Embedding JML-Annotated Programs in the Coq Proof System

Andreas Kägi

Master Project Report

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

http://www.pm.inf.ethz.ch/

March 2009

Supervised by:
Hermann Lehner
Prof. Dr. Peter Müller

Chair of Programming Methodology
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

JML is a complex specification language for Java. Its large scale and manifold features make it hard to precisely define its semantics in a reference manual. It is thus desirable to formally specify the syntax and semantics of JML.

This thesis builds on top of a formalisation of JML and Java in the Coq Proof Assistant. Its goal is to extend the formalisation with a translation frontend, translating JML-annotated Java programs into the embedding of JML and Java in Coq. Furthermore it envisions to extend the formalised syntax with constructs not previously covered, most notably with the different forms of method specifications.

The translation frontend enables to link JML-annotated programs with the formalisation which is particularly useful if the formalisation would be used as a verification environment. The separate extension of the basic syntax to more advanced JML constructs is assistant in covering all features of JML, while still having a basic syntax subset to consider for the formalisation of JML's semantics.

The thesis starts by defining a syntax superset and rewritings of these extended syntax constructs in terms of basic syntax constructs, all within the Coq Proof Assistant. A second part consists of implementing the translation frontend in Java. The overall goal is to define the largest possible part within Coq and to have a minimal frontend that focuses on generating code that resembles the original code as much as possible. Finally, the thesis conducts a case study evaluating the feasibility of proving on top of the formalisation.

Contents

Chapter 1

Introduction

JML is a widely accepted specification language for Java. It is a complex language offering manifold features. This results in a Reference Manual that counts more than 150 pages. Defining the semantics of a language in free textual form is a big challenge: it is easy to forget special cases or to give ambiguous or even contradictory definitions.

To circumvent these problems, it is desirable to formally specify the syntax and semantics of JML. Specification within a theorem prover has the further advantage that semantic properties may be formulated and proven or the formalisation may be used as a verification environment for JML-annotated programs.

This thesis deals with a formalisation of JML and Java in the Coq Proof Assistant. It contributes four parts to the existing formalisation: First, the definition of a syntax superset, extending the previously defined basic syntax with constructs that are expressible in terms of basic constructs. Second, a transformation of the extended syntax superset into the basic syntax set, realised within the Coq Proof Assistant itself. Third, a frontend that translates JML-annotated Java source programs into Coq source files that represent the programs in the formalised syntax. Finally, the thesis conducts a case study that investigates the feasibility of formulating and discharging proofs on top of the existing formalisation.

The goal of this approach is to define the largest possible part of JML in the Coq Proof Assistant and to have a frontend that is as simple as possible. Following this approach, even the simpler desugarings of constructs from the extended syntax superset are part of the formalisation. Defining a syntax superset that is independent of the basic syntax used to formalise the semantics of JML has two major advantages: First, the definition of the semantics is not cluttered with mere syntactic sugar, while second, the syntax superset can still be built in a way that retains the readability of the original source code as much as possible.

The implementation of a frontend is still of value be it as part of a verification environment that requires programs to be represented in the formalisation or to represent programs used in testing and evaluating the formalisation.

The next chapter gives the necessary background information on JML and Coq. Chapter 3 gives an overview about the extensions to the formalisation as done by this thesis. Chapter 4 gives the details of the individual components: basic syntax – extended syntax – rewritings – translation frontend – case study. The thesis ends with its conclusions in chapter 5.

Chapter 2

Background

2.1 JML

JML [10] [9], the Java Modelling Language, is a specification and modelling language for Java. It offers a broad range of features, including support for floating-point numbers and concurrency.

The most important features relevant to this thesis are *type specifications* and *method specifications*. JML also adds new statements and expressions to the Java language, most notably the logical connectives
`forall` and
`exists` of predicate logic that allow to quantify in an expression over arbitrary values of a given data type.

```
// In the following examples, we assume the existence of a function
// "boolean isPrime(int)" and an object "set1" of type java.util.Set.

\exists int i; 0 <= i && i <= 10; isPrime(i);
\forall Object o; set1.contains(o); o != null;
```

Listing 2.1: Example predicates of quantified expressions

2.1.1 Type Specifications

JML supports the notion of *invariants* and *history constraints*. An invariant for an object `o` is a predicate that has to hold in all visible states of object `o`. A history constraint is a predicate that is a relationship that should hold for the combination of each visible state of object `o` and any visible state of `o` that occurs later in the program's execution.

Further type specification clauses include the `initially` clause, a predicate that each constructor has to establish, and the `represents` clause that deals with *model fields*.

A model field `mf` is a field declared with modifier `model`. It is an abstract, specification-only field of an object, whose value is determined by the `represents` clause: either the value is given by the value of another field `f` (`represents mf <- f`) or the value is determined non-deterministically such that it satisfies a given predicate `P`: `represents mf \such_that P`.

Besides model fields, JML supports another kind of specification-only field, so called *ghost fields*. In contrast to model fields, ghost fields are concrete fields and their value is set via JML statement `set`.

The other type specification clauses are less relevant for this thesis.

2.1.2 Method Specifications

JML supports method specifications with a variety of clauses, of which the most important ones are:

requires a predicate that has to hold in the pre-state of a method.

ensures a predicate that has to hold in the post-state of a method, if it returns normally.

signals a predicate that has to hold in the post-state of a method, if it returns exceptionally, i.e. throws an exception.

signals_only a list of exception type, the method is allowed to throw. (runtime exceptions are always allowed.)

diverges a predicate that specifies under which condition a method is allowed to diverge, i.e. not return.

assignable a list of storage locations, the method is allowed to assign values to.

working_space, duration an expression specifying the maximum amount of heap space (the maximum number of JVM cycles) the method is allowed to use to finish its computation.

Storage locations include references to individual fields (e.g. '`this.f`'), all fields of an object (e.g. '`o.*`'), individual elements within an array (e.g. '`this.a[1]`') or multiple elements within an array (e.g. '`this.a[1..3]`' or '`this.a[*]`').

JML supports *lightweight* specification cases that do not assume a full specification and generate reasonable default values for omitted clauses and *heavyweight* specification cases that assume a full method specification to be present (see section 4.3.4 for details).

JML also supports so called *nested* specification cases that allow to group multiple specification cases into groups, where a list of `forall-` and `old` variable declarations and `requires` clauses can be declared that apply to all cases in this group.

```
normal_behaviour
  requires x != null
  {|   /* 1 */
    requires x.getValue() >= 0;
    ensures \result == x.getValue();
  also /* 2 */
    requires x.getValue() < 0;
    ensures \result >= 0;
  |}
```

Listing 2.2: Example of a nested specification case

In the above example, requires clause '`requires x.getValue() >= 0`' applies to both cases */* 1 */* and */* 2 */*.

2.1.3 Data Groups

Data groups allow to conveniently group together sets of storage locations that can then be used within `assignable` clauses. A data group named `f` is automatically generated for every field declaration `f`, initially containing only the storage location of field `f`. Other storage locations can be added to a data group by adding another data group explicitly (static data group inclusion using `in`) or by mapping dynamically a set of storage locations into the data group (using clause `maps ... into`). Example 2.3 illustrates the inclusion possibilities.

```
int f = 0; // Implicit creation of a data group f = ["this.f"]

int g = 1; // Implicit creation of a data group g = ["this.g"]
//@ in f // add data group g to data group f: f = ["this.f", "this.g"]

Object[] arr = ...;
//@ \maps arr[*] \into f; // add storage location arr[*] to data group f

// data group f = ["this.f", "this.g", "this.a[*]"]
```

Listing 2.3: Examples of data group inclusion

2.2 Coq Proof Assistant

Coq is a formal proof management system acting at the same time as an interactive theorem prover and as a full-fledged functional programming language[1]. Coq is based on a framework called ‘Calculus of Inductive Constructions’ which is turn is based on constructive logic. As such, according to the Curry-Howard correspondence, statements can be seen as types and proofs as programs. Following this idea, Coq makes no difference between statements and types or proofs and programs. Consequently, proofs are functional programs and checking whether a proof of a statement is correct is equivalent to checking whether a program satisfies a given type.

```
Section Program.
  (* Identity function on natural numbers *)
  Definition id_nat : nat → nat := fun (n:nat) ⇒ n.
End Program.

Section Proof.
  Variables P : Prop.
  (* Proof of P ⇒ P (written P → P in Coq),
     i.e. the identity on proofs of proposition P *)
  Definition P_imp_P : P → P := fun (p:P) ⇒ p.
End Proof.
```

Listing 2.4: Programs vs. Proofs

Coq offers a very strong type system with dependent types (called dependent products) that allows to express predicate logic formulae as well: As an example, `∀ P:Prop, P → P` states that `P_imp_P` holds for every proposition `P` and is at the same time the type of all programs proving this fact.

[1]Its syntax is similar to those of languages from the ML family

Using inductive definitions, the other logical connectives are embedded in this system as well: `Inductive and (A B : Prop) : Prop := conj : A → B → A ∧ B.`
That is, a proof of `and A B` (written `A ∧ B`) can be constructed from proofs `a:A` and `b:B` (`a` is a proof of `A`, `b` a proof of `B`) using constructor `conj`: `conj a b : (A ∧ B)`.

Coq offers various proof tactics and (semi-)automatic decision procedure algorithms for some (semi-)decidable logics. Furthermore it offers a standard library implementing the most common functional data structures (lists and maps) as well as formalisations of natural numbers, integer numbers, rational numbers and real numbers.

Coq is well-suited for programming language formalisations due to its close link between (functional) programming and proving as well as its flexibility in supporting additional syntactic notations. Further desirable features include the presence of a relatively small certification "kernel" to check proofs, as well as the ability to extract certified (i.e. verified) functional programs [2] out of Coq program developments.

2.2.1 Module System

This section gives a short introduction to the module system of Coq. As the formalisation makes extensive use of this mechanism, it is necessary to understand its basic concepts. It is described in detail in Coq'Art[1] and the Coq Proof Assistant Reference Manual [5].

The module system augments Coq with two concepts: *name spaces* and *signatures*.

Name Spaces

In Coq, every declaration is part of a module. Top-level declarations in a file `F` are part of an implicit module `F`. Every module defines its own name space. Within the same name space `M`, a declaration with name `f` can be accessed by using the unqualified name `f`, but from outside of the module, the qualified name `M.f` has to be used. This is true unless module `M` is explicitly imported to also allow unqualified access to names declared in `M`.

Signatures

A signature describes the *interface* of a module. A module `M` implements signature `S`, denoted `M : S`, if every declaration with name `x` and type `t` is also declared in `M` with the same name `x` and the same type `t`. A client should only rely on the types and functions declared in the signature of a module.

Signatures are well-suited to specify abstract data types. Thereby a signature `S` declares an abstract type `t` and functions that operate on type `t`. The desired behaviour is then specified as a set of axioms that are also part of the signature. A module implementing signature `S` then not only provides an implementation of type `t` and its functions but can also proves the axioms stated in `S`.

```
Module Type STACK_TYPE.
  Parameter t : Type.
  Parameter new  : t.
  Parameter push : t → nat → t.
```

[2]Objective Caml or Haskell programs

```
  Parameter top  : t → option nat.

  Axiom push_new : ∀ n:nat, top (push new n) = Some n.
End STACK_TYPE.

Module STACK : STACK_TYPE.
  Definition t := list nat.
  Definition new : t := nil.
  Definition push (stack:t) (n:nat) : t := cons n stack.
  Definition top : t → option nat := head (A := nat).

  Lemma push_new : ∀ n:nat, top (push new n) = Some n.
  Proof.
    auto.
  Qed.
End STACK.
```

Listing 2.5: Example of an abstract data type and its implementation.
Type `option` [3] is used to specify partial functions, that is, undefined results (`None`).

A signature `S` that depends on functionality declared in another signature `S'` can *declare* a module `M'` with signature `S'` and access from `M'` the functions declared in `S'`. A module `M` implementing `S` then has to *define* an existing module `M' : S'`.

A module `M` that depends on functionality declared in another signature `S'` can either use a module `M'` with signature `S'` directly or declare itself as a parametric module with parameter `M' : S'`.

Chapter 3

Project Overview

3.1 Overview

Extensions to the formalisation of JML and Java in Coq as done by this thesis divide into four parts.

1. Translation frontend
2. Extended syntax superset definition
3. Extended syntax rewriting
4. Basic syntax extensions/adaptations

The frontend translates a given JML-annotated Java program into the extended syntax embedding of JML and Java in Coq. This embedding is called a *deep embedding* since a program is represented as a syntax tree with newly defined data types for the different syntactical constructs.

As an example, a JML expression '`\fresh v1, v2`' is translated by the frontend into the extended syntax embedding '`Fresh [(var v1); (var v2)]`' [1], where `Fresh : list Expr` → `Expr` is a constructor of type `Expr`, the deep embedding type for expressions. `var : Var` → `Expr` is another constructor of this type. In the given example, the extended syntax and the basic syntax data type do not differ, but in other cases the two data types may be distinct (for example in the case of method specifications).

An extended syntax rewriting could [2] syntactically rewrite '`\fresh v1, v2`' as '`\fresh v1 && \fresh v2`'. This would result in a basic syntax deep embedding '`InfixOp ConditionalAnd (Fresh [var v1]) (Fresh [var v2])`', where `InfixOp ConditionalAnd : Expr` → `Expr` → `Expr` is the basic syntax pendant to the `&&`-connective.

The definition of the semantics of JML, which is not part of this thesis, would then translate this `Expr` value into a higher-order logic formula, a *shallow embedding* of JML in Coq: '`EvalFresh ( InfixOp ConditionalAnd (Fresh [var v1]) (Fresh [var v2]) )`', where `EvalFresh` is defined as

[1] We assume that `v1` and `v2` are variable declarations of type `Var`.

[2] although it doesn't...

```
Definition EvalFresh (locations : list Expr) : option value :=
  Some ( Bool (
    ∀ loc,
      In loc (map (EvalFreshExpression EvalExpression h fr) locations) →
      Heap.typeof (PreHeap fr) loc = None)).
```

Listing 3.1: Definition of `EvalFresh`

'`EvalFresh [var v1]`' is thus equal to '`Some (Bool True)`', if for location `loc`, that corresponds to the evaluation of expression '`var v1`', the `PreHeap` does not contain a a value at this storage location `loc`.

3.1.1 The Different Components

As seen in the introductory example, the translation frontend translated a given JML-annotated Java program into the extended syntax embedding. The frontend thereby generates for each input class a corresponding Coq output file. The frontend is the only component implemented in Java.

The extended syntax superset defines additional syntactical constructs to the syntax, most notably for the different forms of specification cases and a type for modifiers.

The extended syntax rewritings consists of the rewriting of loop annotations, quantified expressions with multiple variables, implicit invariants due to non_null constraints and the desugaring of method specifications.

The basic syntax extensions amount to the reorganisation of the existing interface (block and statement data types, loop annotations and type specifications), the addition of new data types (data groups) and the implementation of the abstract data types defined in the basic syntax interface. Of all abstract data type implementations, the implementation of the block type (block/compound statements) is of most interest. It is separately discussed in section 4.1.5.

3.2 Implementation Overview

The following section presents the architecture of the JML and Java formalisation from an implementation point of view.

Figure 3.1 illustrates the dependencies between the different Coq modules defined. The convention in this figure is that signatures are written in capital letters, whereas modules are written in normal letters.

PROGRAM This signature (file `JMLProgram.v`) declares the abstract data types of the basic syntax interface.

PROGRAM_PLUS This signature (file `JMLProgramPlus.v`) declares the abstract data types of the extended syntax interface (within subsignature `METHODSPEC_PLUS`). Signature `PROGRAM_-PLUS` also declares a module of type `PROGRAM`, the `PROGRAM_PLUS` signature thus contains a superset of the definitions of the `PROGRAM` signature.

Full2Basic This module (file `JMLFull2Basic.v`) declares several signatures (incl. `METHODSPEC_RE-WRITINGS_TYPE`) that declare the different rewriting functions.

Make This module (file `JMLProgramPlusImpl.v`) implements the `PROGRAM` signature and the `METHODSPEC_PLUS` signature.

MakePlus This module (file `JMLProgramPlusImpl.v`) implements the `PROGRAM_PLUS` signature (trivially by including module `Make`).

Full2BasicImpl This module (file `JMLFull2BasicImpl.v`) contains submodules that implement the signatures declared in the `Full2Basic` module. This file thus contains the definitions of the rewriting functions.

ExpressionNotations This module (file `JMLExpressionNotations.v`) define notation shorthands for expressions.

Notations This module (file `JMLNotations.v`) define the rest of the notation shorthands (for statements, method specification cases, etc.).

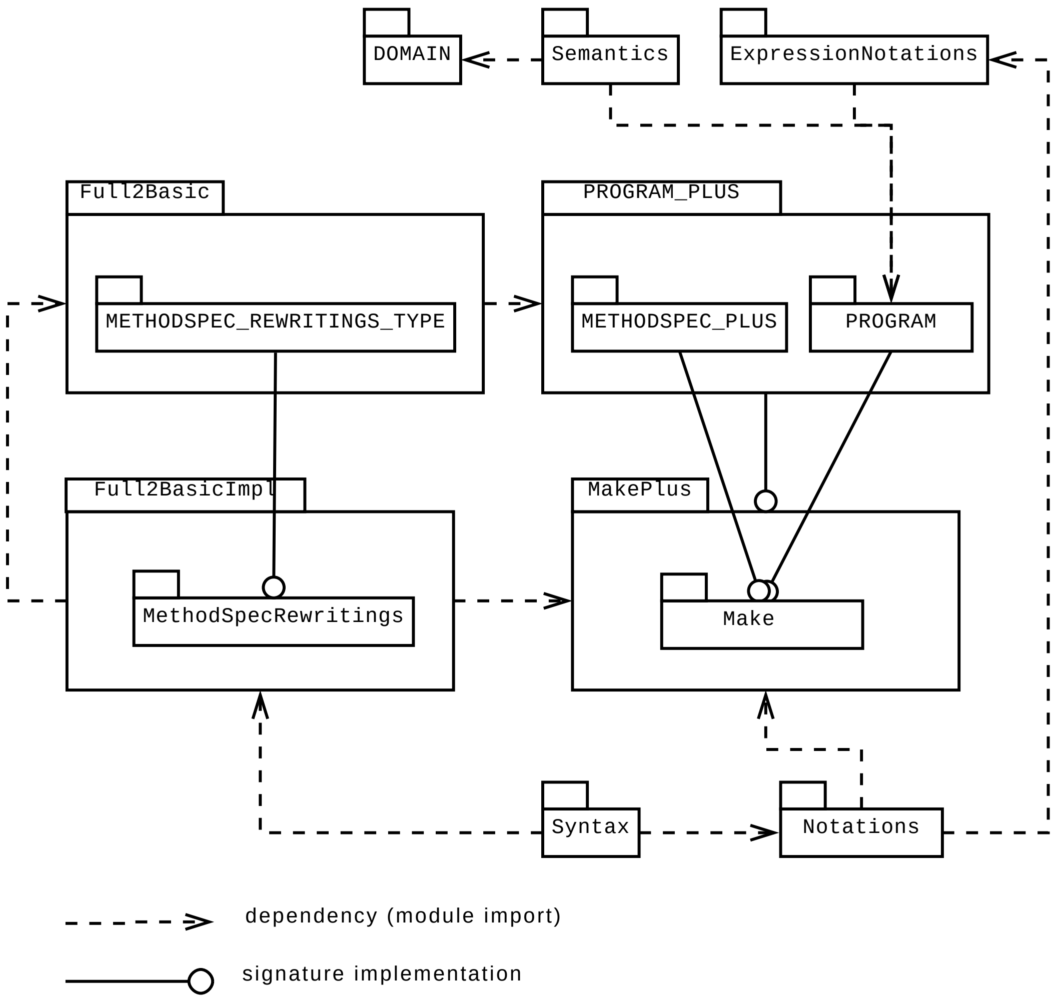
Syntax This module (file `JMLSyntax.v`) gathers together all syntax modules, and is included in the output files, generated by the translation frontend.

DOMAIN This signature (file `JMLDomain.v`) declares the data types and functions of the semantic domain (formalisation of heap, etc.).

Semantics This module (file `JMLSemantics.v`) defines the semantic functions of JML.

Notice that modules `Full2Basic`, `ExpressionNotations` and `Semantics` are implementation independent, they only depend on the interface signatures `PROGRAM` or `PROGRAM_PLUS`. Modules `Notations` and `Syntax` in contrast is dependent on the signature implementation modules `Make/MakePlus` and `Full2BasicImpl`.

Figure 3.1: Implementation overview



Chapter 4

Formalisation Details

4.1 Basic Syntax

The part of the Coq formalisation of Java and JML called 'Basic Syntax' contains a formalisation of the syntax of Java and a part of the syntax of JML. The Basic Syntax formalises all JML constructs that are not mere syntactic sugar.

In the following, we give a short presentation of the basic syntax interface. Its definition was not part of this thesis. The rest of the chapter discusses adaptations and extensions that were made to the basic syntax as part of this thesis. The chapter ends with some comments on the implementation of abstract data types declared in the basic syntax interface.

4.1.1 Basic Syntax Interface

Most syntactic constructs are formalised as abstract data types, declaring a type `t` and functions operating on type `t`. As an example, signature `METHODSIGNATURE_TYPE` declares functions `name`, `parameters` and `result` that operate on the declared type `METHODSIGNATURE.t` and give access to the name, the list of parameters and the result type of the given method signature.

```
Module Type METHODSIGNATURE_TYPE.
  Parameter t : Type.
  Parameter name       : t → ShortMethodName.
  Parameter parameters : t → list Param.
  Parameter result     : t → option type.
End METHODSIGNATURE_TYPE.
```

Listing 4.1: Example abstract data type `METHODSIGNATURE_TYPE`

Notable exceptions to this pattern of defining syntactic constructs as abstract data types are *expressions* and *statements*. For these constructs, the basic syntax declares `inductive` data types. This allows convenient pattern matching on values of these types, a mechanism that is not supported for abstract data types. Pattern matching is syntactically convenient as it allows to directly refer to children in the different cases. Furthermore, the type checker checks that a case analysis is exhaustive, meaning all possible cases are covered.

```
(* inductive data type approach ... *)
match (type stmt) with
| Compound block ⇒ process1 block
| WhileStmt test body ⇒ process2 test body
| ...

(* ... vs. abstract data type approach *)
if isCompound stmt
  then process1 (block stmt)
else if isWhileStmt stmt
  then process2 (test stmt) (body stmt)
else
  ...
```

Listing 4.2: Illustration of the two approaches to statements: Definition as an inductive data type vs. definition as an abstract data type.

Types are formalised as an inductive data type `type`. A distinction is made between primitive types (`BOOLEAN`, `BYTE`, `SHORT` and `INT`) and reference types (array types and class/interface types). Type `char` is not supported.

```
Inductive type : Set :=
  | ReferenceType : refType → type
  | PrimitiveType : primitiveType → type
with refType : Set :=
  | ArrayType   : type → utsModifier → refType
  | EntityType  : EntityName → utsModifier → refType
with  primitiveType : Set :=
  | BOOLEAN | BYTE | SHORT | INT.
```

Listing 4.3: Formalisation of types

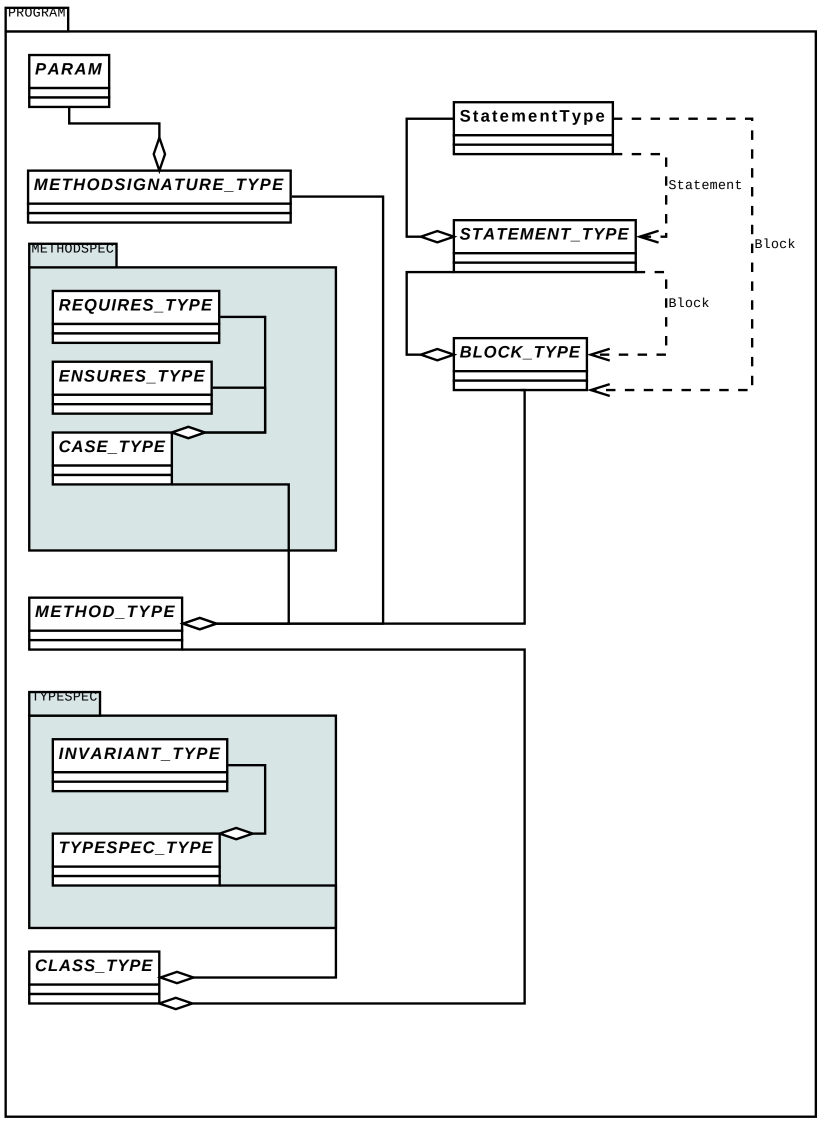
The interface to the formalisation of the basic syntax of JML and Java is given as a single signature `PROGRAM`. Figure 4.1 shows the most important components of signature `PROGRAM`.

JML

JML expressions (and Java expressions as well) are formalised as part of the inductive `Expr` data type. For instance $a \iff b$ is represented by constructor `InfixPredOp : Operator → Expr → Expr → Expr` and operator `Equivalence`. Quantifiers, like the universal quantifier are formalised by constructor `Quantification`, where `Quantification q v r e` corresponds to a quantifier of type `q` (`Forall`, `Exists`, `Max`, `Min`, `NumOf`, `Product` or `Sum`) declaring variable `v` and having range `r` (optional) and quantifier expression `e`. Notice that the basic syntax only supports quantifications with a single variable.

JML statements are formalised as part of the inductive `StatementType` data type. For instance `assume(_redundantly)` is represented by constructor `LocalAssumption`, where `LocalAssumption e l r` corresponds to an assumption `e` with an optional label `l`, either redundant or non-redundant (`r`).

Figure 4.1: Components of signature PROGRAM



```
Quantification : Quantifier → Var → option Expr → Expr → Expr
LocalAssumption : Expr → option Expr → bool → StatementType [1]
```

Listing 4.4: `Expr` and `StatementType` constructor signatures

A JML method specification case is formalised in the basic syntax by abstract data type `SPECIFICATION_CASE_TYPE`.

```
Module Type SPECIFICATION_CASE_TYPE.
  Parameter t : Type.
  Parameter visibility    : t → Visibility.

  Parameter forallVarDecl : t → FORALL_VAR_DECL.t.
  Parameter oldVarDecl    : t → OLD_VAR_DECL.t.

  Parameter requires      : t → REQUIRES.t.
  Parameter ensures       : t → ENSURES.t.
  Parameter signals       : t → SIGNALS.t.
  Parameter signalsOnly   : t → SIGNALS_ONLY.t.
  Parameter diverges      : t → DIVERGES.t.
  Parameter when          : t → WHEN.t.
  Parameter assignable    : t → ASSIGNABLE.t.
  Parameter accessible    : t → ACCESSIBLE.t.
  Parameter callable      : t → CALLABLE.t.
  Parameter measuredBy    : t → MEASURED_BY.t.
  Parameter captures      : t → CAPTURES.t.
  Parameter workingSpace  : t → WORKING_SPACE.t.
  Parameter duration      : t → DURATION.t.

  Parameter requiresRedundantly      : t → REQUIRES.t.
  (* ... all other redundant clauses ... *)

  Parameter isRedundant    : t → bool.
  Parameter isCodeContract : t → bool.
End SPECIFICATION_CASE_TYPE.
```

Listing 4.5: Specification Case Type

Notice that type `SPECIFICATION_CASE.t` supports only a very basic kind of specification case: namely a specification case, where every kind of clause is present exactly once. For the support of more sophisticated specification cases, see section 4.2. `visibility` is equal to `Private`, `Protected`, `Package` or `Public`. Every kind of clause is represented by its own data type. As an example, type `ENSURES.t` represents an ensures clause. Selector function `ENSURES.pred : t → optional Expr` gives access to the expression of the ensures clause. The types of the other clause kinds and its selector functions are defined similarly.

Storage locations are formalised as additional constructors to the `Expr` inductive data type:

- `Nothing : Expr` (‘`\nothing`’), `Everything : Expr` (‘`\everything`’)
- `field : FieldSignature → option Expr (* target *) → Expr` (e.g. ‘`this.f`’)

[1]The actual type of LocalAssumption is `∀ (S:Type) (B:Type), Expr → option Expr → bool → Statement-Type S B`. For the discussion of the formalisation of statements and blocks, see section 4.1.3

- `fieldAll : Expr (* target *) → Expr` (e.g. "`this.*`")
- `array : Expr (* target *) → Expr (* index *) → Expr` (e.g. '`a[1]`')
- `arrayAll : Expr (* target *) → Expr` (e.g. '`a[*]`')
- `arrayRange : Expr (* target *) → Expr (* from *) → Expr (* to *) → Expr` (e.g. '`a[1..3]`')

4.1.2 Issues with the Existing Bicolano Interface

The formalisation of the syntax of Java source code is realised as an extension to the formalisation of the syntax of Java byte code as part of the 5.1.1 project. Our goal has been to reuse the parts of the Coq formalisation that are common to both Java source code and Java bytecode such that eventually, our changes could be reintegrated into Bicolano. Unfortunately, the Bicolano interface makes "incorrect" usage of the module system that leads to problems in the implementation of the basic syntax interface. The problem is illustrated by means of the following excerpt from the basic syntax:

```
Module Type PROGRAM. (* Problem 1 *)
  Parameter Param : Set. (* Problem 2 *)
  Module Type PARAM_TYPE.
    Parameter isFinal : Param → bool.
    ...
  End PARAM_TYPE.
End PROGRAM.
```

The first problem is that `PROGRAM` is incorrectly declared as a module *type* (signature). A module `P` implementing `PROGRAM` has to declare signature `PARAM_TYPE`, too. This is clearly undesired as it leads to code duplication and hence to maintenance problems. Furthermore, `P` is not obliged to implement `PARAM_TYPE`, which obviously was not the intent.

The second problem, is that abstract type `Param` is declared outside of the corresponding signature `PARAM_TYPE`. The problem is seen on a hypothetical implementation of `P1`:

```
Module P1 : PROGRAM.
  Module PARAM.
    Inductive t : Set := Build_t { isFinal : bool; ... }
  End PARAM.
  Definition Param := PARAM.t.
  Module Type PARAM_TYPE.
    Parameter isFinal : Param → bool.
    ...
  End PARAM_TYPE.
End P1.
```

Implementation `P1` is correct in the sense that it provides an implementation of the `PARAM_TYPE` signature, but this fact is not checked! Due to problem 2, the declaration of Param outside of `PARAM_TYPE`, the order of the declarations `PARAM` module – `Param` type – `PARAM_TYPE` signature is fixed, and `PARAM` cannot be declared as `PARAM : PARAM_TYPE`. This second problem can be lessened by defining an additional module `PARAM' : PARAM_TYPE := PARAM`. So in a roundabout way, checking the adherence of module `PARAM` to signature `PARAM_TYPE` is still possible. Nevertheless, the nice solution would be to avoid problems problem 1 and problem 2 altogether and provide the interface and implementation as follows:

```
Module PROGRAM.
  Module Type PARAM_TYPE.
    Parameter t : Set.
    Parameter isFinal : t → bool.
    ...
  End PARAM_TYPE.
End PROGRAM.
Module P.
  Module PARAM : PARAM_TYPE.
    Inductive t : Set := Build_t {
      isFinal : bool;
      ...
    }
  End PARAM.
End P.
```

4.1.3 Interface Adaptations

Statements and Blocks

The statement and block constructs make up the trickiest part of the syntax formalisation. The two constructs are mutually dependent: the statement construct depends on the block statement and vice versa, since a block is a kind of statement, and a block consists of a sequence of statements.

Two requirements constrain the possibilities to formalise these constructs: First, a statement must give access to its label (if present) and its program counter. Second, a block must give access to its local variable declarations and its ordered sequence of statements. Adhering to these constraints, the original interface contained two abstract data types for statements and blocks:

```
Module Type STATEMENT_TYPE.
  Parameter label : Statement → Label.
  Parameter pc    : Statement → PC.
End STATEMENT_TYPE.

Module Type BLOCK_TYPE.
  Parameter localVariables : Block → list Var.
  Parameter first : Block → PC
  Parameter last  : Block → PC
  Parameter statementAt : Block → PC → Statement.
  Parameter next : Block → PC → option PC.
End BLOCK_TYPE.
```

As discussed in the basic syntax interface introduction, for convenient use, the different kind of statements are best described as an inductive type. Thus the original interface contained the declaration:

```
Inductive StatementType : Type :=
  | Compound  (block:Block)
  | WhileLoop (test:Expr) (body:Statement)
  | ...
```

As such, this inductive type does not enable to implement the functions `label` and `pc`. We identified two possible solutions to integrate an inductive statement type with the required functions from the abstract `STATEMENT_TYPE`.

1. Addition of a field `info` to each constructor of the inductive `StatementType` type. An implementation can then store the `label` and `pc` information in this field. This makes it straightforward to implement ADT `STATEMENT_TYPE` by realising the abstract type `Statement` as `StatementType`.
2. Addition of a parameter `kind` to the ADT enabling access to the statement kind. An implementation can then realise the abstract type `Statement` by defining a new record type Statement that stores the `label`, `pc` and `kind` information.

```
(* "info" solution... *)
Inductive StatementType (Info:Type) : Type :=
    | Compound (info:Info) (block:Block)
    | WhileLoop (test:Expr) (body:Statement)
    | ...
Parameter info : ∀ Info:Type,  StatementType Info → Info.

(* ... vs. "kind" solution *)
Module Type STATEMENT_TYPE.
  Parameter label : Statement → Label.
  Parameter pc    : Statement → PC.
  Parameter kind  : Statement → StatementType.
End STATEMENT_TYPE.
```

Listing 4.6: Two possible statement formalisations

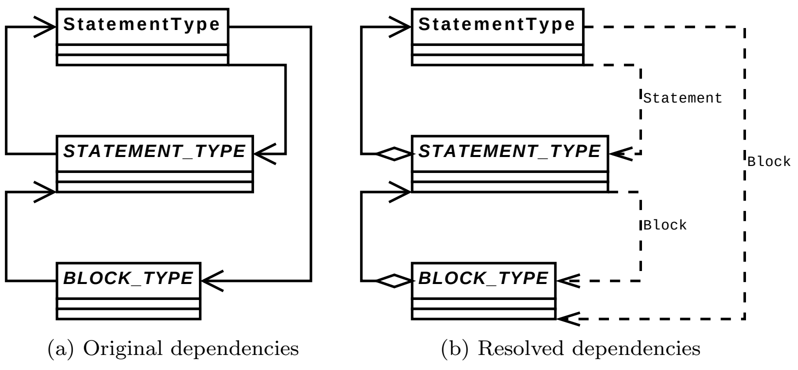
We decided for the second solution in favour of the first one, mainly because of the additional info field. This info field should not be visible to clients. It is part of the solution domain (implementation) and does not correspond to a Java construct (problem domain).

The dependency graph (see figure 4.2) of `StatementType`, `STATEMENT_TYPE` and `BLOCK_TYPE` make some problems apparent. It nicely shows the mutual dependence between the types in the form of cycles. This makes it impossible to define the types separately. The most natural way to break these cycles is to make `StatementType` type parametric in terms of Statement and Block types. Hence the `StatementType` type can be declared without knowledge of the other types. Only when the type is used, the parameters have to be set.
However, one cycle still remains: the mutual dependencies between `STATEMENT_TYPE` and `BLOCK_TYPE`. `STATEMENT_TYPE` depends on `BLOCK_TYPE` as the Block type parameter of field kind has to be instantiated. This problem is solved by declaring the Block type as an additional abstract type parameter within `STATEMENT_TYPE`.

The final interface to the three types is depicted in the following listing:

```
Inductive StatementType (Statement:Type) (Block:Type) : Type :=
```

Figure 4.2: Dependencies between statements and blocks



(a) Original dependencies (b) Resolved dependencies

```
  | Compound (block:Block)
  | WhileLoop (test:Expr) (body:Statement)
  | IfStmt (test:Expr) (_then : Statement) (_else : option Statement)
  | ...

Module Type STATEMENT_TYPE.
  Parameter t : Type.
  Parameter b : Type.
  Parameter label : t → Label.
  Parameter pc    : t → PC.
  Parameter kind  : t → StatementType t b.
End STATEMENT_TYPE.

Module Type BLOCK_TYPE.
  Declare Module STATEMENT : STATEMENT_TYPE.
  Parameter t : Type.
  Parameter localVariables : t → list Var.
  Parameter first : t → PC
  Parameter last  : t → PC
  Parameter statementAt : t → PC → STATEMENT.t.
  Parameter next : t → PC → option PC.
End BLOCK_TYPE.
```

Listing 4.7: Final form of statement and block formalisation

Block Type Specification

The functions declared in signature `BLOCK_TYPE` are non-trivial: they do not merely return fields of a record as it is the case for signature `METHODSIGNATURE_TYPE`. Thus it is safer to specify their meaning formally. For this reason, we added several axioms to signature `BLOCK_TYPE`, that have to be proven as lemmas in an implementation. This greatly increases the trustworthiness of an implementation. For our list-based implementation, some details are given in section 4.1.5.

Function `statementAt` is specified by axiom
`statementAt_def : ∀ t pc s, statementAt t pc = Some s → STATEMENT.pc s = pc.`
The program counter of the statement returned by `statementAt t pc` must be equal to the given `pc` value.

For the specification of the other functions, a useful helper function is introduced:
`elem : t → PC → bool`, were `elem t pc` states for a given block `t` and program counter `pc` whether `pc` corresponds to a program counter of a statement contained in block `t`. This is specified formally with the help of `statementAt`:

```
elem_def : ∀ t pc,
  elem t pc = true → ∃ s, statementAt t pc = Some s ∧ STATEMENT.pc s = pc.
```

`next` can now easily be specified:
`next_elem : ∀ t pc1 pc2, next t pc1 = Some pc2 → elem t pc2 = true.`
That is, for every block `t` and program counter `pc1`, if the result of next is a program counter `pc2`, `pc2` must be the program counter of a statement contained in `t` (i.e. `elem t pc2 = true`).

The specification of `first` is also straightforward to understand:
`first_elem : ∀ t pc, first t = Some pc → elem t pc = true.`
The same must be true for `last`:
`last_elem : ∀ t pc, last t = Some pc → elem t pc = true.`

Using the current interface, it can unfortunately not be stated that the program counter found in `first t` corresponds to the first statement of `t`. But it can be stated that the program counter found in `last t` indeed corresponds to the last statement:
`last_def : ∀ b pc, last b = Some pc → next b pc = None.`
That is, `next t` gives no further program counter, if the program counter found in `last t` is given as argument.

```
Axiom statementAt_def : ∀ t pc s,
  statementAt t pc = Some s → STATEMENT.pc s = pc.
Axiom elem_def : ∀ t pc,
  elem t pc = true → ∃ s, statementAt t pc = Some s ∧ STATEMENT.pc s = pc.
Axiom next_elem  : ∀ t pc1 pc2, next t pc1 = Some pc2 → elem t pc2 = true.
Axiom first_elem : ∀ t pc, first t = Some pc → elem t pc = true.
Axiom last_elem  : ∀ t pc, last t = Some pc → elem t pc = true.
Axiom last_def   : ∀ b pc, last b = Some pc → next b pc = None.
```

Listing 4.8: Axiomatisation of signature `BLOCK_TYPE`

Loop Annotations

In the initial design, loop annotations (loop invariants and loop variants) were realised as particular kind of statements, independent of the loop they precede. We dropped this approach in favour of loop statements that have an additional loop annotation field. The new loop annotation type is depicted in listing 4.9. Function `expression` gives access to a single loop invariant expression that is a rewrite of all given invariants and variants.
We think that this new approach is superior for the following reasons:

1. It follows more closely the syntax of (annotated) loops described in the JML Reference Manual, section 12.2 [10].
2. From the viewpoint of a loop statement, it gives direct access to the annotations of this loop.
3. It conveniently allows the extended syntax rewriting of loop annotations to be formulated *locally*, that is only in terms of the loop annotation type.

With the old approach, to achieve both 2. and 3., implementations would have had to consider all statements within the enclosing block.

```
Module Type LOOP_ANNOTATION_TYPE.
  Parameter t : Type.

  Parameter expression : t → optional Expr.
  Parameter invariants : t → list Expr.
  Parameter variants   : t → list Expr.
  Parameter expressionRedundantly : t → optional Expr.
  Parameter invariantsRedundantly : t → list Expr.
  Parameter variantsRedundantly   : t → list Expr.
End LOOP_ANNOTATION_TYPE.
```

Listing 4.9: Loop annotation formalisation

Data Groups

The initial syntax definition did not contain a data type for data groups. The basic syntax interface now contains signature `DATA_GROUP_TYPE` that formalise data group inclusion clauses.

```
Module Type DATA_GROUP_TYPE.
  Parameter t : Type.
  Parameter isDynamic   : t → bool.
  Parameter expression  : t → Expr. (* only valid if isDynamic == true *)
  Parameter dataGroups  : t → list FieldSignature.
  Parameter isRedundant : t → bool.
End DATA_GROUP_TYPE.
```

Listing 4.10: Basic syntax data type for data groups

Values of implementation type `DATA_GROUP.t` are linked to (model-) fields as follows: Signatures `FIELD_TYPE` and `MODELFIELD_TYPE` declare a function `dataGroupInclusions : Field -> list DATA_GROUP.t`.

This is illustrated on example 2.3:

```
int f = 0; // implicit creation of a data group f = ["this.f"]

int g = 1; // implicit creation of a data group g = ["this.g"]
//@ in f // add data group g to data group f: f = ["this.f", "this.g"] (dg1)

Object[] arr = ...;//
//@ \maps arr[*] \into f; // add storage location arr[*] to data group f (dg2)

// data group f = ["this.f", "this.g", "this.a[*]"]
```

Assuming the existence of constructor
```
DATA_GROUP.Build_t
  : bool (* isDynamic *) → Expr → list FieldSignature
  → bool (* isRedundant *) → DATA_GROUP.t,
```
the above example translates to:

```
Definition dg1 :=
  DATA_GROUP.Build_t
    false      (* isDynamic *)
    null%jml   (* irrelevant *)
    f_sig      (* field signature of f *)
    false.     (* isRedundant *)

Definition dg2 :=
  DATA_GROUP.Build_t
    true              (* isDynamic *)
    "Tr[this.arr[*]" (* maps expression *)
    f_sig             (* field signature of f *)
    false.            (* isRedundant *)
```

`dataGroupInclusions` f_{decl} would then contain `DATA_GROUP.t` values `dg1` and `dg2`.

Type Specifications

In the initial design the type specification clauses, such as invariants or history constraints, were realised as abstract data types with “uplinks” to the class or interface they specify. In contrast, the type itself did not contain a reference to its type specification clauses. This is somewhat opposite to the rest of the syntax formalisation. All other constructs are modelled with references to their children but not to the parent. It follows that all constructs are organised in a tree that closely resembles the abstract syntax tree defined by the grammar of Java and JML. The advantage of this organisation is that all constructs that logically belong to a construct can be reached from it be traversing the tree.

We decided to adapt the type specification clause data types in order to be more consistent with the rest of the syntax. We removed the “uplinks” from the clause types and instead added a reference to a ‘type spec’ data type to the class and interface data types. The ‘type spec’ data type is merely a conglomerate that holds references to the declared clauses. We added this additional level of indirection for two reasons: To avoid cluttering the interface of the class and interface data types and to have a similar design for type specification clauses and for method specification clauses. The `SPECIFICATION_CASE_TYPE` data type of the latter corresponds to the `TYPESPEC_TYPE` type of the former.

```
Module Type TYPESPEC_TYPE.
  Parameter t : Type.
  Parameter invariant    : t → list INVARIANT.t.
  Parameter constraint   : t → list CONSTRAINT.t.
  Parameter represents   : t → list REPRESENTS.t.
  Parameter initially    : t → list INITIALLY.t.
  Parameter axiom        : t → list AXIOM.t.
  Parameter readable_if  : t → list READABLE_IF.t.
  Parameter writable_if  : t → list WRITABLE_IF.t.
  Parameter monitors_for : t → list MONITORS_FOR.t.
End TYPESPEC_TYPE.

Module Type INVARIANT_TYPE.
  Parameter t : Type.
  Parameter pred       : t → Expr.
  Parameter visibility  : t → Visibility.
  Parameter isStatic    : t → bool.
  Parameter isRedundant : t → bool.
End INVARIANT_TYPE.
```

Listing 4.11: Formalisation of type specification clauses and invariants

The other kinds of type specification clauses are formalised similarly.

4.1.4 Notations

It has not yet been showed how a program or parts of a program are represented in the data types declared in the basic syntax. This section begins by showing, how expressions and statements are represented. The representation of other parts of a program is shown in section .

The data types defined for statements and blocks have already been presented in section 4.1.3. In the following, the existence of a module `STATEMENT` implementing abstract type `STATEMENT_TYPE` and a module `BLOCK` implementing `BLOCK_TYPE` is assumed, as well as constructors
`BLOCK.Build_t : list STATEMENT.t → BLOCK.t` and
`STATEMENT.Build_t : PC → option Label → StatementType t b → STATEMENT.t` that create values of the corresponding types.

The type defined for expressions is a simple inductive type (only parts used in the following are shown):

```
Inductive Expr : Type  :=
  | ...
  | literal (l : Literal)
  | var (var : Var)
  | AssignOp (op : Operator) (left : Expr) (right : Expr)
  | InfixOp (op : Operator) (left : Expr) (right : Expr)
  | InfixPredOp (op : Operator) (left : Expr) (right : Expr)
  | ...
```

The representation of expressions and statements is shown by means of the following example:

```
expr1 = i%2 == 0;
stmt1 = sumEven += i;
stmt2 = sumOdd += i;

if (i%2 == 0) {
  sumEven += i;
} else {
  sumOdd += i;
}
```

It turns out that with the previously presented functions and types the definition of these examples is very cumbersome:

```
(* In the following, we assume that variables x, I, sumEven and sumOdd are
   previously defined and the type parameters for the StatementType
   Constructors already fixed. *)

expr1 := InfixOp Equal (InfixOp Mod (var x)
  (literal (IntLiteral 2)))
  (literal (IntLiteral 0)).

stmt1 :=
  STATEMENT.Build_t 6%Z None (ExprStmt (AssignOp Add (var sumEven) (var i))).
stmt2 :=
  STATEMENT.Build_t 7%Z None (ExprStmt (AssignOp Add (var sumOdd) (var i))).

STATEMENT.Build_t 5%Z None
  (IfStmt expr1
    (STATEMENT.Build_t 5%Z None (Compound (BLOCK.Build_t [] [stmt1])))
    (Some (STATEMENT.Build_t 7%Z None (Compound (BLOCK.Build_t [] [stmt2]))) )
  ).
```

Fortunately, Coq supports what it calls *notations*. By defining a notation, one can extend the parser of Coq to recognise a new syntactic construct and to replace it by user-defined code. Notations are very flexible and allow to specify their precedence level and associativity. Notations can even be defined for constructs with recursive patterns.

This embedding of Java and JML in Coq makes heavy use of notations. It defines notations or other shorthands for most syntactic constructs. The goal has been to make a piece of source code expressed in the embedding resemble as much as possible the original source code. Here is how the above examples are expressed using notations:

```
expr1 := (var i) mod (int 2) == (int 0)
stmt1 := stmt ((var sumEven) += (var i))
stmt2 := stmt ((var sumOdd) += (var i))
5% :> ife (expr1) {: 5 :>>
        [6%N :> stmt1]
      :} else_ {: 7 :>>
        [7%N :> stmt2]
      :}
```

Expression Notations

Notations for expressions have already been defined for the basic syntax and are therefore not part of this thesis. Most notations define operators that are identical to the original Java/JML operators. For example, an expression '`e1 <==> e2`' is written as '`e1 <==> e2`' in the embedding, too. This is defined as a notation:
`Notation "x <==> y" := (InfixPredOp Equivalence x y) (at level 96) : jml_scope.`
Some operators conflicted with existing Coq notations. In these cases, the operator in the embedding is suffixed with a prime character (`'`). For example, an expression '`e1 && e2`' is written as '`e1 &&' e2`' in the embedding.

Statement Notations

As seen in the notation examples above, statements are built using the `STATEMENT.Build_t` constructor that is given a pc, an optional label and the statement type as arguments. As statements are so frequent and can be nested, it is nice to have a notation that replaces the use of the constructor. The notation defined for this case is: '`pc :> stmt`' For example, '`30%N :> nop`' [2] defines a new no operation (skip) statement at program counter 30.

There are also notations and shorthand functions that simplify the use of the different `StatementType` constructors. A switch `StatementType` is defined with the help of notation '`switch expr {{` *cases* `default` *dflt* `}}`'. All other statement types (except Compound) are created with simple shorthand functions. The following two tricks were used to beautify their syntax:

Dummy arguments Dummy arguments are added to functions to mimic Java syntax. The `ife` (if-then-else) function expects an argument of type `Else_` whose constructor `else_` is merely used to make a call '`ife (test) stmt1 else_ stmt2`' look more like the Java equivalent '`if (test) stmt1 else stmt2`'. (functions `ife`, `do_`, `try`)

Dummy functions Similar to dummy arguments. As an example, function '`catch v s := (v,s)`' is merely used to make a list of catch clauses look more like the Java equivalent: '`[(v1, stmt1), (v2, stmt2)]`' vs. '`[catch v1 stmt1; catch v2 stmt2]`' (functions `cases`, `catch`)

Block statements (Compound statement type) are treated differently. Once could define a notation like '`{ ... }`' for the Compound statement type and then create a block statement using '`pc :> { ... }`'. However, this is ugly since block statements often need to be passed to other statement creation functions such as `ife`. Then, '`pc :> { ... }`' would have to be surrounded by additional parenthesis to allow correct parsing.
Instead, a new notation '`{:` *pc* `:>>` *statements* `:}`' is defined that directly creates a new block statement (instead of a `StatementType` value).

Listing 4.1 gives an overview of `StatementType` notations that differ from the Java notations:

JML statements like `assert` or `assume` have notations identical to normal JML syntax.

Loop Annotation Notations

As table 4.1 already indicated, loop notations require a loop annotation value as argument. Section 4.1.3 has already presented the `LOOP_ANNOTATION_TYPE` signature. The implementation also provides notations to create values of type `LOOP_ANNOTATION.t` that implements signature `LOOP_ANNOTATION_TYPE`. This is best seen with an example:

```
/*@ maintaining i <= 10;
  @ decreasing 10-i;
  @*/
for (int i = 0; i < 10; i++) ;
```

The loop annotations '`maintaining i <= 10`' and '`decreasing 10-i`' are created using the `maintaining` / `decreasing` notations and summarised into a single `LOOP_ANNOTATION.t` value using helper function `loop_annotation`:

[2] `%N` opens the scope of natural numbers; `30` is thus interpreted as a natural number.

Statement kind	`StatementType` value/`STATEMENT.t` value [a]
Empty Statement “`;`”	`nop`
Labelled statement “*label*`:` *statement*”	`(`*pc*`,` *label*`) :->` *statement* [a]
Expression statement “*e*`;`”	`stmt` *e*
Block: “`{` *block* `}`	`{:` pc_{block} `:>>` *block* `:}`” [a]
Labelled block “*label*`: {` *block* `}`”	`{: (`pc_{block}`,` *label*`) :->>` *block* `:}` [a]
“`for (`*init*`;` *test*`;` *step*`)` *body*”	`for_` la^b *init test step body*
“`while (`*test*`)` *body*”	`while` la^b *test body*
“`do` *body* `while (`*test*`)`”	`do_` la^b *body* `while_` *test*
“`if (`*test*`)` *then-part*”	`if_ (`*test*`)` *then-part*
“`if (`*test*`)` *then-part else-part*”	`ife (`*test*`)` *then-part else-part*
“`switch (`*test*`) {` `case` L_1`: case` L_2`:` $body_1$ `...` `default:` $body_n$ `}`”	`switch (`*test*`) {{` `cases [`L_1`;` L_2`]` $body_1$ `...` `default` $body_n$ `}}` [c]
“`try {` *try-block* `} catch` *P*1 `{` $block_1$ `} ...` `} finally {` *finally-block* `}`”	`tryF {:` $pc_{try\text{-}block}$ `:>>` *try − block* `:} catch` *P*1 `{:` pc_{block_1} `:>>` $block_1$ `:} ...` `:} finally {:` $pc_{finally\text{-}block}$ `:>>` *finally-block* `:}` [d]
“`break`”/“`break` *label*”	`break`/`breakL` *label*
“`continue`”/“`continue` *label*”	`continue`/`continueL` *label*
“`return`”/“`return` *expr*”	`return`/`returnE` *expr*
Java assertion “`assert` *expr*”	`jassert` *expr*
Variable declaration “*modifiers type name*”	[e]

[a]These notations directly create `STATEMENT.t` values instead of `StatementType` values.
[b]Loop annotations are discussed separately.
[c]If default case is missing, just omit default $body_n$ in translation.
[d]If *finally-block* is missing, use `try` instead of `tryF` and omit part `finally {: ... :}`
[e]Variable declaration statements are discussed in section 4.2.5.

Table 4.1: `StatementType` notations

```
loop_annotation loop-label
  [maintaining (var i) <= (int 10);
   decreasing 10 - (var i)].
```

Declaration function `loop_annotation` and the need for a *loop-label* are explained in section 4.3.1.

The whole for-loop translates to:

```
(* We assume that i c.t. the variable declaration of i
   and i_name to the name declaration of i. *)

(loop-label, pc_for) :-> for_
  loop_annotation loop-label [
    maintaining (var i) <= (int 10);
    decreasing 10 - (var i)
  ]
  [22 :> var_decl_stmt int_t i_name (Some (int 0))]
  ((var i) < (int 10))
  [(var i)++]
  nop
```

4.1.5 Implementation Remarks

The implementation of the abstract data types defined in the basic syntax interface is based on lists. That is, all collections of values are implemented as lists. It would be very well possible to use more sophisticated data structures like maps in the implementation. We think that this is not necessary since the formalisation will most often be used for proving. In this context, runtime efficiency of the operations given in the abstract data types is irrelevant. In case the formalisation is used to evaluate concrete Java programs, we assume that classes and methods have reasonable size.

Most abstract data type implementations are straightforward: they are implemented by means of a newly defined record with appropriate fields.

```
Module Type PARAM_TYPE.
  Parameter signature : Param → ParamSignature.
  Parameter isFinal   : Param → bool.
End PARAM_TYPE.

Module PARAM : PARAM_TYPE.
  Record t : Set := Build_t {
    signature : ParamSignature;
    isFinal   : bool
  }.
d PARAM.
```

Listing 4.12: Example abstract data type implementation of the signature for parameters. Compared to the stack example from the background section 2.5, this implementation is based on a record type and exploits the fact that Coq automatically generates a selector functions for every field within a record.

Block Type Implementation

Recall the BLOCK_TYPE signature:

```
Module Type BLOCK_TYPE.
  Declare Module STATEMENT : STATEMENT_TYPE.
  Parameter t : Type.
  Parameter localVariables : t → list Var.
  Parameter first : t → PC
  Parameter last  : t → PC
  Parameter statementAt : t → PC → STATEMENT.t.
  Parameter next : t → PC → option PC.

  Parameter elem : t → PC → bool.

  Axiom statementAt_def : ∀ t pc s,
    statementAt t pc = Some s → STATEMENT.pc s = pc.
  Axiom elem_def : ∀ t pc,
    elem t pc = true → ∃ s, statementAt t pc = Some s ∧ STATEMENT.pc s = pc.
  Axiom next_elem  : ∀ t pc1 pc2, next t pc1 = Some pc2 → elem t pc2 = true.
  Axiom first_elem : ∀ t pc, first t = Some pc → elem t pc = true.
  Axiom last_elem  : ∀ t pc, last t = Some pc → elem t pc = true.
  Axiom last_def   : ∀ b pc, last b = Some pc → next b pc = None.
End BLOCK_TYPE.
```

Listing 4.13: BLOCK_TYPE signature

Signature BLOCK_TYPE is realised as a record of two lists: a list of local variables and a list of statements. Function localVariables is thus trivially defined. For convenience, the implementation also defines the other selector function statements : BLOCK.t → list Statement. Based on function statements the other functions are defined. The implementation uses so called *strong specifications*.

The idea is that the result of a strongly-specified function not only contains the final value of the desired computation, but also a proof that the computed result is correct. As an example, a strongly-specified square root function might look as follows:
sqrt : ∀ x:nat, x >= 0 → {y:nat | y*y = x}
That is, function sqrt takes as arguments a natural number x, a *proof* that x is larger or equal to zero and results in a natural number y that satisfied the equality '$y^2 = x$'.
'{y:nat | y*y = x}' is syntactic sugar for 'sig (fun y:nat ⇒ y*y = x)', an instance of type sig:

```
Inductive sig (A : Type) (P : A → Prop) : Type :=
  | exist : ∀ y : A, P y → sig P.
```

Listing 4.14: Signature type sig

A value of type 'sig nat (fun y:nat ⇒ y*y = x)' is given using constructor exist: 'exist y P', where in the example, y would correspond to the computed square root number and P to a proof of '$y^2 = x$'.

The advantage of using strongly-specified functions is that the specifications are always at hand (no separate lemmas necessary) and that strongly-specified functions can better be composed into larger strongly-specified functions. As such, it allows for a modular creation of large programs and proofs.

Continuing the square root example, it may be desirable to also allow negative numbers as input arguments. In this case, the result cannot be expressed as '{y:nat | y*y = x}' since a proof '$y^2 = x$' does not exist for negative numbers x. In this case, the result can be built using another helper type:
`sqrt2 : ∀ x:nat, {y:nat | y*y = x}+(x < 0)`
This is read as: Either the result is a value y together with the proof that y is the square root of x (as previously) or the result is a proof that x is negative. [3]

The whole implementation of the BLOCK_TYPE signature makes heavy use of the above concepts by defining strongly-specified counterparts to the functions declared in the signature and defining the signature versions in terms of the strongly-specified versions. This section now focuses on two particularly interesting lemmas, last_elem and last_def, that demonstrate the use of strong specifications but also the need to define new helper lemmas and even a new axiom.

`last_elem : ∀ t pc, last t = Some pc → elem t pc = true` is the simpler of the two lemmas. Its proof is based on a strongly-specified variant of last,
`lastS : lastS (l:list A) : {x:A | Last x l}+{l=nil}` (last is defined in terms of lastS). This allows to derive from 'last t = Some pc' that there exists a statement s satisfying the predicate 'Last s (statements t)' (s is the last statement of list 'statements t') and 'STATEMENT.pc s = pc'.

From a second lemma
`elem_def_inv : ∀ t s0, In s0 (statements t) → elem t (STATEMENT.pc s0) = true`, and knowing that element x of 'Last x (statements s)' is part of 'statements s' (Lemma Last_In), it follows that 'elem t pc' is indeed equal to true.

```
(* sum A B is written as A ∧ B *)
Inductive sum (A B : Type) : Type :=
  | inl : A → A + B
  | inr : B → A + B.

Inductive Last (x:A) : list A → Prop :=
  | Last_base : Last x (x::nil) (* base case *)
  | Last_step : ∀ (y:A) (l:list A),
      Last x l → Last x (y::l). (* step case *)
```

Listing 4.15: sum type and definition of Last

The proof of `last_def : ∀ b pc, last b = Some pc → next b pc = None` is more involved. Again, as in the proof of last_elem it follows that there exists a statement s, satisfying 'Last s (statements t)' and 'STATEMENT.pc s = pc'. Then the fact that next is defined in terms of a helper function suffixS is exploited:

```
suffixS : ∀ (stmts:list STATEMENT.t) (pc:PC),
  {l:list STATEMENT.t &{s | Suffix l stmts ∧ head l = Some s
                              ∧ STATEMENT.pc s = pc}}+{AllS (neq_PC_Stmt pc) stmts}
```

In plain English: for all lists of statements stmts and program counters pc, the result of suffixS stmts pc is

- either a suffix list l of list stmts where the statement in front of list l has a program counter

[3] '{y:nat | y*y = x}+(x < 0)' is syntactic sugar for 'sum {y:nat | y*y = x} (x < 0)' the widespread functional sum type.

equal to `pc` or

- all statements in list `stmts` have a program counter different from `pc`.

Knowing that statement `s` is in the list and its program counter is equal to `pc`, only the first case applies and it follows that there exists such a suffix list `x0`. Now comes the tricky part: It is now known that the program counter of `s` is `pc` and that the program counter of the first element of list `x0` is also `pc`. As no two distinct statements can have the same program counter (to be discussed), it follows that the first statements of list `x0` is indeed `s`. Using some arguments, it also follows that `s` is the last element of `x0`. (From `lastS` already follows '`Last s (statements t)`', and since `x0` is a suffix of '`statements t`', '`Last s x0`' is true as well. A last lemma (`Last_head_singleton`) ensures that the list `x0` is a singleton list (since both the first and the last element of `x0` is `x` and no two distinct statements may have identical program counters, this must indeed be the case). From this the desired result '`next b pc = None`' follows, since the implementation of `next` selects the second element of suffix list `x0` (that does not exist) and thus selects `None`.

```
(* The result of next is the pc of the second element of
   suffix list x0 = suffixS (statements block) pc0. *)
Definition next (block:t) (pc0:PC) : option PC :=
  match suffixS (statements block) pc0 with
  | inleft (existT l _) ⇒
      match tail l with
      | n::_ ⇒ Some (STATEMENT.pc n)
      | nil ⇒ None
      end
  | inright _ ⇒ None
  end.
```

Listing 4.16: Implementation of `next`

An important part of the proof is that two statements with the same program counters can be *unified*. This is ensured by lemma `PC_unique_impl` which in turn is not possible to be proven without a new axiom `Stms_unique`. `Stmts_unique` states that every program counter of a statement in b is unique.

```
Lemma PC_unique_impl : ∀ x y b,
      In x (statements b) →
      In y (statements b) →
      pc x = pc y →
      x = y.

(* For every suffix list x::s it holds that the program counter of list head x
   is different from the program counter of any element in tail s. *)
Axiom Stmts_unique : ∀ b x ss,
      Suffix (x::ss) (statements b) →
      AllS (neq_PC_Stmt (pc x)) ss
```

Listing 4.17: A lemma and an axiom for the *uniqueness* of program counters.

4.2 Extended Syntax

The part of the Coq formalisation of Java and JML that we call ‘Extended Syntax’ contains a formalisation of all syntactic constructs of Java and JML that are rewritable in terms of basic syntax formalisation constructs. For Java, this amounts to the addition of declaration functions that generate default values omitted in the declaration. For JML, the extended syntax adds support for specification cases in all its forms. Furthermore it provides rewritings that desugar specification cases, loop annotations and non_null modifiers.

4.2.1 Extended Syntax Interface

First and foremost, the extended syntax defines new data types to support method specification cases in all its sugared forms:

- support for omitted and multiple method specification clauses per specification case
- support for nested specification cases
- support for lightweight-, normal behaviour- and exceptional behaviour specification cases

A *(full) specification case* is one that may use the above named specification case features, whereas it is called a *basic specification case* when these features are not allowed. In the formalisation, a basic specification case is a value of type `SPECIFICATION_CASE.t`. The type for full specification cases will be defined below.

4.2.2 Omitted- and Multiple Method Specification Clauses

In a basic specification case every method specification clause has to be declared exactly one. This syntax is thus first extended to support omitted clauses and multiple clauses. The existing `SPECIFICATION_CASE_TYPE` signature could have been adapted to provide a (possibly empty) list of clauses for every kind of clause but the extended syntax takes another approach: It provides *one* algebraic data type `MethodSpecClause` for all clause kinds. This gives more flexibility in omitting clauses and writing them in arbitrary order by having a single list of `MethodSpecClause` values per specification case.

```
Inductive MethodSpecClause : Type :=
  | ensuresC (redundant : bool) (pred : optional Expr)
  | signalsC (redundant : bool) (pair : Var * optional Expr)
  | signalsOnlyC (redundant : bool) (types : list type)
  | divergesC    (redundant : bool) (pred : optional Expr)
  | whenC (redundant : bool) (pred : optional Expr)
  | assignableC (redundant : bool) (storeRefs : optional (list Expr))
  | accessibleC (redundant : bool) (storeRefs : optional (list Expr))
  | callableC   (redundant : bool) (storeRefs : optional CallableList)
  | measuredByC (redundant : bool) (pair : optional Expr * option Expr)
  | capturesC   (redundant : bool) (storeRefs : optional (list Expr))
  | workingSpaceC (redundant : bool) (pair : optional Expr * option Expr)
  | durationC (redundant : bool) (pair : optional Expr * option Expr).
```

Listing 4.18: Extended syntax data type for method specification clauses

4.2.3 Nested Specification Cases

JML allows specification cases to be nested. Section 9.4 of the JML Reference Manual [10] defines the syntax of a nested specification case (generic-spec-case):

```
generic-spec-case ::=
    [ spec-var-decls ] spec-header [ generic-spec-body ]
  | [ spec-var-decls ] generic-spec-body

generic-spec-body ::=
    simple-spec-body
  | {| generic-spec-case-seq |}

generic-spec-case-seq ::= generic-spec-case [ also generic-spec-case ] ...

spec-header ::= requires-clause [ requires-clause ] ...

simple-spec-body ::= simple-spec-body-clause [ simple-spec-body-clause ] ...
```

Listing 4.19: Nested specification case grammar

A nested specification case contains a possibly empty list of variable declarations, an optional specification header (one ore more requires clauses) and an optional specification body that is either one or more body clauses (all clauses except requires) or one or more nested specification cases. It is not allowed that a specification case has neither header nor body.

Grammar non-terminal `generic-spec-case` is formalised by means of the new signature `GENERIC-_SPEC_CASE_TYPE`:

```
Module Type GENERIC_SPEC_CASE_TYPE.
  Parameter t : Type.
  Parameter forallVarDecl : t → list FORALL_VAR_DECL.t.
  Parameter oldVarDecl    : t → list OLD_VAR_DECL.t.
  Parameter specHeader    : t → list SpecHeader.
  Parameter genericBody   : t → (list MethodSpecClause) + (list t).
End GENERIC_SPEC_CASE_TYPE.
```

Listing 4.20: Nested specification case type

That is, a value of type `GENERIC_SPEC_CASE.t` implementing signature `GENERIC_SPEC_CASE_TYPE` gives access to the list of variable declarations, the list of specification headers and a body that is either a list of clauses or a list of nested `GENERIC_SPEC_CASE.t` values. [4] Notice that this abstract data type does not enforce the above consistency criterion that requires the existence of either a header or a body. This criterion has to be enforced by a *well-formedness predicate* that guarantees that all values have the desired form. Furthermore an empty body could be represented as both '`inl nil`' or '`inr nil`', but this does not bother too much. The abstract data type in the present form has been chosen to be as simple as possible in order to allow short and clear definitions of the rewriting functions.
Types `FORALL_VAR_DECL.t` and `OLD_VAR_DECL.t` are part of the basic syntax interface:

[4] sum type `A+B` has already been presented in section 4.1.5.

```
Module Type FORALL_VAR_DECL_TYPE.
  Parameter t : Type.
  Parameter vars : t → list Var.
End FORALL_VAR_DECL_TYPE.

Module Type OLD_VAR_DECL_TYPE.
  Parameter t : Type.
  Parameter varDecls : t → list (Var * Expr).
End OLD_VAR_DECL_TYPE.
```

Listing 4.21: Basic Syntax signatures for ∀- and `old` variable declarations.

Type `SpecHeader` is defined to be a "pendant" to the `spec-header` grammar non-terminal and allows for a convenient definition of the `requires`/ `requires_redundantly` notations as explained later.

```
Inductive SpecHeader : Type :=
  | requiresSH (redundant : bool) (pred : optionalSame Expr).
```

Listing 4.22: Extended syntax data type for a `spec-header`.

Lightweight-, Normal Behaviour- and Exceptional Behaviour Specification Cases

The different specification cases only differ with respect to the allowed clauses (normal- and exceptional behaviour) [5], disallowed visibility (lightweight) [6] and the semantics of omitted clauses. But the (possibly nested) specification clauses are described by `generic-spec-case` for all kind of specification cases. For simplicity, we therefore define one single data type for all kinds of cases. Again, a well-formedness predicate ensures that for every kind of specification case only allowed clauses are given. The semantics of omitted clauses is discussed in section 4.3.4.

```
Module Type FULL_SPEC_CASE_TYPE.
  Parameter t : Type.
  Parameter specCaseType    : t → SpecCaseType.
  Parameter visibility      : t → option Visibility.
  Parameter isRedundant     : t → bool.
  Parameter isCodeContract  : t → bool.
  Parameter genericSpecCase : t → GENERIC_SPEC_CASE.t.
End FULL_SPEC_CASE_TYPE.
```

Listing 4.23: Extended syntax data type for the different kind of specification cases.

The specification case *type* (→ `lightweight`, `behaviour`, `normal_behaviour` or `exceptional-_behaviour`) is used to distinguish between the different case types.

[5]Normal behaviour cases do not allow `signals` clauses, whereas exceptional behaviour cases do not allow `ensures` clauses.

[6]A lightweight case inherits the visibility of the enclosing method.

Declarations with Omitted Modifiers

The basic syntax does not provide a type for modifiers. Instead, every modifier is realised as a separate boolean selector on the corresponding type. As an example, type `PARAM.t` provides a selector `isFinal : Param → bool` that corresponds to the final modifier.

```
Module Type PARAM_TYPE.
  Parameter signature  : Param → ParamSignature.
  Parameter isFinal    : Param → bool.
  Parameter isNullable : Param → bool.
End PARAM_TYPE.
```

Thus, the basic syntax requires an entity declaration to provide explicit values for all modifiers. As an example, a parameter declaration '`final int x`' would translate to '`PARAM.Build_t (PARAMSIGNATURE.Build_t x_name int_t) true false`'. (`true` for `isFinal` and `false` for `isNullable`.) The full syntax changes these cumbersome declarations into forms that resemble more closely the original syntax: '`paramDecl [final] int_t` x_{name}', where `int_t` corresponds to the integer type and x_{name} defines the translation of the name of `x`. As seen in the example, the new declaration function can be given a list of modifiers, defined through the new Modifier type.

```
Inductive Modifier : Set :=
  | public   | protected | private
  | abstract | static    | final
  | native
  | spec_public | spec_protected
  | model | ghost
  | pure
  | instance
  | helper
  | non_null | nullable
  | monitored
  | uninitialized
  | code
  | implicit_constructor  7 .
```

Listing 4.24: Extended syntax data type for modifiers.

Declaration function `paramDecl : list Modifier → type → ParamName → PARAM.t` not only supports omitted modifiers (in the example, modifier `nullable` was omitted, and is thus implicitly `non_null`) but also flattens the nested application of constructor `PARAMSIGNATURE.Build_t`: its arguments are given to `paramDecl` directly.

The full syntax provides declaration functions in the style of `paramDecl` for all kinds of declarations. The values for the different kind of modifiers are taken from the JML Reference Manual, Appendix B [10].

[7]modifier `implicit_constructor` is unofficial and private tois formalisation. It tags the implicit zero-argument default constructor that needs special treatment inthod specification desugarings. (see section 4.3.4)

4.2.4 Rewritings

The rewritings of extended syntax constructs in terms of basic syntax constructs are initiated in the implementations of the declaration functions. For better understanding, we differentiate between *structure-transforming* and *structure-preserving* and between *local* and *non-local* rewritings.

A structure-transforming rewriting is one that transforms a full syntax data type into a basic syntax data type, whereas a structure-preserving rewriting operates on a basic syntax type only.

A local rewriting is one that has no information about the program other then the data type it is rewriting. A non-local rewriting in contrast has information about other data types from the program as well.

The declaration function for loop annotations,
`loopAnnotationDecl`
  `: Label → list (LoopAnnotationTag * Expr) → LOOP_ANNOTATION.t`,
is a structure-transforming local rewriting. It is structure-transforming since it transforms a list of loop invariants and loop variants into a single loop invariant value (`LOOP_ANNOTATION.expression`).

A second structure-transforming local rewriting is `rewriteFullQuantifier`:
`rewriteFullQuantifier : FullQuantifier → Expr`. Type FullQuantifier provides a single constructor
`FullQuantification`
  `: Quantifier → list Var → optional Expr`
  `→ Expr → FullQuantifier`,
the “pendent” to the `Quantification` constructor of type `Expr`. `rewriteFullQuantifier (FullQuantification q vs r e)` thereby rewrites a quantified expression with possibly multiple variables `vs` in terms of a simple basic syntax expression.

Another rewriting is `rewriteInvariants`:
`rewriteInvariants : list INVARIANT.t → ENTITY.t → list INVARIANT.t`.
`rewriteInvariant invs e` results in a list of invariants, where for every `non_null` field `f` of reference type a new invariant ‘`f != null`’ is appended to `invs`. This rewriting is structure-preserving and non-local (it requires information about fields of type `e`).

The most complex of the extended syntax rewritings is the desugaring of method specification cases:
`rewriteFullSpecification`
  `: list FULL_SPEC_CASE.t → Method → ENTITY.t → list SpecificationCase`.
`rewriteFullSpecification scs m e` desugars the list of full specification cases `scs` of method `m` declared in type `t` into a list of basic specification cases. `rewriteFullSpecification` is a non-local, structure-transforming rewriting (transformation of `list FULL_SPEC_CASE.t` into `list SpecificationCase`).

Fortunately, both non-local rewritings `rewriteInvariants` and `rewriteFullSpecification` only require parent node values as additional information: In the case of `rewriteInvariants` type `e` is the direct parent of the list of invariants `invs`. In the case of `rewriteFullSpecification`, type `e` is the parent of method `m`, and `m` is the parent of the list of specification cases `scs`. Thus, these rewritings can be done with a simple traversal of the syntax tree of the corresponding type declaration `e`. This AST traversal is initiated in the common implementation `typeDecl` of the corresponding type declaration functions `classDecl` and `interfaceDecl`.

4.2.5 Notations

In section 4.1.4 on basic syntax notations, the notation `loop_annotation` for loop annotations has not yet fully been explained. The reason being that this notation is actually just a synonym for `loopAnnotationDecl` [8].

The use of quantified expressions using either constructor `Quantification` of `Expr` or `rewrite-FullQuantifier/FullQuantification` is very cumbersome and ugly to read. The formalisation thus defines notations that make translated quantified expressions appear very much like the original forms:

```
Notation "forall vs ; r ; e"  :=
  (rewriteFullQuantifier (FullQuantification Forall vs r e))
  (at level 0, e at level 200) : jml_scope.
```

Thus, a quantified expression '`forall vs; r; e`' is in fact just a shorthand for the more cumbersome '`rewriteFullQuantifier (FullQuantification Forall vs r e)`'.

A variable declaration statement, such as '`final int` i `= 0`' is represented in the formalisation using constructor `varDeclStmt : VAR.t` $\rightarrow$ `Expr` $\rightarrow$ `StatementType`. Two notations `var-_decl_stmt` and `var_decl_stmtM` [9] are defined for this case to hide the use of the `VAR.t` and `VARSIGNATURE.t` constructors as has been discussed for parameters in section 4.2.3. The notation uses the declaration function `varDecl : list Modifier` $\rightarrow$ `type` $\rightarrow$ `VarName` $\rightarrow$ `VAR.t` to make the declaration appear like the original form. The example '`final int` i `= 0`' translates to '`var_decl_stmtM [final] int_t` i_{name} `(int 0)`'.

The full syntax also defines notations to make the creation of specification cases more natural. Using notations, a specification case

```
normal_behaviour
  requires x != null
  {|
    requires x.getValue() >= 0;
    ensures \result == x.getValue();
  also
    requires x.getValue() < 0;
    ensures \result >= 0;
  |}
```

Listing 4.25: Example of a `normal_behaviour` specification case. (JML notation)

can be represented as

```
spec_case [public] normal_behaviour (
  nested_case nil nil
  [requires (: (var x_decl) != null :)]
  {|
    simple_case nil nil
      [requires (: (callT (var x_decl) getValue []) >= (int 0) :)]
      [ensures  (: \result == (callT (var x_decl) getValue []) :)]
  ;
    simple_case nil nil
```

[8] `Definition loop_annotation := loopAnnotationDecl.`

[9] `var_decl_stmtM` is the variant with modifier list argument, `var_decl_stmt` the variant without modifier list argument.

```
      [requires (: (callT (var x_decl) getValue []) < (int 0) :)]
      [ensures  (: \result >= (int 0) :)]
  |}%jml_nb
).
```

Listing 4.26: Example of a `normal_behaviour` specification case. (Extended syntax notation)

Notation `spec_case` is a synonym to declaration function `methodSpecDecl false : list Modifier → SpecCaseType → GENERIC_SPEC_CASE.t → FULL_SPEC_CASE.t`. Notation `simple_case` just hides an application of `GENERIC_SPEC_CASE.Build_t`, were the `genericBody` argument is fixed to be a simple body argument:

```
Definition simple_case fvd ovd sh (simple:list MethodSpecClause) :=
  GENERIC_SPEC_CASE.Build_t fvd ovd sh (inl _ simple).
```

Notation `nested_case` is defined analogously with argument `(nested:list GENERIC_SPEC_CASE.t)` instead of argument `simple` and the use of `inr` instead of `inl`.

Notation '`{|` *generic-spec-case*$_1$`;` `...`; *generic-spec-case*$_n$ `|}%jml`' makes the declaration appear very JML-like but is in fact just a synonym to the normal list notation '`[`*generic-spec-case*$_1$`;` `...`; *generic-spec-case*$_n$`]`'.

4.2.6 Implementation Remarks

This section discusses a technical problem that emerge as a consequence to the formalisation of method specification clauses as a single inductive data type:

```
Inductive MethodSpecClause : Type :=
  | ensuresC (redundant : bool) (pred : optional Expr)
  | signalsC (redundant : bool) (pair : Var * optional Expr)
  | signalsOnlyC (redundant : bool) (types : list type)
  | ...
```

A list of such clauses can be extracted from a simple specification case using selector function: `GENERIC_SPEC_CASE.genericBody : GENERIC_SPEC_CASE.t → (list MethodSpecClause) + (list t)`. However, most rewriting function implementations are only interested in a list of clauses of *one* kind, say ensures clauses. In such a case, it would be easiest to define a function of the form `extractEnsuresClauses : list MethodSpecClause → list (optional Expr)` (for ensures clauses) to extract the desired clauses. Unfortunately, such extraction functions would have to be defined for every clause kind separately. Even more, functions that operate on an arbitrary, but single clause kind would have to make a case analysis on the clause kind and apply the appropriate extraction function.

An example of such a function is `addDefault` that adds the default clause of a certain kind to a list of clauses, if no clause of the given kind is present yet. This function would first have to find all clauses of the given kind which is not possible with the discussed approach of extraction functions.

The solution to this problem is based on the idea of assigning *tags* to data: Every clause kind is associated a unique tag:

```
Inductive tag : Set :=
  | ensuresT
  | signalsT
  | signalsOnlyT
  | ...
```

Listing 4.27: `tag` data type for method specification clauses.

Function `addDefault` is in this way given the signature
`addDefault : tag → list MethodSpecClause → list MethodSpecClause`.
'`addDefault t l`' thus adds a default clause of tag type `t` (e.g. for an `ensures` clause: tag `ensuresT`) to clause list `l` if no such clause is present in `l`. Then, a generalised extraction function should be built on the basis of this tag type. The result of this extraction function must certainly depend on the `tag` type, it is thus a *dependent type*: `mapType : tag → Type`. In the case of method specification clauses this type is defined as

```
Definition mapType (t:tag) : Type :=
  match t with
  | ensuresT       ⇒ optional Expr
  | signalsT       ⇒ (Param * optional Expr)
  | signalsOnlyT   ⇒ list type
  | ..
```

Listing 4.28: Dependent type `mapType`; result type of the generalised extraction function.

Then the extraction function could be defined as
`extract : ∀ t:tag, list MethodSpecClause → list mapType t`, but we took a more general approach. First the extraction function is generalised to work on an arbitrary `data` type (here: `MethodSpecClause`). Then the extraction function is split into two separate functions:
`filterTag (t:tag) (l:list data) : list {d:data | t=tagOf d}` and
`mapData2Type : ∀ t:tag, list {d:data | t=tagOf d} → list (mapType t)`.
'`filterTag t l`' thereby extracts all clauses of tag type `t` resulting in a list `l` not of type '`mapType t`' but type '`{d:data | t=tagOf d}`'. Function `tagOf : data → tag` simply associates the corresponding tag to a data item (= clause kind). Result '`list {d:data | t=tagOf d}`' thus contains all data items `d` whose tag '`tagOf d`' is equal to the desired tag `t` and a proof of this fact. Applying function `mapData2Type` to the result yields a result of the desired type `list (mapType t)`. [10]

The definition of a generalised function `mapData2Type` is only possible with the help of a function `mapF : ∀ t:tag, {d:data | t=tagOf d} → mapType t`. This function is indeed implemented for method specification clauses. As an example, for arguments `ensuresT` and data item '`ensuresC false expr`', it merely results in value `expr`.

Why is this split of `extract` into `filterTag` and `mapData2Type` useful? It is useful because type '`list {d:data | t=tagOf d}`' is useful. The implementation of the rewritings, for instance, use variants of the familiar *map* and *fold* operations defined for this special list type '`list {d:data | t=tagOf d}`'.

For more details on this idea of tagged lists, see the implementation (module `TaggedList.v`, and module `Full2BasicImpl.v` for its application for method specification clauses).

[10] `extract : ∀ t:tag, list data → list mapType t` could thus be defined as
`Definition extract t l := mapData2Type t (filterTag t l).`

4.3 Rewriting Details

4.3.1 Loop Annotations

The rewriting of loop annotations, initiated by `loopAnnotationDecl label annotations` (`loopAnnotationDecl : Label → list (LoopAnnotationTag * Expr) → LOOP_ANNOTATION.t`), rewrites the variants and invariants given in the list of `annotations` as a single invariant. Each variant expression `e` is thereby rewritten as an invariant '`(0 <= e) && (e < \old(e, label))`', i.e. it is always the case that variant `e` is non-negative and that the value of `e` is strictly smaller than the old value of `e` at `label`, where `label` corresponds to the label of the loop body statement.

```
Definition rewriteVariant (lbl:Label) (e:Expr) : Expr :=
    let lit0 := literal (IntLiteral 0%Z) in
    ( (lit0 <= e) &&' (e <= \oldl e at lbl) )%jml.
```

Listing 4.29: Rewriting of loop annotations.

The single invariant (`LOOP_ANNOTATION.expression`) consists of a conjunction of all invariants and all variants rewritten by `rewriteVariant`. The same procedure is applied to all redundant variants and invariants and stored in `LOOP_ANNOTATION.expression_redundantly`.

4.3.2 Quantified Expressions with Multiple Variables

The rewriting of quantified expressions with multiple variables, initiated by '`rewriteFullQuantifier q`' (`rewriteFullQuantifier : FullQuantifier → Expr`) rewrites quantified expression `q` in terms of a nested expression of simple (single variable) quantified expressions. The rewriting is defined by:

```
Definition rewriteFullQuantifier (fq:FullQuantifier) : Expr :=
  let (q,l,range,expr) := fq in rewriteFullQuantifier_rec q l range expr.

Fixpoint rewriteFullQuantifier_rec
  (q:Quantifier) (l:list Var) (range:option Expr) (expr:Expr) {struct l} : Expr :=
  match l with
  | nil      ⇒ false'%jml (* error *)
  | (v::nil) ⇒ Quantification q v range expr
  | (v::vs)  ⇒ Quantification q v None (rewriteFullQuantifier_rec q vs range expr)
  end.
```

Listing 4.30: Rewriting of quantified expressions with multiple variables.

That is, `rewriteFullQuantifier` unpacks the `FullQuantifier` argument record and delegates the rewriting to `rewriteFullQuantifier_req` a recursively defined function (structurally decreasing in argument `expr`). Note that case `nil` of the case analysis on `l` never matches `l` since type-checking prevents quantified expressions without a declaration of at least one variable.

Example '`forall int i,j; 0 <= i && i < j && j < 10; a[i] < a[j]`' from the JML Reference Manual, section 11.4.24.1 [10] is translated into quantified expression

```
(* We assume that a c.t. the declaration of array a,
   i and j to the variable declarations of i and j,
   and $i_{name}$ and $j_{name}$ to the corresponding name declarations.
 *)

(* with use of notations: *)
forall
  [var_decl int_t $i_{name}$; var_decl int_t $j_{name}$];
  0 <= i && i < j && j < 10;
  (array a i) < (array a j)

(* without use of notations: *)
rewriteFullQuantifier Forall (
  FullQuantification
    [var_decl int_t $i_{name}$; var_decl int_t $j_{name}$]
    (0 <= i && i < j && j < 10)
    ((array a i) < (array a j))
  )
```

The rewriting of this quantified expression results in
'`forall int i; ; (forall int j; 0 <= i && i < j && j < 10; a[i] < a[j])`' which is represented as

```
Quantification (var_decl int_t $i_{name}$) None
  (Quantification (var_decl int_t $j_{name}$
    (0 <= i && i < j && j < 10)
    ((array a i) < (array a j))
```

in the formalisation.

This rewriting is done for all kinds of quantified expressions: universal and existential quantifiers `\forall` and `\exists`, generalised quantifiers `\max` and `\min`, `\product` and `\sum` and numerical quantifier `\num\_of`.

4.3.3 Implicit Invariants

Every `non_null` field (or model field) `f` (of reference type) of a type `e` defines an implicit invariant '`f != null`'. '`rewriteInvariants invs e`' adds these invariants to the list of invariants `invs` of type `e`.

```
Definition rewriteInvariants (invs:list INVARIANT.t) (e:ENTITY.t)
    : list INVARIANT.t :=
  let fieldInvs :=
    fieldNonNullInvariants (ENTITY.name e) (ENTITY.fields e) in
  let modelFieldInvs :=
    modelFieldNonNullInvariants (ENTITY .name e) (ENTITY.modelFields e) in
  invs +++ fieldInvs +++ modelFieldInvs.
```

Listing 4.31: Rewriting of implicit invariants.

`fieldNonNullInvariants : EntityName → list Field → INVARIANT.t` gathers together all additional invariants generated by `non_null` fields:

```
Definition fieldNonNullInvariant (eName:ClassName) (f:Field)
    : option INVARIANT.t :=
  if  FIELD.isNullable f
      || negb (isReferenceType (FIELDSIGNATURE.type (FIELD.signature f)))
    then None
    else let target := if FIELD.isStatic f then None else Some (this%jml) in
          let fExpr := field (eName, (FIELD.signature f)) target in
          Some (INVARIANT.Build_t
                (fExpr != null)%jml
                (FIELD.visibility f)
                (FIELD.isStatic f)
                false
              ).
```

Listing 4.32: `fieldNonNullInvariant`

‘`fieldNonNullInvariant (ENTITY.name e) f`’ results in `None` if field `f` is nullable or not of reference type, otherwise in an invariant ‘`f != null`’. Invariant ‘`f != null`’ has the same visibility as `f` and is static iff `f` is static.

4.3.4 Method Specification Cases

The desugaring of JML method specification cases in discussed in depth in technical report #00-03e by Raghavan and Leavens [14]. The desugarings that are part of this formalisation closely follow report #00-03e. To facilitate comparison, nomenclature is mostly taken from the report as well.
The author’s divide the desugaring into several steps:

3.1 Desugaring Non_Null for Arguments

3.2 Desugaring Non_Null Results

3.3 Desugaring Pure

3.4 Desugaring Empty Specifications

3.5 Desugaring Nested Specifications

3.6 Desugaring Lightweight, Normal, and Exceptional Specifications

3.7 Desugaring Inheritance and Refinement

3.8 Standardising Signals Clauses

3.9 Desugaring Multiple Clauses of the Same Kind

3.10 Make Assignable and Signals Only Clauses the Same

3.11 Desugaring Also Combinations

The desugarings as done by `rewriteFullSpecification : list FULL_SPEC_CASE.t → Method → ENTITY.t → list SpecificationCase` include desugarings 3.1 – 3.6, with the exception of not combining heavyweight specification cases of the same visibility (part of desugaring 3.6), and desugarings 3.8 and 3.9. Desugarings 3.7, 3.10 and 3.11 are not part of the rewrite from full specification cases to basic specification cases but are treated within the definition of the semantics of JML (module `JMLSemantics.v`). The individual desugaring steps are designed as transformation functions that operate on a list of `FULL_SPEC_CASE.t` values, that is, each rewriting is a function `list FULL_SPEC_CASE.t → ... → list FULL_SPEC_CASE.t`.

In the nomenclature of section 4.3, the rewritings are structure-preserving and mostly non-local, needing information from the parent class and/or parent type declaration. The advantage of this design is that the individual transformation functions could easily be applied in different orders or some transformation could be disabled for debugging purposes. More importantly, each desugaring is an independent unit, can be tested separately and could be specified and verified modularly.

The following subsections discuss the individual desugarings (transformation functions). The section ends with the discussion of the consolidation function `rewriteFullSpecification`.

The following listing remains the formalisation of full method specification cases.

```
Module Type FULL_SPEC_CASE_TYPE.
  Parameter t : Type.
  Parameter specCaseType    : t → SpecCaseType.
  Parameter visibility      : t → option Visibility.
  Parameter isRedundant     : t → bool.
  Parameter isCodeContract  : t → bool.
  Parameter genericSpecCase : t → GENERIC_SPEC_CASE.t.
End FULL_SPEC_CASE_TYPE.

Module Type GENERIC_SPEC_CASE_TYPE.
  Parameter t : Type.
  Parameter forallVarDecl : t → list FORALL_VAR_DECL.t.
  Parameter oldVarDecl    : t → list OLD_VAR_DECL.t.
  Parameter specHeader    : t → list SpecHeader.
  Parameter genericBody   : t → (list MethodSpecClause) + (list t).
End GENERIC_SPEC_CASE_TYPE.
```

Desugaring Non_Null for Arguments

This desugaring adds an additional requires clause ‘`p != null`’ for every `non_null` parameter `p` of reference type to every specification case. If no specification case is given, these additional requires clauses are added as a lightweight specification case.

```
Definition desugarNonNullArguments (specs: list FULL_SPEC_CASE.t) (m:METHOD.t)
    : list FULL_SPEC_CASE.t :=
  let reqs :=
    paramsNonNullRequires (METHODSIGNATURE.parameters (METHOD.signature m)) in
  let defaultCase  := GENERIC_SPEC_CASE.Build_t nil nil reqs (inl _ nil) in
  let defaultSpecs :=
    FULL_SPEC_CASE.Build_t lightweight None false false defaultCase in

  match reqs, specs with
  | nil,  _ ⇒ specs (* 1 *)
  | _, nil   ⇒ defaultSpecs :: nil (* 2 *)
  | _, _     ⇒ map (fun s ⇒ addHeaders s reqs) specs (* 3 *)
  end.
```

Listing 4.33: Desugaring Non_Null for Arguments

`paramsNonNullRequires : list Param → list SpecHeader` results in a list of specification headers of '`requires p != null`' clauses for a given list of parameter declarations. It is defined analogously to `fieldNonNullInvariants` discussed in section 4.3.3 on implicit invariants.

Case analysis on '`reqs, specs`' distinguished three cases:

1. If no additional requires clauses are generated (there exist no `null_null` parameter of reference type), the result is equivalent to the given `specs` argument.
2. If no specification case is given ('`specs=nil`'), the result is a lightweight specification case with the additional requires clauses.
3. In all other cases, the result is equivalent to the given `specs` where the additional requires clauses are added to the header of each specification case (`addHeaders`).

```
Definition addHeaders (specs:FULL_SPEC_CASE.t) (reqs:list SpecHeader)
    : FULL_SPEC_CASE.t :=
  let (sct,v,r,cc,case) := specs in

  FULL_SPEC_CASE.Build_t sct v r cc
    (GENERIC_SPEC_CASE.Build_t nil nil reqs (inr _ (case::nil))).
```

Listing 4.34: Definition of the `addHeaders` helper function.

Desugaring Non_Null Results

This desugaring adds an additional ensures clause '`\result != null`' to every simple body of every (non-exceptional) specification case, if the method is declared *non_null*. [11] If no specification case is given, the additional ensures clause is added as a lightweight specification case with a simple body.

```
Definition desugarNonNullResult (specs:list FULL_SPEC_CASE.t) (m:METHOD.t)
    : list FULL_SPEC_CASE.t :=
  let ens := resultNonNullEnsures m in
  let defaultCase ens'  :=
    GENERIC_SPEC_CASE.Build_t nil nil nil (inl _ (ens'::nil)) in
  let defaultSpecs ens' :=
    FULL_SPEC_CASE.Build_t lightweight None false false (defaultCase ens') in

  match ens, specs with
  | None, _ ⇒ specs (* 1 *)
  | Some ens', nil ⇒ defaultSpecs ens' :: nil (* 2 *)
  | Some ens', _   ⇒ map (fun s ⇒ addBody s ens') specs (* 3 *)
  end.
```

Listing 4.35: Desugaring Non_Null Results

[11] Exceptional specification cases cannot have ensures clauses.

`resultNonNullEnsures : Method → option MethodSpecClause` results in '`\result != null`' if the method is declared `non_null`, otherwise it results in `None`.

Again, the function is defined with a case analysis: '`match ens, specs`' distinguishes three cases:

1. If the method is declared `nullable` ('`resultNonNullEnsures m = None`'), the result is equivalent to the given `specs`.
2. If the method is declared `non_null`, but no specification case is given ('`specs=nil`'), the result is a lightweight specification case with a simple body containing an ensures clause '`\result != null`'.
3. In all other cases, the additional ensures clause is added to every simple body of every (non-exceptional) specification case (`addBody`).

Function `addBody` enables adding additional simple body clauses to a full specification case. If the given clause is an ensures clause and the case is exceptional or if the given clause is a signals or signals_only clause and the case is a normal_behaviour case, `addBody` has no effect.

```
Definition addBody (specs0:FULL_SPEC_CASE.t) (clause:MethodSpecClause)
    : FULL_SPEC_CASE.t :=
  let (sct,v,r,cc,case) := specs0 in
  let addClause (clauses:list MethodSpecClause) := (clause :: clauses) in
  let specs1 := FULL_SPEC_CASE.Build_t sct v r cc (clausesMap addClause case) in

  match sct with
  | lightweight ⇒ specs1
  | behaviour   ⇒ specs1
  | normal_behaviour ⇒
      if isSignalsClause clause then specs0 else specs1
  | exceptional_behaviour ⇒
      if isEnsuresClause clause then specs0 else specs1
  end.
```

Listing 4.36: Definition of the `addHeaders` helper function.

Desugaring Pure

This desugaring adds additional simple body clauses to every specification case, if the method is declared `pure`. The additional clauses are '`diverges false`' and, in case of a constructor '`assignable this.*`' and in case of a normal method '`assignable \nothing`'. Again, if no specification case is given, the clauses are added as a lightweight specification case with a simple body.

```
Definition desugarPure (specs:list FULL_SPEC_CASE.t) (m:METHOD.t)
    : list FULL_SPEC_CASE.t :=
  let addBody2 clause1 clause2 spec := addBody (addBody spec clause1) clause2 in
  let div := divergesC false (: false' :)%jml in
  let ass :=
    match METHOD.kind m with
    | Constructor ⇒ assignableC false (: [fieldAll this] :)%jml
    | _           ⇒ assignableC false (: [\nothing] :)%jml
    end in

  let defaultCase  :=
    GENERIC_SPEC_CASE.Build_t nil nil nil (inl _ (div :: ass :: nil)) in
  let defaultSpecs :=
    FULL_SPEC_CASE.Build_t lightweight None false false defaultCase in

  match METHOD.isPure m, specs with
  | false, _ ⇒ specs (* 1 *)
  | true, nil ⇒ defaultSpecs :: nil (* 2 *)
  | true, _ ⇒ map (addBody2 ass div) specs (* 3 *)
  end.
```

Listing 4.37: Desugaring Pure

The case analysis ‘`METHOD.isPure m, specs`’ is similar to those in `desugarNonNullArguments` and `desugarNonNullResult`.

Desugaring Empty Specifications

This desugaring adds a default specification to the given method if no (non-redundant) cases were explicitly declared with the method and no cases implicitly created as part of desugarings 3.1 – 3.3. The default specification amounts to a lightweight specification case ‘`requires \not_specified`’ for any non-override method or ‘`also requires false`’ for an override method.

As discussed in section 3.4 of TR #00-03e [14], the implicit zero-argument default constructor is treated specially in this desugaring: its default specification amounts to a lightweight specification case with an assignable clause that is “a copy of the superclass’s default constructor’s assignable clause”. *The* assignable clause of the superclass’s default constructor is equivalent to a *union* of all assignable clauses of the different specification cases. This is described in desugaring 3.10.

```
Definition desugarEmptySpecification (specs:list FULL_SPEC_CASE.t) (m:METHOD.t)
    (enc:ENTITY.t) : list FULL_SPEC_CASE.t :=
  let req :=
    match METHOD.override m with
    | None   ⇒ requiresSH false \not_specified%jml
    | Some _ ⇒ requiresSH false (: false’ :)%jml
    end in

  let defaultCase  := GENERIC_SPEC_CASE.Build_t nil nil (req::nil) (inl _ nil) in
  let defaultSpecs :=
    FULL_SPEC_CASE.Build_t lightweight None false false defaultCase in

  match METHOD.isImplicitDefaultConstructor m, isEmptySpecification specs with
  | true, _      ⇒ (implicitCtorDefaultSpecs enc) :: nil (* 1 *)
  | false, true  ⇒ defaultSpecs :: nil (* 2 *)
  | false, false ⇒ specs (* 3 *)
  end.
```

Listing 4.38: Desugaring Empty Specification

Again, the function is defined with a case analysis:
'`METHOD.isImplicitDefaultConstructor m, isEmptySpecification specs`' distinguishes three cases:

1. If the method is the implicit zero-argument default constructor (modifier `implicit_constructor` set as argument to `methodDecl`), the default specification is given by `implicitCtorDefaultSpecs`.
2. If the method has no specification, the result is equivalent to the default specification '`requires \not_specified`' or '`also requires false`'.
3. In all other cases, the result is equivalent to the given argument `specs`.

The default specification for the implicit default constructor, `implicitCtorDefaultSpecs`, is defined as:

```
Definition implicitCtorDefaultSpecs (enc:ENTITY.t) : FULL_SPEC_CASE.t :=
  let parentCtor :=
    match ENTITY.superClass_ enc with
    | Some sup ⇒ zeroArgCtor sup
    | None ⇒ None
    end in

  let parentSpecs :=
    match parentCtor with
    | Some ctor ⇒ METHOD.fullSpecs ctor
    | None ⇒ nil
    end in

  let clauses :=
    match assignableUnion parentSpecs with
    | Some ole ⇒ (assignableC false ole) :: nil
    | None ⇒ nil
    end in

  let case  := GENERIC_SPEC_CASE.Build_t nil nil nil (inl _ clauses) in
  FULL_SPEC_CASE.Build_t lightweight None false false case.
```

Listing 4.39: Default specification for the implicit constructor

`assignableUnion : list FULL_SPEC_CASE.t → option (optional (list Expr))` thereby creates the union of all assignable clauses (type of an assignable list: '`optional (list Expr)`') of the given specification case list. [12]

Two assignable clauses `a1` and `a2` are merged as follows: If `a1` or `a2` is `NotSpecified`, the result is the other clause. Otherwise the two clauses are equivalent to two lists `l1` and `l2` of locations. These are merged by `appLocations`:

```
Definition appLocations (l1 l2:list Expr) : list Expr :=
  match l1, l2 with
  | Nothing::nil, _     ⇒ l2
```

[12] `Inductive optional (A:Type) := Specified A → optional A | NotSpecified : optional A.`

```
  | Everything::nil, _ ⇒ l1
  | _, Nothing::nil    ⇒ l1
  | _, Everything::nil ⇒ l2
  | _, _ ⇒ List.app l1 l2
  end.
```

Desugaring Nested Specifications

This desugaring flattens all nested specification cases. A specification case `sc` is called *nested* if its `generic-body` is nested:
`GENERIC_SPEC_CASE.genericBody (FULL_SPEC_CASE.genericSpecCase sc))`
`  = inr _ nestedBody`.
A specification case is called *flat* if its `generic-body` is a `simple-body`:
`GENERIC_SPEC_CASE.genericBody (FULL_SPEC_CASE.genericSpecCase sc))`
`  = inl _ simpleBody`.

The desugaring is described in the technical report [14] as well as in the JML Reference Manual, section 9.6.5 [10] (Semantics of nested behavior specification cases), by means of a semi-formal example:

```
spec-var-decls
spec-header                                  spec-var-decls
{|                                           spec-header
     GenSpecCase1                            GenSpecCase1
   also                                    also
     ...              -- desugars to --      ...
   also                                    also
     GenSpecCaseN                            spec-var-decls
|}                                           spec-header
                                             GenSpecCaseN
```

Listing 4.40: Semi-formal description of the desugaring of nested specifications.

In the formalisation this amounts to:

```
Definition desugarNested (specs:list FULL_SPEC_CASE.t) : list FULL_SPEC_CASE.t :=
  let desugar1 (spec:FULL_SPEC_CASE.t) : list FULL_SPEC_CASE.t :=
    let (sct,v,r,cc,case) := spec in
    map (FULL_SPEC_CASE.Build_t sct v r cc) (flatten case) in
  flat_map desugar1 specs.

Fixpoint flatten (case:GENERIC_SPEC_CASE.t) : list GENERIC_SPEC_CASE.t :=
  let (fvd, ovd, sh, genBody) := case in

  let addDecls (flattenedCase:GENERIC_SPEC_CASE.t) :=
    let (fvd1, ovd1, sh1, simpleBody1) := flattenedCase in
      GENERIC_SPEC_CASE.Build_t
        (List.app fvd fvd1)
        (List.app ovd ovd1)
        (List.app sh sh1)
        simpleBody1 in

  let fix flattenNested (l:list GENERIC_SPEC_CASE.t)
      : list GENERIC_SPEC_CASE.t :=
    match l with
    | nil         ⇒ nil
    | (nb::nbs) ⇒ List.app (flatten nb) (flattenNested nbs)
    end in

  match genBody with
  | inl simpleBody ⇒ case :: nil
  | inr nested     ⇒ map addDecls (flattenNested nested)
  end.
```

Listing 4.41: Desugaring Nested Specifications

That is, the flattening function `flatten` is applied to the generic-spec-case of every specification case. `flatten` recursively flattens its nested generic-spec-cases and adds the variable declarations (`fvd`, `ovd`) and requires clauses (`sh`) to every recursively flattened case (`addDecls`). Finally, a full specification case is built out of every generic-spec-case in the result of `flatten`.

Desugaring Lightweight, Normal, and Exceptional Specifications

This desugaring transforms lightweight, normal- and exceptional behaviour specification cases into behaviour specification cases. For `normal_behaviour` cases this amounts to adding a clause ‘`signals (Exception) false`’ to each simple body. For `exceptional_behaviour` cases, a clause ‘`ensures false`’ is added to every simple body. For lightweight specification cases, the visibility is set to the visibility of the enclosing method and a default clause is is added for every clause kind that is missing in this case. Notice that this desugaring is required to be applied *after* the flattening process such that every specification case has a simple body only.

```
Definition desugarBehaviour (specs:list FULL_SPEC_CASE.t) (m:METHOD.t)
    : list FULL_SPEC_CASE.t :=
  let desugar1 (spec0:FULL_SPEC_CASE.t) : FULL_SPEC_CASE.t :=
    let vis :=
      match FULL_SPEC_CASE.visibility spec0, FULL_SPEC_CASE.specCaseType spec0 with
      | Some v', _          ⇒ v' (* case only possible for heavyweight case *)
      | None, lightweight ⇒ METHOD.visibility m
      | None, _             ⇒ Package (* default visibility for heavyweight case *)
      end in

      match FULL_SPEC_CASE.specCaseType spec0 with
      | lightweight ⇒
          let spec1 := setSpecCaseType spec0 behaviour in
          let spec2 := setVisibility spec1 (Some vis) in
          let spec3 := addDefaults spec2 m (lightweightDefaults spec2 m) in
          spec3
      | normal_behaviour ⇒
          let spec1 := setSpecCaseType spec0 behaviour in
          let spec2 := setVisibility spec1 (Some vis) in
          let spec3 :=
            addBody spec2 (signalsC false (Exception_e, (: false' :)%jml)) in
```

```
        spec3
    | exceptional_behaviour ⇒
        let spec1 := setSpecCaseType spec0 behaviour in
        let spec2 := setVisibility spec1 (Some vis) in
        let spec3 := addBody spec2 (ensuresC false (: false’ :)%jml) in
        spec3
    | behaviour ⇒ setVisibility spec0 (Some vis)
    end in
  map desugar1 specs.
```

Listing 4.42: Desugaring Lightweight, Normal, and Exceptional Specifications

The adding of default clauses (e.g. a default ensures clause is added to a lightweight specification case, if no ensures clause is present), is formalised by function `addDefaults : FULL_SPEC_CASE.t -> METHOD.t -> FullSpecCaseDefaults -> FULL_SPEC_CASE.t`. Function `addDefaults` draws default clauses from argument `defaults : FullSpecCaseDefaults`. `defaults` provides a function `bodyDefault` that returns the default clause for any given kind of clause (see section 4.2.6 for an explanation of type `ClauseTag`).

```
Record FullSpecCaseDefaults : Type := Build_FullSpecCaseDefaults {
    headerDefault : SpecHeader;
    bodyDefault   : ClauseTag → MethodSpecClause
  }
```

Listing 4.43: Data type for default clause values.

The formalisation provides two `FullSpecCaseDefaults` values: `lightweightDefaults` and `heavyweightDefaults`, both of which have signature `FULL_SPEC_CASE.t -> Method -> FullSpecCaseDefaults`. The use of the two arguments is explained below. The default values for the different clause kinds are given in the table 4.2 [13]:

For `signals_only`, the default clause is the list of checked exceptions declared in the methods throws clause. Unchecked exceptions listed in the throws clause do not enter this default clause. In the formalisation this is expressed through the use of function `filterCheckedExceptions`: ‘`filterCheckedExceptions (METHOD.throws m)`’. For `accessible` and `callable` clauses, it is not entirely clear from the Reference Manual if these clauses can be used in non-code contract specification cases. In any case, sections 9.9.10 and 9.9.11 only define a default value for code contract specification cases, which is `\everything` for both `accessible` and `callable` clauses. We decided to not explicitly disallow these clauses to be part of non-code contract specification cases. We use a default clause of `\not_specified` for both lightweight and heavyweight non-code contract specification cases. Considering these default values it becomes clear why both functions `lightweightDefaults` and `heavyweightDefaults` require arguments `sc:FULL_SPEC_CASE.t` and `m:Method`: Argument `sc` is required to determine whether a specification case is a code contract (`FULL_SPEC_CASE.isCodeContract`) and argument `m` to extract the list of checked exceptions.

Standardising Signals Clauses

This desugaring standardises every signals clause ‘`signals (ET n) P`’ declaring an exception variable `n` of type `ET` into a signals clause ‘`signals (Exception e) (e instanceof ET) ⇒ [(ET)e/n] P`’ declaring a signals variable `e` of type `Exception`. ‘`[(ET) e/n] P`’ denotes expres-

[13]The numbers in parenthesis give the section from the Reference Manual that specify the corresponding default value.

Clause Kind	Default for lightweight case	Default for heavyweight case
`requires`	`\not_specified` (9.9.2)	`(: true' :)` (9.9.2)
`ensures`	`\not_specified` (9.9.3)	`(: true' :)` (9.9.3)
`signals`	`(Exception_e, \not_specified)` (9.9.4)	`(Exception_e, (: true' :))` (9.9.4)
`signals_only`	see below (9.9.5)	see below (9.9.5)
`diverges`	`(: false' :)` (9.9.7)	`(: false' :)` (9.9.7)
`when`	`\not_specified` (9.9.8)	`(: true' :)` (9.9.8)
`assignable`	`\not_specified` (9.9.9)	`(: [\everything] :)` (9.9.9)
`accessible`	`\not_specified` (9.4)	`(: [\everything] :)`
`callable`	`\not_specified` (9.4)	`\not_specified` [a] / `(: EveryMethod :)` [b]
`measured_by`	`(\not_specified, None)` (9.9.12)	`(\not_specified, None)` (9.9.12)
`captures`	`\not_specified` (9.4)	`(: [\everything] :)` (9.9.13)
`working_space`	`(\not_specified, None)` (9.9.14)	`(\not_specified, None)` (9.9.14)
`duration`	`(\not_specified, None)` (9.9.15)	`(\not_specified, None)` (9.9.15)

[a]within a non-code contract specification case
[b]within a code contract specification case

Table 4.2: Default values for method specification clauses

sion `P`, where every occurrence of `n` is substituted by '`(ET) e`'. If exception type `ET` is equivalent to `Exception`, the normalisation is equivalent to '`signals (Exception e) [e/n] P`', i.e. the normalisation ensures the use of always the same signals variable `e`.

`desugarSignals : list FULL_SPEC_CASE.t → list FULL_SPEC_CASE.t` [14] applies a normalisation to every signals clause found in the given list of specification cases:

```
Definition normaliseSignals (data:{d:MethodSpecClause | signalsT = TAG.tagOf d})
    : MethodSpecClause :=
  let (par, oexpr0) := TAG.mapF signalsT data in
  let paramType := PARAMSIGNATURE.type (PARAM.signature par) in
  let oexpr1 :=
    match oexpr0 with
    | NotSpecified ⇒ NotSpecified
    | Specified expr0 ⇒
        if isExceptionType paramType
          then
            let expr1 := SIGNALS_SUBST.subst par (param Exception_e) expr0 in
            Specified expr1
          else
            let expr1 :=
              SIGNALS_SUBST.subst par (Cast paramType (param Exception_e)) expr0 in
            let expr2 :=
              (((param Exception_e) instanceof paramType) =⇒' expr1)%jml in
            Specified expr2
    end in
  signalsC false (Exception_e, oexpr1).
```

Listing 4.44: Normalisation function for signals clauses

[14]The implementation of `desugarSignals` is not shown as it used some advanced features of Coq that does not help in understanding its meaning.

‘`normaliseSignals d`’ exactly performs the standardisation explained above. The substitution function `SIGNALS_SUBST.subst` is based on a more general substitution function `expressionSubstitute : Expr → (Expr → bool) → Expr → Expr → Expr`, where ‘`expressionSubstitute x eq_x t e`’ substitutes term `x` in expression `e` by term `t`. Function `eq_x` is used to determine if a given expression is equal to substitution term `x`.

```
Module SIGNALS_SUBST.
  (* bool equality between a parameter and an expression:
    * Param_Expr_eqb p e = true iff e = (param q) and PARAM.eq_t p q
    *)
  Definition Param_Expr_eqb p e : bool :=
    match e with
    | param q ⇒ PARAM.eq_t p q
    | _ ⇒ false
    end.

  (* subst p t e
    * Substitute expression /t/ for parameter p in e.
    *)
  Definition subst p t e :=
    expressionSubstitute (param p) (Param_Expr_eqb p) t e.
End SIGNALS_SUBST.
```

Listing 4.45: Substitution function for the signals clause normalisation

Desugaring Multiple Clauses of the Same Kind

This desugaring merges multiple clauses of the same kind within a single body into a single clause. For instance, two requires clauses ‘`requires p1; requires p2`’ are merged into a single requires clause ‘`requires p1 && p2`’. The following section describes the individual mergings in Coq code. The mergings presented here are simplified to the point of only merging two clauses into one clause. The implementation generalises this to lists of clauses. Section 4.3.4 contains an example.

Forall Variable Declarations and Old Variable Declarations

```
Definition mergeForallVarDecls (c1 c2 : FORALL_VAR_DECL.t) : FORALL_VAR_DECL.t :=
  let l1 = FORALL_VAR_DECL.vars c1 in
  let l2 = FORALL_VAR_DECL.vars c2 in
  FORALL_VAR_DECL.Build_t (List.app l1 l2).
```

Listing 4.46: Merging of forall variable declarations

Merging is analogous for old variable declarations.

Requires Clauses

Note that a requires clause may be a specified expression, `\not_specified` or `\same`. This is formalised by type `optionalSame`.

```
Inductive optionalSame (A:Type) :=
  | SpecifiedOS : A → optionalSame A
  | NotSpecifiedOS : optionalSame A
  | Same : optionalSame A.
```

Listing 4.47: Extended syntax data type `optionalSame`

```
Definition mergeRequires (c1 c2 : optionalSame Expr) : optionalSame Expr :=
  match c1, c2 with
  | SpecifiedOS e1, SecifiedOS e2 ⇒ SpecifiedOS (e1 &&’ e2)%jml (* 1 *)
  | NotSpecifiedOS, _ ⇒ c2 (* 2 *)
  | _, NotSpecifiedOS ⇒ c1 (* 2 *)
  | Same, _ ⇒ Same (* 3 *)
  | _, Same ⇒ Same (* 3 *)
  end.
```

Listing 4.48: Merging of two requires clauses

Notice that a requires clause `\same` is always the only requires clause in a simple body. Thus *(* 3 *)* is correct.

Ensures Clauses, Diverges Clauses and When Clauses

An ensures clause may be a specified expression or `ensures \not_specified`. This is formalised by type `optional`:

```
Inductive optional (A:Type) :=
  | Specified : A → optional A
  | NotSpecified : optional A.
```

Listing 4.49: Extended syntax data type `optional`

```
Definition mergeEnsures (c1 c2 : optional Expr) : optional Expr :=
  match c1, c2 with
  | Specified e1, Specified e2 ⇒ Specified (e1 &&’ e2)%jml
  | NotSpecified, _ ⇒ c2
  | _, NotSpecified ⇒ c1
  end.
```

Listing 4.50: Merging of two ensures clauses

The merging of diverges clauses and when clauses is done identically.

Signals Clauses

```
Definition mergeSignals (c1 c2 : Param * optional Expr)
    : Param * optional Expr :=
  match c1, c2 with
  | (p1, Specified e1), (p2, Specified e2) ⇒ (p1, Specified (e1 &&' e2)%jml)
  | (p1, NotSpecified), _ ⇒ c2
  end.
```

Listing 4.51: Merging of two signals clauses

Note that the correctness of this desugaring depends on the normalisation of signals clauses as performed by desugaring "Standardising Signals Clauses". In this case, the exception parameter is always equal to `Exception e` and it does not matter if the parameter of `c1` or `c2` is taken.

Assignable Clauses, Accessible Clauses and Captures Clauses

Assignable and accessible clauses may both be `\not_specified` or specify a list of storage locations. The merging of storage location lists has already been presented in section "Desugaring Empty Specifications":

```
Definition appLocations (l1 l2:list Expr) : list Expr :=
    match l1, l2 with
    | Nothing::nil, _     ⇒ l2
    | Everything::nil, _ ⇒ l1
    | _, Nothing::nil     ⇒ l1
    | _, Everything::nil ⇒ l2
    | _, _ ⇒ List.app l1 l2
    end.
```

Listing 4.52: Merging of two lists of storage locations

Storage location lists `l1` and `l2` are appended unless either list is equal to `\nothing` or `\everything`.

```
Definition mergeAssignable (c1 c2 : optional (list Expr))
    : optional (list Expr) :=
  match c1, c2 with
  | Specified e1, Specified e2 ⇒ Specified (appLocations e1 e2)
  | NotSpecified, _ ⇒ c2
  | _, NotSpecified ⇒ c1
  end.
```

Listing 4.53: Merging of two assignable clauses

Accessible and captures clauses are merged in the same way.

Callable Clauses

A callable clause is `\not_specified`, `\everything` or a list of method signatures. This is formalised as type `optional CallableList`, where inductive type `CallableList` is defined as:

```
Inductive CallableList : Type :=
  | EveryMethod : CallableList
  | These list MethodSignature → CallableList.
```

```
Definition mergeCallable (c1 c2 : optional CallableList)
    : optional CallableList :=
  match c1, c2 with
  | Specified cl1, Specified cl2 ⇒
      match cl1, cl2 with
      | These l1, These l2 ⇒ Specified (These List.app (l1 l2))
      | EveryMethod, _ ⇒ Specified EveryMethod
      | _, EveryMethod ⇒ Specified EveryMethod
  | NotSpecified, _ ⇒ c2
  end.
```

Listing 4.54: Merging of two callable clauses

Working Space Clauses and Duration Clauses

According to TR #00-03e, section 3.9 [14], two specified working_space clauses are merged as follows:

‘`working_space` E_1 `if` C_1`; working_space` E_2 `if` C_2’ desugars to
‘`duration Long.min(`C_1 `?` E_1 `: Long.MAX VALUE,` C_2 `?` E_2 `: Long.MAX VALUE) if` C_1 `||` C_2’.

```
Definition mergeWorkingSpace (c1 c2 : optional Expr * option Expr)
    : optional Expr * option Expr :=
  let (expr1, cond1) := c1 in
  let (expr2, cond2) := c2 in

  let cond (ws:optional Expr * option Expr) : Expr :=
    match snd ws with
    | None   ⇒ true'%jml
    | Some c ⇒ c
    end in

  let rewrite1 (ws:optional Expr * option Expr) : optional Expr :=
    match fst ws with
    | NotSpecified   ⇒ NotSpecified
    | Specified expr ⇒ Specified ((cond ws)
                          then' expr
                          else' field java.lang.Integer.F_MAX_VALUE None)%jml
    end in

  let merge2 (e1:Expr) (e2:Expr) : Expr :=
        (method java.lang.Integer.M_min_5 None [e1; e2])%jml in

  match rewrite1 c1, rewrite1 c2 with
  | Specified e1, Specified e2 ⇒ (Specified (merge2 e1 e2),
                                          ((cond c1) ||' (cond2))%jml)
  | NotSpecified, _ ⇒ c2
  | _, NotSpecified ⇒ c1
  end.
```

Listing 4.55: Merging of two working_space clauses

Notice the use of field signature `java.lang.Integer.F_MAX_VALUE` and method signature `java.lang.Integer.M_min_5`. Both are defined in a prelude library. The merging of two duration clauses is done analogously, where `Long.MAX_VALUE` and `Long.min` are used instead of `Integer.MAX_VALUE` and `Integer.min`.

Signals_Only Clauses and Measured_By Clauses

This formalisation does not allow the usage of more than one signals_only clause per flat specification case. Section 9.9.5 of the JML Reference Manual describes the merging of multiple signals_only clauses, but strongly suggests not to use it: "Since this may be confusing, only one signals_only clause should ever be used in a given specification case."

Multiple measured_by clauses per flat specification case are not supported as well.

Generalisation to Lists of Clauses

The generalisation of the above presented merging functions to lists of clauses is straightforward. As an example, the following definition is the actual merging function for ensures clauses.

```
Definition mergeEnsures (l:list (optional Expr)) : option (optional Expr) :=
  let optionalAnd := liftOptional2 (InfixOp ConditionalAnd) in
  match l with
  | nil ⇒ None
  | _   ⇒ Some (fold_left1 optionalAnd l NotSpecified)
  end.
```

Listing 4.56: Merging of a list of ensures clauses

`liftOptional2` is used to abstract

```
match c1, c2 with
| Specified e1, Specified e2 ⇒ Specified (e1 &&' e2)%jml
| NotSpecified, _ ⇒ c2
| _, NotSpecified ⇒ c1
end
```

This case analysis is used in a similar form is almost all merging functions.

```
Definition liftOptional2 (A:Type) (op:A→A→A) (oa ob:optional A) : optional A :=
  match oa, ob with
  | Specified a,  Specified b ⇒ Specified (op a b)
  | NotSpecified, b ⇒ b
  | a, NotSpecified ⇒ a
  end.
```

Listing 4.57: Abstraction of merging function case analysis

Putting it All Together

The desugarings can easily be applied in sequence. Function `desugarAll` does exactly that:

```
Definition desugarAll (specs0:list FULL_SPEC_CASE.t) (m:METHOD.t) (enc:ENTITY.t)
    : list FULL_SPEC_CASE.t :=
  let specs1 := desugarNonNullArguments specs0 m in
  let specs2 := desugarNonNullResult specs1 m in
  let specs3 := desugarPure specs2 m in
  let specs4 := desugarEmptySpecification specs3 m enc in
  let specs5 := desugarNested specs4 in
  let specs6 := desugarBehaviour specs5 m in
  let specs8 := desugarSignals specs6 in
  let specs9 := desugarMultiple specs8 in
  specs9.
```

Listing 4.58: Desugaring consolidation function `desugarAll`

There is still a little gap between `desugarAll` and `rewriteFullSpecification : list FULL-_SPEC_CASE.t -> METHOD.t -> ENTITY.t -> list SpecificationCase`: The resulting list of full specification cases has to be transformed into a list of basic specification cases. This is rather straightforward as desugaring `desugarNested` has already flattened the specification cases and desugaring `desugarMultiple` ensures that *at most* one clause is declared per clause kind. What remains is to add default clauses to ensure that *exactly one* clause is declared per clause kind. This is only necessary for specification cases that were originally not lightweight, as for those the adding of default clauses is already done in desugaring `desugarBehaviour`. For the other specification cases it is done in the same fashion, with the help of `addDefaults` and `heavyweightDefaults` (see section 'Desugaring Lightweight, Normal, and Exceptional Specifications').

The final transformation of a `FULL_SPEC_CASE.t` value into a `SpecificationCase` value is semantically trivial but not shown here, as it again uses some advanced Coq features.

4.4 Proving Case Study

As part of this thesis, a small case study was carried out to evaluate the feasibility of doing proofs on top of the presented Coq formalisation of JML and Java. The proofs carried out were intentionally chosen to be small example proofs. The focus did not lay on proving interesting semantic properties but rather to build a basis for later, more interesting proofs.

The two proofs that the case study looks at, treat a simple rewriting of invariants:

```
Definition mergeInvariants (invs:list INVARIANT.t) : INVARIANT.t :=
  let and := InfixOp ConditionalAnd in
  let lPred := map INVARIANT.pred invs in (* extract invariant exprs *)
  INVARIANT.Build_t (fold_right and true'%jml lPred) Public false false.
```

Listing 4.59: Merging of invariants

The rewriting `mergeInvariants` simply merges all invariants of a given type into one big conjunction consisting of the individual invariants. The case study ignores the visibility and static/instance property of invariants.

The first theorem to prove states that the merged invariant of a given type `c` is equivalent to the list of individual invariants of `c`. This can be formalised by using the semantic function `EvalInvariant`:

```
Definition EvalInvariant (p : Program) (c : Class) (m : Method) (h : Heap.t)
    (fr : Frame.t) : Prop :=
  if METHOD.isHelper m then
    True
  else
    ∀ inv ,
      DefinedInvariant p c inv →
      EvalPredicate p (INVARIANT.pred inv) h fr.
```

Listing 4.60: Semantic function `EvalInvariant`

'`EvalInvariant p c m h fr`' holds exactly if `m` is a helper method or all invariants defined for type `c` hold (`EvalPredicate : Program → Expr → Heap.t → Frame.t → Prop`).

For the first proof, `EvalInvariant` is simplified to only treat invariants *declared* in type `c`, i.e. premise '`DefinedInvariant p c inv`' is replaced by '`In inv (TYPESPEC.invariant (ENTITY.typeSpec c))`'.

Then, the first theorem can be stated as

```
∀ p c c' m h fr,
  c' = rewriteEntity c →
  (EvalInvariant p c m h fr ↔ EvalInvariant p c' m h fr).
```

('`rewriteEntity c`' thereby performs the rewriting `mergeInvariants`.) Since the rewriting merges two invariants `inv1` and `inv2` into a single invariant `inv1 && inv2`, it soon became clear that an important helper lemma is necessary to prove the above theorem:

```
∀ p h fr e1 e2,
  (EvalPredicate p (e1 &&' e2) h fr ↔
  (EvalPredicate p e1 h fr ∧ EvalPredicate p e2 h fr)),
```

that is, the evaluation of '`e1 && e2`', '`Eval [e1 && e2]`' is equivalent to '`Eval [e1] ∧ Eval [e2]`'. Without any further assumptions, this lemma can unfortunately not be proven, since `EvalPredicate` is defined in terms of a more general function `EvalExpression : Program → Expr → Heap.t → Frame.t → option value`. Not having an assumption that the given expressions `e1` and `e2` are *well-formed* boolean expressions, it is not necessarily the case that the evaluation of these two expressions (`EvalExpression`) results in a boolean value. Thus, the definition of a *well-formedness predicate* for boolean expressions became necessary.

For simplicity, the case study restricts well-formed boolean expressions to constants `true` and `false` and the two connectives `&&` and `||`:

```
Inductive Wf_pred : Expr → Prop :=
  | Wf_false : Wf_pred false'
  | Wf_true  : Wf_pred true'
  | Wf_and : ∀ e1 e2, Wf_pred e1 → Wf_pred e2 → Wf_pred (e1 &&' e2)
  | Wf_or  : ∀ e1 e2, Wf_pred e1 → Wf_pred e2 → Wf_pred (e1 ||' e2).
```

Listing 4.61: Well-formedness predicate for boolean expressions

Using this well-formedness predicate, it can be proven that the evaluation of a well-formed boolean expression always results in a boolean value:

```
Lemma EvalExpression_pred : ∀ p h fr e,
  Wf_pred e →
  ∃ b, EvalExpression p e h fr = Some (DOMAIN.Bool b).
```

Listing 4.62: Boolean expressions evaluate to boolean values

Then, a modified version of the above "`Eval [e1 && e2] ⇔ Eval [e1] ∧ Eval [e2]`" can be proven as well:

```
Lemma EvalPredicate_and : ∀ p h fr e1 e2,
  Wf_pred e1 →
  Wf_pred e2 →
    (EvalPredicate p (e1 &&' e2) h fr ↔
    (EvalPredicate p e1 h fr ∧ EvalPredicate p e2 h fr)).
```

Listing 4.63: `Eval [e1 && e2] ⇔ Eval [e1] ∧ Eval [e2]`

Using lemma `EvalPredicate_and`, another helper lemma can be proven that is already very close to the desired goal:

```
Lemma mergeInvariants_ok : ∀ p h fr l,
  (∀ inv, In inv l → Wf_pred (INVARIANT.pred inv)) →
    ((∀ inv, In inv l → EvalPredicate p (INVARIANT.pred inv) h fr) ↔
    EvalPredicate p (INVARIANT.pred (mergeInvariants l)) h fr).
```

Listing 4.64: `mergeInvariants` "preserves" `EvalPredicate`

The proof makes use of the fact that the result of `mergeInvariants` is well-formed as well:

```
Lemma mergeInvariants_Wf : ∀ l,
  (∀ inv, In inv l → Wf_inv inv) →
  Wf_inv (mergeInvariants l).
```

Listing 4.65: The result of mergeInvariants is well-formed

Finally, using lemma `mergeInvariants_ok`, the desired theorem can be proven:

```
Theorem merge_invariants_simple1 : ∀ p c c' m h fr,
  (∀ inv, In inv (TYPESPEC.invariant (ENTITY.typeSpec c)) → Wf_inv inv) →
    c' = rewriteEntity c →
    (EvalInvariant p c m h fr ↔ EvalInvariant p c' m h fr).
```

Listing 4.66: Main theorem of Proof 1

Compared to the originally presented theorem, an additional well-formedness premise has been added.

4.4.1 Proof Details

To get an impression of the proof scripts involved, we look at an example proof script in this section. The example is direction "→" of lemma `mergeInvariants_ok`.

```
Lemma mergeInvariants_ok_lr : ∀ p h fr l,
  (∀ inv, In inv l → Wf_inv inv) →
  ((∀ inv, In inv l → EvalPredicate p (INVARIANT.pred inv) h fr) →
  EvalPredicate p (INVARIANT.pred (mergeInvariants l)) h fr).
Proof.
  intros p h fr.
  induction l.
  (* base case l=nil *)
  intros.
  simpl.
  intuition.
  compute.
  trivial.
```

Listing 4.67: Proof of direction → of lemma `mergeInvariants_ok` (1)

The proof is by induction. The base case '`l = nil`' is easy to prove:
'`∀ inv, In inv l → EvalPredicate p (INVARIANT.pred inv) h fr)`' simplifies to
'`∀ inv, False → ...`' which simplifies to `True`.
'`EvalPredicate p (INVARIANT.pred (mergeInvariants l)) h fr)`' simplifies to
'`EvalPredicate p (true'%jml) h fr`' since '`mergeInvariants nil`' is equals to an invariant `true'%jml`.

'`EvalPredicate p (true'%jml) h fr`' is proven by tactic `compute` that computes the result `True`.

```
(* ... continuing the proof... *)

  (* step case a::l *)
  intro Hwf.
  set (inv' := mergeInvariants (a::l)).

  assert (Wf_pred (INVARIANT.pred inv')) as Hwf_inv'.
  apply mergeInvariants_Wf with (l := a::l); assumption.

  (* (∀ inv, In inv (a::l) → EvalPredicate p inv) → EvalPredicate p inv' *)

  intro H. (* ∀ inv, In inv (a::l) → EvalPredicate p inv *)
  simpl.
  apply EvalPredicate_and_lr.
    (* Wf_pred a *)
    apply Hwf; intuition.
    (* Wf_pred inv, In inv l *)
    inversion Hwf_inv'; assumption.

    (* EvalPredicate a *)
    apply H; intuition.
    (* EvalPredicate p inv, In inv l *)
    apply IHl.
      (* ∀ inv, In inv l → Wf_inv inv *)
      intros inv Hinv.
      apply Hwf; intuition.

      (* ∀ inv, inv l → EvalPredicate p (INVARIANT.pred inv) h fr *)
      intros inv Hinv.
      apply H; intuition.
```

Listing 4.68: Proof of direction → of lemma `mergeInvariants_ok` (2)

In the step case, a helper lemma '`Wf_pred (INVARIANT.pred inv')`' is proven, i.e. that the result `inv'` of '`mergeInvariants (a::l)`' is well-formed. Then "`EvalPredicate inv'`" [15] is simplified (tactic `simpl`) into "`EvalPredicate (a &&' fold_right (InfixOp ConditionalAnd) true' (map INVARIANT.pred l))`". This is split by applying `EvalPredicate_and_lr` (direction "→" of `EvalPredicate_and`) into "`EvalPredicate a ∧ EvalPredicate (fold_right ...)`". First, the two well-formedness premises are proven. Then, "`EvalPredicate a`" is proven by applying hypothesis '`H : In inv (a::l) → EvalPredicate inv`'. Lastly, "`EvalPredicate (fold_right ...)`" is proven by applying the induction hypothesis `IHl`. Both premises of `IHl` are easily proven by applying hypothesis `Hwf` and `H`.

4.4.2 Results

All above conjectured lemmas and theorems have been proven as part of this thesis. The full proof strict can be seen in Appendix B.

The following table indicates the length of the various proofs:

Lemma/Theorem	Lines of Code
`EvalExpression_pred`	15
`EvalPredicate_and`	30
`mergeInvariants_Wf`	10
`mergeInvariants_ok`	35
`merge_invariants_simple1`	45
Other lemmas	20
Total	155

With comments and empty lines, the whole proof script has a length of 430 lines.

4.4.3 Discussion

The proof of theorem `merge_invariants_simple1` has successfully shown that proofs on top of the JML formalisation are indeed possible. However, the proof script is surprisingly long for such a simple proof. In our opinion this has two major reasons:

First, the proof script also contains the development of the well-formedness predicate and the associated proof of `EvalExpression_pred` (i.e. a well-formed boolean expression evaluates to a boolean value). The need for this lemma stems from missing typing information in the semantics.

[15] In the code comment and this discussion, for simplicity, we write "`EvalPredicate inv`" for '`EvalPredicate p (INVARIANT.pred inv) h fr`'.

Second, the proof script only makes use of rather basic tactics. By defining higher-level tactics and automating rewritings and unfoldings of definitions specially suited to our formalisation, the proofs may be simplified considerably.

4.4.4 Second Proof

A second proof has been discharged as part of the case study: theorem `merge_invariants_simple2`. The difference to `merge_invariants_simple1` being that the simplification of the semantic function `EvalInvariant` has been undone: `EvalInvariant` in this case treats all invariants *defined* for a type `c` (instead of invariants *declared* in type `c`).

```
Definition EvalInvariant (p : Program) (c : Class) (m : Method) (h : Heap.t)
    (fr : Frame.t) : Prop :=
  if METHOD.isHelper m then
    True
  else
    ∀ inv ,
      DefinedInvariant p c inv →
      EvalPredicate p (INVARIANT.pred inv) h fr.
```

The proof of lemma `merge_invariants_simple2` could reuse large parts of the first proof script, in particular lemma `mergeInvariants_ok`. In addition, two lemmas lay the basis of the second proof:

```
Lemma invariant_defined_2 : ∀ (c : Class) (inv : INVARIANT.t),
  DefinedInvariant c inv ↔
  In inv (TYPESPEC.invariant (ENTITY.typeSpec c)) ∨
  (∃ super : Make.ENTITY.t,
    direct_subtype c super ∧ DefinedInvariant super inv).
```

Listing 4.69: Case split on `DefinedInvariant`

That is, for every class `c` and invariant `inv`, invariant `inv` is *defined* for class `c` if and only if the invariant is *declared* in `c` or there exists a direct super type `super` of `c` and invariant `inv` is *defined* for type `super`. This enables a proof by cases on `invariant_defined_2` that is very much like a proof by induction.

The second helper lemma is `rewriteEntity_maintains_direct_supertypes` that states the fact that '`mergeInvariants (rewriteEntity)`' on a type `c` maintains the direct supertypes of type `c`.

```
Definition maintains_direct_supertypes (p:Program) (f:ENTITY.t → ENTITY.t) :=
  ∀ e super,
  direct_subtype e super <-> direct_subtype (f e) super.

(** rewriteEntity maintains the direct super types *)
Lemma rewriteEntity_maintains_direct_supertypes : ∀ p,
  maintains_direct_supertypes p (fun e ⇒ rewriteEntity e).
```

Listing 4.70: `rewriteEntity` maintains direct supertypes

Although the second proof case study is more involved, the second proof script is still shorter then the first proof script ($\sim$ 330 lines of codes and comments vs. $\sim$ 430 lines). This is due to the large amount of reuse of the first proof script.

4.5 Java Translation Frontend

The Java Translation Frontend translates a set of JML-annotated Java source files (Java version 1.4) into a set of Coq output files. These output files make up an embedding of the Java/JML source code in the full syntax definition of Java and JML in Coq.

Besides this pure syntactic translation, the frontend desugars nullable_by_default and pure modifiers on type level. These desugarings are discussed in subchapter 4.5.5.

In the following, when referring to a Java source file, we always mean a JML-annotated Java source file.

The translation of a Java source file into a set of Coq output files is done as is common practice in compiler design: the Java source file is first parsed into an abstract syntax tree (AST). A traversal of the tree is then used to build the translation.

4.5.1 Design

We had two major design goals in mind for this translation frontend:

First, we wanted the output generation to be *composable*. That is, the output should be built bottom-up in the same way the tree is traversed: the output for an AST node is built from the outputs of its children nodes. As an example serves the AST of '`1 + 2`'.

```
[+]
 ⊢ [1]
 ⊢ [2]
```

The output `"1+2"` of node '`+`' is built from the outputs `"1"` and `"2"` of its children and its own information.

Second, we wanted modifications on the output syntax to be performed easily (*maintainability*), and even more, the whole output backend to be replaced easily (*adaptability*).

These design goals lead to a split of the visitor into three parts: the *output visitor*, the *outputter* and the *pretty printer*. The connection between the three parts is the *output document* representing the translated Coq file. A visit method for a node X first visits the child nodes of X. Then, the output document corresponding to node X is created by a delegating call to the outputter, giving the output documents, resulting from the children's visit methods, as arguments. The output document of the root node can then be pretty-printed into a Coq output file.

That is, the split into output visitor, outputter and pretty printer is a division of labour: the visitor traverses the tree, the outputter builds output documents and the pretty printer provides facilities to create and compose those documents.

The output visitor is part of a visitor pattern [7]. The pattern is implemented in the visitor-controlled variant, that is, the traversal of the tree is controlled within the visit methods and not within the accept methods, that solely call the appropriate visit methods. The visit methods

Document	Pretty print
`nil`	`""`
`text "ab" <> nil <> text "cd"`	`"abcd"`
`nest 2 (text "hello" <> line <> text "world")`	`"hello\n␣␣world"`

Table 4.3: Prettier printer illustration

have the form '`Object visitT(T x, Object o)`', where `T` is one of the node classes. The result object of the visit/accept methods of the output visitor is always the output document of the corresponding subtree. The argument object `o` is partially used to give additional information to visit methods of child nodes.

The pretty printer provides methods to create, compose and pretty-print documents. The pretty printer we used is an imperative implementation of the functional prettier printer by Philip Wadler [15]. The transformation of this functional pretty printer into an imperative version and its implementation in Java has been carried out as part of this thesis. To keep the link to the original prettier printer and to simplify notation, we use in the following Coq syntax to describe its functionality. The pretty printer operates on an abstract type Doc. The following functions create or compose documents:

- `nil : Doc`
- `text : String` → `Doc`
- `line : Doc`
- `concat : Doc` → `Doc` → `Doc`
- `nest : int` → `Doc` → `Doc`

New documents are created using function `nil`, function `text` creates a new document representing a given string `s` and function `line` a new document representing a newline character. Function '`concat d1 d2`' (written as '`d1 <> d2`') creates a document that is equivalent to document `d2` appended to document `d1` and function '`nest n d`' creates a document that is equivalent to document `d` where every new line (except the first one) is indented by `n` characters.

Table 4.3 gives example documents built using the different functions and their pretty print.

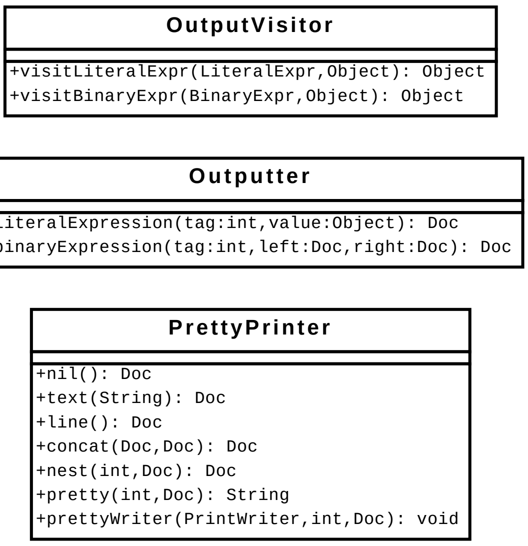
Documents are pretty-printed to an output `String` by function `pretty : int` → `Doc` → `String`[16]. In the imperative implementation, function `void prettyWriter(PrintWriter, int, Doc)` can be used to pretty print into any `PrintWriter`.

In the following, we give an example that illustrates the interaction between output visitor, outputter and pretty printer: the translation of the binary expression '`this.f = 10`'. Notice the division of labour: methods `Visitor.visitBinaryExpr` and `Visitor.visitLiteralExpr` only deal with visiting child nodes and calling the outputter. It is only the outputter's methods `Outputter.literalExpression` and `Outputter.BinaryExpression` that deal with the syntax of the Coq formalisation. UML diagram 4.3 gives an overview about the functions of the three classes.

```
// translation of this.f = 10
```

[16]For an explanation of the first parameter, see the chapter 4.5.4.

Figure 4.3: Visitor, outputter and pretty printer


OutputVisitor
+visitLiteralExpr(LiteralExpr,Object): Object
+visitBinaryExpr(BinaryExpr,Object): Object
Outputter
+literalExpression(tag:int,value:Object): Doc
+binaryExpression(tag:int,left:Doc,right:Doc): Doc
PrettyPrinter
+nil(): Doc
+text(String): Doc
+line(): Doc
+concat(Doc,Doc): Doc
+nest(int,Doc): Doc
+pretty(int,Doc): String
+prettyWriter(PrintWriter,int,Doc): void

```
1. Doc Visitor.visitBinaryExpr(BinaryExpr x, Object o)
    // x.op    == TagConstants.ASSIGN
    // x.left  == [x.f]
    // x.right == [10]
    Object leftArg = x.left.accept(this, VisitorArgument.NO_ARGUMENT); // 1.1
    Object rightArg = x.right.accept(this, VisitorArgument.NO_ARGUMENT); // 1.2

    // leftArg == "field Test.F_f (Some this)"
    // rightArg = "int 10"

    return out.binaryExpression(x.op, leftArg, rightArg); // 1.3
    // \result == "field Test.F_f (Some this) ==' int 10"

1.2. Doc Visitor.visitLiteralExpr(LiteralExpr x, Object o)
    // x.tag == TagConstants.INTLIT, x.value == 10

    switch (x.tag) {
    case TagConstants.INTLIT: {
      return out.literalExpression(x.tag, x.value); // 1.2.1
    }

1.2.1 Doc Outputter.literalExpression(int tag, Object value)
    // tag == TagConstants.INTLIT, value == 10
    switch (tag) {
    case TagConstants.INTLIT: {
      Integer i = (Integer) value;
      return pp.concatSpace(  [17]
          pp.text("int"),
          pp.intValue(i))
        );
```

[17]'concatSpace d1 d2 := d1 <> text " " <> d2'; 'concatSpace d1 d2 d3' is defined analogously.

```
    }
    }

1.3 Doc Outputter.binaryExpression(int tag, Doc left, Doc right)
    // tag == TagConstants.ASSIGN
    // left == "field Test.F_f (Some this)", right == "int 10"
    String tagString = exprTag2String(tag); // tagString == "=’"
    return pp.concatSpace(
        left,
        pp.text(tagString),
        right
      );
```

Listing 4.71: Example translation

4.5.2 Translation Overview

Now, after having described the design of the translator and its three main components, we look at the different steps involved in the translation process, beginning with the set of Java source files as input.

Input: $f_1.java$, ..., $f_n.java$

1. Javefe parser, ESC/Java2 parser $\rightarrow$ compilation units cu_1, ..., cu_n
2. $\rightarrow$ split into compilation units cu_{1,m_1}, ..., cu_{n,m_n}
3. for every compilation unit cu_{ij} in cu_{1,m_1}, ..., cu_{n,m_n}:
   - 3.1 apply the rewritings (section 4.5.5)
   - 3.2 apply the output visitor to traverse the AST of the compilation unit (result: output document d_{ij})
   - 3.3 pretty print output document d_{ij} to an output file g_{ij}

Coq output files $g_{1,m_1}.v$, ..., $g_{n,m_n}.v$

As previously described, an abstract syntax tree has to be built first for every input file. The translation frontend builds on top of ESC/Java2 (see section 5.1.3) that is used to parse JML and Java source code and to build the AST. ESC/Java2 creates a compilation unit (AST node) cu_i for every input file f_i. In a second step, every compilation unit cu_i containing m_i type declarations is split into m_i compilation units containing one type declaration only. The motivation behind this splitting is explained later. Every compilation unit $cu_{i,j}$ is then first desugared by applying the type level rewritings. Then, the output visitor is applied to its accept method to create the output document d_{ij}. Finally, output document d_{ij} is pretty-printed to a Coq output file $g_{ij}.v$. The output file name g_{ij} is built as follows: If compilation unit cu_{ij} declares a class `X` within package `A.B`, the output file name g_{ij} is equivalent to `A_B_X.v`.

4.5.3 Translation Details

In the following we discuss some interesting details of the translation done in step 3.2, by means of example 4.72. The full translation of this example is given in Appendix A.

```
package test;

public class A {
  private int tally = 0;

  void inc(int n) {
    tally += n;
  }

  int tally() {
    inc();
    return tally;
  }
}
```

Listing 4.72: Example class A

The compilation unit of class `A` is translated into the following Coq output file:

```
Tr [package test; public class A {...}] =
  Require Import JMLSyntax.
  Require Import java_lang_Object.

  (* Names and signature variables *)

  (* Import the names and signatures defined in the context of class test.A *)
  Import test.A.
  Definition Def' := Tr [public class A {...}]
```

First, two files are imported, `JMLSyntax` that provides the formalisation and `java_lang_Object`, the Coq output file defining the translation of base class `Object` of class `A`. Next comes a bulk of name and signature variable definitions. These variables are used as shorthands to make the translation more readable. Finally, variable `Def'` defines the translation of the declaration of class `A`.

The use of a name variable can be seen in the translation of the declaration of field `tally`:

```
Tr [int tally = 0] =
  field_decl of_class [private] int_t F_tally_ (Some (int 0))
```

Field name variable `F_tally_` (‘`Definition F_tally_ := 1002`’) is used instead of the real field name `1002` to make the translation more readable.

Name variables are generated for all identifiers appearing in the compilation unit to translate...

- the package name
- the class name
- field and routine names
- formal parameter names

- local variable names
- label names

... and for all referenced identifiers of other types

- package names
- class names
- field and routine names.

The translation of expression '`return tally`' is:

```
Tr [return tally] =
  returnE (field test.A.F_tally (Some this))
```

In this translation, field signature variable `test.A.F_tally` is used. Signature variables are used as a shorthand such that the signature does not have to be repeated over and over again. ('`Definition F_tally := field_signature_decl test.A_ int_t F_tally_`') Signature variables are generated for fields and routines of the type to translate and all referenced fields and routines of other types, as well as for formal parameters and local variables.

An example, illustrating the need for formal parameter signatures is the translation of the expression '`tally += n`':

```
Tr [tally += n] =
  (field test.jtf_only.A.F_tally (Some this)) += (param P_n_1004).
```

Here, parameter `n` is referenced via its signature variable `P_n_1004`.

Name- and Signature Variable Generation

The first idea regarding signature generation had been to only define name and signature variables for entities declared in type `T` within the Coq output file for type `T`. Then, signatures for fields or methods of other types (that `T` depends on) would be imported by importing the corresponding output files. This turned out to be impossible because of *mutual dependencies*: classes `Object` and `String` for example have mutual dependencies ('`String Object.toString()`' references '`boolean String.equals(Object)`' and vice versa). In this case, output file `java_lang_Object` would have to import `java_lang_String` and vice versa, which is not possible in Coq.

The second idea had been to define all [18] name and signature variables within the Coq output file for type `T` and only import the output file of `T`'s base class [19]. In this approach, name and signature variables have to be organised in name spaces to prevent conflicts between variables of different types. Still, the idea had been to use one output file for a compilation unit with possibly multiple types. Two new problems emerged: *"crosswise subtyping"* between two compilation units and *dependencies* between signatures.

"Crosswise subtyping" occurs in the following situation:

[18] Name and signatures of entities declared in type `T` and of referenced entities of other types.
[19] Since the subtype graph is free of cycles, this would not pose any problems.

```
Compilation unit 1:                      |  Compilation unit 2:
class PublicSuper {}                     |  class PublicSub extends PublicSuper {}
class PackageSub extends PackageSuper {} |  class PackageSuper {}
```

Listing 4.73: Illustration of “crosswise subtyping” subtyping

The output file of compilation unit 1 imports the output file of compilation unit 2 (base class `PackageSuper`) and vice versa (base class `PublicSuper`). This problem is solved by allowing only one type definition per compilation unit as achieved by the splitting described in the translation overview.

The problem of dependencies between signatures is illustrated in the following scenario:

```
package a                 |  package b;
public class A {          |  public class B {
  void testB(b.B b) {}    |    void testA(a.A a) {}
}                         |  }
```

Listing 4.74: Illustration of signature dependencies

This example would lead to the following name and signature definitions in the output file of class `A`:

```
Module a.
  Module A.
    Definition PKG_a_ := ...
    Definition C_A_   := ...
    Definition A_ := (PKG_a_, C_A_).

    Definition P_b_1004_     := ...
    Definition M_testB_1004_ := ...

    Definition P_b_1004 := param_decl (class_t b.B_ peer) P_b_1004_.
    Definition M_testB_1004 :=
      method_signature_decl a.A.A_ Void M_testB_1004_
        [param_decl (class_t b.B.B_ peer) P_b_1004_].
  End A.
End a.

Module b.
  Module B.
    (* names and signatures of type B *)
  End B.
End b.
```

In Coq, a definition can only refer to identifiers that are previously defined; either previously in the same file, or in an imported file. But in the definition of `M_testB`, class name `b.B.B_` is not yet defined. (The same problem would occur with class name `a.A.A_` if module `B` were defined before module `A`.) Notice that the problem is restricted to references on other class names, since method/field signatures can only refer to other class names (as types of parameters). Thus the solution is to define all class names before any field/method signature definition.

Identifier kind	Required "uniqueness" scope	Ensured "uniqueness" scope
Labels	Label statement	Type
Local variables	Routine	Type
Formal parameters	Routine	Type
Fields	Global	Global
Routines	Global	Global
Types	Global	Global
Packages	Global	Global

Table 4.4: =
"Uniqueness" scope of identifiers

A technical problem remains: Coq does not allow to use a name space again, once it is closed. But this would indeed be necessary in the proposed solution, first the name space would be used for the class name definition, then for the other name and signature definitions. This problem is overcome by enclosing the first name space in a dummy name space `N'` [20]. This name space `N'` is imported to create the illusion that the class name definitions are within the same name spaces as the rest of the definitions.

The full translation (see Appendix A) of example 4.72 gives an example of name and signature variable generation in this last and final form.

Identifier Generation

The translation has to ensure that it generates unique identifiers. This problem is twofold: First, entity names, such as class or field names, represented as natural numbers in the formalisation have to be unique up to a certain point. Field-, routine-, type- and package names have to be globally unique since they can be referenced from anywhere. Local variable- and formal parameter names have to be unique within the enclosing routine. Label names only have to be unique within the enclosing label statement.

The translation frontend not only ensures the aforementioned "uniqueness" scope, but ensures to generate name numbers for labels, local variables and formal parameters that are unique within the enclosing type. The reason for this widening becomes clear, when looking at the second form of identifiers (besides name numbers) the translation frontend has to generate, namely the name- and signature variable identifiers.

Name and signature variable definitions are used in the output file to make the translation shorter and more readable. Recalling the example translation '`Tr [package test; public class A {...}]`', all name and signature variables are defined in one place, before the translation of the class declaration, within name spaces corresponding to their enclosing type. Despite the use of name spaces, name clashes could occur between different identifiers, if Java identifier names were used directly as name and signature variable identifiers:

- two different kind of identifiers with the same name (e.g. a field `tally` and a method `tally`)
- two identifiers of the same kind within different scopes in Java but within the same enclosing type (e.g. parameters `n` of two methods '`void inc(int n)`' and '`void dec(int n)`')
- overloaded methods (e.g. '`void inc()`' and '`void inc(int)`')

[20]The prime character is used to avoid name conflicts with Java identifiers of the same name.

Identifier Kind	Translated name	Declaration/signature identifier
Label x [a]	L_x_id_	
Local variable x	V_x_id_	V_x_id
Formal routine parameter x	P_x_rid_	P_x_rid
Other formal parameter x	P_x_id_	P_x_id
Field x	F_x_	F_x
Routine x	M_x_id_	M_x_id
Type x	C_x_	C_x
Package x	PKG_x_	

[a] x refers to the name of the original Java identifier.

Table 4.5: Identifier naming scheme

To prevent these name clashes, the scheme outlined in table 4.5 is used for name and signature variable identifiers.

The translated name is used for the corresponding name definitions while the entry of the third row is used for declarations of variables, parameters and types and for the signatures of fields and routines. The natural number `id` used, corresponds to the entity name number. This is the reason why uniqueness within the enclosing type is guaranteed for entity name numbers of labels, variables and parameters. Otherwise, this simple naming scheme could not be used.

A distinction is made between formal routine parameters and other formal parameters (catch clause parameter and signals clause parameter). For routine parameters, the scheme uses the name number of its enclosing routine (`rid`) instead of the parameters number, to make an easy association between routines and their parameters possible. Examples of signature (and name-) identifiers from the full translation of example 4.72: Field signature id `F_tally`, routine signature id `M_inc_1004` and parameter declaration id `P_n_1004` of routine `inc`.

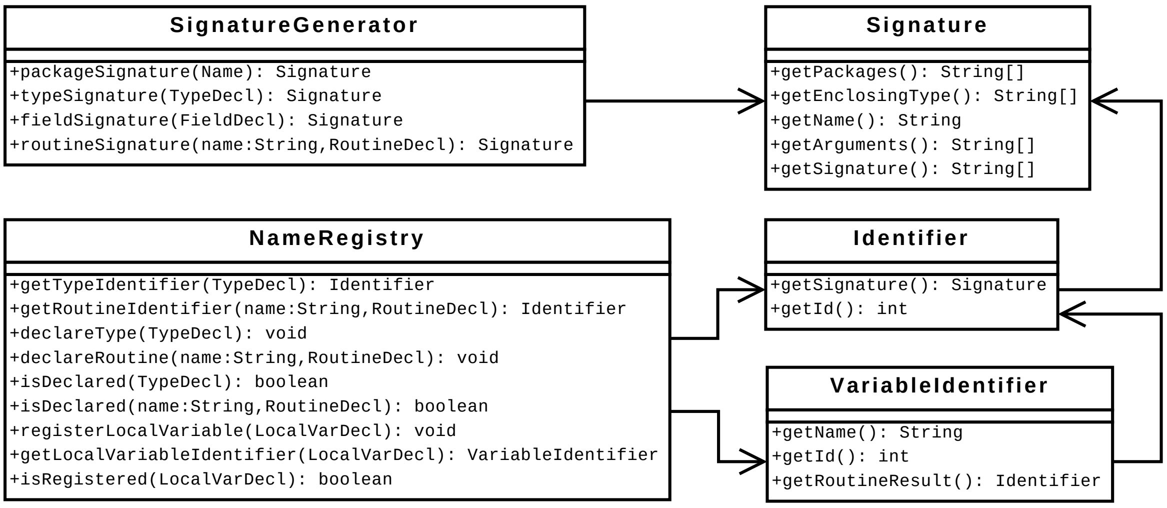
4.5.4 Implementation

In the implementation, class `NameRegistry` is in charge of generating the entity name numbers. UML diagram 4.4 shows the relevant classes. The name registry draws a distinction between *local* entities (labels, variables and parameters) and *global* entities (fields, routines, types and packages). Local entities have to be registered with the name registry before the corresponding identifier can be retrieved. This goes hand-in-hand with the output visitor: A label, local variable or parameter is always declared before being used. Thus, for reasons of safety, the name registry also requires that a local entity is registered (`NameRegistry.isRegistered`) before the corresponding identifier (entity name number) can be retrieved.

```
// Example: Tr[void inc(int n) { tally += n; }]

1. Object OutputVisitor.visitMethodDecl(MethodDecl, Object) {
     ...
     for (int i = 0; i < nParams; i++) {
       formalParamArgs[i] = x.args.elementAt(i).accept(
           this,
           VisitorArgument.paramArgument(true) /* method param */
         );
     }
     ...
```

Figure 4.4: Name registry and signature generator


SignatureGenerator
+packageSignature(Name): Signature
+typeSignature(TypeDecl): Signature
+fieldSignature(FieldDecl): Signature
+routineSignature(name:String,RoutineDecl): Signature
Signature
+getPackages(): String[]
+getEnclosingType(): String[]
+getName(): String
+getArguments(): String[]
+getSignature(): String[]
NameRegistry
+getTypeIdentifier(TypeDecl): Identifier
+getRoutineIdentifier(name:String,RoutineDecl): Identifier
+declareType(TypeDecl): void
+declareRoutine(name:String,RoutineDecl): void
+isDeclared(TypeDecl): boolean
+isDeclared(name:String,RoutineDecl): boolean
+registerLocalVariable(LocalVarDecl): void
+getLocalVariableIdentifier(LocalVarDecl): VariableIdentifier
+isRegistered(LocalVarDecl): boolean
Identifier
+getSignature(): Signature
+getId(): int
VariableIdentifier
+getName(): String
+getId(): int
+getRoutineResult(): Identifier

```
2. Object OutputVisitor.visitFormalParaDecl(FormalParaDecl x, Object o) {
     ...
     // register parameter x with registry → registry.isDeclared(x) == true
     registry.registerParameter(x, arg.getArgument());
     // query name result for param x from name registry
     NameRegistry.VariableIdentifier name = registry.getParameterIdentifier(x);
     ...
   }
```

Listing 4.75: Example of visitor and name registry interaction

With a global entity `e` in contrast, it is not possible in every case to process the declaration of `e` before every use of `e`. The mutual dependency between classes `String` and `Object` again serve as an example: Whatever class is processed first by the output visitor, the name registry has to be queried for the entity name number of the other class. Thus, the name registry does not require the entities to be registered before queries can be done.

The name registry stores name numbers in *hash tables*. The hash values used are generated by class `SignatureGenerator` that generates unique strings for all types of entities, based on the package name, the name of the enclosing type, the name and eventual arguments (for routines) of the entity. Table 4.6 summaries the signature generation scheme.

The name registry uses a most simple scheme to ensure the uniqueness scopes presented in table 4.4: it uses a global counter that it queries and increments for each newly generated entity name number.

Advanced Pretty Printer features

The pretty printer provides the possibility to generate alternative layouts and to choose the optimal layout based on a maximum line width. This possibility is not used in the current implementation of the outputter but it could easily be adapted to use this feature.

	package	enclosing type	name	arguments	hash value
Default package	`{"unnamed"}`	`{}`	`"unnamed"`	`{}`	`{"unnamed"}`
Package `x.y`	`{"x", "y"}`	`{}`	`"y"`	`{}`	`{"x", "y"}`
Type `x.T`	`{"x"}`	`{}`	`"T"`	`{}`	`{"x", "T"}`
Type `x.T1.T2`	`{"x"}`	`{"T1"}`	`"T2"`	`{}`	`{"x", "T1", "T2"}`
Routine `x.T.r(T1, T2)`	`{"x"}`	`{"T"}`	`"r"`	`{"T1", "T2"}`	`{"x", "T", "r", "T1, "T2"}`
Field `x.T.f`	`{"x"}`	`{"T"}`	`"f"`	`{}`	`{"x", "T", "f"}`

Table 4.6: Signature generation scheme

To generate alternative layouts, the pretty printer provides function `group : Doc` → `Doc`. ‘`group d`’ returns a new document representing document `d` (first layout) plus an alternative layout of `d`, where every newline character is replaced by a space character (second layout).

Pretty print function `pretty : int` → `Doc` → `String` then chooses the best layout based on its first argument, the maximum line width: If the first line of both layouts fit the maximum line width, the layout with the longer line is chosen, otherwise the layout with the shorter first line is chosen.

This is best seen with an example: We either want a function call like ‘`main(arg1, arg2)`’ to be pretty printed as

```
main(arg1, arg2)
```

or as

```
main(arg1,
     arg2)
```

depending on whether the line width is sufficient for the first layout. The pretty printer code (functional pseudo code) to output a document with those two layouts is the following:

```
(* The following code pretty prints a function 'name' (length of name 'n')
   with arguments 'args':
*)
Definition ppFunction (name : Doc) (n : int) (args : list Doc) : Doc :=
  text "("  <>
  group (nest (n+1)
    (vsep (punctuate (text ",") args)) <>
  text ")"
```

‘`punctuate punct docs`’ concatenates document `punct` to every document in `docs`. ‘`vsep docs`’ vertically separates `docs`, i.e. it concatenates a newline character to every document in `docs`.

The following listing illustrate `ppFunction` on ‘`main(arg1, arg2)`’:

```
name = text "main"
args = [text "arg1", text "arg2"]
punctuate (text ",") args =
  [text "arg1" <> text ",", text "arg2"] ["arg1,", "arg2"]
```

```
vsep (punctuate (text ",") args) =
  text "arg1" <> text "," <> line <> text "arg2" ["arg1,\narg2"]
```

Pretty print of first layout: `main(arg1,\n␣␣␣␣␣arg2)`

Pretty print of second layout: `main(arg1,␣arg2)`

Thus depending on whether the first line of layout 1 (length: 10) fits the maximum line length, the first or the second layout is chosen.

For implementation details, in particular how to efficiently implement alternative layouts, see paper 'A prettier printer' by Philip Wadler [15].

As part of this thesis, this prettier printer library has been transformed to imperative style and implemented in Java. For details on this transformation of the original functional code into imperative code and in particular, how lazy evaluation is simulated, see the source code (`pp.PrettierPrinter`, `pp.impl.PrettierPrinterImpl`) and the Javadoc documentation.

4.5.5 Type Level Rewritings

Before translating the abstract syntax trees of the parsed compilation units, the Java translation frontend does two simple rewritings: the desugaring of the JML modifiers `nullable_by_default` and `pure` on the type level.

Modifier `nullable_by_default`, set on a top-level type declaration `T`, makes all declarations within `T` (and nested types of `T`) implicitly `nullable`. (JML Reference Manual, section 6.2.12 [10]).

Modifier `pure`, set on any type declaration `T` (`T` may be a nested or local type), makes all constructors and instance methods within `T` (not within nested types of `T`) implicitly `pure`. (JML Reference Manual, section 6.1.2).

We decided to perform these rewritings in the frontend and not within the Coq formalisation, mainly because they are easier to write down in the frontend. Both rewritings require traversing the AST of the affected types and, in case of `nullable_by_default`, the whole AST of method specification cases (to set forall- and old declarations `nullable`). These AST traversals would unnecessarily complicate the rewritings of the extended syntax. Furthermore these rewritings are straightforward: they replace one syntactic modifier keyword (`nullable_by_default/pure`) by other modifier keywords (`nullable/pure`). Also, the rewritings do not strongly affect the readability of the Coq syntax outputted by the frontend. In the end, the decision is a tradeoff between defining all JML semantics within the Coq formalisation and having a short and comprehensible semantics. In this case, we decided for the latter.

The exact details of the two rewritings is described in the Javadoc documentation of the classes `NullableByDefaultRewriter` and `PureRewriter`.

4.5.6 Implementation Remarks

Program Counter Handling

For compatibility reasons with Bicolano (see section 5.1.1), every statement in our Java source code formalisation has the program counter of the first corresponding bytecode instruction associated.

In the current implementation, dummy program counters are used: a global counter is used as program counter, that is incremented with each visit of a statement node. Nevertheless, the implementation can easily be adapted to use the real program counters of the bytecode instructions. The only place in the source code of the translation frontend to change for this adaptation is method '`void OutputVisitor.prepareOutput(ASTNode, Object)`'. Every statement visit method calls `prepareOutput` before calling the outputter.

```java
Object visitAssertStmt(AssertStmt x, Object o) {
  Object predArg = x.pred.accept(this, VisitorArgument.NO_ARGUMENT);
  Object labelArg =
    x.label == null ? null : x.label.accept(this, VisitorArgument.NO_ARGUMENT);
  prepareOut(x, o);
  return out.assertStatement(predArg, labelArg);
}
```

Listing 4.76: Example visit method showing the use of `prepareOutput`

With the current implementation, `prepareOutput` sets the program counter within the outputter and increments it. The outputter in turn uses the most recent program counter value set by `prepareOutput` as current program counter in its statement output functions.

Using ESC/Java2 as Parsing Frontend

The following section described how ESC/Java2 (see section 5.1.3) is used as parsing frontend. ESC/Java2 is thereby used as a library only, to reduce mixing of foreign code and our own code to a minimum. The easiest way to use ESC/Java2 is to let the application's main class extend `escjava.Main`. Then, the application merely has to call method `run` and do its processing in an override of method `postprocess`. It is best to also override method `makeOptions` to returns an `Options` object that has field `stages` set to "type-checking only", unless the application needs the other functionalities of ESC/Java2 as well. Code snippet 4.77 gives an example.

```java
public class Main extends escjava.Main {

  public javafe.Options makeOptions() {
    escjava.Options options = new escjava.Options();
    // set type-checking only
    options.stages = 1;
    // set other default options...
    return options;
  }

  public static void main(String[] args) {
    new Main().run(args);
  }

  public void preprocess() {
    nUnitsToProcess = loaded.size();
    super.preprocess();
  }

  public void postprocess() {
    // do your processing here...
  }
}
```

Listing 4.77: Example use of ESC/Java as parsing frontend

Note that ESC/Java2 only fully parses the explicitly given input files. Further compilation units, automatically loaded by ESC/Java2 during parsing are not fully parsed, unless argument '`-Depend`' is given to ESC/Java2. Unfortunately, this argument makes ESC/Java2 very slow. To know which compilation units were given explicitly and are thus fully parsed, it is easiest to remember the number of loaded units (`this.loaded`) at the time of `preprocess`. Example 4.77 also illustrates this issue.

Disabling the Annotation Handler

ESC/Java2 does a lot of JML desugarings during the parsing/type-checking phase that are unsuitable for tools desiring to obtain an abstract syntax tree of the sources as written by the programmer. More specifically, the following desugarings are done within the `escjava.AnnotationHandler`:

- it adds default specs if none are present (`defaultSignalsOnly`, `defaultModifiers`)
- it performs unnesting of nested specification cases (`deNest`)
- it desugars lightweight, normal- and exceptional_behaviour specification cases (`Nested-PragmaParser`)

The solution to disable these desugarings we found is to override the annotation handler and the nested pragma parser (`escjava.AnnotationHandler.NestedPragmaParser`):

- override all exported methods of the annotation handler (that are actually called) with an empty implementation:
  - `void process(TypeDeclElement)`
  - `void desugar(TypeDecl)`
  - `void desugar(RoutineDecl)`
  - `void parseAllRoutineSpecs(CompilationUnit)`
- override `parseSeq` of the nested pragma parser to keep the behaviour-/normal_behaviour-/ exceptional_behaviour modifier pragmas. (see listing 4.78)

```
void parseSeq(...) {
  // Save 'behavior' modifier in the 'result' ModifierPragmaVec if non-null
  // and the result is non-empty

  int retval = super.parseSeq(pos, pm, behaviorMode, behavior, result, rd);
  if (behavior != null && result.size() > 0) {
    result.insertElementAt(behavior, 0);
  }
  return retval;
}
```

Listing 4.78: Override of `NestedPragmaParser.parseSeq`

The two overridden classes (say they are called `MyAnnotationHandler` and `MyNestedPragmaParser`) are installed as follows:

- override `javafe.tc.TypeCheck Main.makeTypeCheck()` to return a `TypeCheck` object whose `annotationHandler` field is an instance of `MyAnnotationHandler`.
- set the static field `specparser` of class `escjava.AnnotationHandler` (type `NestedPragma-Parser`) to an instance of `MyNestedPragmaParser` before ESC/Java2 begins its processing.

4.5.7 Limitations

The following section lists Java and JML features not supported by the Java Translation Frontend.

Concerning Java, the features not supported by the frontend (and not supported by the formalisation at all) are

- concurrency features
- floating-point numbers
- string- and character literals
- nested classes

Concerning JML, the features not supported by the frontend (and not supported by the formalisation at all) are

- model programs
- redundant examples
- set comprehensions

Features not supported by the frontend (but part of the formalisation) because not supported by ESC/Java2:

- `\not_assigned` expression
- `\not_modified` expression (supported per se, but ESC/Java2 does not support the use of `\nothing` or `\everything` in the `\not_modified` clause)
- `\only_accessed`, `\only_assigned`, `\only_called`, `\only_captured` expressions
- `\lockset` (only keyword supported by ESC/Java2), `\max` expressions
- informal predicates (translated to true by ESC/Java2)
- `assume` statement (no support for variant with a given label)
- `debug` statement

Universe Type System [6] annotations are part of the formalisation. However, their support in ESC/Java2 is very limited. Thus, the frontend makes no guarantees for correct translation of Universe Type System modifiers.

Chapter 5

Conclusions

We conclude this thesis by looking at related and future work, and finally by recapitulating the results from our work.

5.1 Related Work

5.1.1 Bicolano

Bicolano [8] (BYte COde LANguage in cOq) is a formalisation of Java bytecode in the Coq Proof System. It is part of the Mobius project that concerns establishing a security architecture for Java on mobile platforms. Our formalisation of JML and Java source code is based on Bicolano, in particular we tried to reuse parts of their program syntax-, semantic domain- and machine arithmetic definitions.

5.1.2 Desugaring JML Method Specifications

The work of Raghavan and Leavens [14] on the desugaring of method specifications builds the basis of the rewriting of our extended syntax. In particular the splitting of the desugaring into individual steps has shown to be very useful for our rewritings as it allows to express them as individual transformation functions.

5.1.3 ESC/Java2

ESC/Java2 [2], [3], [11], the Extended Static Checker for Java version 2, is a programming tool that attempts to find common run-time errors in JML-annotated Java programs by static analysis of the program code and its formal annotations.
Similar to the method specification desugarings performed as part of the extended syntax rewriting, ESC/Java2 desugars JML method specifications in their frontend prior to static analysis. However, we prefer our approach of having the desugarings be part of the formalisation instead of a parsing/type-checking frontend. We think that our approach leads to more transparent desugarings that can be reordered, disabled or even be replaced.

5.1.4 jmldoc

jmldoc is a documentation generation tool similar to Javadoc [12] but extended to support the JML language. It was initially designed by Raghavan [13]. The tool also performs method specification desugaring prior to output generation. The desugarings are performed in Java (class `JmlTreeSurgery`) and directly operate on the parsing AST. The comments on the ESC/Java2 method specification desugarings also apply here.

5.1.5 bico

bico [4] is a tool which transforms Java class files into the Bicolano (see section 5.1.1) embedding of Java bytecode in Coq. For each class file it generates three files:

- a type file which contains the class name definition,
- a signature file which contains the signatures of all the fields and all the methods, and
- a main file which contains a translation of the Java bytecode found in the file to Bicolano formalisation and the proper definition of the class, with all its dependencies.

bico can be seen as the bytecode "pendant" to our Java translation frontend. bico has a similar approach to generating identifiers as natural numbers. However it is somewhat different in that it generates multiple output files per class file and that it summarises type information. bico has an easier job to do than our frontend, as Java bytecode is much less complex then the Java source code language. Even more, bico does not support a specification language like BML.

5.2 Future Work

The case study has proven the equivalence of the original invariants of a type and a single merged invariant with respect to semantic function `EvalInvariant`. In the future, it would be interesting to drive this idea even further and define a minimal syntax subset consisting of only those features of JML that are directly treated in the semantics. Rewritings of other syntactic constructs in terms of the minimal constructs could then be proven to conform to the defined semantics of JML. A further example of such a rewriting is the desugaring of a signals_only clause as a signals clause.

In the same spirit, the presented method specification desugarings could be extended to also perform desugaring steps 3.7 (Desugaring Inheritance and Refinement), 3.10 (Make Assignable and Signals Only Clauses the Same) and 3.11 (Desugaring Also Combinations) [14]. As these desugarings are at present part of the semantics, the desugarings could be proven to conform to the semantics as well.

Concerning the translation frontend, it would be worth evaluating, if replacing ESC/Java2 as parsing frontend is an option, as ESC/Java2 does not fully support JML[1]; an improvement of this situation in the near future does not seem likely.
The translation frontend itself could be improved to output even nicer code (make use of the pretty printer feature of alternative layouts, see section 4.5.4), or to support parsing in multiple steps [2], very much like C allows to compile multiple source files of the same program in multiple compiler "runs" through the use of object files.

[1]In particular, it does not fully support the Universe Type System.
[2]This would require to make the name registry permanent.

5.3 Conclusions

This thesis presents extensions to an existing formalisation of JML and Java in the Coq Proof Assistant. It defines an extended syntax superset within Coq that allows to embed JML-annotated Java classes in a form that closely resemble the original syntax. Furthermore it allows to represent the different forms of specification cases.

The rewritings of the extended syntax constructs in terms of basic syntax constructs consists of the rewriting of loop annotations, quantified expressions with multiple variables, implicit invariants due to non_null constraints and the desugaring of method specifications.
Concerning method specifications, the desugarings are designed and implemented as transformation functions in a way that conveniently allows to reorder or disable existing, or add further desugaring steps.

The Java translation frontend is deliberately kept rather small. Nevertheless is is designed and implemented in three parts – visitor, outputter and pretty printer – to keep the frontend maintainable and flexible for adaptations: being changes on the current output backend or even replacing the whole output backend.

The case study, as a last point, showed that proving on top of the formalisation is indeed feasible, once the necessary basis is laid. The basis consisted of the definition of a well-formedness predicate for boolean expressions, an associated lemma guaranteeing "correct" evaluation of boolean expressions and a lemma showing the equivalence of the syntactic `&&`-connective and the semantic $\wedge$-connective. Finally, the case study highlighted the need for higher-level tactics defined specifically for the needs of proofs within this formalisation.

Appendix A

Full Translation of Example 4.72

This appendix contains the full translation of example 4.72:

```java
package test;

public class A {
  private int tally = 0;

  void inc(int n) {
    tally += n;
  }

  int tally() {
    inc();
    return tally;
  }
}
```

```coq
Require Import JMLSyntax.
Require Import java_lang_Object.

Module N'.
Module java.
Module lang.
  Definition PKG_lang_ := 1.
  Definition C_Object_ := 1.

  Definition Object_ := (PKG_lang_, C_Object_).
End lang.
End java.
Module test.
Module jtf_only.
  Definition PKG_jtf_only_ := 1000.
  Definition C_A_ := 1001.

  Definition A_ := (PKG_jtf_only_, C_A_).
```

```
End jtf_only.
End test.
End N'.
Import N'.


Module java.
Module lang.
Module Object.
  Definition M_Object_1_ := 1.

  Definition M_Object_1 :=
    method_signature_decl java.lang.Object_ Void M_Object_1_ [].
End Object.
End lang.
End java.
Module test.
Module jtf_only.
Module A.
  Definition F_tally_ := 1002.
  Definition P_n_1004_ := 1003.
  Definition M_inc_1004_ := 1004.
  Definition M_tally_1005_ := 1005.
  Definition M_A_1006_ := 1006.

  Definition F_tally := field_signature_decl test.jtf_only.A_ int_t F_tally_.
  Definition P_n_1004 := param_decl int_t P_n_1004_.
  Definition M_inc_1004 :=
    method_signature_decl test.jtf_only.A_ Void M_inc_1004_
      [param_decl int_t P_n_1004_].
  Definition M_tally_1005 :=
    method_signature_decl test.jtf_only.A_ ((Return int_t)) M_tally_1005_ [].
  Definition M_A_1006 := method_signature_decl test.jtf_only.A_ Void M_A_1006_ [].
End A.
End jtf_only.
End test.

Module A'.
Import test.jtf_only.A.

Definition Impl :=
 class_decl [public] test.jtf_only.A_
  (extends java_lang_Object.Object'.Impl)
  no_implements
  [
    field_decl of_class [private] int_t F_tally_ (Some (int 0))
  ]
  no_model_fields
  [
    method_decl of_class
    []
    no_override
    no_modifiers Void M_inc_1004_ [param_decl int_t P_n_1004_] no_throws
    {{:
```

```
    1 :> stmt ((field test.jtf_only.A.F_tally (Some this)) += (param P_n_1004))
  :}};

  method_decl of_class
  []
  no_override
  no_modifiers (Return int_t) M_tally_1005_ [] no_throws
  {{:
    3 :> stmt (callR this test.jtf_only.A.M_inc_1004 [int 1]);
    4 :> returnE (field test.jtf_only.A.F_tally (Some this))
  :}};

  method_declS of_class Constructor
  []
  no_override
  [public; implicit_constructor] Void M_A_1006_ [] no_throws
  {{:
    6 :> super_call java.lang.Object.M_Object_1 []
  :}}
 ]
 (type_spec_decl [] [] [] [] [] [] [] []).

End A'.
```

Appendix B

Full Proof Script of Proof 1

```
Add LoadPath "./Library".
Add LoadPath "./Helpers".

Require Import JMLSyntax.
Require Import JMLSemantics.

Declare Module DOMAIN : SEMANTIC_DOMAIN with Module Prog := P.Program.
Export DOMAIN.

Module AT := AnnotationTable DOMAIN.
Export AT.

Import TYPESPEC_REWRITINGS.
Import ALL_REWRITINGS.

(**
 * First proof case study:
  (EvalInvariant p c m h fr ↔ EvalInvariant p c’ m h fr) where
  c’ contains one signle invariant that corresponds to the
  conjunction of all invariants in c.

  The proof is simplyfied in that rewriteEntity only performs the mergeInvariants step
  and EvalInvariant only quantifies over its own invariants instead of all
  (transitively) owned invariants.

  see FULL2BASIC.mergeInvariants
  author Andreas Kaegi
 *)

(**
  In the following sections we override and simplify some functions
  from the semantics/full2basic implementation to
  make this first proof case study as simple as possible.
 *)

 (* --- BEGIN OVERRIDE SECTION --- *)
```

```
    Definition rewriteTypeSpecs (ts:TYPESPEC.t) (enc:ENTITY.t) : TYPESPEC.t :=
      let invs0 := TYPESPEC.invariant ts in
      (* let inv1 := rewriteInvariants invs0 enc in *)
      let invs1 := mergeInvariants invs0 in
      TYPESPEC.setInvariant ts (invs1::nil).

    Definition rewriteEntity (e0:ENTITY.t) : ENTITY.t :=
      let ts1 := rewriteTypeSpecs (ENTITY.typeSpec e0) e0 in
      let methods1 := map (fun m ⇒ rewriteMethod m e0) (ENTITY.methods e0) in
      let e1 := ENTITY.setTypeSpec e0 ts1 in
      (* let e2 := ENTITY.setMethods e1 methods1 in *)
      e1.

    Definition EvalInvariant (p : Program) (c : Class) (m : Method) (h : Heap.t)
      (fr : Frame.t) : Prop :=
      if METHOD.isHelper m then
        True
      else
        ∀ inv ,
          In inv (TYPESPEC.invariant (ENTITY.typeSpec c)) →
    (* DefinedInvariant p c inv → *)
          EvalPredicate p (INVARIANT.pred inv) h fr.

(* --- END OVERRIDE SECTION --- *)

(** Every bool b is either true or false. *)

Lemma bool_tnd : ∀ b:bool, b=true ∨ b = false.
Proof.
  intro b.
  case b; auto.
Qed.

Require Import List.

(**
  Well-formedness predicate (Wf) for JML predicates (= boolean expressions).
  To simplify this case study, only the constants true/false, and the connectors and/or
  are considered.
  This definition could easily be generalised, though in this case,
  EvalExpression_pred has to be adapted, too.
 *)

 Inductive Wf_pred : Expression → Prop :=
  | Wf_false : Wf_pred false'
  | Wf_true  : Wf_pred true'
  | Wf_and : ∀ e1 e2, Wf_pred e1 → Wf_pred e2 → Wf_pred (e1 &&' e2)
  | Wf_or  : ∀ e1 e2, Wf_pred e1 → Wf_pred e2 → Wf_pred (e1 ||' e2).
Hint Resolve Wf_false Wf_true Wf_and Wf_or : Wf_base.

(** An invariant is well-formed if its predicate is well-formed *)

Definition Wf_inv : INVARIANT.t → Prop := fun inv ⇒ Wf_pred (INVARIANT.pred inv).
```

```
(** If an expression is well-formed, EvalExpression e is a boolean value. *)

Lemma EvalExpression_pred : ∀ p h fr e,
  Wf_pred e →
  ∃ b, EvalExpression p e h fr = Some (DOMAIN.Bool b).
Proof.
  intros p h fr.
  apply Wf_pred_ind.

  (* false' *)
  ∃ False; auto.

  (* true' *)
  ∃ True; auto.

  (* &&' *)
  intros e1 e2 Hwf1 H1 Hwf2 H2.
  destruct H1 as [b1 H1]; destruct H2 as [b2 H2].
  ∃ (b1 ∧ b2).
  simpl.
  rewrite H1; rewrite H2.
  reflexivity.

  (* ||' *)
  intros e1 e2 Hwf1 H1 Hwf2 H2.
  destruct H1 as [b1 H1]; destruct H2 as [b2 H2].
  ∃ (b1 ∨ b2).
  simpl.
  rewrite H1; rewrite H2.
  reflexivity.
Qed.

(**
  The resulting invariant of mergeInvariants is well-formed
  (under the assumption that the input invariants are well-formed.)
 *)

Lemma mergeInvariants_Wf : ∀ l,
  (∀ inv, In inv l → Wf_inv inv) →
  Wf_inv (mergeInvariants l).
Proof.
  unfold Wf_inv.
  induction l.
  (* base case l=nil *)
  intros.
  simpl.
  auto with Wf_base.

  (* step case l'=a::l *)
  intros.
  unfold mergeInvariants in IHl ⊢ *.
  simpl in IHl ⊢ *.
  apply Wf_and.
  (* case a *)
  apply H; intuition.
  (* case IH *)
  apply IH1; intuition.
Qed.


(**
  If EvalPredicate holds for a well-formed predicate p,
  and the result of EvalExpression for the same predicate is b,
  b holds.
 *)

Lemma EvalPredicate_EvalExpression : ∀ p h fr e b,
  Wf_pred e →
  EvalPredicate p e h fr →
  EvalExpression p e h fr = Some (DOMAIN.Bool b) →
  b.
Proof.
  intros.
  unfold EvalPredicate in H0.
  rewrite H1 in H0.
  assumption.
Qed.

(**
  Eval [e1 && e2] ↔ Eval [e1] ∧ Eval [e2]
 *)

Lemma EvalPredicate_and : ∀ p h fr e1 e2,
  Wf_pred e1 →
  Wf_pred e2 →
  (EvalPredicate p (e1 &&' e2) h fr ↔ (EvalPredicate p e1 h fr ∧ EvalPredicate p e2 h fr)).
Proof.
  intros p h fr e1 e2 Hwf1 Hwf2.

  assert (Wf_pred (e1 &&' e2)) as Hwf; auto with Wf_base.
  assert (∃ b1, EvalExpression p e1 h fr = Some (DOMAIN.Bool b1)) as Hex1.
    apply EvalExpression_pred; assumption.
  assert (∃ b2, EvalExpression p e2 h fr = Some (DOMAIN.Bool b2)) as Hex2.
    apply EvalExpression_pred; assumption.
  assert (∃ b, EvalExpression p (e1 &&' e2) h fr = Some (DOMAIN.Bool b)) as Hex.
    apply EvalExpression_pred; assumption.

  destruct Hex1 as [b1 Hb1].
  destruct Hex2 as [b2 Hb2].
  destruct Hex as [b Hb].

  inversion Hb as [H0].
  rewrite Hb1 in H0; rewrite Hb2 in H0.
  inversion H0 as [Heq]; clear H0.

  split.
  (* Eval [e1 && e2] → Eval [e1] ∧ Eval [e2] *)
  intro H.
  unfold EvalPredicate.
  rewrite Hb1; rewrite Hb2.
  rewrite <- Heq in Hb.
  apply (EvalPredicate_EvalExpression p h fr (e1 &&' e2)); assumption.

  (* Eval [e1] ∧ Eval [e2] → Eval [e1 && e2] *)
  intro H.
  destruct H as [H1 H2].
  unfold EvalPredicate.
  rewrite Hb.
  rewrite <- Heq.
  split.
  apply (EvalPredicate_EvalExpression p h fr e1); assumption.
  apply (EvalPredicate_EvalExpression p h fr e2); assumption.
Qed.

Lemma EvalPredicate_and_lr : ∀ p h fr e1 e2,
  Wf_pred e1 →
  Wf_pred e2 →
  EvalPredicate p e1 h fr →
  EvalPredicate p e2 h fr →
  EvalPredicate p (e1 &&' e2) h fr.
Proof.
  intros p h fr e1 e2 Hwf1 Hwf2 H1 H2.
  apply (EvalPredicate_and p h fr e1 e2 Hwf1 Hwf2).
  auto.
Qed.

Lemma EvalPredicate_and_rl : ∀ p h fr e1 e2,
  Wf_pred e1 →
  Wf_pred e2 →
  EvalPredicate p (e1 &&' e2) h fr →
  EvalPredicate p e1 h fr ∧ EvalPredicate p e2 h fr.
Proof.
  intros p h fr e1 e2 Hwf1 Hwf2 H.
  apply (EvalPredicate_and p h fr e1 e2 Hwf1 Hwf2).
  assumption.
Qed.

Lemma mergeInvariants_ok : ∀ p h fr l,
  (∀ inv, In inv l → Wf_inv inv) →
  ((∀ inv, In inv l → EvalPredicate p (INVARIANT.pred inv) h fr) ↔
  EvalPredicate p (INVARIANT.pred (mergeInvariants l)) h fr).
Proof.
  intros p h fr.
  induction l.
  (* base case l=nil *)
  intros.
  simpl.
  intuition.
  compute.
  trivial.

  (* step case a::l *)
  intro Hwf.
  set (inv' := mergeInvariants (a::l)).

  assert (Wf_pred (Make.TYPESPEC.INVARIANT.pred inv')) as Hwf_inv'.
  apply mergeInvariants_Wf with (l := a::l); assumption.

  split.
  (* (∀ inv, In inv (a::l) → EvalPredicate p inv) → EvalPredicate p inv' *)

  intro H. (* ∀ inv, In inv (a::l) → EvalPredicate p inv *)
  simpl.
  apply EvalPredicate_and_lr.
    (* Wf_pred a *)
    apply Hwf; intuition.
    (* Wf_pred inv, In inv l *)
    inversion Hwf_inv'; assumption.

    (* EvalPredicate a *)
    apply H; intuition.

    (* EvalPredicate inv, In inv l *)
    apply IHl.
    (* ∀ inv, In inv l → Wf_inv inv *)
    intros inv Hinv.
    apply Hwf; intuition.
    (* ∀ inv, inv l → EvalPredicate inv *)
    intros inv Hinv.
    apply H; intuition.

  (* EvalPredicate p inv' → ∀ inv, In inv (a::l) → EvalPredicate inv *)
  intros H inv Hin.
  simpl in H.
  apply EvalPredicate_and_rl in H.
  destruct H as [H1 H2].
  apply in_inv in Hin.
  destruct Hin as [Heq | Heq].
  (* EvalPredicate p a *)
  rewrite <- Heq; assumption.
  (* EvalPredicate p inv, In inv l *)
  apply IHl.
  (* Wf inv0, In inv0 l *)
  intros; apply Hwf; intuition.
  (* EvalPredicate p (mergeInvariants l) *)
  apply H2.
  assumption.
  (* Wf_pred a *)
  apply Hwf; intuition.
  inversion Hwf_inv'; assumption.
Qed.

Lemma mergeInvariants_ok_lr : ∀ p h fr l,
  (∀ inv, In inv l → Wf_inv inv) →
  (∀ inv, In inv l → EvalPredicate p (INVARIANT.pred inv) h fr) →
  EvalPredicate p (INVARIANT.pred (mergeInvariants l)) h fr.
```

```
Proof.
  intros p h fr l Hwf H.
  Check mergeInvariants_ok.
  apply (proj1 (mergeInvariants_ok p h fr l Hwf)).
  auto.
Qed.

Lemma mergeInvariants_ok_rl : ∀ p h fr l,
  (∀ inv, In inv l → Wf_inv inv) →
  EvalPredicate p (INVARIANT.pred (mergeInvariants l)) h fr →
  (∀ inv, In inv l → EvalPredicate p (INVARIANT.pred inv) h fr).
Proof.
  intros p h fr l Hwf H.
  apply (mergeInvariants_ok p h fr l Hwf).
  assumption.
Qed.

(*
(Make.TYPESPEC.TYPESPEC.invariant
           (Make.CLASS.typeSpec
              (Make.INTERFACE.setTypeSpec c
                 (Make.TYPESPEC.TYPESPEC.setInvariant
                    (Make.INTERFACE.typeSpec c)
                    (mergeInvariants
                       (Make.TYPESPEC.TYPESPEC.invariant
                          (Make.INTERFACE.typeSpec c)) :: [])))))
*)
Lemma setInvariant_def : ∀ ts inv,
  TYPESPEC.invariant (TYPESPEC.setInvariant ts inv) = inv.
Proof.
  intros.
  compute.
  destruct ts.
  reflexivity.
Qed.

Lemma setTypeSpec_invariant : ∀ e inv,
  TYPESPEC.invariant (
    ENTITY.typeSpec (
      ENTITY.setTypeSpec e (
        TYPESPEC.setInvariant (ENTITY.typeSpec e) inv
      )
    )
  ) = inv.
Proof.
  intros.
  compute.
  destruct e.
  apply setInvariant_def.
Qed.



Theorem merge_invariants_simple1 : ∀ p c c' m h fr,
  (∀ inv, In inv (TYPESPEC.invariant (ENTITY.typeSpec c)) → Wf_inv inv) →
  c' = rewriteEntity c →
  (EvalInvariant p c m h fr ↔ EvalInvariant p c' m h fr).
Proof.
  intros p c c' m h fr Hwf Hc'.
  split.
  (* → *)
  unfold EvalInvariant.

  intro H0.
  generalize (bool_tnd (METHOD.isHelper m)); intro H.
  case H.

  (* case METHOD.isHelper m *)
  intro H2.
  rewrite H2.
  trivial.

  (* case not METHOD.isHelper m *)
  intro H2; rewrite H2 in H0 ⊢ *.
  clear H H2.
  intros inv HIn.

  unfold rewriteEntity in Hc'.
  unfold rewriteTypeSpecs in Hc'.
  simpl in Hc'.

  rewrite Hc' in HIn.

  set (invs := (mergeInvariants
                    (TYPESPEC.TYPESPEC.invariant
                       (ENTITY.typeSpec c)) :: [])) in * ⊢ *.

  rewrite (setTypeSpec_invariant c invs) in HIn.

  elim HIn.

  intro Hinv.
  rewrite <- Hinv.
  apply mergeInvariants_ok_lr; assumption.

  intro Hdis.
  elim Hdis.

  (* <- *)
  unfold EvalInvariant.
  intro H0.
  generalize (bool_tnd (METHOD.isHelper m)); intro H.
  case H.

  (* case METHOD.isHelper m *)
  intro H2; rewrite H2; trivial.

  (* case not METHOD.isHelper m *)
  intro H2; rewrite H2 in H0 ⊢ *.
  clear H H2.

  unfold rewriteEntity in Hc'.
  unfold rewriteTypeSpecs in Hc'.
  simpl in Hc'.
  rewrite Hc' in H0.

  set (invs := TYPESPEC.invariant (Make.ENTITY.typeSpec c)) in * ⊢ *.
  set (invs' := (mergeInvariants invs :: [])) in * ⊢ *.

  rewrite (setTypeSpec_invariant c invs') in H0.

  intros inv Hinv.
  apply mergeInvariants_ok_rl with (l := invs).
  apply Hwf.
  apply H0.
  apply in_eq.
  assumption.
Qed.
```

Listing B.1: Full proof script of case study proof 1

Bibliography

[1] Ives Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[2] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *FMCO'2005*, volume 4111 of *LNCS*, pages 342–363. Springer, 2005.

[3] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas*, pages 322–328. CSREA Press, 2002.

[4] MOBIUS Consortium. Deliverable 4.2: Certificates. Available online from `http://mobius.inria.fr`, 2007.

[5] Coq development team. *The Coq Proof Assistant reference manual, version 8.2*, February 2009.

[6] Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[8] Marieke Huisman and Gustavo Petri. BicolanoMT: a formalization of multi-threaded Java at bytecode level. In *BYTECODE*, ENTCS. Elsevier, 2008.

[9] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML. Technical report, Department of Computer Science, Iowa State University, 2001.

[10] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. JML reference manual (DRAFT). Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, May 2008.

[11] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Report Technical Note 2000-02, Compaq Systems Research Center, 2000.

[12] Sun Microsystems, Inc. Javadoc. http://java.sun.com/j2se/javadoc/.

[13] Arun D. Raghavan. Design of a JML documentation generator. Technical Report TR #00-12, Department of Computer Science, Iowa State University, March 2000.

[14] Arun D. Raghavan and Gary T. Leavens. Desugaring JML method specifications. Technical Report TR #00-03e, Department of Computer Science, Iowa State University, March 2000.

[15] Philip Wadler. A prettier printer. In *Journal of Functional Programming*, pages 223–244. Palgrave Macmillan, 2003.