiler proceeds by recursion over the structure of the syntax tree. Only a few cases require special care. The translation of a `switch` statement, for instance, requires the semantics compiler to collect all `case` labels that occur in the statement's body (so that when the statement is "executed" in PVS, we can determine at the point of the `switch` whether the body will be entered with a `Case` or with a `Default` state). Likewise, all variables declared in a block need to be collected, so that they can be allocated in our memory model in PVS when the block is entered, and deallocated upon block exit. The semantics compiler makes explicit all type information on (overloaded) operations, as well as all C++ conversions.

C++ is a quite complex programming language; the standard alone (including a description of the C++ Standard Library) consist of almost 800 pages. Developing a compiler for all of C++ is therefore a tremendous task that was well beyond the scope of the Robin project. For our current implementation of the semantics compiler, we have focused on the subset of C++ that is actually used in the Nova sources. No attempt has been made to specify this subset more formally. The semantics compiler currently cannot handle templates, virtual functions, or multiple inheritance. Also other, more basic language features (e.g. variable declarations) are restricted to the syntactic forms that they have in the Nova sources, and support for compiler-specific language extensions is incomplete at present.

We assume that the semantics compiler is applied to valid C++ sources only, that were preprocessed with GCC or some other C++ compiler/pre-processor. The semantics compiler is written quite defensively though, in the sense that unsupported language constructs will either raise an assertion in the semantics compiler itself (thereby aborting the translation to PVS), or be translated to a syntactically invalid theory that will not be accepted by PVS. In either case, illegal or unsupported language constructs in the C++ sources will not lead to provable proof obligations in our verification environment.

Aside from the supported language subset, which should eventually be extended, there are other directions for possible future improvement of the semantics compiler. Pretty-printing of the generated PVS theories currently ignores the layout of the C++ program. To obtain more readable output, one might want to preserve whitespace, certain comments etc. Furthermore, the verification goal (e.g. a pre- and postcondition, and possibly loop invariants as well) should eventually be taken from annotations in the C++ program. Currently, the properties that one wants to prove need to be stated in PVS directly.

4 Verification environment in PVS

In this chapter we present our PVS formalization. It consists of the following parts:

- our formalization of the IA32 hardware,
- the semantic combinators used to build the semantics of C++ programs in PVS,
- the formalization of C++ data types,
- and finally our example verifications.

During the project our level of knowledge increased of course a lot. Certain parts of the PVS source code are therefore obsolete. Other parts of the formalization are currently not used because of changes in our plans and priorities. These obsolete and unused parts are still contained in the sources, either because they might be useful in a successor project, or because cleaning up the sources would incur inappropriate costs. We silently omit these obsolete and unused parts in the description in this chapter. In the printed PVS sources in Appendix B they are of course present, because the appendix is generated automatically from our CVS repository.

The complete PVS formalization is spread over about 220 theories[1] in 33 files. The PVS sources (without the proofs) are about 16,500 lines of code. They contain more than 860 handwritten theorems and almost 850 automatically generated type-check conditions (TCCs). Altogether they have been proved with more than 17,500 interactive proof commands.[2] A validation run to automatically check all the proofs takes about 25 minutes on a 2.4 GHz CPU. A detailed description of the complete material would fill several hundred pages(see Appendix B and C). In this chapter we give an overview of the material, with a description of the contents of each file. Figure 4.1 shows the (transitive reduction) of the dependency graph of all source files. (And, just for illustration, Figure 4.2 shows all theories with their dependencies.) Appendix B contains the complete PVS sources, and Appendix C contains the complete proofs, automatically generated by PVS.

4.1 General concepts

Every theorem-proving project bigger than a student exercise inevitably contains formalizations of standard notions, because they are either not available in the chosen theorem

[1]PVS sources are organized in theories. A theory has a unique name and can contain definitions, axioms, theorems and proofs.

[2]The most frequent proof command is `expand` (about 3,000 occurrences) followed by `use` (1,970) and `smash` (1,940). `grind` is on place 7 with 870 occurrences.

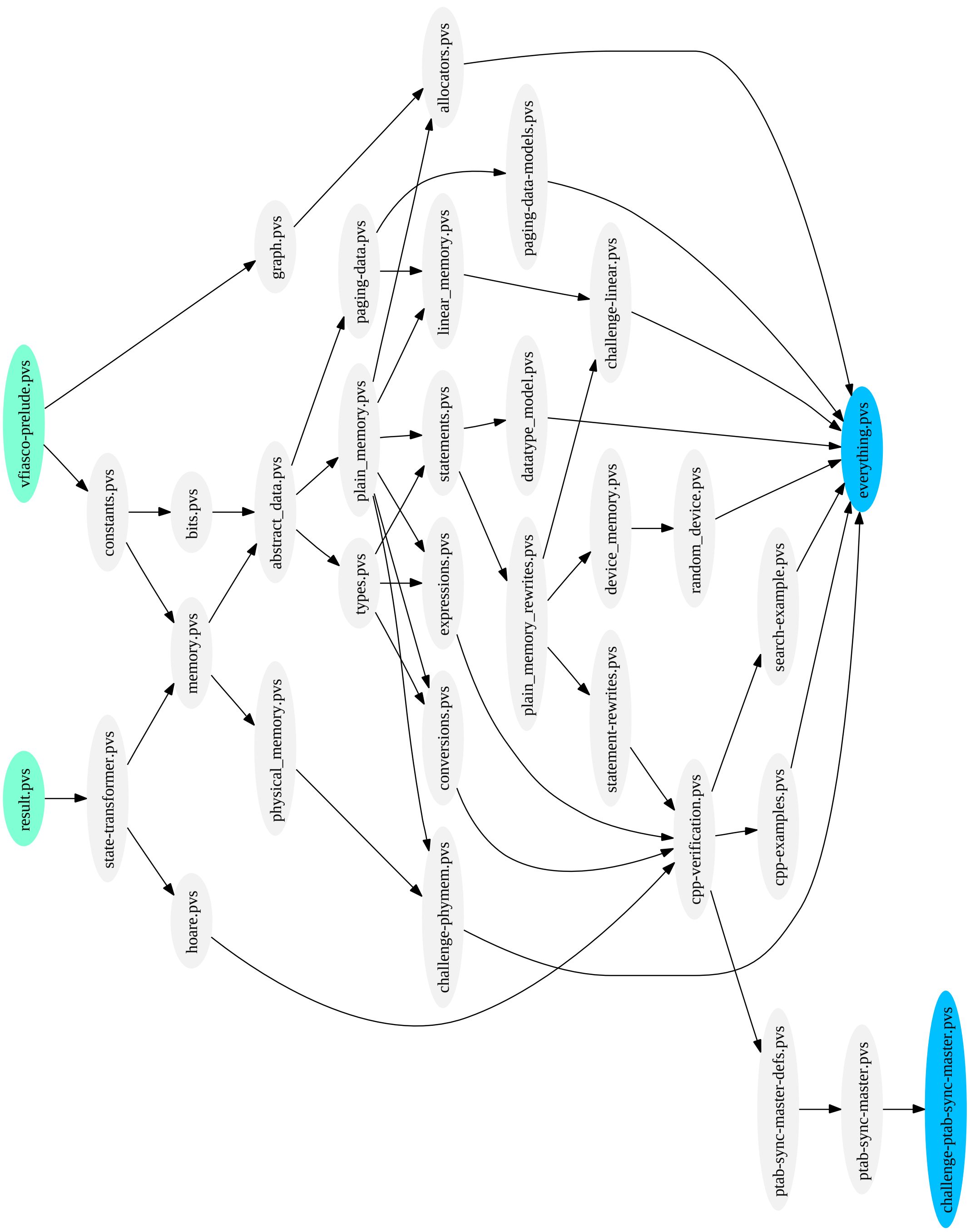


Figure 4.1: File dependency of the PVS sources

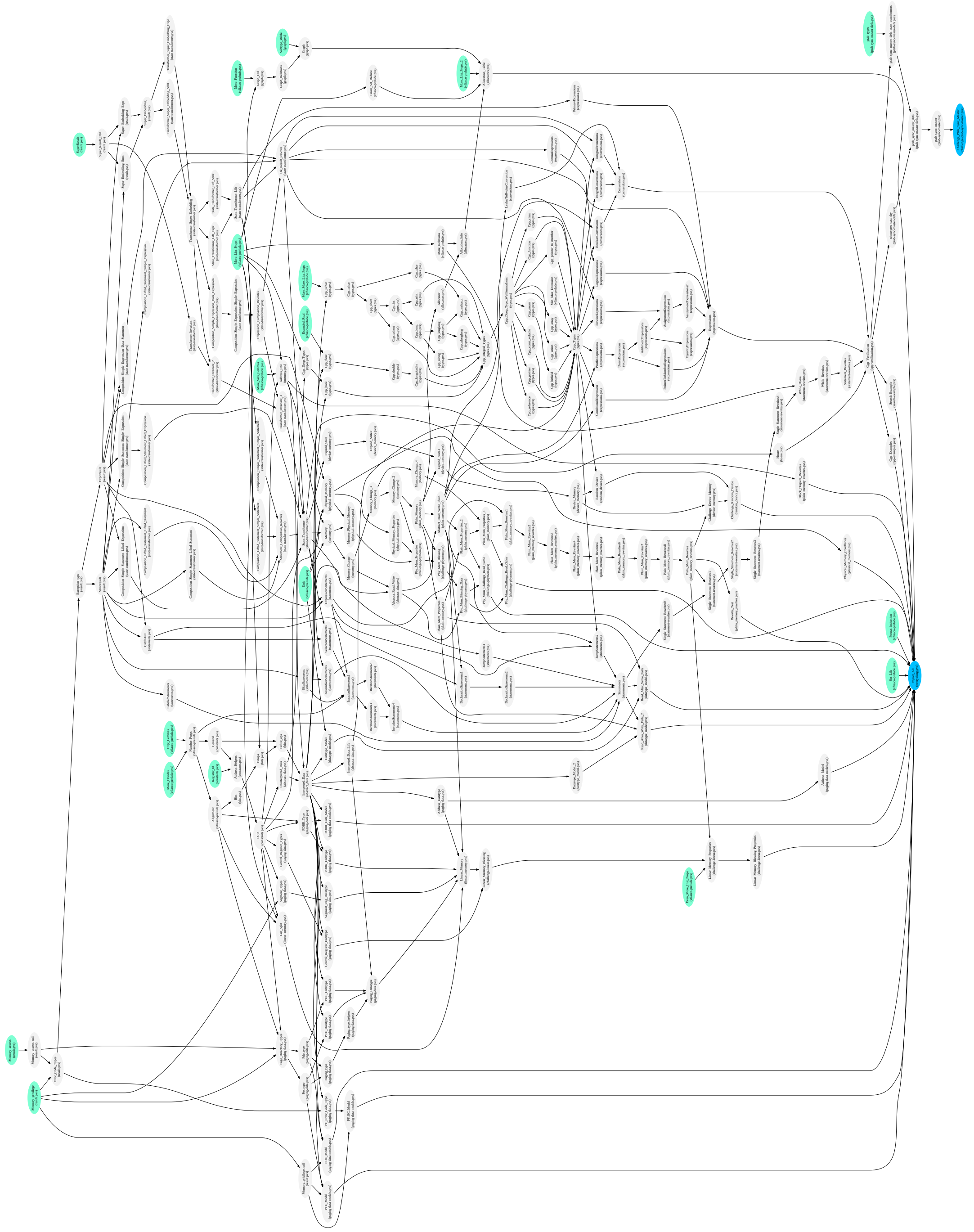


Figure 4.2: Theory dependency of the PVS sources

prover or they are available only in an unsuitable form. In this section we describe our formalizations of general notions or properties that are not included in the PVS prelude or the standard libraries. The first subsection (especially the part on lists) is much more detailed than necessary. It thereby serves as a gentle introduction to PVS for the uninitiated reader.

4.1.1 File vfiasco-prelude.pvs

The file `vfiasco-prelude.pvs` contains stuff that we felt is missing from the PVS prelude.

4.1.1.1 Unit: the one element type

The logic of PVS provides finite product and coproduct (disjoint union) types, but surprisingly the empty product and coproduct types are missing. We formalize the empty product, which is a one-element type as follows:

```
Unit : Datatype
% the unit type (aka semantic of void)
Begin
  unit : unit?
End Unit
```

This defines Unit as a type with precisely one element unit. The recognizer predicate unit?[3] is useless here, but syntactically required. The Unit type is used as result type for state transformers that perform only side effects on the state but do not yield an actual result (such as the semantics of assignment in the hardware model).

4.1.1.2 Additional list function and properties

The PVS prelude defines the data type list[T] of lists over a base type T together with null (the empty list), cons (adding an element), car and cdr (accessing head and tail of non-empty lists), length, append (list concatenation) and nth (returning the n-th element of a list). Most of these functions are defined by recursion and, when needed, exploiting predicate subtyping for the arguments. For instance, the prelude defines nth as follows:

```
nth(l : list[T], n : below(length(l))) : Recursive T =
  IF n = 0 Then car(l) Else nth(cdr(l), n−1) Endif
Measure length(l)
```

Here below(n) denotes the finite subtype of natural numbers strictly less than n. This predicate subtype formalizes the natural requirement that one can only extract elements up to the length of the list. Every use of nth is associated with a type-check condition (TCC) that requires the user to prove that the second argument is within the bounds. Many of those TCC's can be proven automatically, some are so simple that the type-checker can already discharge them. However, there are also some that require a lot

[3]In PVS the question mark is a letter and permitted in identifiers. It is usually used as last letter of predicates, though this is not mandatory.

of effort. The measure in the last line of the definition of nth is required for recursive definitions in order to prove that they are total. For each recursive call the type checker generates a termination TCC, in which one must prove that the measure strictly decreases for the recursive call. This ensures that the recursive function is well-defined for all argument values.

In addition to the list functions from the prelude we need head (extracting the first n elements of a list), tail (deleting the first n elements), list_remove (deleting one element in a list, regardless of its position) and flatten. As an illustration, here are the definitions of head and tail:

```
head(l : list[T], n : upto(length(l))) : Recursive list[T] =
  If n = 0 Then null
  Else cons(car(l), head(cdr(l), n − 1)) Endif
Measure n

tail(l : list[T], n : upto(length(l))) : Recursive list[T] =
  If n = 0 Then l
  Else tail(cdr(l), n − 1) Endif
Measure n
```

Here, upto(n) is the subtype of natural numbers less than or equal to n. The definition of head requires 3 TCCs to be proven. The first one is about taking the car in the **Else** branch, which requires l to be a nonempty list. This is indeed the case because we reach the **Else** branch only if n and therefore also the length of l is greater than zero. The second TCC requires us to prove that the second argument n − 1 of the recursive call is less than or equal to the length of the first argument cdr(l). The third TCC is the termination TCC for the recursive call.

For the definition of tail the type checker of PVS notices that the necessary TCCs are trivially subsumed by the TCCs of head. Therefore the generation of these TCCs is suppressed.

The main contents of the formalization are however not these function definitions, but the properties we proved about them. We proved about 20 lemmas about functions on lists. The formulation of these 20 lemmas requires another 13 TCCs to be proven (this TCC ratio is very typical, in the whole repository 40% of all proven statements are TCCs).

None of our proven list lemmas expresses a complicated fact. On the contrary, some of them are completely trivial:

```
cons_length : Lemma Forall(l : (cons?[T])) : length(l) > 0
```

The lemma cons_length states that a list, which is build using cons and which is therefore not the empty list, has a length greater than zero. In a theorem-proving environment it makes sense to state such trivialities, because (1) they might be needed very often and (2) they are required for discharging preconditions during automated proving.

Most of the proven list lemmas are simplification rules, such as the following.

```
nth_tail : Lemma Forall(l : list[T], n, i : nat) :
  n <= length(l) And i + n < length(l) Implies
    nth(tail(l, n), i) = nth(l, n + i)
```

Other simplification rules are about applying length, car or cdr to the result of head and tail.

Perhaps the most involved lemma about lists (which is still rather shallow) is the following.

```
list_extensionality : Lemma Forall(l1, l2 : list[T]) :
   l1 = l2 IFF
   (length(l1) = length(l2) AND
    Forall(i : below(length(l1))) : nth(l1,i) = nth(l2,i))
```

It makes it possible to derive the equality of two lists from the equality of their elements without induction (the lemma list_extensionality itself is of course proven by induction).

In every complex formalization one needs some results that only deal with peculiarities of the theorem prover. A very typical example is the following.

```
every_extend : Lemma
  Forall (l : list[S], P : PRED[T]) :
    every(Lambda (t : T) : P(t))(l) Implies every(Lambda (s : S) : P(s))(l)
```

Here S is an arbitrary subtype of T and every(P)(l), which is defined internally in PVS, expresses that the predicate P holds on all elements of the list l. The function every is in principle polymorphic in the type of the list elements. However, in PVS types cannot be truly polymorphic. Instead they can contain type parameters, declared for the enclosing theory. When such pseudo-polymorphic functions are used, the type checker must find instantiations for all type parameters.

The lemma every_extend speaks about a list l, containing only elements of S, and an arbitrary predicate P over T. Because S is a subtype of T, PVS knows that the list l can also be considered to be of type list[T]. Because of the predicate argument the type checker derives that the left-hand side every in the assumption works on elements of type T. Thus the type parameter of the left-hand side every is instantiated with T, written in angle brackets as every[T]. At the right-hand side every, in the conclusion, the predicate P is restricted to the subtype S, therefore the right-hand side every is instantiated with S.

The lemma states that if a predicate P holds on all elements of a list (expressed via every[T]), then you can restrict the predicate to any subtype that covers the elements of the list (expressed via every[S]).

The lemma every_extend is used in the proof that linear memory (memory with page-table based address translation) is a plain memory (see Section 4.5.5 below). There, a predicate on a list of arbitrary large addresses must be restricted to addresses within the 4 gigabyte virtual memory.

4.1.1.3 Additional results on numbers

Our underlying memory is organized byte-wise (see Sections 4.4 and 4.5 below). As in reality most data must be encoded into several bytes in order to be stored in memory. For the encoding into bytes we use ndiv (integer division, usually by 2^8 or multiples thereof), rem (remainder or modulo) and expt (r^n for a real r and a natural n). These three functions are defined in the PVS prelude. However, we need many more properties than the prelude provides. In the theories More_Divides, Expt_Lemmas and Number_Props we state about 40 properties of these functions. Many of them are basic facts about integer division and modulus. For example:

```
ndiv_times_2 : Lemma Forall(a : int, b, c : posnat) :
  ndiv(a, b * c) = ndiv(ndiv(a, b), c)
```

This says that the familiar law $a/(b * c) = (a/b)/c$ does also hold for integer division. A second example is

```
ndiv_rem_divisible : Lemma Forall(a, c : posnat, b : int) :
  divides(c, a) And c <= a Implies
    ndiv(rem(a)(b), c) = rem(ndiv(a, c))(ndiv(b,c))
```

Here we first come across rem, where rem(d)(i) is the nonnegative remainder of the division of integer i by a positive divisor d.

The lemma ndiv_rem_divisible is best understood in terms of a positional number system. Think of c as the divisor to separate the last digit (e.g., $c = 10^1$), and a as the divisor to separate the last two digits (e.g., $a = 10^2$) of a number b. The lemma then states two equivalent ways to obtain the second last digit of b. On the left-hand side one first takes the last two digits of b (by taking the remainder of b modulo a) and shifts the result one digit to the right. On the right-hand side one first shifts the whole b one digit to the right and then separates the last digit.

The lemmas about ndiv and rem are used in our formalization of bit vectors (see Section 4.1.2), which in turn is used in the formalization of C++ and hardware data types (such as 32 bit unsigned integers or page-table entries).

4.1.1.4 Additional lemmas about sets

Sets are formalized in PVS (as usual in higher-order logic) with their characteristic functions. A set or predicate over type T is a function that maps the elements in the set or predicate to true:

```
  PRED: TYPE = [T -> bool]
  setof: TYPE = [T -> bool]
```

The PVS prelude defines the usual set operations in the obvious way. For instance

```
member(x : T, a : setof[T]): bool = a(x)
subset?(a, b : setof[T]): bool = Forall x: member(x, a) Implies member(x, b)
union(a, b : setof[T]): setof[T] = Lambda(x : T) : member(x, a) OR member(x, b)
```

To enable the automatic simplification of proof obligations (called automatic rewriting in PVS) we need about 15 additional, mostly trivial properties of sets. The PVS prelude contains for instance the following lemma.

```
subset_reflexive: LEMMA subset?(a, a)
```

However, in automatic rewriting one needs the completely equivalent form

```
subset_equal : Lemma a = b Implies subset?(a, b)
```

The difference between the two forms is that subset_reflexive is only applicable if the two subsets are syntactically identical (because otherwise no substitution can be found). In contrast subset_equal is applicable even if a and b are syntactically different and the equality of a and b can only be established with the help of the decision procedures and the database of known facts.

Another example is a property of disjointness of sets.

```
disjoint_mono : Lemma Forall(a1, a2, b1, b2 : set[T]) :
  subset?(a1, a2) And subset?(b1, b2) And
  disjoint?(a2, b2) Implies
    disjoint?(a1, b1)
```

It says that two sets a1 and b1 are disjoint if there are two disjoint supersets of a1 and b1, respectively.

4.1.1.5 Converting finite sets to lists

In the formalization of memory allocators (see Section 4.6 below) it is necessary to compute the list of allocated memory blocks from a finite set of allocators (by querying the allocated memory blocks from each allocator and concatenating these lists). Somewhere in this computation a finite set has to be converted into a list, where the order of the resulting list is totally irrelevant.

Finite sets are formalized in the PVS prelude as those sets for which an injection into a set of the form below(n) exists. The smallest such n is then defined to be the cardinality of the finite set, denoted by card(S)[4]. The conversion from finite sets to lists is done by the following function.

```
list_of_finite_set(S : finite_set[T]) : Recursive list[T] =
  IF empty?(S) Then null
  Else cons(choose(S), list_of_finite_set(rest(S)))
  Endif
Measure card(S)
```

The function choose uses the Axiom of Choice to select an arbitrary (but constant) element from a set, and rest(S), which is defined in the PVS prelude, removes the chosen element from S:

```
rest(a : setof[T]): setof[T] = If empty?(a) Then a Else remove(choose(a), a) Endif
```

As results for list_of_finite_set we proved that the length of the resulting list equals the cardinality of the original set, and that each element in the list is unique.

4.1.1.6 Alignment

Alignment denotes the requirement of many computer architectures that the addresses of certain operands must be divisible by a certain power of 2. Sparc, for instance requires that 4-byte integers are aligned on a 4-byte boundary, i.e. that the address is divisible by 4. Our target architecture, IA32, has no hard alignment requirements for normal

[4] PVS defines card(S) only for finite sets S, there is no predefined cardinality for infinite sets.

instructions (though alignment checks can be performed in user mode by setting the AM and AC bits). Page tables are however required to be 2^{12}-aligned.

Here we define alignment as a property of natural numbers:

```
aligned?(n) : PRED[nat] = { a : nat | divides(expt(2,n), a) }
aligned(n) : Type = (aligned?(n))
```

The first line defines aligned?(n) as predicate recognizing n-aligned numbers. The brace notation { a : nat | ... } is syntactic sugar for **Lambda**(a : nat) : ... and comes handy for defining sets. The second line defines the type of n-aligned numbers. In PVS parenthesis around a predicate (P) turn the predicate into a type, containing just the elements for which the predicate holds.

For alignment we proved a number of properties, for instance that bigger alignment implies smaller alignment, and that the alignment of the sum of two numbers equals the minimum of the alignment of the operands. The proofs of these properties rely of course on certain properties of ndiv and rem, see Section 4.1.1.3 (on page 22). This formalization of alignment is used in the hardware data types for page tables.

4.1.1.7 Zorn's Lemma

For the termination proof of while loops using variants on a well-founded order we need one direction of Zorn's Lemma.

```
zorn_lr : Lemma Forall(R : PRED[[T,T]]) :
  well_founded?(R) Implies
      Not Exists (f : [nat -> T]) : Forall (n : nat) : R(f(n+1), f(n))
```

A relation in PVS is simply a predicate on the Cartesian product $\mathsf{T} \times \mathsf{T}$, in PVS denoted by [T, T]. The definition of well_founded? in the prelude is the standard one (every non-empty subset of the relation contains a minimal element). The lemma zorn_lr expresses that in a well-founded order there is no infinite descending chain. (The converse, namely that an order is well-founded if there are no infinite descending chains, holds as well. However, its proof requires unbounded ordinals, because it proceeds by ordinal induction up to the size of R. PVS currently contains only ordinals up to ε_0.)

4.1.1.8 Turning an injective function into a bijection

With the predicative sub-typing of PVS every function can be given a type such that it is surjective. If the original function was injective one of course obtains a bijection. The restriction of the type is done with the following function.

```
restrict_to_image(f : [Domain -> Range]) : [Domain -> (image(f, fullset[Domain]))] = f
```

Here Domain and Range are type parameters, fullset[Domain] gives the predicate containing all inhabitants of Domain, and image yields the image of a function as a predicate.

The whole purpose of this exercise is to prove the following.

```
bijective_restrict_to_image : Lemma
  injective?(f) Implies bijective?(restrict_to_image(f))
```

This lemma is used in order to prove that the length of a list in which all elements are unique is bound by the cardinality of the set from which the elements are drawn. This latter result is needed in turn to prove that cycle free paths in finite graphs are bound in size, which again is needed to discharge a TCC in the definition of the unique path from an arbitrary node to a root node in a finite tree.

4.1.2 File bits.pvs

PVS is distributed with a library of bit vectors. The type of those bit vectors is a finite sequence of bits of fixed size:

```
bvec : TYPE = [below(N) -> bit]
```

Here, the natural number N is a theory parameter, encoding the length of the bit vector and bit is a type alias for bool. With this definition the length of the bit vector is encoded in its type. When concatenating two bit vectors of length 5 the result has type bvec[5 + 5]. With several cuts and concats those length expression on the type level can get quite long. In PVS, arithmetic expressions on the type level are not simplified, instead they require special treatment. In our experience this matter complicates proofs about standard PVS bit vectors a lot.

Another complication of the standard PVS bit vectors is that for the formalization of hardware, as we intended in this project, a lot of conversions from bit vectors to natural numbers and back are needed.

In Robin we therefore decided to develop an alternative formalization of bit vectors, where no length information is present on the type level. We just use natural numbers as bit vectors of arbitrary length. The representation of bits is not explicit, but we assume a standard representation of naturals to the base 2. With this assumption operations for cutting, shifting and extraction of single bits are defined. For instance

```
% cut k bits out of n, starting at i
cut_bits(n, i, k : nat) : nat = rem(expt(2, k))(ndiv(n, expt(2, i)))

% return true if the i-th bit in n is 1
cut_bit(n, i) : bool = rem(2)(ndiv(n, expt(2, i))) = 1

% shift n i bits to the left
shift_bits_left(n, i) : nat = n * expt(2, i)
```

Intuitively our bit vectors are all infinitely long (filled with leading zeros). Therefore there is no operation for concatenation, instead we define replacement:

```
% replaces k bits starting from i in word n with k bits starting at i in word m
overwrite_bits(n, m, i, k : nat) : nat =
  n - rem(expt(2, i + k))(n)
      + shift_bits_left(cut_bits(m, i, k), i)
      + rem(expt(2, i))(n)

% replaces bit i in n with the one of m
overwrite_bit(n, m, i : nat) : nat = overwrite_bits(n, m, i, 1)
```

```
% overwrites bit i in n with boolean b
overwrite_bool_bit(n : nat, b : bool, i : nat) : nat =
  overwrite_bit(n, shift_bits_left(bool_to_nat(b), i), i)
```

Our formalization of bit vectors is certainly less elegant and less precise from a modelling perspective. However, we believe, it is more efficient in proofs, at least for us.

The additionally required effort for developing this formalization is less than it sounds because of two reasons. Firstly, reusing an existing library of proofs requires much more effort than reusing a software library, especially proofs must be planned with respect to the properties the library provides. Secondly, in order to get good automation support from the library in the form of automatic rewrites one has to understand and then use the usage pattern that the library developer had in mind.

In our evaluation the alternative bit-vector formalization payed off. The consistency proofs for the data-type formalization of page-table types is almost fully automatic, when it comes to bit manipulations. This hinges admittedly on a special usage pattern, including a certain ordering or bit operations. The following lemma plays a key role in the obtained automation:

```
cut_bit_overwrite_bool_bit : Lemma
  Forall(n : nat, b : bool, i, j : nat) :
    cut_bit(overwrite_bool_bit(n, b, i), j) =
      IF i = j Then b
      Else cut_bit(n, j)
      Endif
```

The overwrite_bool_bit stems from encoding various page-table bits into a list of bytes. The cut_bit comes from the corresponding decoding operation. This rewrite rule is the work horse for proving that encoding into bytes and then decoding gives the original value back.

4.1.3 File graph.pvs

In this file we formalize those pieces of graph theory that are needed to define finite trees, to express the function that gives the unique path of a node to the root above it and finally to prove a certain proof principle over such trees. Strictly speaking, we formalize finite forests (i.e., finite sets of finite trees) however, we sloppily stick to the term *finite tree.* These trees are used to formalize dependencies between different memory allocators. The proof principle for trees, we develop here, is used to prove many of the interesting properties of memory allocators, see Section4.6.

4.1.3.1 Utility notions and results

Before we can define graphs and trees we need some utility notions and properties. The first result has already been pointed to.

```
list_pred_card : Lemma
  Forall(l : list[T], P : PRED[T]) :
    is_finite(P) And every(P)(l) And length(l) > card(P) Implies
      Exists(n1, n2 : below(length(l))) :
```

```
    Not n1 = n2 And
    nth(l, n1) = nth(l, n2)
```

It says that a list in which all elements are drawn from a finite predicate P and whose length is greater than the cardinality of P must contain two identical elements. This lemma could probably be proven by induction. However, we preferred indirect reasoning: If all elements in the list were unique then nth on that list would be injective. Restricting nth to a bijection (using restrict_to_image from Section 4.1.1.8, on page 25) and inverting it gives a bijection from a subset of P (those elements which occur in l) into below(length(l)), which would prove that the cardinality of P must be at least length(l).

We continue with the definition of the base type for paths in graphs together with operations to select the first and the last element of a path.

```
path_list?(l : list[T]) : bool = length(l) >= 1
path_start(l : (path_list?)) : T = car(l)
path_end(l : (path_list?)) : T = nth(l, length(l) −1)
```

The type of nodes of the graph will later instantiate the type parameter T. Path lists are lists of nodes of at least length one.

A number of results will later be proved by induction on paths lists. Using ordinary list induction works, but is cumbersome, because our base case, a path list of length one, must always be split of manually. We therefore prove the following induction theorem for paths.

```
path_list_induction: Lemma
  Forall(p: [(path_list?) −> boolean]):
    (Forall(t : T) : p(cons(t, null))) And
    (Forall(t : T, tail : (path_list?)): p(tail) Implies p(cons(t, tail)))
      Implies
        Forall (l : (path_list?)): p(l)
```

We finally define concatenation of path lists.

```
concatable?(left, right : list[T]) : bool =
  path_list?(left) And path_list?(right) And
  path_end(left) = path_start(right)

concat_path(lr : (concatable?)) : (path_list?) =
  append(Proj_1(lr), cdr(Proj_2(lr)))
```

The argument for concat_path must be a pair of path lists that are concatable. Two path are concatable if the last node of the first path is identical to the first node of the second path. We use the projections Proj_1 and Proj_2 to extract the first and the second path from the pair lr.

For our definition of cycle-freeness we need the transitive closure of an arbitrary relation, for which we exploit the usual higher-order definition.

```
transitive_closure(R : PRED[[T, T]]) : PRED[[T, T]] = Lambda(x, y : T) :
  Forall(Q : PRED[[T, T]]) : subset?(R, Q) And transitive?(Q) Implies
      Q(x,y)
```

The use of the universal quantifier blurs it a bit, but transitive_closure(R) is really the intersection of all transitive relations bigger than R. For proofs we need a characterization result that says that two elements are in the transitive closure of some relation R if and only if there is a finite path in R between these two elements.

```
trans_closure_char : Lemma
  Forall(R : PRED[[T, T]], x, y : T) :
    transitive_closure(R)(x,y) IFF
    Exists(l : list[T]) :
      length(l) >= 2 And
      path_start(l) = x And path_end(l) = y and
      Forall(i : below(length(l) -1)) : R(nth(l, i), nth(l, i+1))
```

4.1.3.2 Graphs and Trees

We can now define a (directed) graph as a record consisting of the set of nodes and the edge relation.

```
graph_type : Type = [#
  nodes : PRED[T],
  edges : PRED[[(nodes), (nodes)]]
#]
```

Note that the type of the second record field depends on the first field. This kind of dependent types is fully supported in PVS.

A path in a directed graph is a path list such that adjacent nodes are connected with an edge.

```
path?(g : graph_type)(l : (path_list?[T])) : bool =
  (Forall(i : below(length(l))) : g`nodes(nth(l, i))) And
  (Forall(i : below(length(l) -1)) : g`edges(nth(l,i), nth(l, i+1)))
```

Note that the first line, requiring that all elements of l are nodes of g is necessary for discharging a TCC in line two. There, the edges relation can only be tested on actual nodes. We need a few lemmas saying that paths are closed under some list operations such as concatenation or taking the head and tail, but they are excluded here.

The set of root nodes of a graph g is defined straightforwardly.

```
roots(g : graph_type) : PRED[(g`nodes)] =
      {y : (g`nodes) | Forall(x : (g`nodes)) : Not g`edges(x,y) }
```

We define freeness of cycles with the transitive closure of the edges relation

```
cycle_free?(g : graph_type) : bool =
  Forall(x : T) : g`nodes(x) Implies
    Not transitive_closure(g`edges)(x, x)
```

A tree is then a cycle-free graph in which all nodes have at most one predecessor.

```
tree?(g : graph_type) : bool =
  cycle_free?(g) And
  Forall(x, y, z : (g`nodes)) :
    g`edges(x,z) And g`edges(y,z) Implies x = y
```

A tree is finite if its set of nodes is finite.

```
finite?(g : graph_type) : bool = is_finite(g`nodes)
finite_tree?(g : graph_type) : bool = finite?(g) and tree?(g)
```

Note, that we are not requiring a unique root. Structures fulfilling finite_tree? can contain several (disjoint) trees with separate root nodes.

We restrict ourselves to finite trees, because the reasoning we want to carry out in the context of memory allocators (see Section 4.6 on page 47) crucially depends on the property that from each node in a tree there is a unique path upwards to a root node.

It turns out to be a nontrivial exercise to define a function that yields the unique root path for each node. First, we define a predicate to recognize the root path for each node.

```
root_path(g : graph_type, n : (g`nodes)) : PRED[(path?(g))] =
  Lambda(p : (path?(g))) :
      roots(g)(path_start(p)) And path_end(p) = n
```

Because in a finite tree, there is precisely one path from each node to a root, we can simply define:

```
path_to_root(g : (finite_tree?), n : (g`nodes)) : (root_path(g, n)) =
      the(root_path(g, n))
```

Here the function the returns the only element of a singleton set. However, it takes 9 proofs and about 160 PVS proof commands to discharge the TCC that root_path(g, n) contains indeed precisely one element for every node n. Our reasoning consists of the following steps (compare the PVS sources in Appendix B on page 67)

- The length of every path in a finite, cycle free graph is bound by the number of nodes (Lemma path_length_bound).
- For any non-root node there exists a path of at least length 2 going upward (Lemma non_root_path).
- Every path not starting in a root can be extended at the front (Lemma non_root_path_extend).
- For every node there exists a root path (Lemma tree_path_to_root).
- Every two root paths for a given node are identical (Lemma tree_unique_path_to_root).

4.1.3.3 A path based proof principle for finite trees

In the context of memory allocators we prove important properties for two nodes (which are memory allocators then) by considering both their path to a root node. We therefore device the following proof principle.

Theorem 1 *Consider a symmetric predicate P on two nodes of a finite tree tr. In order to prove P for all pairs of nodes of tr it is sufficient to prove P for two nodes n_1 and n_2 in each of the following cases:*

- $n_1 = n_2$
- *There exists a path p in tr starting at n_1 and ending in n_2.*

- *There exists a node n_3 and two path p_1, p_2 such that p_1 and p_2 both start in n_3 and p_1 ends in n_1 and p_2 ends in n_2. Further, p_1 and p_2 have at least length 2 and their second nodes are not equal.*

- *There exist two disjoint root path ending in n_1 and n_2, respectively.*

Our PVS version of this theorem is slightly optimized for interactive proof: It makes all the cases disjoint (such that, for instance, for case 2 one can assume $n_1 \neq n_2$).

```
symmetric_node_pair_distinction : Lemma
  Forall(P : PRED[[(g`nodes), (g`nodes)]]) :
    finite_tree?(g) And
    symmetric?(P) And
    (Forall(n : (g`nodes)) : P(n, n)) And
    (Forall(n1, n2 : (g`nodes), p : (path?(g))) :
      NOT n1 = n2 And
      path_start(p) = n2 And path_end(p) = n1 And length(p) >= 2 Implies
        P(n1, n2))
    And
    (Forall(n1, n2 : (g`nodes), p1, p2 : (path?(g))):
      NOT n1 = n2 And root_path(g, n1)(p1) And root_path(g, n2)(p2) And
      NOT path_start(p1) = path_start(p2) Implies
        P(n1, n2))
    And
    (Forall(n1, n2, n3 : (g`nodes), p1, p2 : (path?(g))):
      NOT n1 = n2 And NOT n1 = n3 And NOT n2 = n3 And
      g`edges(n3, path_start(p1)) And g`edges(n3, path_start(p2)) And
      path_end(p1) = n1 And path_end(p2) = n2 And
      NOT path_start(p1) = path_start(p2) Implies
        P(n1, n2))
    Implies
      Forall(n1, n2 : (g`nodes)) : P(n1, n2)
```

4.1.4 File hoare.pvs

This file defines the notion of validity of Hoare triples for our verification environment, both with respect to partial and total correctness. The definition depends on state transformers, which are only introduced in Section 4.3.2 below. A Hoare triple consists of a precondition P, a program (i.e., a state transformer) c, and a postcondition Q. Informally the triple $\{P\}\, c\, \{Q\}$ is *valid* (with respect to partial correctness) iff all states reachable by executing the program c in any state that satisfies P satisfy Q. Total correctness additionally requires that c, whenever executed in a state that satisfies P, terminates normally (i.e., without an error, and without an infinite loop or infinite recursion).

4.2 Hardware details

This section discusses the formalization of some hardware details of the IA32 architecture, such as registers and address sizes. These hardware details form the basis of the

IA32 hardware model.

4.2.1 File constants.pvs

This file defines bytes, registers and addresses.

4.2.1.1 Bytes

In principle our formalization works with bytes of eight bits, where each byte can store one of 256 different values. The C++ standard imposes that bytes must contain at least 8 bits (by including the part of the C Standard which requires that CHAR_BITS is at least 8). The C++ standard further requires that all bits in a byte are visible at the C++ level. For every bit combination one gets a different char object.

For certain aspects of the C++ data type formalization (see Section 4.4.2 below) it would however be beneficial to be able to associate additional data with each byte, that can not be accessed or changed from C++. As laid out in the introduction, one of our main goals is that every concrete IA32 system should give rise to a model of our hardware formalization in a direct way. We therefore exploit under-specification for the formalization of bytes:

```
bits_per_byte : posnat
min_bits_per_byte : nat = 8
bits_per_byte_minimum : Axiom bits_per_byte >= min_bits_per_byte
max_byte : nat = expt(2, bits_per_byte)
Byte : Nonempty_Type = below(max_byte) Containing 0
```

We declare a constant bits_per_byte, which we require to be at least 8, but whose precise value remains unknown. Bytes are then required to store at least $2^{\text{bits\_per\_byte}}$ different values. Obviously, any IA32 system meets these conditions. In proofs one can only rely on the axiom bits_per_byte_minimum, therefore no fact can be derived that requires bytes to contain precisely 8 bits.

Currently, our formalization does not exploit the possibility of invisible bits in bytes. However, future extensions of the data type formalization might exploit this feature in order to catch certain obscure error patterns.

4.2.1.2 Registers and addresses

Our hardware model provides access to memory and registers. We achieve big simplifications by formalizing memory as a special (rather big) register. This way many functions can treat memory and registers uniformly without making a case distinction.

Each IA32 register gets an identifier in an enumeration data type(shortened here, the full sources are in Section B on page 120):

```
Register_Id : Datatype
Begin
  Mem_ : Mem? % ordinary memory
  EAX_ : EAX?
  EBX_ : EBX?
```

```
    ⋮
  PDBR_ : PDBR? % (CR3)
End Register_Id
```

A generalized memory/register address is a tuple of a register identifier and an offset.

```
Address : Type = [# type_of : Register_Id, offset : int #]
```

For real registers the possible values for the offset will of course be severely limited. The register identifiers above end in an underscore to leave the register name free for an address constant, for instance

```
PDBR : Address = (# type_of:= PDBR_, offset := 0 #)
```

4.3 State transformers

State transformers are a particular type of functions that are used as semantic domain for C++ expressions and statements. In addition, various other kinds of internal actions of our hardware model are formalized as state transformers. State transformers come in two flavors, statement state transformers and expression state transformers. Statement state transformers are used as semantics for C++ statements, expressions state transformers are used for the remainder (C++ expressions and internal actions). Statement state transformers cannot yield a value, they can only perform side effects in memory. Further, both flavors of state transformers differ in the kind of abnormal termination conditions they permit.

In order to formulate state-transformer invariants in a uniform way we invented a kind of super type for both variants of state transformers: super state transformers. Statement and expression state transformers can be converted into super state transformers by dropping any possible yielded values and details about abnormalities.

For certain manipulations of the control flow (such as in the semantics of the loops or the `case` statement) yet another variation of state transformers is needed: *complex state transformers*. To minimize the confusion complex state transformers are only introduced in Subsection 4.3.2.2 on the composition of state transformers on page 35 below.

4.3.1 File result.pvs

The formalization starts with expression state transformers.

```
ExprResult[State, Data : Type] : Datatype
Begin
  OK(state: State, data: Data) : OK?
  Hang : Hang?
  Fatal : Fatal?
End ExprResult
```

The type parameter State gets later instantiated with a state space of one of the memory models. It captures the relevant parts of the state of an IA32 system. Data describes the data the state transformer can yield.

An expression state transformer can terminate normally with OK and yield a result of type Data. Alternatively, it can diverge, which is modeled with a result of Hang or it can produce a fatal error condition. Divergence can occur because of non-terminating while loops or page faults that keep occurring at the same position. Fatal is reserved for conditions that we consider an unrecoverable programming error.

The PVS sources currently contain already one field for modelling arbitrary hardware interrupts and exceptions (such as page faults, real interrupts). However, this is not used yet, so we don't describe it here.

An expression state transformer is a function of type

$$\mathsf{State} \quad \longrightarrow \quad \mathsf{ExprResult[State,\ Data]}$$

Next, we formalize statement state transformers.

```
StmtResult[State : Type] : Datatype
Begin
  OK(state : State) : OK?
  Break(state : State) : Break?
  Continue(state : State) : Continue?
  Return(state : State) : Return?
  Switch(state : State, case : int) : Switch?
  Default(state : State) : Default?
  Hang : Hang?
  Fatal : Fatal?
End StmtResult
```

Statement state transformers do not yield a result. They do have a greater variety of abnormalities. Break, Continue and Return are used to model jumps in the control flow for the respective C++ statements. Switch selects a case label inside a switch statement and Default selects the default branch in a switch. The state that these abnormalities carry is the last state before the abnormality occurred and in which the execution is to be continued.

A statement state transformer is a function of type

$$\mathsf{State} \quad \longrightarrow \quad \mathsf{StmtResult[State,\ Data]}$$

```
SuperResult[State : Type] : DATATYPE
Begin
  OK(state : State) : OK?
  Abnormal(state : State) : abnormal?
  Bottom : bottom?
End SuperResult
```

SuperResult is the common denominator of ExprResult and StmtResult. Bottom combines Hang and Fatal, because their distinction is not of interest at the level of super state transformers.

The conversion into super state transformers is done in the obvious way, for instance:

```
stmt_2_super_res(stmt : StmtResult[State]) : SuperResult[State] =
  IF OK?(stmt) Then OK(state(stmt))
```

```
  Elsif has_next_state(stmt) Then Abnormal(state(stmt))
  Else Bottom
  Endif

stmt_2_super(stmt : [State -> StmtResult[State]])(s : State) : SuperResult[State] =
  stmt_2_super_res(stmt(s))
```

There are similar functions expr_2_super_res and expr_2_super for expression state transformers.

In proofs and preconditions it is often necessary to distinguish state transformers whose result carries a successor state.

```
has_next_state(res : SuperResult[State]) : bool =
  Cases res OF
    OK(state) : true,
    Abnormal(state) : true,
    Bottom : false
  EndCases
```

This function is overloaded in the obvious way for ExprResult and StmtResult.

4.3.2 File state-transformer.pvs

This file contains the sequential composition of state transformers and state-transformer invariants. Further it contains a huge number more or less trivial lemmas that are needed to get automatic simplification of C++ statements working.

4.3.2.1 State transformer trivialities

One of the many trivial results needed for automatic rewriting is the following.

```
ok_stmt_2_super : Lemma OK?(stmt_2_super(stmt)(s)) = OK?(stmt(s))
```

Such trivialities are needed because otherwise one would have to unfold the definition of stmt_2_super and do a case split according to the contained **Cases** expression. As one can not control the order of rewriting such a case split has usually a big performance cost. Moreover, often rewrite rules that require case splits are disabled, because automatic case splits can easily produces hundreds of subgoals.

Another triviality are empty state transformers that are needed in certain circumstances.

```
ok_result(data : Data)(s : State) : ExprResult[State, Data] = OK(s, data)
fatal_result(s : State) : ExprResult[State, Data] = Fatal
```

Of course they come each with their own bunch of trivial results, such as that OK? does always hold on the result of ok_result.

4.3.2.2 Composition

The basic case for composition of state transformers is as follows.

```
lift(stmt : [State -> StmtResult[State]])(sres : StmtResult[State]) : StmtResult[State] =
  Cases sres OF
    OK(state) : stmt(state)
    Else sres
  EndCases

##(stmt_1, stmt_2 : [State -> StmtResult[State]])(s : State) : StmtResult[State] =
  lift(stmt_2)(stmt_1(s))
```

In a composition (stmt1 ## stmt2)(s) the start state s is first applied to stmt1 and if this terminates with OK, the possibly modified successor state is passed to stmt2. If stmt1 terminates with something else stmt2 is dropped. This way, a break abnormality, raised by a **break** statement will skip all subsequent statements until the end of the loop. At the end of the loop a special *complex* state transformer is inserted, for instance

```
catch_break(res : StmtResult[State]) : StmtResult[State] =
   CASES res OF
     Break(state): OK(state)
     ELSE res
   ENDCASES
```

(Which is actually defined in `statements.pvs`, see Section 4.7.2 below.)

Composition between normal and complex state transformers is simply function composition. For two expression state transformers we define a variant of the composition that permits binding the result value of the first expression state transformer in a convenient way.

```
##(expr : [State -> ExprResult[State, Data1]],
     fexpr : [Data1 -> [State -> ExprResult[State, Data2]]])(s : State) : ExprResult[State, Data2] =
  Cases expr(s) OF
    OK(state, data) : fexpr(data)(state),
    Fatal : Fatal,
    Hang : Hang
  EndCases
```

This way we can write

```
  expr_1 ## Lambda(r : ...) : expr_2 ...
```

to bind the result of expr_1 to r in expr_2.

For the other combinations of expression and statement state transformers with and without argument binding we overload the ##-operator in a very similar way. Each overloaded ##-operator has its own theory, to make it easier to distinguish the different ## instances, if necessary. Associativity of composition is proved in a number of separate lemmas.

4.3.2.3 State transformer invariants

The notion of invariance plays an important role to describe certain well-formed states and well-behaved operations on them. Given a set transformers of super state transformers, a predicate states is an invariant (with respect to the transformers in transformers) if it is closed under all state transformers in transformers.

```
result_pred(states : PRED[State]) : PRED[SuperResult[State]] =
  Lambda(res : SuperResult[State]) :
    has_next_state(res) Implies states(state(res))

transformer_invariant?(states : PRED[State], transformers : PRED[[State -> SuperResult[State]]]) : bool =
  Forall(s : State, q : [State -> SuperResult[State]]) :
    states(s) AND transformers(q) IMPLIES
      result_pred(states)(q(s))
```

Note, that a state transformer invariant does not ensure normal termination. Normal termination is captured in the following definition.

```
transformers_ok?(states : PRED[State], transformers : PRED[[State -> SuperResult[State]]]) : bool =
  Forall(s : State, q : [State -> SuperResult[State]]) :
    states(s) AND transformers(q) Implies
      OK?(q(s))
```

Invariance and normal termination are very well-behaved notions: They are stable under union and enjoy certain monotonicity predicates. All this is expressed in about 20 lemmas, see Appendix B (on page 351). For example:

```
transformer_invariant_mono_transformers : Lemma
  Forall(transformers_1, transformers_2 : PRED[[State -> SuperResult[State]]]) :
    subset?(transformers_1, transformers_2) AND
    transformer_invariant?(states, transformers_2) Implies
      transformer_invariant?(states, transformers_1)
```

4.4 Abstract memory interface

For maintainability, different features of the real hardware are modelled in separate memory layers. Each such memory layer implements the same abstract interface. This abstract interface consists of the following:

- a set of memory states
- an operation for reading one byte
- an operation for writing one byte
- an operation for performing read side effects for a memory region
- an operation for performing write side effects for a memory region

The operation for side effects are for the formalization of memory-mapped devices (where access to registers of those devices can have arbitrary effects) and for the formalization of alignment and reserved bit checks. In the latter case the side effect functions will not really perform a side effect, they will check the requirements and, if not fulfilled, yield a Fatal result.

Block-wise memory access and reading/writing C++ data from/to memory is implemented on top of the abstract memory interface. Further a large number of properties for block-wise memory access is proven with respect to the abstract memory interface.

4.4.1 File memory.pvs

4.4.1.1 Address blocks

Blocks of addresses are formalized as a start address and a size.

```
% the set of addresses addr <= a < addr + size
address_block(addr : Address, size : nat) : PRED[Address] =
  Lambda(a : Address) :
      type_of(addr) = type_of(a) And
      offset(addr) <= offset(a) And
      offset(a) < offset(addr) + size
```

This definition permits address blocks of size zero, which is rather inconvenient at various other places. The only reason this has not been fixed yet is that the change will break a large number of proofs.

Address blocks are disjoint, if they do not overlap.

```
blocks_disjoint?(addr1 : Address, size1 : nat, addr2 : Address, size2 : nat) : bool =
  Not type_of(addr1) = type_of(addr2) Or
    addr1 + size1 <= addr2 OR
      addr2 + size2 <= addr1
```

There are one additional definitions for address blocks that we explain in the section on memory allocators (see Section 4.6 on page 47 below).

4.4.1.2 Abstract memory structure

The abstract memory interface is formalized as a structure of four operations, where the set of states State is a type parameter.

```
Memory_struct : Type = [#
    memory_read : [Address -> [State -> ExprResult[State, Byte]]],
    memory_write : [Address, Byte -> [State -> ExprResult[State, Unit]]],
    memory_read_side_effect : [Address, list[Byte], bool -> [State -> ExprResult[State, list[Byte]]]],
    memory_write_side_effect : [Address, list[Byte], bool -> [State -> ExprResult[State, list[Byte]]]]
#]
```

All four operations are curried functions that first take additional arguments and yield then an expression state transformer. In general, the side effects are permitted to change the data read or to be written. They therefore receive the data as argument and return it as result in the form of list[Byte]. The additional boolean argument for the side effects is the cross-page flag. It is usually false, but set to true if a contiguous memory access is split into two or more blocks during address translation. This way a side-effect formalization can see if the memory block was specified in the program or only formed during address translation.

On top of the abstract interface we define access to memory blocks recursively using the single byte interface and by combining the side effect.

```
% side effect free list write
memory_write_list_nse(pm : Memory_struct)(addr : Address, bl : list[Byte]) :
                                            Recursive [State -> ExprResult[State, Unit]] =
  Cases bl of
```

```
    null : ok_result(unit),
    cons(b, rl) : (memory_write(pm)(addr, b) ## memory_write_list_nse(pm)(addr + 1, cdr(bl)))
  EndCases
Measure length(bl)

% apply write side effect before register is modified
memory_write_list(pm : Memory_struct)(addr : Address, bl : list[Byte]) : [State -> ExprResult[State, Unit]] =
  memory_write_side_effect(pm)(addr, bl, false) ##
    Lambda (bl1 : list[Byte]) : memory_write_list_nse(pm)(addr, bl1)
```

For writing, the side effect is applied before the actual memory access. This way the side effect can check the data to be written for consistency and/or change it. The definition of memory_read_list is very similar, only that the side effect is done after the actual memory access.

For the use with transformer invariants, sets of memory read and write state transformers are defined as follows.

```
% the set of state transformers that read at addresses
memory_read_transformers(pm : Memory_struct, addresses : PRED[Address]) :
                              PRED[[State -> SuperResult[State]]] =
  Lambda(q : [State -> SuperResult[State]]) :
    Exists(a : Address) : addresses(a) AND
      q = expr_2_super(memory_read(pm)(a))
```

Similarly for writing and for the side effects.

Side effects have been introduced for the formalization of memory mapped devices, alignment and reserved bit checks. In the case of an ordinary memory access the side effects do nothing. This is captured in part with the following definition.

```
% side effect transformers do not change the data read or to be written
side_effect_content_unchanged(addresses : PRED[Address], states : PRED[State],
       se_transformer : [Address, list[Byte], bool -> [State -> ExprResult[State, list[Byte]]]]) : bool =
  Forall(s : State, a : Address, bl : list[Byte], cp : bool) :
    states(s) And
    subset?(address_block(a, length(bl)), addresses) And
    OK?(se_transformer(a, bl, cp)(s)) Implies
      data(se_transformer(a, bl, cp)(s)) = bl
```

4.4.1.3 Localizing memory changes

Virtual memory and memory-mapped devices make memory in general a non-deterministic device. Outside of page tables and memory-mapped devices there is however lots of well-behaved memory. In order to capture this well-behaved memory we developed the following formalisms.

The following definition expresses that a set of state transformers (typically stemming from memory reads) does not change a certain set of addresses.

```
unchanged_memory_invariant?(pm : Memory_struct[State], states : PRED[State],
           transformers : PRED[[State -> SuperResult[State]]], addresses : PRED[Address]) : bool =
  transformer_invariant?(states, transformers) AND
  Forall(s : State, q : [State -> SuperResult[State]], a : Address) :
```

```
    states(s) AND transformers(q) AND addresses(a) AND
    OK?(q(s)) AND
    OK?(memory_read(pm)(a)(s)) AND
    OK?(memory_read(pm)(a)(state(q(s))))
    IMPLIES
      data(memory_read(pm)(a)(state(q(s)))) =
        data(memory_read(pm)(a)(s))
```

For write access to memory we have the following variation, which states that all addresses stay constant, except the one which is actually written.

```
unchanged_memory_write_invariant?(pm : Memory_struct[State], states : PRED[State],
                                            addresses : PRED[Address]) : bool =
  Forall(waddr : Address, b : Byte) :
    addresses(waddr) IMPLIES
      unchanged_memory_invariant?(pm, states,
              singleton(expr_2_super(memory_write(pm)(waddr,b))),
              remove(waddr, addresses))
```

To describe that the memory contents changes in the expected way we use the following.

```
changed_memory_invariant?(pm : Memory_struct[State],
                      states : PRED[State], addresses : PRED[Address]) : bool =
  transformer_invariant?(states, memory_read_transformers(pm, addresses)) And
  transformer_invariant?(states, memory_write_transformers(pm, addresses)) And
  Forall(s : State, a : Address, b : Byte) :
    states(s) And addresses(a) And
    OK?(memory_write(pm)(a,b)(s)) And
    OK?(memory_read(pm)(a)(state(memory_write(pm)(a,b)(s))))
      Implies
        data(memory_read(pm)(a)(state(memory_write(pm)(a,b)(s)))) = b
```

The preceding three definitions enjoy various monotonicity properties and are stable under union in different respects. This is expressed in a number of lemmas, which are not discussed here (see the source code in Appendix B on page 203 for details).

We can now precisely describe what parts of the memory must be well-behaved such that memory block access behaves as expected. There are a number of lemmas that say under which preconditions memory_read_list and memory_write_list terminate normally and return the right results. These lemmas form the heart of the proofs of the plain-memory rewrites, see Section 4.7.3 (on page 52) below. As example, we only show the following lemma that lists the preconditions that are necessary for memory_read_list to yield the same byte list that has just before been written to memory with memory_write_list.

```
unchanged_memory_read_list_write_list : Lemma
  Forall(pm : Memory_struct[State], states : PRED[State], waddr : Address, bl : list[Byte], s : State) :
    unchanged_memory_invariant?(pm, states, union(
      memory_read_transformers(pm, address_block(waddr, length(bl))),
      union(
        memory_read_side_effect_super_transformers(pm, address_block(waddr, length(bl))),
        memory_write_side_effect_super_transformers(pm, address_block(waddr, length(bl))))),
      address_block(waddr, length(bl)))
    AND
    unchanged_memory_write_invariant?(pm, states, address_block(waddr, length(bl)))
```

```
AND
changed_memory_invariant?(pm, states, address_block(waddr, length(bl)))
AND
transformers_ok?(states,
  union(
    memory_read_transformers(pm, address_block(waddr, length(bl))),
    memory_read_side_effect_super_transformers(pm, address_block(waddr, length(bl)))))
AND
transformers_ok?(states,
  union(
    memory_write_transformers(pm, address_block(waddr, length(bl))),
    memory_write_side_effect_super_transformers(pm, address_block(waddr, length(bl)))))
AND
side_effect_content_unchanged(address_block(waddr, length(bl)), states, memory_read_side_effect(pm))
AND
side_effect_content_unchanged(address_block(waddr, length(bl)), states, memory_write_side_effect(pm))
AND
states(s)
IMPLIES
  data(memory_read_list(pm)(waddr, length(bl))(state(memory_write_list(pm)(waddr, bl)(s)))) = bl
```

4.4.2 File abstract_data.pvs

This file contains the framework for the semantics of C++ data types. Intuitively every C++ data type describes a set of values. However, compounds such as structures, unions and classes can be partially initialized. Therefore, the simple approach of describing, for instance, the values of a structure as the Cartesian product of their ingredients does not work. Moreover, there are no operations on whole compounds. The C++ standard specifies that assignment and copying of compounds is done element-wise.

For these reasons we avoid the difficulties of modelling all possible values of compound types. For the semantics of C++ data types we distinguish between compound types, which are modelled without a semantics of their values, and fundamental types (such as `char` and `unsigned int`) whose possible values are formalized as a type in PVS. Diverging from the C++ standard, we treat pointer and reference types as fundamental types.

The formalization of compound types is as follows.

```
Uninterpreted_data_type : Type = [#
      size : nat,
      valid? : [list[Byte], Address -> bool]
  #]

uninterpreted_data_type? : PRED[Uninterpreted_data_type] =
  Lambda(uidt : Uninterpreted_data_type) :
    Forall(l : list[Byte], a : Address) :
      NOT length(l) = size(uidt) Implies valid?(uidt)(l,a) = False
```

A compound type consists of a size (identical to `sizeof`) and a validity predicate. The validity predicate determines whether a list of bytes, found at a particular address in memory, forms a valid object representation of that compound type. The only property required is that valid? must not be true on object representations of the wrong size. For a non-POD class[5] the valid? predicate might check it the hidden type information stored in any object (typically the vtable pointer) has not been tampered with.

The semantics of a particular structure or class consist of additional elements, which will be generated by the semantics compiler. The additional elements are offsets and sizes of the structure elements together with axioms that ensure that the elements do not overlap.

The semantics of fundamental types is based on uninterpreted_data_type?.

```
Interpreted_data_type : Type = [#
      uidt : Uninterpreted_data_type,
      to_byte : [Data, Address -> list[Byte]],
      from_byte : [list[Byte], Address -> lift[Data]]
  #]
```

In addition to size and valid? an interpreted data type consists of a type of all possible values, which is represented as type parameter Data here, and two function that convert any value into their object representation and back.

For historic reasons there exists an intermediate level in the formalization (called interpreted_data_type?) that is not used for any C++ type currently. The semantics of fundamental C++ types is formalized as pod_data_type? (which was originally intended for POD structures and unions as well before we dropped that idea because of the problem with partial initializations).

```
interpreted_data_type? : PRED[Interpreted_data_type] =
  Lambda(idt : Interpreted_data_type) :
    uninterpreted_data_type?(idt`uidt) AND
      % result of to_byte is valid (and has length size)
    (Forall(d : Data, a : Address) :
       valid?(uidt(idt))(to_byte(idt)(d,a), a)) AND

      % from_byte fails on invalid stuff
    (Forall(l : list[Byte], a : Address) :
       valid?(uidt(idt))(l, a) IFF
              up?(from_byte(idt)(l, a))) And

      % 2. the result is the same
    (Forall(d : Data, a : Address) :
       down(from_byte(idt)(to_byte(idt)(d,a), a)) = d )

pod_data_type? : PRED[Interpreted_data_type] =
  Lambda(idt : Interpreted_data_type) :
    interpreted_data_type?(idt) AND
    (Forall(l : list[Byte], a1, a2 : Address) :
        valid?(uidt(idt))(l, a1) IFF valid?(uidt(idt))(l, a2)) And
```

[5]Plain Old Data (POD) comprises all those data (types) of C++ that are present in C too. A non-POD class contains at least one nontrivial constructor, copy constructor, destructor or a virtual function. Downcasts are only possible on non-POD classes, because only they contain runtime type information, typically in the form of a vtable pointer.

```
   size(uidt(idt)) > 0
```

Note, that for the semantics of fundamental types the valid? predicate is superfluous. The axiomatization says that from_byte is a left inverse of to_byte, that from_byte succeeds precisely if the object representation is valid and that the object representation is independent from the address on which the data is stored.

A typical formalization of a fundamental type (see Section 4.7.1) will define the type of values, but leave to_byte and from_byte underspecified. This way the verification results do not depend on a particular object representation of a particular compiler (like for instance the endianness). Moreover, similarly to the C++ standard our formalization leaves the possibility open that the object representation contains runtime type information. Therefore our verification will find all type errors, because it is impossible to derive anything about the result of from_byte on a object representation of a different type.

Finally we define functions to read and write values of arbitrary C++ types from and to memory.

```
read_data(pm : Memory_struct[State], dt : (interpreted_data_type?[Data]))
                                    (addr : Address) : [State -> ExprResult[State, Data]] =
  (memory_read_list(pm)(addr, size(uidt(dt))) ##
     Lambda(bl : list[Byte]) : ok_lift(from_byte(dt)(bl, addr)))

write_data(pm : Memory_struct[State], dt : (interpreted_data_type?[Data]))
                           (addr : Address, data : Data) : [State -> ExprResult[State, Unit]] =
      memory_write_list(pm)(addr, to_byte(dt)(data, addr))
```

The utility function ok_lift converts from_byte into an expression state transformer.

```
ok_lift(d : lift[Data])(s : State) : ExprResult[State, Data] =
  CASES d OF
    bottom : Fatal,
    up(b) : OK(s, b)
  ENDCASES
```

4.4.3 File plain_memory.pvs

The memory in an IA32 system is a sophisticated device: segments and page tables specify access rights, a given piece of memory might be visible in different virtual-address ranges, the address translation in the CPU from virtual to physical addresses might differ from what is specified in the page table because of bogus TLB entries, and much more. We cannot ignore all these effects, not even for most innocent kernel code, because of the errors that they might cause.

As a consequence we designed the *plain-memory* abstraction for the verification of those parts of the kernel that require only the standard C++ memory model. It deals with the following issues.

- Writing or reading a single byte in memory can have devastating effects if one hits a memory-mapped device, a page table or simply an unmapped address. For correctness, the verification must therefore be carried out against a faithful model of IA32 memory. Plain memory provides a (comparatively) simple abstraction that can be used for those parts of the sources that only need well-behaved memory without special effects.

- The IA32 hardware provides several memory configurations: real-address mode, protected mode with and without paging. Our hardware model multiplies the number of different memories because we prefer to model different memory features in different memory models. Most of the code does not depend on a concrete memory model and should consequently be verified against a suitable set of memory models. Plain memory permits precisely this because every memory model of interest will give rise to a model of plain memory.

Technically plain memory is a specification that provides byte-wise read and write access to memory, where special properties are guaranteed for *read-blessed* and *read/write-blessed* address regions. The general idea is simple. Memory at blessed addresses is sane: read access does not change anything in the blessed address range, and write access only changes the bytes written (in the expected way). Moreover, these special properties are maintained as long as only blessed addresses are accessed. No guarantees are made however for a memory access outside the blessed address regions. We have shown in PVS that these properties are satisfied under suitable preconditions by physical memory (see Appendix B on page 115) and also by normal virtual memory outside of memory-mapped devices (see Appendix B on page 94).

We want the plain-memory specification to be usable with all concrete memory models (including physical real-address memory). Therefore the specification must describe all properties only with the observations that can be made by reading and writing single bytes. In PVS the specification is split into a record of functions (capturing the plain-memory signature) and a predicate for the required properties. With this technique the axioms of the plain-memory specification do not show up as axioms in the PVS formalization, hence they do not affect consistency. Instead, any use of a plain-memory property in a proof will spawn a subgoal requiring the proof of the plain-memory axioms. The plain-memory signature is defined as follows:

```
Plain_Memory : Type = [#
  mem : Memory_struct[State],        % see Section 4.4.1.2 on page 38
  states  :  PRED[State],            % states fulfilling the plain memory properties
  ro_addr  :  PRED[Address],         % read−blessed addresses
  rw_addr :  PRED[Address]           % write−blessed addresses
#]
```

The properties of plain memory are specified as follows.

```
plain_memory?(pm : Plain_Memory) : bool =
  unchanged_memory_invariant?(pm‘mem, pm‘states,
    union(
      union(pm‘other_actions, memory_read_transformers(pm‘mem, union(pm‘ro_addr, pm‘rw_addr))),
      union(memory_read_side_effect_super_transformers(pm‘mem, union(pm‘ro_addr, pm‘rw_addr)),
            memory_write_side_effect_super_transformers(pm‘mem, pm‘rw_addr))),
```

```
    union(pm`ro_addr, pm`rw_addr))
  AND
  unchanged_memory_invariant?(pm`mem, pm`states,
              memory_write_transformers(pm`mem, pm`rw_addr),
              pm`ro_addr)
  AND
  unchanged_memory_write_invariant?(pm`mem, pm`states, pm`rw_addr)
  AND
  changed_memory_invariant?(pm`mem, pm`states, pm`rw_addr)
  AND
  transformers_ok?(pm`states,
    union(
      union(memory_read_transformers(pm`mem, union(pm`ro_addr, pm`rw_addr)),
            memory_write_transformers(pm`mem, pm`rw_addr)),
      union(memory_read_side_effect_super_transformers(pm`mem, union(pm`ro_addr, pm`rw_addr)),
            memory_write_side_effect_super_transformers(pm`mem, pm`rw_addr))))
  And
  side_effect_content_unchanged(union(pm`ro_addr, pm`rw_addr), pm`states,
                                          memory_read_side_effect(pm`mem))
  And
  side_effect_content_unchanged(pm`rw_addr, pm`states, memory_write_side_effect(pm`mem))
```

The first clause states that read accesses to blessed addresses and all possible side effects do not change the contents of any of the blessed addresses. The second clause expresses the same for write accesses and the read-blessed addresses. The third clause requires that a write access to one address leaves all other read/write-blessed addresses intact. The fourth clause states that write accesses actually change the written address in the right way. The utility predicates used in the first four clauses additionally require that the set of states forms an invariant with respect to the respective set of state transformers. This makes the plain-memory property an invariant: permitted state transformers must stay in the set of plain-memory states, in which all the nice properties hold. The fifth clause makes all memory accesses terminate with OK (which prohibits e.g. unhandled page-faults). The last two clauses require that side effects do not change the data read or written.

The plain-memory specification entails that only explicit writes change a memory cell. This property enables us to prove the following lemma.

```
plain_memory_read_write_other_res : Lemma
  plain_memory?(pm) AND
  pm`states(s) AND
  in_blessed_memory?(dt1, addr1, pm`rw_addr) AND
  in_blessed_memory?(dt2, addr2, union(pm`ro_addr, pm`rw_addr)) AND
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) AND
  valid_in_mem(pm,dt2)(addr2)(s)
    IMPLIES
      data( (write_data(pm,dt1)(addr1, data1) ##
             read_data(pm,dt2)(addr2))(s) )
        =
      data( read_data(pm,dt2)(addr2)(s) )
```

This lemma expresses that for two variables of type dt1 and dt2 that lie disjoint in blessed memory, writing the first one does not change the contents of the second. There are similar lemmas for the other combinations of read_data and write_data. These lemma are used in a rewrite engine that enables PVS to symbolically compute the value of a variable by going back to the last write access to that variable.

4.5 Concrete memories

Based on the abstract memory interface described in Section 4.4.1.2, we have defined several concrete memory instances: physical memory (RAM), linear memory with page-table based address translation, device memory, etc. For these memories, we have shown that they satisfy (under suitable assumptions) the plain memory conditions explained in Section 4.4.3 (on page 43 above).

4.5.1 File physical_memory.pvs

Our hardware model contains physical memory (RAM) as a base of all memory models. Physical memory provides one byte of storage for every address up to a certain maximum. Accesses above the maximum yield Fatal as result.

4.5.2 File challenge-phymem.pvs

Unsurprisingly we can prove that (under suitable assumptions about the possible side effects) all states of physical memory form a plain memory, with all addresses below the maximum read/write-blessed.

```
phy_mem_plain_memory : Lemma
    unchanged_memory_invariant?(phy_mem,
              fullset[Physical_memory], oact, in_memory(min, max))
      Implies
        plain_memory?(phy_pm(oact))
```

This theorem and the necessary lemmas for its proof are contained in this file.

4.5.3 Files paging-data.pvs and paging-data-models.pvs

These two files contain the formalization of those hardware data types that are important for page-table traversal. This are in particular the page-table base register (CR3), first level 4K and 4M page-table entries and second level page-table entries. For these hardware types a PVS type is defined and then to_byte and from_byte functions are axiomatically introduced.

In order to exclude the possibilities of inconsistencies introduced with those axioms the file `paging-data-models` defines concrete to_byte and from_byte functions and proves all axioms as theorems (in particular that those functions fulfill pod_data_type?). These to_byte and from_byte functions convert records of booleans and addresses into lists of four bytes and vice versa. For that they use our own formalization of bit vectors from `bits.pvs`, see Section 4.1.2. The proofs are almost fully automatic.

4.5.4 File linear_memory.pvs

Our model of linear memory contains page-table based address translation, but no TLB or page-fault handler yet. The linear memory is stacked on top of a plain memory using the general memory-structure interface. This plain memory is typically instantiated with physical memory (possibly containing devices). The formalization follows closely the hardware manuals, but is far from trivial, see Appendix B on page 194.

4.5.5 File challenge-linear.pvs

We have shown in PVS that the plain-memory properties are obtained for linear memory under the following preconditions:

- The code segment register (`CS`), determining the code privilege level, the page-table base register, and the page tables themselves remain unchanged (with the exception of access bits).
- Any translation for read or execute accesses succeeds for the entire blessed range of virtual addresses. Translations for writes succeed for the writable subset.
- Blessed writable virtual addresses map to blessed writable physical addresses, blessed read-only addresses map to blessed readable or writable physical addresses.
- Page tables reside in a memory area that is disjoint from the targets of the above mappings.
- There is no blessed shared-memory alias to a writable virtual address. (Virtual read-only regions may be shared arbitrarily.)

One point to highlight is that we only have to require those page-table entries to remain unchanged that are used in the translation of the virtual blessed address range. This allows us to modify unrelated page-table entries without loosing the blessing properties. We achieve this by requiring

```
disjoint?(virt_to_phys_range(s, union(pm`ro_addr, pm`rw_addr)),
          address_in_pt_range?(s, union(pm`ro_addr, pm`rw_addr))),
```

where virt_to_phys_range translates all addresses in the virtual blessed address range, and address_in_pt_range? returns the corresponding physical addresses containing the page-table entries for this range.

4.6 Allocation, File allocators.pvs

In the running Nova kernel several memory allocators will be active at the same time. One for global and static variables (that actually have been allocated at link time), one for local variables on the stack (where allocation is in-lined by the compiler), a buddy allocator managing all dynamic memory and several allocators (typically slab-allocators) optimized for a certain use that obtain their memory from the buddy allocator.

All together these allocators maintain a simple looking kernel invariant: Correctly allocated variables are disjoint. The aim of the formalization in this file is to derive this kernel invariant from an abstract formalization of a set of allocators. (This differs from a complete axiomatization or characterization of memory allocators, which we do not try to achieve here.)

We formalize a single memory allocator as a record of six functions.

```
Allocator : Type = [#
  memory_pool : [(plain_memory?) -> [State -> PRED[Address]]],
  allocated : [(plain_memory?) -> [State -> PRED[Address]]],
  private_mem : [(plain_memory?) -> PRED[Address]],
  freed_size : [(plain_memory?) -> [Address -> [State -> lift[posnat]]]],

  alloc: [(plain_memory?) -> [posnat -> [State -> ExprResult[State, Address]]]],
  free: [(plain_memory?) -> [Address -> [State -> ExprResult[State, Unit]]]]
#]
```

The first four are model functions, which means that they do not appear explicitly in the source code of the allocator. The last two are the semantics of the alloc and free functions of the allocator source code. The information that the first four function provide is somewhere implicit in the source code of the allocator. The formalization of allocators builds on plain memory, therefore all these functions take a plain memory as argument.

The function memory_pool gives the address range that the allocator controls, allocated gives the part of the memory_pool that has already been allocated, private_mem describes the memory area that contains the state variables of the allocator and freed_size tells the size of a memory block that will be freed by a call to free. The state variables of the allocator (private_mem) are typically outside of the memory pool of the allocator itself and are possibly allocated in a different allocator. Dynamic memory allocators (such as a buddy or slab allocator) can internally associate the size of a memory block to every allocated address. However, this is not the case in a stack allocator, which simply increases and decreases a stack pointer. But even in a stack allocator one can tell the size of the memory block that will be freed (everything up to the top of the stack).

The required properties are rather lengthy (about 80 lines of PVS code). For an allocator we require

- private_mem is inside the plain memory and alloc and free stay in the states invariant of the plain memory.
- allocated is a subset of memory_pool
- If alloc terminates normally and does not return the null pointer, then the newly allocated region is disjoint from allocated in the pre-state and the allocated predicate changes accordingly. Further the memory pool does not change.
- free is successful precisely if freed_size is
- If free is successful than allocated changes accordingly and the memory_pool stays constant.

- allocated and memory_pool do not change as long as the private_mem stays constant.
- alloc and free do only change writable private_mem.

What is really of interest for us is the list of *allocation points* of an allocator. An allocation point is a pair of an address and a size that comes from a successful allocation which has not been freed since. Typically an allocation point precisely describes the memory region of a correctly allocated variable. While some allocators contain enough internal state to compute the list of their allocation points, others, most notably stack allocators, do not. For this reason there is no model function returning the allocation points. Instead we formalize allocation points independently of allocators and define a consistency criterion between a list of allocation points and a memory allocator.

A list of allocation points is captured with the type Allocator_Info.

```
Allocator_Info : Type = list[[Address, posnat]]
allocator_info? : PRED[Allocator_Info] = Lambda(ai : Allocator_Info) :
  blocks_pairwise_disjoint(ai)
```

Note that in the context of allocators all memory blocks are nonempty. Here, blocks_pairwise_disjoint is a utility function, coming from memory.pvs, compare Section 4.4.1 (on page 38).

```
blocks_pairwise_disjoint(blocks : list[[Address, posnat]]) : bool =
  Forall(addr_1, addr_2 : Address, size_1, size_2 : posnat) :
    member((addr_1, size_1), blocks) And
    member((addr_2, size_2), list_remove((addr_1, size_1), blocks))
      Implies
        blocks_disjoint?(addr_1, size_1, addr_2, size_2)
```

The consistency criterion simply states that all allocation points are inside the allocated part.

```
consistent_allocator?(pm : (plain_memory?))(ac : (allocator?(pm)), ai : Allocator)(s : State) : bool =
  allocator_info?(ai) And
  Forall(addr : Address, size : posnat) : member((addr, size), ai) Implies
    subset?(address_block(addr, size), ac`allocated(pm)(s))
```

The set of allocators that are active in a certain state are described as a finite forest (i.e., a finite tree with multiple root nodes) of memory allocators. If allocator a_2 is a child of allocator a_1 (there is an edge from a_1 to a_2) then a_2 obtained its memory pool by allocating a suitable block in a_1. For instance, a slab allocator obtains its memory pool by allocating unstructured memory somewhere else.

The formalization of active allocators, called Allocation_Table is based on our formalization of graph theory, see Section 4.1.3 (on page 27).

```
Allocation_Table(pm) : Type = [#
  hierarchy : (finite_tree?[(allocator?(pm))]),
  info : [(hierarchy`nodes) -> Allocator_Info]
#]
```

An allocation table consists of a finite forest of allocators together with a list of allocation points for every allocator. We call an allocation table consistent (in PVS allocation_ok?) if it fulfills the following properties.

- The memory pool of a child must be allocated in the parent. Further, the memory pool of a child must not be listed as allocation point of the parent (it must be disjoint from all allocation points of the parent).
- The memory pools of two children of the same parent must be disjoint.
- The memory pools of two root allocator nodes must be disjoint.
- The private memory of any two allocators must be disjoint.
- Any allocator in the tree must be consistent with its list of allocation points.

With Theorem 1 (on page 30) we can proof the following theorem, which forms the basis of the other results on allocation tables.

Theorem 2 (Lemma allocation_memory_pools) *For two allocators of an consistent allocation table precisely one of the following is true.*

- *The two allocators are equal.*
- *Their memory pools are disjoint.*
- *The memory pool of one allocator, say a_1, is allocated in the other, say a_2, such that the allocation points of a_2 are disjoint from the memory pool of a_1.*

So far we proved the following two top level theorems about allocation tables. We first consider the list of all allocation points of an allocation table. This list of all allocation points is not so easy to define.

```
allocation_info(pm : (plain_memory?))(at : Allocation_Table(pm)) : Allocator_Info =
  flatten(map(at`info)(list_of_finite_set(at`hierarchy`nodes)))
```

We first convert the finite set of allocators in an allocation table into a list, substitute each allocator with its list of allocation points (using conventional list map) and flatten the resulting list of lists.

Theorem 3 (Lemma allocation_consistent) *All allocation points of a consistent allocation table are pairwise disjoint (fulfilling allocator_info?).*

Assuming that the allocation points are maintained during the proof, the preceding theorem precisely matches the before-mentioned invariant that all correctly allocated variables are disjoint.

The second result considers a successful allocation. It facilitates maintaining a consistent allocation table during the symbolic evaluation of the program.

Theorem 4 (Lemma allocation_alloc) *Consider a consistent allocation table at and assume that alloc for one of its allocators a is successful, not returning the null pointer. Let the allocation table at′ be identical to at except that the newly allocated block is added to the allocation points of a. Then at′ is consistent.*

4.7 C++ Semantics

Our C++ semantics is based on a shallow embedding of C++ expressions and C++ statements as state transformers (see Section 4.3) in PVS. The development of the semantics was driven by two design goals: *standard compliance* and *modularity*. Standard compliance means that in general we only want to be able to prove those C++ programs correct that run on *any* C++ implementation, based only on the guarantees that are provided by the C++ standard. The correctness of verified programs should not hinge on certain compiler- or architecture-specific assumptions, say about memory layout or data-type sizes. On the other hand, the Nova sources cannot be verified without relying on such assumptions to some extend. Therefore our C++ semantics needs to be modular, in the sense that additional guarantees (provided e.g. by the compiler and/or hardware architecture) should be easy to add as additional axioms.

The shallow embedding of statements and expressions is complemented by a deep embedding of C++ types. This deep embedding is necessary because types with otherwise identical behavior may still give rise to different compound types: e.g. pointers to **signed char** and **char** could differ, even on architectures where these two types are otherwise identical. The details of our formal C++ semantics are briefly discussed in the following subsections.

4.7.1 File types.pvs

This file contains PVS definitions for the various C++ types and type constructors, as defined in Clauses 3, 7, 8, and 9 of the C++ standard. Our model of C++ types is based on a generic data-type formalization; data types are given by records that contain a size value, encoding and decoding functions that transform between semantic values and their bit representation in memory, and more (see Section 4.4.2 (on page 41) for details). In `types.pvs`, we provide (uninterpreted, but axiomatically restricted) records for all *fundamental* types: **unsigned char**, **signed char**, **char**, **short**, **int**, **long**, **long long**, **unsigned short**, **unsigned int**, **unsigned long**, **unsigned long long**, **wchar_t**, **bool**, and **void**. In addition, we provide (parameterized) records to construct pointer and reference types. For this, the file contains a deep embedding of C++ types as a recursive data type in PVS.

4.7.2 File statements.pvs

This file contains PVS definitions for various C++ statements, as defined in Clause 6 of the C++ standard: labeled statements (**case**, **default**) (6.1), expression statements (6.2), selection statements (**if**, **switch**) (6.4), iteration statements (**while**, **do** ... **while**, **for**) (6.5), jump statements (**break**, **continue**, **return**) (6.6), and declaration statements (6.7). Compound statements (blocks) (6.3) do not need to be modeled explicitly in our shallow embedding. A notable omission is the **goto** statement (6.6.4), which would complicate the formalization of other statements, compare [Tew04]. Similarly **case** and **default** labels are permitted at top-level statements only, that is, we do not allow jumps into loop bodys or other nested sub-statements. Although such jumps would not be difficult to add in principle, they would complicate the formalization of all other statements.

4.7.3 File plain_memory_rewrites.pvs

This file contains several rewrite lemmas to simplify expressions of the form f(q ## rw1 ## rw2), where f is either OK? or data, q is a state transformer, and rw1 and rw2 are read or write accesses to a plain memory. For instance, the following lemma states that a previous write access can be ignored when reading from a disjoint address range:

```
plain_memory_read_write_other_q_data_expr : Lemma %(P: 6)
  Forall (q : [State -> ExprResult[State, Data_q]]) :
    plain_memory?(pm) And
    in_blessed_memory?(dt1, addr1, union(pm`ro_addr, pm`rw_addr)) And
    in_blessed_memory?(dt2, addr2, pm`rw_addr) And
    blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
    pm_q_prop_read(pm, dt1, addr1)(q(s))
  Implies
    data((q ## write_data(pm, dt2)(addr2, data2) ## read_data(pm, dt1)(addr1))(s)) =
      data((q ## read_data(pm, dt1)(addr1))(s))
```

Similar lemmas are stated and proved for other combinations of read and write operations.

4.7.4 File datatype_model.pvs

In this file, it is shown that our generic data type model for POD ("plain old data") types permits *type tags*, which can be used to implement dynamic type checking: for instance read_data(pm, pod_2)(addr) after write_data(pm, pod_1)(addr) fails, where pod_1 and pod_2 are PODs that use different type tags.

The data types defined in this file are merely examples, and are not used anywhere else in our verification environment. They do not coincide with actual C++ types.

4.7.5 File conversions.pvs

This file contains PVS definitions for various C++ conversions, as defined in Clause 4 of the C++ standard: lvalue-to-rvalue conversions (4.1), various integral promotions (4.5) and integral conversions (4.7), and Boolean conversions (4.12).

4.7.6 File expressions.pvs

This file contains PVS definitions for various C++ expressions, as defined in Clause 5 of the C++ standard. Expressions are modeled as (shallowly embedded) expression state transformers. We have defined state transformers for primary expressions (5.1), postfix expressions (5.2), unary expressions (5.3), explicit type conversion (5.4), pointer-to-member operators (5.5), multiplicative operators (5.6), additive operators (5.7), shift operators (5.8), relational operators (5.9), equality operators (5.10), bitwise operators (5.11-5.13), logical operators (5.14, 5.15), the conditional operator (5.16), assignment operators (5.17), and the comma operator (5.18).

Two notable deviations from the C++ standard are the evaluation order of subexpressions, and our treatment of modulo arithmetic. For subexpressions, the C++ standard says that the behavior is undefined unless a (somewhat obscure) side condition is satisfied (cf. 5.4) which implies that the evaluation order is actually irrelevant. We have not modeled this undefinedness; instead, we have chosen some fixed evaluation strategy. A simple pre-processing step could be applied to the Nova C++ sources to ensure that the code meets the side condition of 5.4.

The C++ standard requires that unsigned integer types implement arithmetic modulo their first unrepresentable value, for instance, 32-bit integers implement arithmetic modulo 2^{32}. The Nova code rarely relies on this guarantee. In most cases an arithmetic overflow at runtime, even if permitted by the C++ standard, indicates a programming error, and should be visible in the verification environment. Therefore an underspecified function is used to implement arithmetic on unsigned integer types. This function can be defined to implement modulo arithmetic (as the standard requires), or to yield an error in case of an overflow (as we believe is reasonable in most cases).

Another issue is the treatment of function calls. The C++ standard does not specify just where exactly function arguments or return values are stored in memory. (This is usually defined in the ABI documents for an architecture, e.g. in Section 3.9 of the "System V ABI Intel 386 Architecture Processor Supplement".) To implement function calls on our hardware model, we made some (hopefully reasonable) choices here.

4.7.7 File statement-rewrites.pvs

This file contains rewriting rules for C++ statements. These rules simplify code by symbolic simplification of the control flow constructs. For example a break will remove all following statements until it reaches a catch_break, in which case break ## catch_break is eliminated from the code. Statements are simplified right to left. Here one expression transformer (respectively one transformer of type StmtResult $\rightarrow$ ExprResult) is allowed to occur after the simplification point. The forward rules simplify these two cases plus those cases with a leading statement automatically with the help of the respective rules for the standalone case. Because of the requirement on rewriting rules these lemmas have to be defined for any form a standalone simplifcation may assume. The forms are denoted by the `s-c->s, ...` sequence which stands for "statement1 ## complex_statement simplifies to statement1".

4.8 Verification examples

Our C++ semantics, defined on top of a generic hardware and memory model, forms the basis of a versatile verification environment for C++ programs in PVS. We have used this verification environment for several case studies, showing correctness of (relatively small and simple) C++ implementations of various algorithms.

4.8.1 File cpp-examples.pvs

This file contains the PVS semantics of some (rather simple) C++ programs and program fragments, together with suitable Hoare specifications. These programs serve as examples and benchmarks for our C++ semantics, the Hoare logic, the verification environment, and the degree of automation achieved in verification proofs.

4.8.2 File search-example.pvs

This file contains a correctness proof for a simple linear search algorithm, implemented in C++:

```
/**
 * search the array element whose value is val
 *
 * @param val the value to search for
 * @param first pointer to the first element of the array (from which
 * to start the search)
 * @param last pointer beyond the last element of the array
 * @return pointer to the first occurrence of val after (including) first
 */
int const * search(int const val, int const * const first,
                   int const * const last) {
  assert(first);
  assert(last);

  int const * current = first;

  while(current < last) {
    if (*current == val)
      break;

    current++;
  }

  return current;
}
```

It is shown that this algorithm (under suitable preconditions) terminates normally, that it returns a pointer to the first occurrence of val in the array located at base address first, and that it returns last if val is not contained in the array (provided first <= last).

4.8.3 File device_memory.pvs

This file defines a generic way to extend a given memory structure with an additional state component, e.g. to add device state to the (memory) state of the hardware. We model the (possibly concurrent) behavior of devices by using state transformers that can update both the device state and the other state of the hardware upon memory (including device memory) access.

The main result that is established is the following. The extension of a plain memory structure with a sufficiently well-behaved device (i.e. without side effects that modify the memory at blessed addresses) forms a plain memory again:

```
Device_memory : Type = Expand_state[Physical_memory, Device_state]

pm : Var Plain_Memory[Device_memory]

device_memory_plain_memory : Lemma
  Forall (pm) :
    is_device_plain_memory?(device_read_side_effect,
                            device_write_side_effect)(pm)
      Implies plain_memory?(pm)
```

4.8.4 File random_device.pvs

This file contains the formalization of a random number generator as a memory-mapped device. The device state consists of a seed value, a counter for memory accesses, and a "random" (underspecified) value. It is shown that integrating this device with a plain memory again establishes a plain memory (with suitable assumptions about the blessed address ranges).

4.8.5 File cpp-verification.pvs

This file is merely a top-level wrapper theory. It imports all other theories that together form our generic verification environment for C++ programs in PVS.

4.8.6 File ptab-sync-master-defs.pvs

This file contains hand-translated semantics for parts of the Nova C++ implementation of Ptab::sync_master. It is necessary for a verification of Ptab::sync_master because the implementation of this function, either directly or through calls to other functions, uses some C++ constructs that our semantics compiler cannot (yet) translate to PVS automatically. If the semantics compiler was extended to handle a sufficiently rich subset of C++, the file `ptab-sync-master-defs.pvs` should eventually become unnecessary.

4.8.7 File ptab-sync-master.pvs

This file is generated by the semantics compiler. It contains the semantics of the method Ptab::sync_master from the Nova sources. We are currently working on first verification goals for this Nova method.

5 Conclusion

This document describes the Robin verification environment for systems-level C++ programs that we have developed in work package 4 (kernel specification and verification) of the Robin project. An important part of the verification environment is the formalization of the IA32 hardware and the semantics of C++ in PVS. Appendix B and C contain the complete PVS sources and the proofs, generated automatically from our CVS repository. The PVS sources can also be downloaded from `http://www.cs.ru.nl/~tews/Robin`.

The Robin verification environment can be used for the verification of systems-level C or C++ code, especially for operating-system kernel code. The formalization closely follows the standard documents, in particular the IA32 manuals [Int07a] and the C++ standard [Int07b]. However, compiler specific assumptions and/or best practices (which are known to work, although not backed up by the standard) can be added in an orthogonal way if necessary.

Even with the verification environment presented here, the formal verification of systems-level code remains challenging. Our verification environment contains several solutions to achieve a reasonable level of abstraction and automation; most notably the plain memory specification presented in Section 4.4.3. The case studies that were carried out show that our approach is promising, and that the implemented solutions can greatly reduce the human effort required to reason about systems-level code fragments in an interactive theorem prover.

A Bibliography

[Chi] Shigeru Chiba. Open C++ Website. http://opencxx.sourceforge.net/.

[Chi93] Shigeru Chiba. Open C++ Programmer's Guide. Technical Report 93-3, Department of Information Science, University of Tokyo, 1993.

[CW] Karl Chen and Daniel S. Wilkerson. Oink: a Collaboration of C++ Static Analysis Tools. http://www.cubewano.org/oink.

[FHJ+07] Ansgar Fehnker, Ralf Huuck, Patrick Jayet, Michel Lussenburg, and Felix Rauch. Goanna – A Static Model Checker. In L. Brim, B. Haverkort, M. Leuker, and J. van de Pol, editors, *Formal Methods: Application and Technology*, volume 4346 of *Lecture Notes in Computer Science*, pages 297–300. Springer Verlag, 2007.

[Gle07] T. Glek. Dehydra source analysis tool. In H. Tews, editor, *Proceedings of the C/C++ Verification Workshop*, pages 81–93, July 2007. Technical report ICIS-R07015, Radboud University Nijmegen.

[Gra] Graphviz – graph visualization software. available from www.graphviz.org.

[HT] M. Hohmuth and H. Tews. The vfiasco project. Website www.vfiasco.org.

[HT05] M. Hohmuth and H. Tews. The VFiasco approach for a verified operating system. In *Proceedings of the 2nd ECOOP Workshop on Programm Languages and Operating Systems*, Glasgow, 2005.

[Int07a] Intel Corporation, Denver, CO. *Intel 64 and IA-32 Architectures Software Developer's Manual*, May 2007. Order Number: 25366[5-9]-023US.

[Int07b] International C++ Committee of the Organization for Standardization. *The C++ Standard, Incorporating Technical Corrigendum 1.* John Wiley & Sons Ltd, Chichester, England, 2007.

[lon] longjmp, siglongjmp — non-local jump to a saved stack context. Linux manual page.

[McP] Scott McPeak. Elsa: An Elkhound-based C++ Parser. http://www.cs.berkeley.edu/ smcpeak/elkhound/.

[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.

[Tew04] H. Tews. Verifying Duff's device: A simple compositional denotational semantics for Goto and computed jumps. Draft, 2004. Available from `wwwtcs.inf.tu-dresden.de/~tews/science.html`.

[Vel] T. L. Veldhuizen. C++ templates are turing complete. Available at citeseer.ist.psu.edu/581150.html.