Certification of a Tool Chain for Deductive Program Verification
Paolo Herms

UNIVERSITÉ DE PARIS-SUD

École doctorale d'Informatique

THÈSE

présentée
pour obtenir le

Grade de Docteur en Sciences de l'Université Paris-Sud
Discipline : Informatique

PAR

Paolo HERMS

———✕———

SUJET :

Certification of a Tool Chain for Deductive Program Verification

soutenue le 14 janvier 2013 devant la commission d'examen

MM.	Roberto	Di Cosmo	Président du Jury
	Xavier	Leroy	Rapporteur
	Gilles	Barthe	Rapporteur
	Emmanuel	Ledinot	Examinateur
	Burkhart	Wolff	Examinateur
	Claude	Marché	Directeur de Thèse
	Benjamin	Monate	Co-directeur de Thèse
	Jean-François	Monin	Invité

Résumé

Cette thèse s'inscrit dans le domaine de la vérification du logiciel. Le but de la vérification du logiciel est d'assurer qu'une implémentation, un programme, répond aux exigences, satisfait sa spécification. Cela est particulièrement important pour le logiciel critique, tel que des systèmes de contrôle d'avions, trains ou centrales électriques, où un mauvais fonctionnement pendant l'opération aurait des conséquences catastrophiques.

Les exigences du logiciel peuvent concerner la sûreté ou le fonctionnement. Les exigences de sûreté, tel que l'absence d'accès à la mémoire en dehors des bornes valides, sont souvent implicites, dans le sens que toute implémentation est censée être sûre. D'autre part, les exigences fonctionnelles spécifient ce que le programme est censé faire. La spécification d'un programme est souvent exprimée informellement en décrivant en anglais la mission d'une partie du code source. La vérification du programme se fait alors habituellement par relecture manuelle, simulation et tests approfondis. Par contre, ces méthodes ne garantissent pas que tous les possibles cas d'exécution sont capturés.

La preuve déductive de programme est une méthode complète pour assurer la correction du programme. Ici, un programme, ainsi que sa spécification formalisée à l'aide d'un langage logique, est un objet mathématique et ses propriétés désirées sont des théorèmes logiques à prouver formellement. De cette façon, si le système logique sous-jacent est cohérent, on peut être complètement sûr que la propriété prouvée est valide pour le programme en question et pour n'importe quel cas d'exécution.

La génération de *conditions de vérification* est une technique censée aider le programmeur à prouver les propriétés qu'il veut sur son programme. Ici, un outil (VCG) analyse un programme donné avec sa spécification et produit une formule mathématique, dont la validité implique la correction du programme vis à vis de sa spécification, ce qui est particulièrement intéressant lorsque les formules générées peuvent être prouvées automatiquement à l'aide de *solveurs SMT*.

Cette approche, basée sur des travaux de Hoare et Dijkstra, est bien comprise et prouvée correcte en théorie. Des outils de vérification déductive ont aujourd'hui acquis une maturité qui leur permet d'être appliqués dans un contexte industriel où un haut niveau d'assurance est requis. Mais leurs implémentations doivent gérer toute sorte de fonctionnalités des langages et peuvent donc devenir très complexes et contenir des erreurs elles mêmes – au pire des cas affirmer qu'un programme est correct alors qu'il ne l'est pas. Il se pose donc la question du niveau de confiance accordée à ces outils.

Le but de cette thèse est de répondre à cette question. On développe et certifie, dans le système Coq, un VCG pour des programmes C annotés avec ACSL, le langage logique pour la spécification de programmes ANSI/ISO C. Notre première contribution est la formalisation d'un VCG exécutable pour le langage intermédiaire *Whycert*, un langage impératif avec boucles, exceptions et fonctions récursives, ainsi que sa preuve de correction par rapport à la *sémantique opérationnelle bloquante à grand pas* du langage. Une deuxième contribution est la formalisation du langage logique ACSL et la sémantique des annotations ACSL dans Clight de Compcert. De la compilation de programmes C annotés vers des programmes Whycert et sa preuve de préservation de la sémantique combiné avec une axiomatisation en Whycert du modèle mémoire Compcert résulte notre contribution principale : une chaîne intégrée certifiée pour la vérification de programmes C, basée sur Compcert. En combinant notre résultat de correction avec celui de Compcert, on obtient un théorème en Coq qui met en relation la validité des l'obligations de preuve générées avec la sûreté du code assembleur compilé.

Abstract

This thesis belongs to the domain of software verification. The goal of verifying software is to ensure that an implementation, a program, satisfies the requirements, the specification. This is especially important for critical computer programs, such as control systems for air planes, trains and power plants. Here a malfunctioning occurring during operation would have catastrophic consequences.

Software requirements can concern safety or functioning. Safety requirements, such as not accessing memory locations outside valid bounds, are often implicit, in the sense that any implementation is expected to be safe. On the other hand, functional requirements specify what the program is supposed to do. The specification of a program is often expressed informally by describing in English or some other natural language the mission of a part of the program code. Usually program verification is then done by manual code review, simulation and extensive testing. But this does not guarantee that all possible execution cases are captured.

Deductive program proving is a complete way to ensure soundness of the program. Here a program along with its specification is a mathematical object and its desired properties are logical theorems to be formally proved. This way, if the underlying logic system is consistent, we can be absolutely sure that the proven property holds for the program in any case.

Generation of *verification conditions* is a technique helping the programmer to prove the properties he wants about his programs. Here a VCG tool analyses a program and its formal specification and produces a mathematical formula, whose validity implies the soundness of the program with respect to its specification. This is particularly interesting when the generated formulas can be proved automatically by external *SMT solvers*.

This approach is based on works of Hoare and Dijkstra and is well-understood and shown correct in theory. Deductive verification tools have nowadays reached a maturity allowing them to be used in industrial context where a very high level of assurance is required. But implementations of this approach must deal with all kinds of language features and can therefore become quite complex and contain errors – in the worst case stating that a program correct even if it is not. This raises the question of the level of confidence granted to these tools themselves.

The aim of this thesis is to address this question. We develop, in the Coq system, a certified verification-condition generator (VCG) for ACSL-annotated C programs.

Our first contribution is the formalisation of an executable VCG for the *Whycert* intermediate language, an imperative language with loops, exceptions and recursive functions and its soundness proof with respect to the *blocking big-step operational semantics* of the language. A second contribution is the formalisation of the ACSL logical language and the semantics of ACSL annotations of Compcert's Clight. From the compilation of ACSL annotated Clight programs to Whycert programs and its semantics preservation proof combined with a Whycert axiomatisation of the Compcert memory model results our main contribution: an integrated certified tool chain for verification of C programs on top of Compcert. By combining our soundness result with the soundness of the Compcert compiler we obtain a Coq theorem relating the validity of the generated proof obligations with the safety of the compiled assembly code.

Contents

Chapter 1

Introduction

This thesis belongs to the domain of software verification. The goal of verifying software is to ensure that an implementation, a program, satisfies the requirements, the specification. This is especially important for critical computer programs, such as control systems for air planes, trains and power plants. Here a malfunctioning occurring during operation would have catastrophic consequences. There are numerous examples of software failures in the past. The interested reader may refer to a very comprehensive list composed by Nachum Dershowitz. [1]

Software requirements can concern safety or functioning. Safety requirements, such as not accessing memory locations outside valid bounds, are often implicit, in the sense that any implementation is expected to be safe. Many high-level programming languages ensure the main safety requirements by providing to the programmer an abstraction of the machine that disallows unsafe operations. On the other hand low-level languages, like the C programming language, grant the programmer the full access to the machine exposing him also to the full risk. Unfortunately the use of low-level languages can often not be avoided – especially in the implementations of aforementioned control systems. Indeed the C language remains nowadays the language of choice for developing critical embedded software.

1.1 The C language and its semantics

The C programming language was initially developed by Dennis M. Ritchie at Bell Labs starting from 1969. Only 10 years later its semantics was informally specified and one has to wait until 1989/1990 for C to become an ANSI/ISO standard. Nevertheless, the C norm is still ambiguous and leaves a certain degree of freedom to the compiler. For instance the evaluation order of function arguments is "unspecified", i.e. code like `f(g(),h())` is completely legal but the compiler may call `g()` and `h()` in any order. More subtle, the semantics of multiple side-effects between two sequence points is "undefined", i.e. code like `++i + i++` is illegal but the compiler doesn't have to reject it and may interpret it arbitrarily. Moreover the notion of sequence point is quite complex and disputed. [2]

C programs containing such problematic code should generally ruled out by a static analyser.

But apart from issues coming from the lack of precise semantics, the most important issues come from the low-level nature of the language itself. *Aliasing* is such an issue and defines the situation in which a memory location can be accessed through different symbolic names in the program. For instance, the following assertion is not generally true:

---

1. `http://www.cs.tau.ac.il/~nachumd/horror.html`
2. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n993.htm`

```
*x=1;
*y=2;
assert (*x + *y == 3);
```

as it is sufficient that the code was preceded by `x= y;`. Harder to find, it is not even generally true that

```
*p=5;
assert (*p == 5);
```

and this code may crash if it was preceded by `p= (int*)&p;`.

1.2   Deductive Program verification

Functional requirements specify what the program is supposed to do. The specification of a program is often expressed informally by describing in English or some other natural language the mission of a part of the program code.

A way of formalising the specification is through a logical language extending the programming language. Consider the following example of a C function `max` along with its formal specification given in the ANSI/ISO C Specification Language (ACSL), its *function contract*:

```
/*@   ensures \result >= x && \result >= y;
      ensures \result == x || \result == y;
      assigns \nothing;
*/
int max (int x, int y) { return (x > y) ? x : y; }
```

This is a simple implementation in C of a function returning the largest of two given integer numbers `x` and `y`. The specification of this function is expressed by saying that the result is larger or equal than both `x` and `y` and either equal to `x` or equal to `y`. Additionally, it is specified that the function does not modify any memory location.

Usually program verification is then done by manual code review, simulation and extensive testing. But this does not guarantee that all possible execution cases are captured. There can be a rare sequence of external inputs or events the programmer did not think of and therefore does not test his program with. And being rare, even randomised black-box testing may miss it.

Deductive program proving is a complete way to ensure soundness of the program. Here a program is a mathematical object and its desired properties are logical theorems to be formally proved. This way, if the underlying logic system is consistent, we can be absolutely sure that the proven property holds for the program in any case.

Since Alan Turing showed that the halting problem is undecidable, we know that it is not possible to automatically prove all properties of a program in the general case. Therefore, we need techniques to help the programmer to prove the properties he wants about his programs.

Generation of *verification conditions* is such a technique. Here a VCG tool analyses a program and its formal specification and produces a mathematical formula, whose validity implies the soundness of the program with respect to its specification.

For instance, the following formula, given in first-order logic with arithmetic, is a verification condition for the `max` function of the previous example:

$$\forall z_1 \forall z_2 \ z_1 > z_2 \implies ((z_1 = z_1 \lor z_1 = z_2) \land z_1 \geq z_1 \land z_1 \geq z_2)$$
$$\land z_1 \leq z_2 \implies ((z_2 = z_1 \lor z_2 = z_2) \land z_2 \geq z_1 \land z_2 \geq z_2)$$

If we can prove this formula logically valid, then we can be sure that the initial C function respects its specification. This is particularly interesting if the generated formulas can be proved automat-

ically: Automatic provers systematically explore the state space of logical formulas to return a "yes/no/don't know" answer to the satisfiability or validity of logical formulas. When automatic provers fail to prove a valid verification condition, proof assistants provide a way to complete the proof. Proof assistants provide a mechanical support for manually proving hard theorems, usually requiring expert users.

This approach is based on works of Hoare and Dijkstra and is well-understood and shown correct in theory. Deductive verification tools have nowadays reached a maturity allowing them to be used in industrial context where a very high level of assurance is required. This is illustrated by experiments done in the context of avionics by Airbus France [54] and Dassault Aviation [53]. But implementations of this approach must deal with all kinds of language features like pointer aliasing as explained above and can therefore become quite complex and contain errors – in the worst case stating that a program correct even if it is not. This raises the question of the level of confidence granted to these tools themselves. The aim of this thesis is to address this question.

1.3 Short History, Overview of the State of the Art

Early Works   The pioneer implementation of a tool for deductive verification was made in ESC/modula 3 [23] introducing the notion of function contracts. These works are in turn based on works for the Simplify theorem prover performed as early as 1981 [48]. The vocabulary of "contract" comes from the Eiffel [43] language developed in the same period, whose aim was to perform runtime checking of the requirements given by those contracts.

Verification of Purely Functional Programs   At approximately the same period of time there were a lot of works about proving of purely functional programs inside theorem proving environments, as in PVS [51], ACL2 [31], HOL88, Isabelle/HOL [49], HOL4, HOL-light and Coq [52].

Verification of Non Purely Functional Programs in General Purpose Proof Assistants   Probably the first attempt to prove non purely functional programs inside a general purpose proof assistant was the SunRise system in 1995 [29] where a simple imperative language is defined in HOL, with a formal operational semantics. A set of Hoare-style deduction rules are then shown valid. A SunRise program can then be specified using HOL assertions and proved in the HOL environment. What has to be noticed is that the programming language is deeply embedded in the logical framework, i.e. the abstract syntax of the language is defined as a data type. A program is then an object of the logical system one can state properties about. On the other hand the specifications are not deeply, but shallowly embedded: they are formulas of the framework. As a consequence proof of a program must be done inside the HOL proof assistant without the help of any external automatic program prover like Simplify.

Later Norrish formalised the C programming language [50] in HOL including a set of Hoare style deduction rules allowing to prove properties of C programs inside HOL. The Isabelle/HOL formalisation used in the L4-verified project [32] was based on this work and allowed to certify a C implementation of a highly secured OS kernel.

Other works proposed proofs of non purely functional programs on top of general purpose proof assistants, like Mehta/Nipkow [42] and Schirmer [55] on top of Isabelle/HOL, Ynot [46, 19] and CFML [17] on top of Coq, which can deal with "pointer" programs via separation logic, and also support higher-order functions.

This gives a very high level of confidence about the program's soundness but theorem provers are usually not well suited for program proving as they don't give much assistance for proving programs and most proofs must be carried out manually. Also, using a theorem prover would

make us dependent of that particular system, whereas we would like to mix different systems, playing on each system's strengths.

Standalone Dedicated Verification Tools   A different approach for program verification has leaded to the implementation of dedicated tools, not based on proof assistants but in the spirit of ESC/Modula 3.

These approaches provide standalone verification condition generators automatically producing verification conditions, usually by means of variants of Dijkstra's weakest precondition calculus. This is the case of ESC/Java [20], B [1] ; the Why platform [26] and its Java [41] and C [25, 44, 21] front-ends (Jessie) ; and Spec# [4] and VCC [22] which are front-ends to Boogie [3]. The role of these front-ends is to transform input programs, where pointer aliasing is allowed, into the alias-free intermediate languages, like Why, using a so-called memory model, which typically encodes the operations on the memory heap by functional updates [40]. Being independent of any underlying proof assistant, these tools analyse programs where formal specifications are given in ad-hoc annotation language such as JML [15, 16] and ACSL [8].

Trusted Code Base   Up to now the aforementioned standalone tools have never been formally proved to be sound. In other words proving a program with such a tool means proving it up to the soundness of the implementation of the tool itself. The implementation needs thus to be trusted, even if it can become quite complex. We say it does not always respect the "de Bruijn criterion", which Freek Wiedijk defines as having the correctness of the system as a whole depends on the correctness of a very small kernel. If we call the *trusted code base* (TCB) the part of the implementation of a tool on which the soundness depends on, then we can say that the standalone approach has a large TCB whereas the approaches on top of proof assistant have a significantly smaller one.

For VCC, recent works [14] aim at validating the axiomatisation, that the generated proof obligations refer to, with respect to an ad-hoc formalisation of a C memory model (CVM). The original axiomatisation contained 900 axioms, some of which where discovered inconsistent, showing the importance of such a work.

To summarise, one can distinguish two main kinds of deductive verification approaches. The first kind is characterised by the use of a deep embedding of the input programming language in a general purpose proof assistant, whereas the second kind takes the form of dedicated standalone tools, offering a higher degree of proof automation but with a larger TCB.

Certifying versus Certified Compilers   A formal certification of the correctness of the source program is even more desirable in presence of technologies permitting to relate this to the correctness of the compiled assembly program.

A famous line of work is the *Proof Carrying Code* approach (PCC) [47, 2]. The general idea is to have executable code accompanied with a *certificate* which guarantees that this code will not make any runtime errors (such as invalid memory access) or satisfy particular functional properties (like respect of safety policies). Such a certificate must be produced by the compiler, thus called a *certifying compiler*. Originally this is limited to simple properties that can be established directly by the compiler, for instance, type and memory safety.

To go beyond simple properties, one should be able to take into account requirements given in the source by means of code contracts. This was proposed by Barthe et al. [5, 6, 7] for the case of Java programs annotated using JML, compiled to Java byte code annotated using BML.

A way to go even further is to develop a *certified compiler*, i.e. a compiler that is proved to preserve the semantics of the source code during the compilation towards assembly code. This

therefore guarantees the preservation of any general property from the source code to the assembly code.

The Verisoft project [3] formalises a subset of C, called C0, and develops both a certified compiler and a proof environment on top of Isabelle/HOL [34]. Independently, a certified optimising compiler for C has been developed and proved in Coq by Leroy, Blazy et al. [35, 12, 36] within the Compcert project [4]. As this compiler covers a sufficiently large part of the C language and moreover produces a sufficiently efficient executable code, its use is considered for production in an industrial context.

Towards Certified Verification Tools The Compcert project showed that it is nowadays possible to develop real-world applicable tools inside a theorem prover. Similarly there are attempts to develop certified verification tools, such as certified abstract interpreters developed in Coq by Besson et al. [11]. The Verasco project [5] precisely aims at developing certified static analysis tools on top of Compcert operational semantics .

The question of trusting formal methods was addressed in work package 5 of the U3CAT project [6]. This thesis was in part in the framework of that project.

1.4 Contributions of this thesis

In this work we develop, in the Coq system, a certified verification condition generator for ACSL-annotated C programs.

Our first contribution is the formalisation of an executable verification-condition generator for the Whycert intermediate language, an imperative language with loops, exceptions and recursive functions and its soundness proof with respect to the *blocking big-step operational semantics* of the language (Chapter 3).

Our second contribution is the formalisation of the ACSL logical language and the semantics of ACSL annotations of Compcert's Clight (Chapter 4).

From the compilation of ACSL annotated Clight programs to Whycert programs and its semantics preservation proof combined with a Whycert axiomatisation of the Compcert memory model results our main contribution (Chapter 5): an integrated certified tool chain for verification of C programs on top of Compcert. By combining our soundness result with the soundness of the Compcert compiler we obtain a Coq theorem relating the validity of the generated proof obligations with the safety of the compiled assembly code.

With respect to the state of the art presented above, our work aims at combining the advantages of the two families of approaches of deductive verification: we have a standalone verification tool able to interact with external automated provers which is, at the same time, proved formally correct with a small TCB.

---

3. `http://www.verisoft.de`
4. `http://compcert.inria.fr`
5. `http://verasco.imag.fr`
6. `http://frama-c.com/u3cat/wp5.html`

Chapter 2

Preliminaries

2.1   A Short Introduction to the Coq Proof Assistant

Coq is specified in the Coq Reference Manual[56] and introduced in the official Coq Tutorial[30] as well as in CoqArt book[10]. In this section we introduce some features we use throughout this development. The main constructs of the Coq language we use in this thesis are *algebraic data types*, *structurally recursive functions* on such types and *inductive* and *co-inductive predicates*.

One of the easiest examples of an algebraic data type is the list:

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A → list A → list A
```

Such an inductive definition allows defining of structurally recursive functions over the type, e.g. the function computing the length of a given list.

```
Fixpoint length A (l : list A) : nat :=
  match l with
    | nil ⇒ 0
    | cons h q ⇒ 1 + length q
  end.
```

In Coq, predicates can be defined inductively just as data types, except that their *sort*, i.e. the type of their type, is `Prop` instead of `Type`. See for instance the definition of the predicate relating any two lists that are permutations of each other:

```
Inductive Permutation (A : Type) : list A → list A → Prop :=
  | perm_nil : Permutation nil nil
  | perm_skip : forall (x : A) (l l' : list A),
                  Permutation l l' → Permutation (cons x l) (cons x l')
  | perm_swap : forall (x y : A) (l : list A),
                  Permutation (cons y (cons x l)) (cons x (cons y l))
  | perm_trans : forall l l' l'' : list A,
                  Permutation l l' → Permutation l' l'' →
                    Permutation l l''
```

The predicate is defined by the *least fixed point* over these four rules. That is, among all the predicates that satisfy the four clauses above, there is one smaller than all the others (i.e. contains less pairs of lists). The existence of this smallest fixed point is guaranteed by the positivity conditions that are checked by the Coq kernel.

Coq offers the notion of co-induction, which is the dual of induction where least fixed points are replaced by greatest fixed points. A typical example found in tutorials about co-induction is the data type of infinite lists, also known as streams. In this thesis we will not use any co-inductive data types, only co-inductive predicates, to talk about non terminating computations.

We illustrate the definition of co-inductive predicates and proofs by co-inductions on a variant of the Collatz problem [33]. We define a sequence of natural numbers as follows. Given an arbitrary starting number x_0, we pose

$$x_n \text{ is even} \quad \to \quad x_{n+1} = x_n/2$$
$$x_n \text{ is odd} \quad \to \quad x_{n+1} = 5x_n + 1$$

If x_n is 1 the sequence stops.

In the original Collatz problem, the coefficient 5 is replaced by 3 and the still open conjecture is that for every starting number the sequence is finite.

For coefficient 5 there are infinite sequences, e.g.

$$13, 66, 33, 166, 83, 416, 208, 104, 52, 26, 13, \dots$$

Now we can formalise in Coq the predicate `P x` that is true when `x` starts an infinite sequence, by a co-inductive definition:

```
CoInductive P n: Prop :=
| Even: n > 1 → even n → P (div2 n) → P n
| Odd: n > 1 → odd n → P (5*n+1) → P n.
```

In other words, `P n` holds if either `n` is even and `P (n/2)` holds, or `n` is odd and `P (5*n+1)` holds.

Proving a goal about `P` typically involves the use of the `cofix` tactic, which introduces the current goal as an hypothesis with the given name. However this hypothesis can be used only after having applied a constructor of the predicate. For example the following proof attempt is rejected by the Coq type checker (on the `Qed` command):

```
Goal P 13.
  cofix CO. apply CO.
Qed.
```

Instead one have to construct an infinite proof tree by exposing the cycle:

```
Ltac even := apply Even; [auto with arith|auto with arith|simpl].
Ltac odd := apply Odd; [auto with arith|auto with arith|simpl].

Goal P 13.
  cofix CO.
    odd. even. odd. even. odd. even. even. even. even. even.
  apply CO.
Qed.
```

or, more simply:

```
Goal P 13.
  cofix CO. repeat ((even;[])||(odd;[]); auto).
Qed.
```

2.1.1 Notations

In order to improve the visual aspect of our development, we make use of several notations. Notations make Coq definitions more readable which is particularly important in case of specifications as they need to be agreed on and thus belong to the trusted code base.

In order to conveniently deal with partial functions, i.e. functions returning a value of type `option A`, we define a notation for the error monad.

```
Notation "'let?' x := e1 'in' e2" :=
   (match e1 with None ⇒ None | Some x ⇒ e2 end).
```

Notations for Inference Rules

A class of notations particularly involved in making specifications more readable are our notations for inference rules. It is clear that when defining type systems or operational semantics by an inductive or co-inductive type then this can be seen as inference system the rules are given by the constructors. To make this even clearer we just need to write the conclusions of the rules under a line. For instance the permutation relation over lists of the Coq standard library can be rewritten as follows:

```
Inductive Permutation A: list A → list A → Prop :=
| perm_nil:
            ∅
   --------------------
     Permutation [] []
|
 perm_skip x l l' :

         Permutation l l'
   -----------------------------
     Permutation (x::l) (x::l')

| perm_swap x y l :
                  ∅
   ----------------------------------
     Permutation (y::x::l) (x::y::l)

| perm_trans l l' l'' :

     Permutation l l'   ·   Permutation l' l''
   ---------------------------------------------
             Permutation l l''
.
```

For co-inductive definitions we also provide a double-line notation.

```
CoInductive P : nat → Prop :=
| Even n:
          P n
      ===========
        P (2*n)
| Odd n:
     P (5*(2*n+1)+1)
   ===================
        P (2*n+1)
.
```

These notations are defined as any other Coq notation:

```
Notation "x '·' y" := (x → y).
Notation "x  -------------  y" := (x → y).
Notation "'∅'  -------------  y" := (y).
```

In order to support horizontal lines of variable lengths we just add one of such declaration per length between 5 and 100.

   Thanks to these notations the rules given in this thesis are directly copied from the Coq development.

Scopes

   The same notation can have different meanings according to the current *interpretation scope*. For instance, the `+` infix symbol denotes integer addition in the `Z_scope` and real addition in the `R_scope`.

   Scopes can be opened and closed globally using `Open Scope` and `Close Scope`. To change scope within sub-expressions, these can be suffixed with a scope delimiter using the `%` syntax.

```
Open Scope positive_scope.
Definition p1 := 1.
Definition p2 := p1 + p1.
Open Scope Z_scope.
Definition z1 := 1.
Definition z2 := z1 + Zpos p1.
Check z1 < z2 ∧ (p1 < p2)%positive.
```

2.1.2 Coq Sections

   Coq sections help defining parametric definitions. Just like modules, sections group variable declarations and type, constant and function definitions together. But on closing section, every Coq variable declared inside a section becomes a formal parameter to every definition relying on that variable. Consider the following example:

```
Section S.
  Variable n : nat.
  Definition f m := n + m.
  Definition g m := m + m.
  Definition h := f 5.
  Definition i := g 6.
  Check f: nat → nat.
  Check g: nat → nat.
  Check (h,i) : nat * nat.
End S.
Check f: nat → nat → nat.
Check g: nat → nat.
Check (h,i) : (nat → nat) * nat.
```

Inside the section `S`, the functions `f` and `g` have the same type `nat -> nat` and the constants `h` and `i` have simply type `nat`. On closing the section, all the definitions are generalised with respect to the variable `n`, so `f` requires it as an additional parameter.

2.1.3 Coercions

In Coq integers are defined as the following inductive type

```
Inductive Z := Z0 : Z | Zpos : positive → Z | Zneg : positive → Z.
```

In a development that works extensively with both, `positive` and `Z`, it could be convenient to consider `positive` a sub type of `Z` such that one can write a `positive` where a `Z` would be expected. This is possible in Coq by declaring a *coercion.*

```
Coercion Zpos: positive ↣ Z.
```

That is, whenever a `Z` is expected and a `positive` is found, Coq automatically inserts the constructor `Zpos`.

Continuing the example of Section 2.1.1 we can now write

```
Check z1 < p2.
```

The shorthand syntax `:>` is available to directly declare constructors of a new inductive type as coercions. With that we could have defined `Z` as

```
Inductive Z :=  Z0 : Z | Zpos :> positive → Z | Zneg : positive → Z.
```

2.1.4 Dependent Types

Coq's underlying formal language is a functional programming language with a powerful type system. For instance we can define `vector n` the type of lists of length `n`.

```
Inductive vector (A : Type) : nat → Type :=
| Vnil : vector A 0
| Vcons : A → (forall n : nat, vector A n → vector A (1 + n)).
```

Such a definition guarantees that a list of type `vector 5` always has exactly 5 elements. This simplifies the definition of function accessing such a vector, for instance the function returning the first element.

```
Definition vhd A n (v: vector A (1 + n)) :=
  match v with
    | Vcons x n x0 ⇒ x
  end.
```

Notice that the definition does not need to handle the case where there is no first element, as this situation is excluded by typing.

2.1.5 The `Program` Environment

The definition of the previous example is especially simple because Coq's type checker can automatically infer the type of the expression and the absurd branch of the case expression. This is not always possible, as it may be necessary to *prove* that a given case is absurd or that two types are equivalent. The `Program` environment aims at automatically inserting these proofs. Consider the following examples, which are automatically completed:

```
Program Fixpoint lhd {A} (l: list A) (_ : List.length l ≠ 0) :=
  match l with
    | [] ⇒ !
    | h::q ⇒ h
  end.

Program Fixpoint mk_vector {A} n (l: list A) (_ : List.length l = n)
  : vector A n :=
  match l with
    | [] ⇒ Vnil
    | h::q ⇒ Vcons h (mk_vector (pred n) q _)
  end.
```

In the first example, the `!` marked case is absurd as it contradicts the hypothesis that the length of the list is not zero. In the following the return type is not trivial. In each case `Program` automatically inserts a cast to the desired return type.

2.1.6 Extraction to OCaml Programs

The Coq extraction mechanism [39, 38] creates executable OCaml or Haskell implementations from Coq definitions. This is a mostly syntactic translation, as Coq definitions are lambda terms as usual in functional programming languages. However non computational contents and unsupported typing features are removed by the extraction. For instance this is the extracted code for the above examples:

```
let lhd = function
| Nil -> assert false (* absurd case *)
| Cons (h, q) -> h

type 'a vector =
| Vnil
| Vcons of 'a * nat * 'a vector

let rec mk_vector n = function
| Nil -> Vnil
| Cons (h, q) -> Vcons (h, (pred n), (mk_vector (pred n) q))
```

Notice that the hypotheses about the length of the lists have been removed. Similarly the type definition of `vector` does not depend on its length. In general it is therefore not safe to call Coq-extracted functions by hand-written OCaml code.

2.2 Compcert

In this thesis we consider the Coq formalisation of C provided by the Compcert project. This formalisation addresses some of the issues explained in the introduction.

In particular Compcert formalises two front-end languages: Clight, a normalised version of C with side-effect free expressions, and CompcertC, aiming to allow ambiguous expressions and to formalise the notion of sequence point. In this work we refer to the former. Many aspects of C are shared between the front-end languages so the following considerations are true for both of them.

Several semantic elements are more precise in Compcert than in generic C. Currently Compcert fixes the word size to 32-bits and arithmetic operations are all defined modulo this word size. Also the order of the fields in a structure is fixed and fields aligned using alignment bits. The function computing the offset for a given field is thus completely defined.

On the other hand, more importantly, the Compcert memory model formalises some of the abstract aspects of the C memory. First, the Compcert memory model defines memory states as collections of blocks, where each block is associated an array of atomic bytes. Compared with a more hardware-like view of memory as a single, big array of bytes, this allows capturing the fact that blocks of memory resulting from different allocations are necessarily separate and the corresponding pointers incomparable. Also, a byte is formalised as being either a real 8-bit integer or a portion of a pointer. This captures the fact that bit-level access to portions of machine integers and floats is possible, whereas pointers should be atomic and their representation opaque.

To summarise the Compcert memory model is a perfect base to abstractly reason about memory operations in program proofs.

2.3 The ACSL Specification Language

ACSL is specified in the reference manual [8]. It can express a wide range of functional properties: from low-level properties, e.g. about the validity of some given pointer, to high-level properties, e.g. that a given list is sorted. Figure 2.1 shows an example of an annotated program for sorting a given array.

The specification employs the user-defined predicates `Swap`, `Sorted` and `Permut`. These logical predicates refer to one, respectively two, memory states, which must be noted as explicit parameters in their definitions. `Swap` specifies that in a given array pointed by `a` the elements at positions `i` and `j` are swapped between the memory states `L1` and `L2` (lines 2 and 3) and that all the remaining positions are unchanged (lines 4 and 5). The predicate `Sorted` states that at memory state `L` in a given array pointed by `a` the sequence of elements between positions `l` and `h` is sorted. Notice that as this predicate refers to only one memory state, this is implicit in the dereferences `a[i]` and `a[j]`, i.e. `a[i]` is here equivalent to `\at(a[i],L)`. This is different within code annotations where dereferences always refer to the memory state at the current program point (e.g. line 47). In code annotations `\at` expressions can refer to previous, labelled memory states (e.g. line 75). `Permut` is defined inductively, similar to the Coq predicate shown above, except that it is over an array in two memory states.

The specification of the program is organised around function contracts which relate the pre-state with the post-state of the function. For instance, the pre-condition of the function `swap` expresses that the given array `t` must be valid at least for the positions `i` and `j`. Its post-condition specifies that the memory locations may be modified at these positions (line 28). This

```
1   /*@ predicate Swap{L1,L2}(int *a, integer i, integer j) =
2   @    \at(a[i],L1) == \at(a[j],L2) &&
3   @    \at(a[j],L1) == \at(a[i],L2) &&
4   @    \forall integer k; k != i && k != j
5   @        ==> \at(a[k],L1) == \at(a[k],L2);
6   @*/
7
8   /*@ predicate Sorted{L}(int *a, integer l, integer h) =
9   @    \forall integer i,j; l <= i <= j < h ==> a[i] <= a[j] ;
10  @*/
11
12  /*@ inductive Permut{L1,L2}(int *a, integer l, integer h) {
13  @  case Permut_refl{L}:
14  @    \forall int *a, integer l, h; Permut{L,L}(a, l, h) ;
15  @  case Permut_sym{L1,L2}:
16  @     \forall int *a, integer l, h;
17  @       Permut{L1,L2}(a, l, h) ==> Permut{L2,L1}(a, l, h) ;
18  @  case Permut_trans{L1,L2,L3}:
19  @     \forall int *a, integer l, h;
20  @       Permut{L1,L2}(a, l, h) && Permut{L2,L3}(a, l, h) ==>
21  @         Permut{L1,L3}(a, l, h) ;
22  @  case Permut_swap{L1,L2}:
23  @     \forall int *a, integer l, h, i, j;
24  @        l <= i <= h && l <= j <= h && Swap{L1,L2}(a, i, j) ==>
25  @       Permut{L1,L2}(a, l, h) ;
26  @  }
27  @*/
```

Figure 2.1: Example of an ACSL user-defined predicates

so called assigns-clause has an exclusive semantics, i.e. all the memory locations not mentioned are ensured to remain untouched. The resulting memory at the array positions in question is specified with the `Swap` predicate. The other function `min_sort` has a specification with named behaviours: `sorted` and `permutation`. These names are referred to in the loop invariants which specify the behaviour of loops and to allow inductive reasoning when proving the specifications.

The Frama-C platform provides parsing, type-checking and AST creation of ACSL annotated C source files. Its plug-in architecture allows the definition and dynamic loading of OCaml written analysers of such annotated programs. Currently two plug-ins, the Jessie plug-in[45] and the WP plug-in [1], allow verifying a program with respect to its specification by generating proof obligations and sending them to external provers. With the Jessie plug-in all the generated proof obligations are proved automatically.

---

1. http://frama-c.com/wp.html

```
28   /*@ requires \valid(t+i) && \valid(t+j);
29     @ assigns t[i],t[j];
30     @ ensures Swap{Old,Here}(t,i,j);
31     @*/
32   void swap(int t[], int i, int j) {
33     int tmp = t[i];
34     t[i] = t[j];
35     t[j] = tmp;
36   }
37   /*@ requires \valid_range(t,0,n-1);
38     @ behavior sorted:
39     @   ensures Sorted(t,0,n);
40     @ behavior permutation:
41     @   ensures Permut{Old,Here}(t,0,n-1);
42     @*/
43   void min_sort(int t[], int n) {
44     int i,j;
45     int mi,mv;
46     if (n <= 0) return;
47     /*@ loop invariant 0 <= i < n;
48       @ for sorted:
49       @  loop invariant
50       @   Sorted(t,0,i) &&
51       @   (\forall integer k1, k2 ;
52       @      0 <= k1 < i <= k2 < n ==> t[k1] <= t[k2]) ;
53       @ for permutation:
54       @  loop invariant Permut{Pre,Here}(t,0,n-1);
55       @ loop variant n-i;
56       @*/
57     for (i=0; i<n-1; i++) {
58       // look for minimum value among t[i..n-1]
59       mv = t[i]; mi = i;
60       /*@ loop invariant i < j && i <= mi < n;
61         @ for sorted:
62         @  loop invariant
63         @   mv == t[mi] &&
64         @   (\forall integer k; i <= k < j ==> t[k] >= mv);
65         @ for permutation:
66         @  loop invariant
67         @   Permut{Pre,Here}(t,0,n-1);
68         @ loop variant n-j;
69         @*/
70       for (j=i+1; j < n; j++) {
71         if (t[j] < mv) {
72           mi = j ; mv = t[j];
73         }
74       }
75     L:
76       swap(t,i,mi);
77       //@ assert Permut{L,Here}(t,0,n-1);
78     }
79   }
```

Figure 2.2: Example of an ACSL annotated C program

Chapter 3

Whycert: A Certified Verification Condition Generator

The purpose of this chapter is to develop a certified VC generator able to produce VCs for multiple provers. We implement and prove sound, in the Coq proof assistant, a VC generator inspired by the former Why tool, called Whycert. To make it usable with arbitrary theorem provers as back-ends, we make it generic with respect to a *logical context*, containing arbitrary abstract data types and axiomatisations. Such a generic aspect is suitable to formalise memory models needed to design front-ends for mainstream programming language, as it is done for C by VCC above Boogie or Frama-C/Jessie above Why. The input programs of our VC generator are imperative programs written in a core language which operates on mutable variables of types declared in the logical context. The output logic formulas are built upon the same logical context. This certified Coq implementation is crafted so it can be extracted into a standalone executable as explained in Section 2.1.6.

Section 3.1 informally describes the core programming language and shows an example. Section 3.2 formalises our notion of generic logical contexts. Section 3.3 formalises our core language and defines its operational semantics. Section 3.4 defines the weakest precondition computation WP and proves its *soundness*. Section 3.5 aims at the extraction of an executable plug-in for the Why3 platform. We introduce a variant wp of the calculus which produces concrete formulas instead of Coq ones.

A significant part of this chapter has been published at the VSTTE conference in 2012 [28]. The main increments are a more general notion of logical contexts and the Why3 back-end.

3.1   Informal Description of the core programming language

Our core language follows most of the design choices of the input language of Why and we use the Why3 concrete syntax for the examples and explications in this chapter. Indeed we reduce to an even more basic set of constructs, nevertheless remaining expressive enough to encode higher-level sequential algorithms. We follow an ML-style syntax; in particular there is no distinction between expressions and instructions. Following again the Why design, our core language contains an exception mechanism, providing powerful control flow structures. As we will see these can be handled by weakest pre-condition calculus without major difficulty. As a difference to the Why programming language, we allow only global references: Local references and references as function parameters are not supported, excluding thus any form of aliasing by construction, whereas Why excludes aliasing by typing. For instance, in the following Why program the call to `f` is rejected:

```
type array
function select array int : int
function store array int int : array

axiom select_eq: forall a :array, i x :int. select (store a i x) i = x

axiom select_neq : forall a :array, i j x: int.  i ≠ j →
  select (store a i x) j = select a j

predicate sorted array int

axiom sorted_def: forall a : array,n:int. sorted a n ↔
  forall i j:int. 0 ≤ i && i ≤ j && j < n → select a i ≤ select a j

predicate swap array array int int
axiom swap_def: forall a b :array, i j : int. swap a b i j ↔
      select a i = select b j && select a j = select b i &&
      forall k : int. k ≠ i && k ≠ j → select a k = select b k

predicate permut array array int
axiom permut_refl: forall a : array, n:int. permut a a n
axiom permut_sym: forall a b: array, n:int. permut a b n → permut b a n
axiom permut_trans:
  forall a b c:array, n:int. permut a b n && permut b c n → permut a c n
axiom permut_swap : forall a b : array, n i j : int. 0 ≤ i && i < n &&
      0 ≤ j && j < n && swap a b i j → permut a b n
```

Figure 3.1: Logical context for sorting

```
  let f(x:int ref, y:int ref) = ...
  let g(z:int ref) = f (z, z)
```

A program in this language is defined by a *logical context* and a finite set of function defin-itions, which can be mutually recursive. A logical context provides a fixed set of abstract types, logical symbols, global references and exceptions that may be used in the program.

The core programming language of Whycert contains a language of logical terms, which will also be the output of the WP calculus. The following concrete syntax illustrates Whycert logic terms and predicates.

$$
\begin{array}{llll}
\texttt{t} & ::= & \texttt{s} & \text{logical symbol} \\
& | & \texttt{t1 t2} & \text{term application} \\
& | & \textbf{let } \texttt{v} = \texttt{t} \textbf{ in } \texttt{t} & \text{local binding} \\
& | & \textbf{forall } \texttt{v:T, t} & \text{quantification} \\
& | & \texttt{v} & \text{local name} \\
& | & \texttt{!r} & \text{dereference} \\
& | & \texttt{at !r l} & \text{dereference at label} \\
& | & \texttt{t \&\& t | t} \rightarrow \texttt{t} & \text{connectives}
\end{array}
$$

These logic expressions always have an associated type and we will call *predicates* the logic terms that have a propositional type and also *proposition* if the term doesn't contain dereferences.

Only the conjunction and implication connectives are built-in, as they are the only ones needed for the WP calculus. Other connectives may be included as logical symbols which can have higher order types and are applied in a curried way.

Terms may refer to a *mutable state* associating a value to every global reference. The syntax `!r` allows dereference `r`. Additionally terms may refer to previous states by means of their label.

Logic terms are used inside program expressions whose concrete syntax is self explanatory:

```
e  ::=  t                            term
     |  e; e                         sequence
     |  let v = e in e               local binding
     |  f(t,...,t)                   function or parameter call
     |  if t then e else e           conditional branching
     |  r := t                       assignment of a reference
     |  'L: e                        labelled expression
     |  assert {t}; e                local assertion
     |  raise (ex t)                 exception throwing
     |  try e with ex v → e          exception catching
     |  loop invariant {t} e         infinite loop
```

The only particularity are the infinite loops which have an associated invariant. The only way to exit them is by using exceptions.

A definition of a function follows the structure

```
| let f(x1:T1) ... (xn:Tn) : T = { p } e { q }
```

where the predicates `p` and `q` are the pre- and the post-condition. In the post-condition, the reserved name `result` is locally bound and denotes the result of the function of type `T` and label `old` is bound to denote the pre-state. Note that exceptions are not supposed to escape function bodies. We could easily support such a feature by adding a family of post-conditions indexed by exceptions as in Why [24].

Example 1 *Fig. 3.1 presents, in Why3 concrete syntax, an appropriate axiomatisation for specifying a program sorting an array. An abstract type* `array` *is introduced to model arrays of integers indexed by integers. It is axiomatised with the well-known theory of arrays. We also define predicates* `sorted t i` *meaning that* $t[0], \ldots, t[i-1]$ *is an increasing sequence, and* `permut t1 t2` *meaning that* t_1 *is a permutation of* t_2. *The latter is axiomatised: it is an equivalence relation that contains all transpositions* `swap` *of two elements.*

All the constants in the example like `select`, `store`, *but also* `0`, `1`, `+` *and* `<`, *are logical symbols belonging to the appropriate logical context as formalised below.*

Fig. 3.2 shows a program that sorts the global array `t` *by the classical selection sort algorithm. Note the use of the exception* `Break` *to exit from the infinite loops. Note also the use of labels in annotations, allowing to specify assertions, loop invariants and post-conditions that link up various states of execution.*

3.2 Logical Contexts

Our background logic is multi-sorted. Models for specifying programs can be defined by declaring types, constant, function and predicate symbols, and axioms. In Coq this is formalised using section variables which all the following definitions depend on.

3.2.1 Logical Signatures

A *logical signature* is composed of a set `utype` of sort names introduced by the user, a set `sym` of constant, function and predicate symbols, a set `ref` of global reference names and a set

```
val t : ref array

let swap (i:int) (j:int) : unit =
 { true }
  let tmp = (select !t i : int) in
  t := store !t i (select !t j); t := store !t j tmp; ()
 { swap !t (old !t) i j }

val mi : ref int
val mv : ref int
val i : ref int
val j : ref int

exception Break

let selection_sort (n:int) : unit =
 { n ≥ 1 }
  i := 0;
  'Pre:
  try loop
    invariant { 0 ≤ !i && !i < n && sorted !t !i && permut !t (at !t 'Pre) n
      && forall k1 k2 : int. 0 ≤ k1 && k1 < !i && !i ≤ k2 && k2 < n →
           select !t k1 ≤ select !t k2 }
    if !i ≥ n-1 then raise Break  else ();
    (* look for minimum value among t[i..n-1] *)
    mv := select !t !i; mi := !i; j := !i+1;
    try loop
      invariant { !i < !j && !i ≤ !mi && !mi < n && !mv = select !t !mi &&
        forall k:int. !i ≤ k && k < !j → select !t k ≥ !mv }
      if !j ≥ n then raise Break else ();
      if select !t !j < !mv then (mi := !j ; mv := select !t !j; ()) else ();
      j := !j + 1
    end with Break → () end;
  'Lab: begin
    swap !i !mi;
    assert { permut !t (at !t  'Lab) n };
    i := !i + 1 end
  end with Break → () end
 { sorted !t n && permut !t (old !t) n }
```

Figure 3.2: Selection sort in our core language

exn of exception names. We require every symbol, exception and reference to have an associated type.

```
Variable utype : Type.

Inductive type :=
| Tuser : utype → type
| Tarrow : type → type → type
| Tprop.

Variable sym : type → Type.
Variables ref exn: type → Type.
```

That is, the types are completed with built-in types for propositions and functions. We will use the following notation for function types:

```
Inductive utype := Tarray | Tint | Tunit.
Notation type := (type utype).
Inductive sym : type → Type :=
| SYMselect: sym (Tarray --> Tint --> Tint)
| SYMstore: sym (Tarray --> Tint --> Tint --> Tint)
| SYMsorted: sym (Tarray --> Tint --> Tprop)
| SYMswap: sym (Tarray --> Tarray --> Tint --> Tint --> Tprop
| SYMpermut: sym (Tarray --> Tarray --> Tprop)
| SYMeq ty: sym (ty --> ty --> Tprop)
| SYMconst_unit: sym Tunit
| SYMconst_int: Z → sym Tint
| SYMlt: sym (Tint --> Tint --> Tprop)
| SYMplus: sym (Tint --> Tint --> Tint)
| SYMopp: sym (Tint --> Tint)
```

Figure 3.3: A formal declaration of a logical signature for sorting

```
Infix "-->" := Tarrow.
```

Example 2 *The logical signature of example 1 can be given by the definitions in Figure 3.3*[1]. *Notice that in addition to the required abstract symbols, we also need to declare some standard types and symbols as they are not built-ins of our language.*

Note that the axioms of Figure 3.1 do not belong to the logical signature as needed to define the program. They are listed here to illustrate the symbols which are completely abstract to our WP calculus. They will be needed only by the provers to reason about the symbols when proving the verification conditions generated by the calculus.

3.2.2 Dependently Typed *De-Bruijn* Indices

A design choice in our formalisation is to define terms and expressions such that they are well typed by construction. This simplifies the definition of the semantics and the weakest precondition calculus on such expressions, as we don't need to handle malformed constructions at those points. To begin we need to ensure that occurrences of variables actually correspond to bound variables in their current scopes and that they are used with the correct type. Here we use so-called dependently typed *de Bruijn* indices following the preliminary approach of Herms [27] as documented in Chlipala [18].

Dependent indices are like regular *de-Bruijn* indices, in that `HIO` refers to the innermost bound variable, `HIS HIO` to the second innermost bound variable, etc. Additionally they carry information about their typing environment and about the type of the variable they represent. We use indices of type `lidx A E` to represent variables of type `A` under a typing environment `E`, that is the list of the types of the bound variables. The type of the innermost bound variable is stored at the first position in the typing environment, the type of the second innermost bound variable at the second position, etc. In Coq we can formalise this constraint using a parametrised annotated inductive type. The following polymorphic definition constrains the type of the first index to match the first element in the type list and recursively for the other elements.

---

1. Note that these definitions, as well as the ones shown in Figs. 3.6 and 3.19 are purely for illustrative purposes; the real example is processed by the Why3 front end as described in Section 3.5.3

```
Variable S : Type.
Inductive lidx A : list S → Type :=
| HI0 E : lidx A (A :: E)
| HIS B E : lidx A E → lidx A (B :: E).
```

Dependent indices, specialised with `S:=type`, are thus placeholders within terms but they can also be used to reference elements within heterogeneous lists, polymorphically defined as:

```
Variable T : S → Type.
Inductive hlist : list S → Type :=
| Hnil : hlist []
| Hcons A E : T A → hlist E → hlist (A :: E).
```

In such a heterogeneous list each element may have a different type. The type `hlist E` of heterogeneous lists then depends on the list of types `E` of their elements. Thanks to the constraints on the type parameters, if an index `i : lidx A E` references an element within a heterogeneous list `l : hlist E`, we are sure to find an element of type `A` at `i`-th position of `l`. This allows us to define the function `accslidx : lidx A E → hlist E → T A` which given an index and an `hlist` returns the element in the list pointed by the index. In Coq, thanks to the **Program** environment its definition is straightforward.

```
Program Fixpoint accslidx A E (i:lidx A E) (l:hlist E) : T A :=
  match l with
    | Hnil ⇒ !
    | Hcons _ _ h q ⇒
        match i with
          | HI0 _ ⇒ h
          | HIS _ _ i1 ⇒ accslidx i1 q
        end
  end.
```

We will use these heterogeneous lists to give semantics to our languages. Precisely, heterogeneous lists, specialised with `T:=dentype`, are the representation of evaluation environments which associate a value to each variable in the current typing environment. The function `accslidx` is then used in the semantics rule for variable access.

Example 3 *The heterogeneous list* `l` *defined as* `[ 5; true; pred ]` *has type*

$$\texttt{hlist (S:=\textbf{Type}) (T:=id) [Z; bool; nat → nat].}$$

The De-Bruijn *indices*

$$\texttt{HI0 : lidx (S:=\textbf{Type}) Z [Z; bool; nat → nat]}$$

and

$$\texttt{HIS (HIS HI0) : lidx (nat → nat) [Z; bool; nat → nat],}$$

are well-typed and can be used to access their values in `l`, *e.g.* `accslidx HI0 l = 5 : Z` *and* `accslidx (HIS (HIS HI0)) l = pred : nat → nat`.

3.2.3   Terms and Propositions

The formal syntax of logic terms is given in Fig. 3.4. Terms depend on the parameters `L`, `E` and `A`, denoting respectively the highest index of a valid label, the typing environment and the type of the value they denote, which is `Tprop` in case of propositional terms. Variables are

```
Inductive builtin : type → Type :=
| Band : builtin (Tprop --> Tprop --> Tprop)
| Bimply : builtin (Tprop --> Tprop --> Tprop)
| Bfalse : builtin Tprop.

Definition var := lidx (S:=type).

Variable L : nat.

Inductive term E : type → Type :=
| Tsym A : sym A → term E A
| Tbuiltin A : builtin A → term E A
| Tvar A : var A E → term E A
| Tderef A : ref A → term E A
| Tapp A A1: term E (A1 --> A) → term E A1 → term E A
| Tlet A A1 : term E A1 → term (A1::E) A → term E A
| Tforall A1 : term (A1::E) Tprop → term E Tprop
| Tat A : label → ref A → term E A
.


Notation prop E := (term E Tprop).
```

Figure 3.4: Inductive definitions of logic terms

represented by our dependent indices `lidx A E`. The constructor `Tlet` expresses let-blocks at the term level. As usual with *de Bruijn* indices, no variable name is given and the body of the block is typed in a typing environment that is enriched by the type of the term to be remembered. Similarly `Tforall` binds a new de Bruijn variable but generalising it instead of assigning a value to it. The symbol application is formalised in a curryfied style.

3.2.4 Logical Contexts, Semantics

The semantics of our generic language depends on an *interpretation* given to types and symbols. From such an interpretation, any term or proposition can be given a value, in a given *evaluation environment* for variables and given *state* for references.

Given a logical signature, an *interpretation* consists of a function `denutype` of type `utype → Type` assigning a semantics to the user types, and a function `densym` assigning a semantics to the introduced function and predicate symbols. Given `denutype` we define `dentype` to interpret all the types.

```
Fixpoint dentype ty : Type :=
  match ty with
    | Tuser uty ⇒ denutype uty
    | ty1 --> ty2 ⇒ dentype ty1 → dentype ty2
    | Tprop ⇒ Prop
  end.
```

The semantics of a term of type `term L E A` is defined under an *evaluation environment* `G` of type `env E`, a *state* `S` of type `state` and a history of previous states `SS` of type `states L`. As described above, `env E` is a heterogeneous list with one element for each entry of the typing environment `E`. `state` is a mapping from references `ref A` to values of type `dentype A` and `states L` is a vector of size `L` of `state`, whose nth element denotes the state memorised at the nth enclosing label. As a special case a term of type `term 0 E A` cannot refer to any previous labels.

```
Variable densym : ∀ty, sym ty → dentype ty.

Definition denbuiltin ty (c: builtin ty) : dentype ty :=
  match c with
    | Bandd ⇒ and
    | Bimply ⇒ impl
    | Bfalse ⇒ False
  end.

Fixpoint evalterm L E A (t:term E A) (G:env E) (SS: states L) (S: state)
  : dentype A :=
  match t with
    | Tsym _ s ⇒ densym s
    | Tbuiltin A a ⇒ denbuiltin a
    | Tvar _ v ⇒ accsvar v G
    | Tderef A r ⇒ S A r
    | Tapp _ _ t1 t2 ⇒ (evalterm t1 G SS S) (evalterm t2 G SS S)
    | Tlet _ _ t1 t2 ⇒ (evalterm t2 (evalterm t1 G SS S::G) SS S)
    | Tforall _ p ⇒ forall x, evalterm p (x::G) SS S
    | Tat A l r ⇒ Vnth l SS A r
  end.

Notation valid := (evalterm (A:=Tprop)).
```

Figure 3.5: Denotational semantics of terms and propositions

The semantics of terms defined by structural recursion is shown in Figure 3.5. Note that `evalterm` is a total function: the semantics is defined for every term as correct typing is ensured by construction.

The programs formalised in the next section will assume a given logical signature and the verification conditions generated from such a program will be logical propositions concerning the types and symbols of that logical signature. Recall that our goal is to prove these formulas with the aid of automatic SMT provers so it is of crucial importance that the semantics these provers give to the formulas is the same as in our Coq formalisation. Except the dereferences which will be output as universally quantified variables, our logical language is very basic, so every reasonable prover should agree on its semantics – provided that they agree on the interpretation of the symbols.

Many symbols can remain *abstract* to the provers. This means that a prover doesn't make any assumption about the interpretation of such a symbol. E.g. in the example this is the case for `select`, `store`, `sorted`, etc. To allow provers reasoning about abstract symbols, the user can specify *axioms* in the form of propositions in our logical language which will be sent to the provers along with the verification conditions. In Coq these logical terms can be evaluated to values of type **Prop** and proving these Coq propositions correct using tactics corresponds to validating the axiomatisation. Indeed the interpretation given by the Coq functions `dentype` and `densym` can be seen as a *model* for the axiomatisation. If a prover, or a combination of several provers, succeeds in proving the verification conditions about the abstract symbols using only information from the axiomatisation, i.e. for *any* interpretation of the abstract symbols, then we are sure the conditions are valid also for the particular interpretation given in Coq.

Some symbols however can not be treated as abstract symbols. Either because they correspond to higher order connectives of the logic of the prover, like disjunction or negation, or because they belong to some built-in theory for which the prover is optimised, like integer or real arithmetic. In these cases, it is up to the user to make sure that the semantics the provers give to these symbols is the same as in our Coq formalisation.

```
Definition denutype ty :=
  match ty with
    | Tarray ⇒ Z → Z
    | Tint ⇒ Z
    | Tunit ⇒ unit
  end.
Notation dentype := (dentype denutype).

Definition densym ty (s: sym ty) : dentype ty :=
  match s in sym ty return dentype ty with
    | SYMselect ⇒ fun a i, a i
    | SYMstore ⇒ fun a i x, fun i', if i == i' then x else a i'
    | SYMsorted ⇒ fun a n, ∀i j, 0 <= i → i <= j → j < n →  a i <= a j
    | SYMswap ⇒ fun a b i j, a i = b j ∧  a j = b i
                              ∧  forall k, k <> i → k <> j →  a k = b k
    | SYMpermut ⇒ fun a1 a2 z, Permutation a1 a2 (Zabs_nat z)
    | SYMconst_unit ⇒ tt
    | SYMeq ty ⇒ eq
    | SYMconst_int x ⇒ x
    | SYMlt ⇒ Zlt
    | SYMplus ⇒ Zplus
    | SYMopp ⇒ Zopp
  end.
```

Figure 3.6: The interpretation of the logical symbols for sorting

Example 4 *A possible interpretation of the logical types and symbols for the sorting example is shown in Figure 3.6. Note that starting from* `SYMeq`*, all the symbols make use of definitions from the Coq standard library: all these symbols will correspond to built-in symbols for the provers as well, whereas the first five stay non interpreted.*

The axioms can then be proved in Coq, thus proving the axiomatisation consistent.

Recall that such an interpretation and proof of consistency is not needed for the WP calculus: when a program is proved with the WP calculus, it is proved with respect to any model of the axiomatisation.

3.3 The Core Programming Language

3.3.1 Formal Syntax of Expressions

Like terms of the logic, expressions of programs are formalised by an inductive type `expr L E A` depending on the parameters `A`, `E` and `L`, denoting respectively the evaluation type, the typing environment and the highest index of a valid label, as shown in Figure 3.7. Notice that variables and labels are left implicit in the inductive definition thanks to *De-Bruijn* representation.

Additionally expressions depend on the parameter `F`, the list of *signatures* of the functions a sub-expression can refer to. Unlike `L,E` and `A` this parameter does not change within sub-expressions, so we can say `F` is a parameter of the program. A signature is a pair of the return type of the function and the list of the function's parameters. A function identifier, as used in function calls, of type `lidx (signature A P) F` is an index pointing to an element with the signature `signature A P` within a heterogeneous list of functions with types `F`. Such heterogeneous lists of type `hlist (func F) F` are precisely the representation of a program

```
Variable par : list type → type → Type.

Record signatureT := signature {
  fu_return: type;
  fu_params: list type }.

Variable F : list signatureT.

Definition fidx sig := lidx sig F.

Inductive expr L E A : Type :=
| Eterm (t:term L E A): expr L E A
| Eseq A1 (e1:expr L E A1) (e2:expr L E A): expr L E A
| Elet A1 (e1:expr L E A1) (e2:expr L (A1::E) A): expr L E A
| Eassign (r:ref A) (t: term L E A): expr L E A
| Eassert (p:prop L E) (e: expr L E A): expr L E A
| Eraise A1 (ex: exn A1) (t: term L E A1): expr L E A
| Eif (p: prop L E) (e1 e2: expr L E A): expr L E A
| Eloop A1 (inv: prop L E) (e:expr L E A1): expr L E A
| Etry Aex (e1:expr L E A) (ex: exn Aex) (e2:expr L (Aex::E) A): expr L E A
| Elab (e: expr (succ L) E A): expr L E A
| Ecall P (f: lidx (signature A P) F) (ps: termlist 0 E P): expr L E A
| Ecallpar P (pa: par P A) (ps: termlist 0 E P): expr L E A.

Record contract P A := {  pre : prop 0 P;  post : prop 1 (A::P)  }.

Record func sig := {
  fun_contract :> contract sig.(fu_params) sig.(fu_return);
  body : expr 1 sig.(fu_params) sig.(fu_return) }.

Definition prog := hlist func F.

Variable pg : prog.

Coercion accsfunc sig (fi: fidx sig) : func sig :=
   (accslidx fi pg).
```

Figure 3.7: Inductive definition of expressions

`prog F`. Notice that in the definition of programs the parameter `F` appears twice: once as parameter of `hlist`, to define the signatures of the functions in the program, and once as parameter of `func` to constrain expressions in function bodies to refer only to functions with a signature appearing in `F`. This way we ensure the well-formedness of the graph structure of programs: it is impossible to refer to an unexisting function.

A function definition `func F (signature A P)` consists of a body of type `expr F 1 E A` and a contract, i.e. a pre-condition of type `prop 0 P` and a post-condition `prop 1 (A::P)`. In the pre-condition no labels may appear, hence its type has the parameter `0`. In the post-condition we allow referring to the pre-state of a function call: in the syntax this corresponds to using the label *old*. The post-condition may additionally refer to the result of the function, hence its type environment is enriched by `A`.

A last parameter of programs is `par`, the set of program parameter names. Like functions, program parameters can be called with a list of arguments, but their implementations are abstract: the only available information about a program parameter is its specification, that is its functional contract and its effects, i.e. the set of potentially modified references.

```
Notation rset := (set (sigT ref)).

Record parspec P A  := {
  par_contract :> contract P A;
  par_effects: rset
}.

Variable get_parspec :> ∀P A, par P A → parspec P A.
```

Like with abstract symbols, the interpretation of program parameters is given in terms of a Coq function `den_par`, which, unlike `densym`, can modify the current state. To ensure soundness of the language we require that the abstract interpretation respects the given specification for every parameter.

```
Variable den_par :
  forall P A, par P A → env P → state → state * dentype A.

Hypothesis den_par_correct : forall P A (pa: par P A),
  ∀G S, evalterm pa.(pre) G [] S →
  let (S', a) := den_par pa G S in
    evalterm pa.(post) (a::G) [S] S' ∧
    assigns S pa.(par_effects) S'.
```

3.3.2 Operational Semantics

The operational semantics is defined in big-step style following the approach of Leroy and Grall [37]. A first set of inference rules inductively defines the semantics of terminating expressions (Figs. 3.8 and 3.9) and a second set defines the semantics of non-terminating expressions, co-inductively (Fig. 3.10 and 3.11). Judgement `G/SS/S/ e ⟹ S'/o` expresses that in environment `G`, history of states `SS` and state `S`, the execution of expression `e` terminates, in a state `S'` with *outcome* o: either a normal value `Outval v` or an exception `Outexn ex v` where `v` is the value held by it. There are two rules for **let** `e1` **in** `e2` depending on the outcome of `e1` (`semlet` and `semletexn`). The rule for assignment `semassign` uses the update operation `update S r a` on states which replaces the mapping for `r` in `S`. A labelled expression is evaluated in an enriched state `S::SS` where the current state is copied on top of the vector `SS` (rule `semlab`). The rule for function calls `semcall` requires the pre-condition to be valid in the current state and the post-condition to be valid in the returning state. Note that there is an implicit coercion from the function identifier to its definition.

Judgement `G/SS/S/ e ⟹ ∞` states that the execution of expression `e` is non terminating in environment `G`, history of states `SS` and state `S`. Its definition is straightforward: the execution of an expression diverges if the execution of a sub-expression diverges. The interesting cases are for the execution of a loop: starting from a given state `S`, it diverges either if its body diverges or if its body terminates on some state `S'` and the whole loop diverges starting from this new state. Of course, non-termination may be caused by infinite recursion of functions, too.

The main feature to notice is that execution blocks whenever an invalid assertion is met: the rules for assertions, loops and function calls are applicable only if the respective annotations are valid. Conversely, as everything is well-typed by construction, the only reason why an expression wouldn't execute is that one of its annotations isn't respected.

Definition 5 (Safe execution) *An expression* `e` *executes safely in environment* `G` *and state* `SS/S` *if either it diverges or it terminates*

```
Inductive sem L E (G:env E) (SS:states L) (S:state) A
  : expr L E A → state → outcome A → Prop :=
| semterm (t:term L E A):
                        ∅
  --------------------------------------------------------
    G/SS/S / Eterm t ⟹ S / Outval (evalterm t G SS S)

| semseq A1 e1 S' (a:dentype A1) e2 S'' (o:outcome A):
    G/SS/S/ e1 ⟹ S' / Outval a  ·  G/SS/S'/ e2 ⟹ S'' / o
  ------------------------------------------------------------
              G/SS/S/ Eseq e1 e2 ⟹ S'' /o

| semseqexn Aex A1 (e1:expr L E A1) S' ex (a:dentype Aex) (e2: expr L E A):
            G/SS/S/ e1 ⟹ S'/Outexn ex a
    ---------------------------------------
      G/SS/S/ Eseq e1 e2 ⟹ S'/Outexn ex a

| semlet A1 e1 S' (a:dentype A1) e2 S'' (o:outcome A):
    G/SS/S/ e1 ⟹ S' / Outval a  ·  (a::G)/SS/S'/ e2 ⟹ S'' / o
  -----------------------------------------------------------------
              G/SS/S/ Elet e1 e2 ⟹ S'' /o

| semletexn Aex A1 e1 S' ex (a:dentype Aex) (e2: expr L (A1::E) A):
            G/SS/S/ e1 ⟹ S'/Outexn ex a
    ---------------------------------------
      G/SS/S/ Elet e1 e2 ⟹ S'/Outexn ex a

| semassign r (t:term L E A) :  let a := evalterm t G SS S in
                          ∅
  ------------------------------------------------------
    G/SS/S/ Eassign r t ⟹ update S r a / Outval a

| semassert P (e: expr L E A) S' o:
      valid P G SS S  ·  G/SS/S/ e ⟹ S'/ o
  --------------------------------------------
        G/SS/S/ Eassert P e ⟹ S'/o

| semraise Aex ex (t:term L E Aex):
                        ∅
  --------------------------------------------------------------
    G/SS/S/ Eraise ex t ⟹ S / Outexn (A:=A) ex (evalterm t G SS S)

| semif1 (p:prop L E) (e1 e2: expr L E A) S' o:
      evalterm p G SS S ·   G/SS/S/ e1 ⟹ S'/o
    ---------------------------------------------
        G/SS/S/ Eif p e1 e2 ⟹ S'/o

| semif2 (p:prop L E) (e1 e2: expr L E A) S' o:
      ¬evalterm p G SS S · G/SS/S/ e2 ⟹ S'/o
    ---------------------------------------------
        G/SS/S/ Eif p e1 e2 ⟹ S'/o
```

Figure 3.8: Operational semantics of terminating expressions

```
| semloop A1 P (e: expr L E A1) S' S'' (o:outcome A) x:
    valid P G SS S  ·  G/SS/S/ e ⟹ S'/ Outval x ·
         G/SS/S'/ Eloop P e ⟹ S''/o
  -------------------------------------------------
             G/SS/S/ Eloop P e ⟹ S''/o

| semloopexn A1 P e S' Aex ex (a:dentype Aex):
    valid P G SS S  ·  G/SS/S/ e ⟹ S'/Outexn (A:=A1) ex a
  -----------------------------------------------------------
         G/SS/S/ Eloop P e ⟹ S'/Outexn (A:=A) ex a

| semtry (e1:expr _ _ A) Aex (ex: exn Aex) e2 S' o:
       G/SS/S/ e1 ⟹ S'/o  ·  (∀a, o =/= Outexn ex a)
    --------------------------------------------------
           G/SS/S/ Etry e1 ex e2 ⟹ S'/o

| semtryexn e1 Aex (ex ex':exn Aex) e2 S' S'' (a:dentype Aex) o:
     G/SS/S/ e1 ⟹ S'/Outexn (A:=A) ex a  ·
  (a::G)/SS/S'/ e2 ⟹ S''/o  ·  ex '=== ex'
  -------------------------------------------
       G/SS/S/ Etry e1 ex' e2 ⟹ S''/o

| semlab (e:expr (succ L) E A) S' a:
       G/(S::SS)/S/ e ⟹ S'/a
  ----------------------------
    G/SS/S/ Elab e ⟹ S'/a

| semcall P (f: fidx (signature A P)) ps S' (a:dentype A):
    let G_args : env P := evaltermlist G [] S ps in
          valid f.(pre) G_args [] S ·
          G_args/[S]/S / f.(body) ⟹ S'/Outval a ·
       valid f.(post) (a::G_args) [S] S'
  ------------------------------------------------------
     G/SS/S/ Ecall f ps ⟹ S' / Outval a

| semcallpar P (pa: par P A) ps S' (a:dentype A):
    let G_args : env P := evaltermlist G [] S ps in
          valid pa.(pre) G_args [] S ·
          den_par pa G_args S = (S', a)
    -------------------------------------------
       G/SS/S/ Ecallpar pa ps ⟹ S' / Outval a

  where "G / SS / S / e ⟹ S' / o" := (@sem _ _ G SS S _ e S' o).
```

Figure 3.9: Operational semantics of terminating expressions (continued)

```
CoInductive seminf L E (G:env E) (SS:states L) S A: expr L E A → Prop :=
| seminfseq A1 (e1:expr L E A1) (e2:expr L _ A) :
         G/SS/S/ e1 ⟹ ∞
  --------------------------
     G/SS/S/ Eseq e1 e2 ⟹ ∞

| seminfseq2 A1 (e1:expr L E A1) (e2:expr L _ A) S' (a:dentype A1):
    G/SS/S/ e1 ⟹ S'/Outval a  · G/SS/S'/ e2 ⟹ ∞
  -------------------------------------------------
              G/SS/S/ Eseq e1 e2  ⟹ ∞
```

Figure 3.10: Operational semantics of non-terminating expressions

```
| seminflet A1 (e1:expr L E A1) (e2:expr L _ A) :
        G/SS/S/ e1 ⟹ ∞
  --------------------------
    G/SS/S/ Elet e1 e2 ⟹ ∞

| seminflet2 A1 (e1:expr L E A1) (e2:expr L _ A) S' (a:dentype A1):
    G/SS/S/ e1 ⟹ S'/Outval a  · (a::G)/SS/S'/ e2 ⟹ ∞
  -------------------------------------------------------
              G/SS/S/ Elet e1 e2 ⟹ ∞

| seminfassert P (e:expr L E A):
    valid P G SS S  ·  G/SS/S/ e ⟹ ∞
  -------------------------------------
       G/SS/S/ Eassert P e ⟹ ∞

| seminfif1 (p:prop L E) (e1 e2: expr L E A):
    evalterm p G SS S · G/SS/S/ e1 ⟹ ∞
  ---------------------------------------
       G/SS/S/ Eif p e1 e2 ⟹ ∞

| seminfif2 (p:prop L E) (e1 e2: expr L E A):
    ¬evalterm p G SS S · G/SS/S/ e2 ⟹ ∞
  ----------------------------------------
       G/SS/S/ Eif p e1 e2 ⟹ ∞

| seminfloop P A1 (e:expr L E A1):
    valid P G SS S · G/SS/S/ e ⟹ ∞
  ----------------------------------
    G/SS/S/ Eloop (A:=A) P e ⟹ ∞

| seminfloop2 A1 P e S' (x:dentype A1):
    valid P G SS S · G/SS/S/ e ⟹ S'/Outval x ·
        G/SS/S'/ Eloop (A:=A) P e ⟹ ∞
  ---------------------------------------------
       G/SS/S/ Eloop (A:=A) P e ⟹ ∞

| seminftry (e1:expr _ _ A) Aex (e2:expr L (Aex::E) A) ex:
        G/SS/S/ e1 ⟹ ∞
  -------------------------------
    G/SS/S/ Etry e1 ex e2 ⟹ ∞

| seminftry2 e1 Aex (ex ex': exn Aex) e2 S' (a:dentype Aex):
    G/SS/S/ e1 ⟹ S'/ Outexn (A:=A) ex a  ·
    (a::G)/SS/S'/ e2 ⟹ ∞   · ex '=== ex'
  ------------------------------------------
        G/SS/S/ Etry e1 ex' e2 ⟹ ∞

| seminflab (e:expr (succ L) E A):
    G/(S::SS)/S/ e ⟹ ∞
  ------------------------
    G/SS/S/ Elab e ⟹ ∞

| seminfcall P (f: fidx (signature A P))  ps:
    let G_args : env P := evaltermlist G [] S ps in
    valid f.(pre) G_args [] S ·  G_args/[S]/S/ f.(body) ⟹ ∞
  ------------------------------------------------------------
              G/SS/S/ Ecall f ps ⟹ ∞

  where  "G / SS / S / e '⟹ ∞'" := (@seminf _ _ G SS S _ e).
```

Figure 3.11: Operational semantics of non-terminating expressions

```
Definition safe L E A G SS S (e: expr L E A) :=
  G/SS/S/ e ⟹ ∞ ∨ ∃S' o, G/SS/S/ e ⟹ S'/o.
```

A program respects its annotations if every function can execute safely.

```
forall P A (fi: lidx (signature A P) F) G S,
  let f := accsfunc fi in
    evalterm f.(pre) G [] S →
    G/[S]/S/ f.(body) ⟹ ∞ ∨
    ∃S' a, G/[S]/S/ f.(body) ⟹ S'/Outval a ∧
           evalterm f.(post) (a::G) [S] S'.
```

Our semantics is quite unusual, in particular it is not executable. Although, if annotations are removed and if the propositional guards in if-then-else blocks and loops are decidable then it becomes executable and coincides with a natural semantics. This approach makes a distinct set of rules for axiomatic semantics *à la* Hoare obsolete: the soundness of the verification condition generator will be stated using this definition of safe execution. Moreover this notion of safe execution is indeed stronger than the usual notion of partial correctness: a safe program that does not terminate will still satisfy its annotations forever. [2]

3.4 Weakest Precondition Calculus

3.4.1 Effect Inference

To carry out the weakest precondition calculus we need to know the *effect* of each expression, i.e. the references it may modify. This is necessary to avoid too verbose loop invariants and function contracts: without effect inference a user would have to specify all the references that do not change between loop iterations or function calls.

Expressions with effects are assignments `r := t`, which modify the reference `r`, and parameter calls, which declare the set of modified references in their specification, so given an expression we need to recursively collect all the references modified by sub-expressions. Because of function calls, this recursion is not structural. Calling a function means possibly modifying all the references appearing in assignments inside the function's body, so we need to know them. Since the functions can be mutually recursive, we compute their effects by iterating a function `writes` collecting additional effects, until it reaches a fixed point. Termination of this algorithm is proven in Coq.

Definition 6 *The* `effects` *of a program associates a finite set of references to each function.*

Given the current effects ϵ, the function `writes` *recursively collects the references modified by the given expression assuming* `accslist f ϵ` *as the references modified by the function* `f`*. Because of the type parameter, references are collected in existential pairs (syntax* `(A & r)` *). The function* `infer_1`*, which returns a new* `effects` *by calling* `writes` *for each function's body, is recursively called by* `infer_n` *until a fixed point is reached. In Coq it is defined by well-founded recursion based on a given decreasing measure. It is not detailed here, but informally the total number of modified references for every function increases and is bounded.*

---

2. Total correctness is not considered in this work; however it is clear that one could add annotations for termination checking: variants for loops and for recursive functions as in ACSL [8].

```
Variable ε : effects.

Fixpoint writes L E A (e:expr L E A) : rset :=
  match e with
    | Eterm _
    | Eraise _ _ _      ⇒ empty
    | Eassign r _       ⇒ singleton (A & r)
    | Eassert _ e
    | Eloop _ _ e
    | Elab e            ⇒ writes e
    | Eseq _ e1 e2
    | Elet _ e1 e2
    | Eif _ e1 e2
    | Etry _ e1 _ e2    ⇒ union (writes e1) (writes e2)
    | Ecall _ i _       ⇒ accslist i ε
    | Ecallpar _ pa _   ⇒ pa.(par_effects)
  end.

Definition infer_1 : effects :=
  smap (fun s (f : func s), writes f.(body)) pg.

Program Fixpoint infer_n ε { measure ... } : effects :=
  let ε' := infer_1 ε in
  if ε == ε' then ε else infer_n ε'.

Definition infer := infer_n ε0.
```

```
Lemma infer_correct:
  forall s (f:fidx s), accslist f infer === writes infer f.(body).
```

In the following we will simply write `writes` for `writes (infer pg)`, as the effects are computed once and for all for a given program.

We define `assigns` which relates two states that differ only in the references appearing in the given set. An intermediate result shows that the execution of an expression `e` really modifies only the references computed by `writes e`.

```
Definition assigns (S:state) (rs : rset) (S':state) :=
  forall A (r:ref A), ¬ In (A&r) rs → S' _ r = S _ r.
```

```
Lemma assigns_correct A E L G SS S (e: expr L E A) S' o:
  G/SS/S/ e ⟹ S'/o → assigns S (writes e) S'.
```

This result helps us in generating less restrictive proof obligations: whenever we need to generalise a state, we can include this syntactic information about which references are modified in the new state and which are not.

3.4.2 Definition of the WP-calculus

We calculate the weakest pre-condition of an expression given a post-condition by structural recursion over expressions (Fig. 3.12). We take as inputs a post-condition `Q` for normal behaviour and a family of post-conditions `R` for exceptional behaviour - one for each exception. In the case of a loop, the pre-condition is calculated using the loop invariant and in the case of a function call we use the pre- and post-condition of that function. In both cases, as customary in WP calculi, we

```
Definition NOP L A := states L → state → dentype A → Prop.
Definition EOP L := states L → state → ∀Aex, exn Aex → dentype Aex → Prop.

Program Fixpoint WP L E A (e:expr L E A)
    (Q:NOP L A) (R:EOP L) (G:env E) (SS:states L) (S:state) :=
  match e return _ with
    | Eterm t ⇒ Q SS S (evalterm t G SS S)
    | Eseq A1 e1 e2 ⇒ WP e1 (fun SS S a, WP e2 Q R G SS S) R G SS S
    | Elet A1 e1 e2 ⇒ WP e1 (fun SS S a, WP e2 Q R (a::G) SS S) R G SS S
    | Eassign r t ⇒ let a := (evalterm t G SS S) in Q SS (update S r a) a
    | Eraise Aex ex t ⇒ R SS S Aex ex (evalterm t G SS S)
    | Eassert P e ⇒ evalterm P G SS S ∧  WP e Q R G SS S
    | Eif p e1 e2 ⇒ (evalterm p G SS S → WP e1 Q R G SS S) ∧
                    (¬evalterm p G SS S → WP e2 Q R G SS S)
    | Eloop _ Inv e1 ⇒ evalterm Inv G SS S ∧
        ∀S', assigns S (writes e1) S' → evalterm Inv G SS S' →
          WP e1 (fun SS S'' tt, evalterm Inv G SS S'') R G SS S'
    | Etry Aex e1 ex e2 ⇒ WP e1 Q
        (fun SS S' Aex' ex' result,
          if ex' `== ex then
            WP e2 Q R (cast result::G) SS S'
          else R SS S' Aex' ex' result)
        G SS S
    | Elab e ⇒ WP e (dnlab; Q) (dnlab; R) G (S::SS) S
    | Ecall P f ps ⇒
        let G_args := evaltermlist G [] S ps in
          evalterm f.(pre) G_args [] S ∧
          forall S' a, assigns S (accslist f F_effects) S' →
            evalterm f.(post) (a::G_args) [S] S' →
            Q SS S' a
    | Ecallpar P pa ps ⇒
        let G_args := evaltermlist G [] S ps in
          evalterm pa.(pre) G_args [] S ∧
          forall S' a, assigns S pa.(par_effects) S' →
            evalterm pa.(post) (a::G_args) [S] S' →
            Q SS S' a
  end.
```

Figure 3.12: Recursive definition of the WP-calculus

need to quantify over all states that may be reached by normal execution starting from the given state S: these are the states S' which differ from S only for the references that are modified in the loop or the function's body. The set of modified references is computed using our effect inference explained above.

Definition 7 *The verification conditions for a whole program are*

```
Definition VCGEN := forall A P (f: fidx (signature A P)) G S,
    evalterm f.(pre) G [] S →
    WP f.(body)
      (fun SS S' (a:dentype A), evalterm f.(post) (a::G) SS S' )
      (fun _ _ _ _ _, False) G [S] S.
```

The `False` *as exceptional post-conditions requires that no function body exits with an exception.*

3.4.3 Soundness Results

A preliminary property to establish is that after a terminating execution, post-conditions are respected if the weakest pre-condition is valid.

```
Lemma WP_correct L E A G SS S (e : expr L E A) S' o:
  G/SS/S/ e ⟹ S'/o →
  forall (Q : NOP L A) (R : EOP L), WP e Q R G SS S →
    match o with
      | Outval a ⇒ Q SS S' a
      | Outexn Aex ex a ⇒ R SS S' Aex ex a
    end.
```

Proof. by induction over the derivation of `G/SS/S/ e ⟹ S'/o`

We now state that if the VCs hold for all functions then any expression having a valid WP executes safely.

Theorem 8 (Soundness) *If* `VCGEN` *holds then the program respects its annotations*

```
Hypothesis VC_prog : VCGEN.

Theorem soundness L E A (e:expr L E A) SS S G Q R:
  WP e Q R G SS S →
  G/SS/S/ e ⟹ ∞ ∨ ∃S' o, G/SS/S/ e ⟹ S'/o.

Corollary global_soundness P A (f: fidx (signature A P)) G S:
    evalterm f.(pre) G [] S →
    G/[S]/S/ f.(body) ⟹ ∞ ∨
    ∃S' a, G/[S]/S/ f.(body) ⟹ S'/Outval a ∧
           evalterm f.(post) (a::G) [S] S'.
```

Proof. by co-induction, using the axiom of excluded middle to distinguish whether the execution of an expression does or does not terminate, following the guidelines of Leroy and Grall [37]. Notice that this means that it is enough to prove the verification conditions for each function separately, even if functions can be mutually recursive, there is no circular reasoning.

The important corollary below states that if the VCs hold for all functions then their bodies all execute safely. By definition of the semantics, this implies that all assertions, invariants and pre- and post-conditions in a given program are verified if the verification conditions are valid.

3.5 Extraction of a Certified Verification Tool

The obtained Coq function for generating verification conditions is not *extractible*, i.e. it cannot be extracted to executable OCaml code: given a program `pg` we obtain a Coq term `VCGEN pg` of Coq type **Prop** which must be proved valid to show the correctness of the program. The process thus remains based on Coq for making the proofs. In this section we show how to extract the calculus into a separate tool so that proofs can be performed with other provers, e.g. SMT solvers.

3.5.1 Concrete WP computation

To achieve this we need the WP calculus to produce a formula in the abstract syntax of Fig. 3.4 instead of a Coq **Prop**. We thus define another function `wp` (Fig. 3.13) which, given an expression `e`, a normal post-condition `q` and a family of exceptional post-conditions `r`, returns a

```
Program Fixpoint wp L E A (e:expr L E A)
    (q:prop L (A::E)) (r: ∀Aex, exn Aex → prop L (Aex::E)): prop L E :=
 match e return _ with
    | Eterm t ⇒ Tlet t q
    | Eseq A1 e1 e2 ⇒ wp e1 (lift_term (E1:=[]) (wp e2 q r)) r
    | Elet A1 e1 e2 ⇒
        wp e1 (wp e2 (lift_term (E1:=[A]) q)
                (fun A ex, lift_term (E1:=[A]) (r A ex))) r
    | Eassign r t ⇒ Tlet (t) (subst_term q r (Tvar HI0))
    | Eraise Aex ex t ⇒ Tlet t (r Aex ex)
    | Eassert P e ⇒ Pand_asym P (wp e q r)
    | Eif p e1 e2 ⇒ Pand (Pimply p (wp e1 q r))
                        (Pimply (Pimply p Pfalse) (wp e2 q r))
    | Eloop A1 Inv e1 ⇒ Pand Inv
        (abstr (Pimply Inv (wp e1 (lift_term (E1:=[ ]) Inv) r)) (writes e1))
    | Etry Aex e1 ex e2 ⇒
        wp e1 q
          (fun Aex' ex' ⇒ if ex' '== ex then
            cast (T:=prop L) ((wp e2 ((lift_term (E1:=[A]) q))
              (fun Aex'' ex'', (lift_term (r Aex'' ex'')))))
            else r Aex' ex')
    | Elab e ⇒ dnlab_term (wp e (uplab_term q) (fun A ex, uplab_term (r A ex)))
    | Ecall P f ps ⇒
        Pand_asym (uplabn_term (substps_term ps (E1:=[]) (f.(pre))))
          (dnlab_term (abstr
            (Tforall (Pimply
              (uplabn_term (substps_term
                (Indexes.map (fun A (t:term 0 E A), uplabx t) ps)
                (E1:=[A]) (f.(post))))
              (uplab_term q)))
            (accslist f F_effects)))
    | Ecallpar P pa ps ⇒
        Pand_asym (uplabn_term (substps_term ps (E1:=[]) (pa.(pre))))
          (dnlab_term (abstr
            (Tforall (Pimply
              (uplabn_term (substps_term
                (Indexes.map (fun A (t:term 0 E A), uplabx t) ps)
                (E1:=[A]) pa.(post)))
              (uplab_term q)))
            pa.(par_effects)))
    end.
```

Figure 3.13: Recursive definition of the concrete WP-calculus

weakest pre-condition. It is defined recursively on `e` similarly to `WP` in Fig. 3.12, but this time `q`, `r` and the result are syntactic propositions which are concretely transformed. Everywhere in `WP` we easily made a semantic operation, like enriching the environment or updating the global state now we need to concretely lift De-Bruijn indices or substitute inside terms. This requires several term operators, and for each a commutation lemma, which we don't show here. Other than that, the concrete `wp` calculus is similar to `WP`, the most notable difference is that quantification over states is now replaced by quantifications over modified references:

```
Definition abstr L E: rset →  prop L E →  prop L E :=
  fold (fun (r:sigT ref) q,
    Tforall (subst_term (lift_term (E1:=[]) q) (dsnd r) (Tvar HI0))).

Lemma abstr_correct L E (q:prop L E) G SS (s : rset) S:
  evalterm (abstr q s) G SS S →
  ∀S', assigns S s S' → evalterm q G SS S'.
```

For each reference `r` in the given set, the function `abstr` precedes the given proposition by `Tforall`, substituting the occurrences of `r` by the newly bound variable. As ensured by the lemma, this leads to the desired result of generalising the term with respect to all the references in `s`.

   We prove that the `wp` calculus is correct w.r.t. the `WP` calculus.

```
Lemma wp_correct L E A (e:expr L E A) (q:prop L (A ::E)) r G SS S:
  evalterm (wp e q r) G SS S →
  WP e (fun SS S a, evalterm q (a::G) SS S)
    (fun SS S Aex ex a, evalterm (r Aex ex) (a::G) SS S)
     G SS S.
```

Proof. By induction over expression `e`, where in each case some commutation lemmas are applied to exchange updates in the state and pushes in the environment by substitutions of references and concrete liftings of *De-Bruijn* indices.
   From `wp` we now define a concrete verification-condition generator `vcgen`.

```
Fixpoint abstrv L E : prop L E → prop L [] :=
  match E with
    | [] ⇒ id
    | A1::E1 ⇒ fun p, abstrv (Tforall p)
  end.

Definition vcgen_f s (f: func s) :=
  abstrv (Pimply f.(pre) (dnlab_term
    (wp f.(body) f.(post) (fun _ _, Pfalse)))) .

Definition vcgen := ` (smap vcgen_f pg).
```

Theorem 9 *An important corollary of* `wp_correct` *is that the concrete verification condition is correct: if the generated concrete conditions are valid then so are the logical ones:*

```
Definition valid_list (l: list (prop 0 [])) :=
  ∀S, List.Forall (fun p, evalterm (A:=Tprop) p [] [] S) l.

Theorem vcgen_correct: valid_list vcgen → VCGEN.
```

   That is, the hypothesis of Definition 8 is established if we prove the formulas generated by `vcgen` valid in any state.

```
Parameter sys : Type.
Parameter sys_lt : relation sys.
Parameter sys_consume : sys → sys.
Axiom sys_lt_SO : StrictOrder sys_lt eq.
Axiom sys_consume_gt : ∀z, z < sys_consume z.

Record IO A :=
  { io: sys → A * sys;
    iosys z := snd (io z);
    ioget z := fst (io z);
    io_prop: ∀z, z < iosys z
  }.

Program Definition iobind {A1 A2} (e1: IO A1) (e2: A1 → IO A2) : IO A2 :=
  {| io z := let '(x, z) := io e1 z in io (e2 x) z |}.

Program Definition ioreturn {A} (a: A) : IO A :=
  {| io z := (a, sys_consume z) |}.

Notation "'let!' x := e1 'in' e2" := (iobind e1 (fun x, e2)).

Extract Inductive IO ⇒ "" [ "" ].
Extract Inlined Constant ioreturn ⇒ "".
Extract Inlined Constant iobind ⇒ " (fun x f →  f x) ".
```

Figure 3.14: The Input-Output Monad

3.5.2 Producing Concrete Syntax with Explicit Binders

Still, formulas of `vcgen` are represented by a De-Bruijn-style abstract syntax. To print out such formulas we need to transform them into concrete syntax with identifiers for variables by generating new names for all the binders. This could be done on the fly in an unproven pretty-printer. Though, being a non trivial transformation it is better to do it in a certified way directly after the generation.

The Why3 language allows to define logical formulas and theories. To be able to directly interface with the Why3 platform we formalise the Why3 logical language as exposed by the Why3 API. This will allow us generating Why3 theories from a list of verification conditions in a certified way.

The Input-Output Monad

The main difficulty consists in handling local variables. In order to translate our logical language with De-Bruijn indices to a language with explicit binders we need to generate fresh variable names. Similarly to many OCaml libraries, the Why3 API relies on the fact that a record value as created by the syntax `{ f1 = e1; ...; fn = en}` is *physically distinct* from any other value created before. Therefore a function like

```
let create_vsymbol name ty = { vs_name = name; vs_ty = ty }
```

returns a fresh `vsymbol` at every call because its implicit side effect is to modify the OCaml heap. In order to model such a function with side effects in Coq we define the input-output monad as in Haskell (Fig. 3.14). A value of type `IO A` is essentially a function from the abstract type `sys`, representing the current state of the system, to `A * sys`, i.e. a value of type A and the resulting state. A function that returns a value of type `IO A` thus actually takes an extra argument, the current state, and returns a new state in addition to its actual result. The `iobind`

```
Parameter ref : Type → Type.
Parameter mkref : ∀A, A → IO (ref A).
Parameter contents : ∀A, ref A → IO A.
Parameter store : ∀A, ref A → A → IO ().

Definition new_counter : () → IO (() → IO nat) :=
  fun _, let! r := mkref 0 in
    let next := fun _,
      let! x := contents r in
      let! _ := store r (succ x) in
        contents r in
      ioreturn next.

Extract Constant ref "'a" ⇒ "'a Pervasives.ref".
Extract Inlined Constant mkref ⇒ "ref".
Extract Inlined Constant contents ⇒ "(!)".
Extract Inlined Constant store ⇒ "(:=)".

Recursive Extraction new_counter.
```

Figure 3.15: Example for the use of the input-ouput monad

operator combines such functions by transparently handling system states and the `let`! notation allows natural programming with `IO`s. At extraction to OCaml the monad type is simply erased as OCaml functions natively support side effects.

Figure 3.15 shows a small example program using the `IO` monad. Using the OCaml reference type and its operators declared as Coq parameters we can define a counter generator – a standard first exercise in imperative OCaml programming. The result of its extraction is shown below. Notice how erasing the monadic operators leads to a rather natural OCaml program. However, it would be more readable if the `fun x f →  f x` could be replaced by a `let x = .. in ..` syntax.

```
type nat = O | S of nat

let new_counter x =
  (fun x f →  f x) (ref O) (fun r →
    let next = fun x0 →
      (fun x f →  f x) ((!) r) (fun x1 →
        (fun x f →  f x) ((:=) r (S x1)) (fun x2 →  (!) r))
    in
     next)
```

In order to prove properties about such a program with references we would need to axiomatise their behaviour, stating for instance that if we write to a reference then the contents of all the others remain untouched.

```
Axiom load_store_eq: forall A (r: ref A) a z,
  ioget (contents r) (iosys (store r a) z) = a.
Axiom load_store_neq: forall A1 (r1: ref A1) A2 (r2: ref A2) a z,
  (&r1) ≠ (&r2) →
  ioget (contents r1) (iosys (store r2 a) z) = ioget (contents r1) z.
```

Notice that to apply this second axiom one would need to prove that two references are distinct. Intuitively a newly created reference is distinct from all the references created before. The best way to formalise this is to postulate an order relation over references and to assume that a reference

is greater than another if is created at a later moment.

```
Parameter ref_lt: relation (sigT ref).
Axiom ref_SO: StrictOrder ref_lt eq.
Axiom mkref_fresh: forall A1 (a1:A1) A2 (a2:A2) z1 z2,
  z1 < z2 → (&ioget (mkref a1) z1) < (&ioget (mkref a2) z2).
```

The notion of "later" is formalised with an order over system states and the definition of the `IO` monad guarantees that the state resulting of an execution of an `IO` operation is greater than the input state (see `ioprop`, Fig. 3.14), which corresponds to the natural notion of "later".

Notice that these order relations are not and don't need to be decidable. The only way to establish hypotheses about the ordering of two elements is using information of this axiomatisation. Technically, notice that because of their type parameter references are compared as existential pairs (`sigT`). Also notice that as `StrictOrder` is a type class, we can use the natural `<` infix notation which automatically refers to the appropriate instance of an order relation.

Strings

Some functions of the API take a string arguments, e.g. the name of a symbol. In order to call such a function from Coq we declare the OCaml string type and a function from Coq strings to OCaml strings as parameters.

```
Parameter ocamlstring : Type.
Parameter ocamlstring_of_string :> string →  ocamlstring.
```

At extraction Coq strings will be lists of characters, so the implementation of the conversion function simply creates a string and fills it with the characters of the list.

```
Extract Inlined Constant ocamlstring ⇒ "String.t".
Extract Constant ocamlstring_of_string ⇒ "fun s →
  let r = String.create (List.length s) in
    let rec fill pos = function
    | [] →  r
    | c::s →  r.[pos] <- c; fill (pos + 1) s
    in fill 0 s".
```

Symbolic Reals

The Why3 logical language provides for built-in types for integers and reals, as well as constructions to express integer and real constants. Whereas integer constants can be concretely represented with the Coq type `Z`, this is not the case for real constants, as the Coq type `R` is currently axiomatised in Coq. If we tried to include the Coq library `Reals` providing this axiomatisation in the part of our development that will be extracted, then we would have to implement all the axioms of that library in OCaml – or at least provide some dummy implementation. Instead we just declare an abstract type of symbolic reals as a parameter.

```
Parameter real : Type.
```

In a different Coq module we then declare that the denotational semantics of a `real` is given as a `R`.

```
Parameter den_real : real →  R.
```

This module, and by transitivity `Reals`, is included only in the specifications of the semantics of our various languages and the proofs about them, but not in the calculus, and need thus not to be extracted.

The Why3 API

With these ingredients we can formalise the logical language as exposed by the Why3 API (Fig. 3.16) and its semantics (Fig. 3.17). We use the `IO` monad to formalise a generator of fresh variable symbols `vsymbol`. Notice that in contrary to its counterpart in OCaml, it has a parameter of type `ty` representing the type of the variable. This is to simplify the proofs about the semantics of variables. Like with a phantom type the functions in the interface ensure the correctness of this type parameter. Similarly `lsymbol`, the type of logical constants, functions and predicates, has two type parameters, representing the types of its arguments and its return type. An `lsymbol` can be created using `make_lsymbol` or obtained accessing the Why3 standard library given the path and the name of the theory and the name of the symbol. For simplicity `stdlib_ls` is formalised as a total function, even if it is clear that it won't succeed if such a symbol does not exist. Also, it should fail if the given argument type list and return type do not match with the ones of the symbol. In this case the OCaml implementation would just raise an exception.

Terms are specified as an inductive, again with a phantom type parameter to ensure correct typing by construction. This allows to specify the semantics in a denotational style (Fig. 3.17): given a term and an environment assigning a value to every local variable the function `den_term` returns the Coq denotation of that term. The semantics of logical symbols is given as a function from a list of arguments to a result value. Here we distinguish between local and global symbols. The semantics of global symbols is assumed as a global parameter while the semantics of local symbols is to be given as an argument to the semantics. With this we try to capture the intended difference between built-in symbols, that provers associate some well known behaviour to, and user defined abstract symbols (see discussion in Section 3.2.4). This way we can define the notion of valid theory as the fact that the validity of a list of axioms implies the validity of a list of goals for *any* interpretation of the local symbols, which is precisely what the external SMT solvers will prove.

A weakness of this semantics is the requirement of the axiom of decidability of any property, used in the semantics of `t_if`, which is generally false. However, this is justified by the fact that the semantics of terms is used only to give semantics to proposition and thus in a **Prop** context. The axiom could possibly be avoided in a more laborious specification making use of `inhabited` or by resigning to a rule based specification.

Generation of Why3 Theories

In order to generate Why3 theories from Whycert verification conditions we essentially need to compile Whycert terms to Why3 terms. As the former may contain some constructions that are not representable in first-order logic, like quantification over functions or partial application of logical symbols, this compilation may fail. We use the following notations to combine the `IO` monad with the error monad:

```
Definition IOx A := IO (option A).
Notation "'return' x" := (ioreturn (Some x)).
Notation "'let!?' x := e1 'in' e2" :=
  (iobind e1
    (fun x, match x with Some x ⇒ e2 | None ⇒ ioreturn None end)).
```

```
Parameter ty : Type.
Parameters ty_int ty_real : ty.
Parameter ty_make: ocamlstring → ty.


Parameter vsymbol : ty → Type.
Parameter vsymbol_lt : relation (sigT vsymbol).
Axiom vsymbol_lt_SO : StrictOrder vsymbol_lt eq.
Parameter create_local_vsymbol : ∀tp, IO (vsymbol tp).
Axiom create_vs_gt: forall tp1 tp2 z1 z2, z2 > z1 →
 (&ioget(create_local_vsymbol tp2) z2) > (&ioget(create_local_vsymbol tp1) z1).
Parameter vs_equal : DependentEqDec (vsymbol) eq.


Parameter lsymbol : list ty → option ty → Type.
Parameter make_lsymbol : ocamlstring → ∀ar vl, lsymbol ar vl.
Parameter is_lsymbol_local: ∀ar vl, lsymbol ar vl → bool.
Axiom make_lsymbol_is_local:
  forall s ar vl, is_lsymbol_local (make_lsymbol s ar vl) = true.
Parameter stdlib_ls : ocamlstring → ocamlstring → ocamlstring →
  ∀ar vl, lsymbol ar vl.
Parameter stdlib_ty : ocamlstring → ocamlstring → ocamlstring → list ty → ty.


Inductive term : option ty → Type :=
| t_var {tp} (vs: vsymbol tp) : term (Some tp)
| t_int_const (z:Z) : term (Some ty_int)
| t_real_const (r:real) : term (Some ty_real)
| t_if {tp} (t0 : term None) (t1 t2: term tp) : term tp
| t_let {tp1 tp2} (vs:vsymbol tp1) (t1:term (Some tp1)) (t2:term tp2): term tp2
| t_forall {tp} (vs: vsymbol tp) (t: term None) : term None
| t_and (t1 t2: term None) : term None
| t_and_asym (t1 t2: term None) : term None
| t_or  (t1 t2: term None) : term None
| t_implies (t1 t2: term None) : term None
| t_iff (t1 t2: term None) : term None
| t_not (t: term None) : term None
| t_true  : term None
| t_false : term None
| t_equ {tp} (t1 t2: term tp) : term None
| t_neq {tp} (t1 t2: term tp) : term None
| t_app {ar vl} (ls: lsymbol ar vl) (tl: termlist ar) : term vl

with termlist : list ty → Type :=
| termlist_nil : termlist []
| termlist_cons {tp tpl}: term (Some tp) →  termlist tpl →  termlist (tp::tpl)
.


Definition prop := term None.


Record theory := {
  th_axioms: list prop;
  th_goals: list prop
}.
```

Figure 3.16: Axiomatisation of the Why3 API

```coq
Parameter den_ty : ty → Type.

Definition denotype ot := match ot with Some x ⇒ den_ty x | None ⇒ Prop end.

Definition Env := forall tp (vs: vsymbol tp), den_ty tp.

Parameter classicT : forall P : Prop, {P} + {¬P}.

Section SEM.

  Parameter den_global_lsymbol :
    forall ar vl, lsymbol ar vl → hlist den_ty ar → denotype vl.
  Variable  den_local_lsymbol :
    forall ar vl, lsymbol ar vl → hlist den_ty ar → denotype vl.

  Definition den_lsymbol ar vl (ls: lsymbol ar vl) :=
    if is_lsymbol_local ls then den_local_lsymbol ls else den_global_lsymbol ls.

  Fixpoint den_term (env: Env) tp (t: term tp) : denotype tp :=
    match t in term tp return denotype tp with
      | t_var _ vs ⇒ (env _ vs)
      | t_int_const z ⇒ den_int_const z
      | t_real_const r ⇒  den_real_const r
      | t_if t t0 t1 t2 ⇒ let P := den_term env t0 in
          if classicT P then den_term env t1 else den_term env t2
      | t_let _ _ vs t1 t2 ⇒ let x := den_term env t1 in
          den_term (add_env env vs x) t2
      | t_forall _ vs t ⇒ forall x, den_term (add_env env vs x) t
      | t_and t1 t2 | t_and_asym t1 t2 ⇒ den_term env t1 ∧  den_term env t2
      | t_or t1 t2 ⇒ den_term env t1 ∨ den_term env t2
      | t_implies t1 t2 ⇒ den_term env t1 → den_term env t2
      | t_iff t1 t2 ⇒ den_term env t1 ↔ den_term env t2
      | t_not t ⇒ ¬ den_term env t
      | t_true ⇒ True
      | t_false ⇒ False
      | t_equ _ t1 t2 ⇒ den_term env t1 = den_term env t2
      | t_neq _ t1 t2 ⇒ den_term env t1 ≠ den_term env t2
      | t_app _ _ ls tl ⇒ den_lsymbol ls (den_termlist env tl)
    end

  with den_termlist env tpl (tl: termlist tpl) : hlist den_ty tpl :=
    match tl with
      | termlist_nil ⇒ []
      | termlist_cons _ _ t tl ⇒ let x : den_ty _ := den_term env t in
        x :: (den_termlist env tl)
    end.

  Definition den_proplist env := List.Forall (den_term env (tp:=None)).

End SEM.

Definition theory_valid tk :=
  forall den_local_lsymbol env,
    den_proplist den_local_lsymbol env th.(th_axioms) →
    den_proplist den_local_lsymbol env th.(th_goals).
```

Figure 3.17: Specification of the Semantics of the logical language

For convenience we extend the type of variable symbols, which must have a first-order type, to variable symbols of any type and accordingly the functions operating over variable symbols:

```
Definition vsym ty :=
  match ty with
    | Tuser t ⇒ vsymbol (comp_utype t)
    | _ ⇒ False
  end.

Definition create_local_vsymbol' ty : IOx (vsym ty) :=
  match ty with
    | Tuser tp ⇒
        let! v := create_local_vsymbol (comp_utype tp) in
          return v
    | _ ⇒ error
  end.
```

Similarly we define `t_var'` and `t_forall'` . De-Bruijn indices are then compiled using a map that assigns such a `vsym` to every valid index:

```
Definition varmap E := forall ty, var ty E → vsym ty.
```

In order to compile successive curryfied applications we compile terms based on their type. If a given sub term has an arrow type, this means that it needs to be applied further, so in this case the compilation returns a function from terms to terms. Otherwise it returns a term, whose type is the compilation of the type of the original term:

```
Variable comp_utype: utype → ty.

Definition fo_comp_wtype ty :=
  match ty with
    | Tuser ty ⇒ Some (comp_utype ty)
    | _ ⇒ None
  end.

Fixpoint termcons ty :=
  match ty with
    | Tarrow ty1 ty2 ⇒ termcons ty1 → termcons ty2
    | _ ⇒ term (fo_comp_wtype ty)
  end.
```

The compilation is shown in Figure 3.18. For simplicity we do not try to compile **let** blocks in case of a higher order type, even if could be possible inlining the term. `compwterm` relies on the compilation of references and symbols: it takes as an argument a function `ref_name: ∀ty, ref ty → ocamlstring`, assigning a name to every reference, as well as a function `comp_sym: ∀ty, sym ty → termcons ty`, mapping every symbol either to some Why3 built-in construction or to an application of a logical symbol, either from some Why3 library or newly created. Figure 3.19 shows the compilation of the symbols for the sorting example as well as some helper functions. Notice that when looking up or creating an `lsymbol`, the appropriate list of parameter types and the appropriate value type is obtained by the type of the symbol.

Intuitively the compilation is correct because every time it goes under a binder adding a variable to the variable mapping, this variable is necessarily fresh. This is formalised with the predicate `vm_before` (Fig. 3.20). Under this hypothesis and the hypothesis that the evaluation environments are related accordingly to the `varmap` (`rel_vm_G`), we prove that the compiled

```
Variable comp_sym: ∀ty, sym ty → termcons ty.
Variable ref_name: ∀ty, ref ty → ocamlstring.

Definition comp_ref ty: ref ty → option (termcons ty) :=
  match ty return ref ty → option (termcons ty) with
    | Tuser tp ⇒ fun r, return
        t_app (make_lsymbol (ref_name r) [] (Some (comp_utype tp)))
          termlist_nil
    | _ ⇒ const error
  end.

Definition compwbuiltin ty (wblt: Why.builtin ty) : termcons ty :=
  match wblt with
    | Why.Band b ⇒ if b then t_and_asym else t_and
    | Why.Bimply ⇒ t_implies
    | Why.Bfalse ⇒ t_false
  end.

Fixpoint compwterm E (vm: varmap E) ty (wt: Wterm L E ty): IOx (termcons ty) :=
  match wt return _ with
    | Why.Tsym _ s ⇒ return comp_sym s
    | Why.Tbuiltin _ wblt ⇒ return compwbuiltin wblt
    | Why.Tvar ty v ⇒ return t_var' (vm _ v)
    | Why.Tderef ty r ⇒
        let? t := comp_ref r in
          return t
    | Why.Tapp tp1 tp2 wt1 wt2 ⇒
        let!? t1 := compwterm vm wt1 in
        let!? t2 := compwterm vm wt2 in
          return (t1: termcons (Tarrow tp2 tp1)) (t2:termcons tp2)
    | Why.Tlet ty ty' wt1 wt2 ⇒
        let!? v := create_local_vsymbol' ty'  in
        let!? t1 := compwterm vm wt1 in
        let!? t2 := compwterm (vm_add vm v) wt2 in
          (match ty return termcons ty → IOx (termcons ty) with
            | Tarrow _ _ ⇒ fun _, error
            | ty ⇒ fun t2, return t_let' v t1 t2
          end t2)
    | Why.Tat _ l r ⇒ False_lab l
    | Why.Tforall ty wt ⇒
        let!? v := create_local_vsymbol' ty  in
        let!? t := compwterm (vm_add vm v) wt in
          return t_forall' v t
  end.

Fixpoint compwproplist (wpl : list (Wprop L [])) : IOx (list prop) :=
  match wpl with
    | [] ⇒ return []
    | wt::wtl ⇒
        let!? t := compwterm wt in
        let!? tl := compwproplist wtl in
          return (t::tl)
  end.

Definition comptheory ax gl : IOx theory :=
  let!? ax' := compwproplist ax in
  let!? gl' := compwproplist gl in
    return {| th_axioms := ax'; th_goals := gl' |}.
```

Figure 3.18: Compilation of Terms

```
Fixpoint ty_args ty :=
  match ty with Tarrow (Tuser t) ty' ⇒ comp_utype t::ty_args ty' | _ ⇒ [] end.
Fixpoint ty_value ty :=
  match ty with Tarrow _ ty' ⇒ ty_value ty' | ty ⇒ fo_comp_wtype ty end.
Fixpoint Is_fo (ty:type) :=
  match ty with
    | Tarrow (Tuser _) ty' ⇒ Is_fo ty'
    | Tarrow _ _ ⇒ False
    | _ ⇒ True
  end.
Fixpoint mk_tc ty
  : Is_fo ty → (termlist (ty_args ty) → term (ty_value ty)) → termcons ty :=
  match ty  with
    | Tarrow (Tuser tp) ty' ⇒
          fun H f t, mk_tc (ty:=ty') H (fun l, f (termlist_cons t l))
    | Tarrow _ _ ⇒ False_rect _
    | _ ⇒ fun _ f, f termlist_nil
  end.

Definition stdlib_tc {ty H} s1 s2 s3: termcons ty :=
  mk_tc H (t_app (stdlib_ls s1 s2 s3)).
Definition make_ls_tc {ty H} s: termcons ty :=
  mk_tc H (t_app (make_lsymbol s _ _)).

Definition comp_sym ty (s: sym ty) : termcons ty :=
  match s with
    | SYMeq ty ⇒ t_equ
    | SYMconst_int x ⇒ t_int_const x
    | SYMlt ⇒ stdlib_tc "int" "Int" "infix <"
    | SYMplus ⇒ stdlib_tc "int" "Int" "infix +"
    | SYMopp ⇒ stdlib_tc "int" "Int" "prefix -"
    | SYMconst_unit ⇒ make_ls_tc "tt"
    | SYMselect ⇒ make_ls_tc  "select"
    | SYMstore ⇒ make_ls_tc "store"
    | SYMsorted ⇒ make_ls_tc "sorted"
    | SYMswap ⇒ make_ls_tc "swap"
    | SYMpermut ⇒ make_ls_tc "permut"
  end.
```

Figure 3.19: Compilation of symbols for the sorting example

term is equivalent by `rel_termcons`. This relation simplifies to standard equality in case of a propositional type, so it is an easy corollary that a compiled proposition is valid if and only if the original proposition is valid (`compwprop_correct`).

Finally we then prove the main soundness theorem of the generation of Why3 theories:

```
Theorem comptheory_correct ax gl th z:
  ioget (comptheory ax gl) z = Some th →
  theory_valid th → valid_list ax → valid_list gl.

Corollary vcgen3_correct ax F (pg: prog sym ref exn par F) z th:
  ioget (comptheory ax (vcgen pg)) z = Some th →
  theory_valid th → valid_list ax → VCGEN pg.
```

It states that if the generated Why3 theory is proved valid by an external solver and the list of axioms is proved in Coq, then the list of goals is valid. This theorem chains up with the soundness theorem of Section 3.5.1 guaranteeing that a program is correct if the Why3 theory generated from

```
Definition vs_before z ty (vs: vsym ty) :=
  ∀ty' vs', ioget (create_local_vsymbol' ty') z = Some vs' →
    (&vs) < (&vs').

Program Definition vm_before z E (vm:varmap E) :=
    forall A v, vs_before z (vm A v).

Variable den_local_lsymbol :
  forall ar vl, lsymbol ar vl → hlist den_ty ar → denotype vl.

Fixpoint rel_termcons (env: Env) ty: termcons ty → dentype ty → Prop :=
  match ty return termcons ty → dentype ty → Prop  with
    | Tuser x ⇒ fun t x,   den_term env t ∼= x
    | Tarrow ty1 ty2 ⇒ fun tc f,
        forall (t1: termcons ty1) (x: dentype ty1),
          rel_termcons  env t1 x → rel_termcons env (tc t1) (f x)
    | Tprop ⇒ fun t x, den_term env t = x
  end.

Hypothesis rel_den_comp_sym:
  forall env ty (s: sym ty), rel_termcons env (comp_sym s) (densym s).

Hypothesis rel_den_refs:
  forall ty (r:ref ty),
    den_local_lsymbol (make_lsymbol (ref_name r) [] (fo_comp_wtype ty)) []
    ∼= S ty r.

Program Definition rel_vm_G (e: Env) E (vm:varmap E) (G: env E) :=
  forall A v, xenv e (vm A v) ∼= accsvar v G.

Lemma compwterm_correct E vm ty (wt : Wterm 0 E ty) tc S env z:
  ioget (compwterm vm wt) z = Some tc →
  vm_before z vm →
  forall G, rel_vm_G env vm G →
    rel_termcons env tc (evalterm wt G [] S).

Corollary compwprop_correct (wt : Wprop 0 []) t S env z:
  ioget (compwprop wt) z = Some t →
  (den_term env t ↔ evalterm wt [] [] S).
```

Figure 3.20: Soundness of the Compilation of Terms

the program's verification conditions is valid.

3.5.3   Extraction and Experimentation

For experimentation purposes we also defined a compilation in the opposite direction, i.e. from programs in a front-end syntax to the corresponding program in *De-Bruijn* syntax, provided that the former is well typed.

We then use the extraction mechanism of Coq to extract an OCaml function that, given an AST of our front-end syntax representing a program, produces a theory.

We finally combine this with the Why3 parser to produce such an AST from concrete input programs in Why3 syntax resulting in a Why3 plugin that can be dynamically loaded by Why3ide allowing us to call automated provers on the proof task. This is illustrated by the diagram in Figure 3.21.
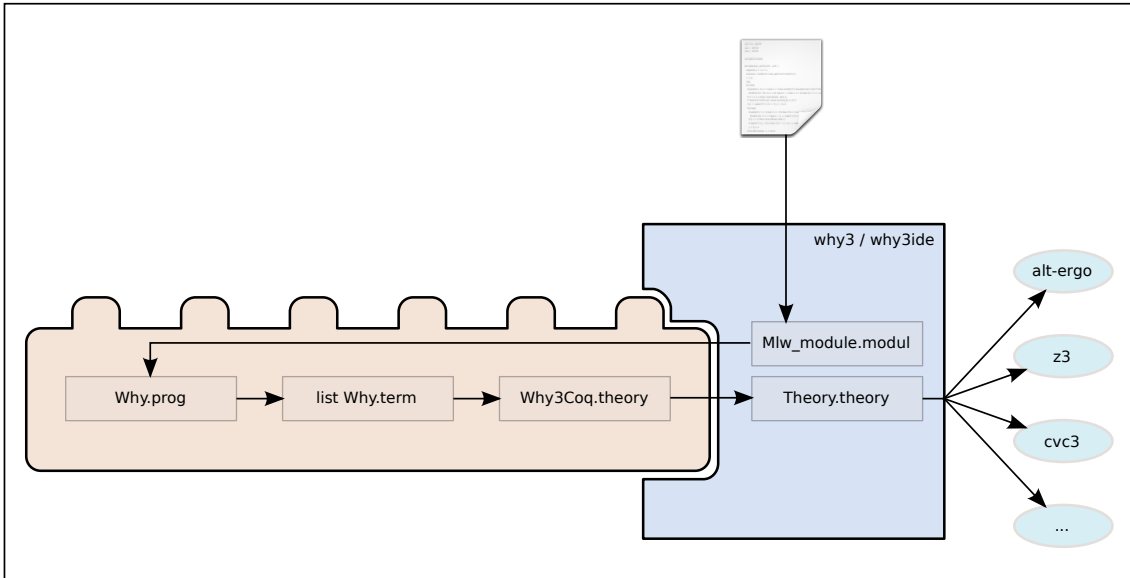
Figure 3.21: Diagram of the Why3 Plugin

We made experiments to validate this process. On our selection sort example, the two VCs for functions `swap` and `selection_sort` are generated in a fraction of a second by the standalone VC generator. These are sent to the Why3 tool, and they are proved automatically, again in a fraction of a second, by a combination of SMT solvers (i.e. after splitting these formulas, which are conjunctions, into parts [13]).

3.6 Conclusions

We formalised a core language for deductive verification of imperative programs. Its operational semantics is defined co-inductively to support possibly non-terminating functions. The annotations are taken into account in the semantics so that validity of a program with respect to its annotations is by definition the progress of its execution. We used an original formalisation of binders so that only well-typed programs can be considered, allowing us to simplify the rest of the formalisation. Weakest precondition calculus is defined by structural recursion, even in presence of mutually recursive functions, assuming the given function contracts. Even if there is an apparent cyclic reasoning, this approach is proved sound by a co-inductive proof. By additionally formalising an abstract syntax for terms and formulas, and relating their semantics with respect to the Coq propositions, we defined a concrete variant of the WP calculus which can be extracted to OCaml code, thus obtaining a trustable and executable VC generator close to Why or Boogie.

In the following chapters we show the certification of the remaining part of a complete chain from ACSL-annotated C programs to proof obligations. Just like the front end described above we will generate Whycert programs but starting from annotated Clight programs. The next chapter starts by formally specifying the semantics of ACSL expressions and ACSL annotated Clight programs.

Chapter 4

Formalisation of C and ACSL

The purpose of this chapter is to formally specify the abstract syntax and the semantics of ACSL terms and propositions and to integrate them into some semantics of Compcert's Clight.

Section 4.1 formalises the ACSL language. One of the characteristics of ACSL is that every logical function, including access to memory, is total, but not necessarily fully specified. Thus in ACSL every well-typed term and predicate has a semantics[8]. We propose a formalisation of such under-specified functions in Coq. We formalise a type system for the ACSL logical language and define a total semantics for well-typed formulas.

Section 4.2 adds ACSL annotations to the Clight abstract syntax and its big-step semantics, in a blocking style as in the previous chapter.

4.1 ACSL

ACSL is informally presented in Section 2.3.

4.1.1 ACSL Types

ACSL types are either C types or the mathematical types integer, real and boolean or arrays.

```
Notation ctype := Csyntax.type.

Inductive type :=
| Linteger | Lreal | Lboolean | Larray (ty:type)
| Tctype :> ctype → type.
```

Unlike arrays at the level of C types, which in the Compcert C formalisation necessarily carry their size (constructor `Tarray`), ACSL array types (constructor `Larray`) represent arrays of any size. In ACSL this is different from a pointer type, because arrays exist also as right values. Furthermore, ACSL arrays can contain any type of elements, even logic types. Not treated are structures containing logical elements. [1]

In the following variables denoted `cty` and `ty` will implicitly have type `ctype` and `type`, respectively.

```
Implicit Type cty : ctype.
Implicit Type ty : type.
```

---

1. Aggregate types with logical elements are required by the reference manual, but are currently not supported by the FramaC implementation

```
Implicit Type n : nat.
Implicit Type z : Z.
Implicit Type b : bool.
Implicit Type i : ident.


Inductive binop :=
| Bplus | Bminus | Bmult | Bdiv | Bmod
| Bbwand | Bbwor | Bbwxor | Band | Bor | Bimply.


Definition relop := Ceq | Cne | Clt | Cle | Cgt | Cge.


Inductive unop :=
| Uplus | Uminus | Uneg | Ubw_compl | Uderef | Uaddr.


Definition label := ident.


Inductive term :=
| Tbooleanconst b
| Tintconst z
| Trealconst (r: real)
| Tnull
| Tvar i
| Tprgvar i cty
| Tunop (op:unop) (t1:term)
| Tbinop (op:binop) (t1 t2:term)                       (* t1 + t2, t1 * t2, ...*)
| Tarray_access (t1 t2:term)                           (* t1[t2] *)
| Tfield_access (t: term) i                            (* t.i *)
| Tcomp (op:relop) (t1 t2: term)                       (* t1 < t2, t1 <= t2, ...*)
| Tcondition (t0 t1 t2:term)                           (* t0 ? t1 : t2 *)
| Tlet i (t1 t2:term)                                  (* \let i = t1; t2 *)
| Tcast ty (t:term)
| Tat (l:label) (t: term)
| Tarray_update (ta ti tx: term)                       (* ta \with [ti] = tx *)
| Tstruct_update (ts: term) (fi: ident) (tx: term)  (* ts \with . fi = tx *)
| Tempty_array ty (tl: term)
| Tempty_struct (si: ident) (fld: fieldlist)
.
```

Figure 4.1: ACSL logical terms

4.1.2 Abstract Syntax

ACSL logical terms and predicates are defined by the inductive types shown in Figs. 4.1 and 4.2. They formalise most of the constructions described in Section 2.3.

As a difference to the concrete syntax, we distinguish between logic variables and program variables. Logic variables are introduced by quantification or local **\let** blocks. In function contracts, the functions' formal parameters are considered as logic variables, too. Program variables lie in the C memory and will be evaluated by accessing their address in the memory. For program variables their type has to be given in the abstract syntax. This is to simplify the typing rules so they don't have to depend on the C environment. In both cases variable identifiers are Compcert identifiers. Another difference to the concrete syntax is the lack of array and structure initialisers. They can still be expressed by successive functional updates starting from an empty array or structure, i.e. an array, structure in which every element, field is uninitialised. The most important feature not yet formalised is user-defined logic functions and predicates.

```
Inductive pred :=
| Ptrue | Pfalse
| Pnot (p: pred)
| Pcomp (op:relop) (t1 t2: term)
| Pcomb (op:binop) (p1 p2: pred)
| Pforall i ty (p:pred)
| Plet i (t:term) (p:pred)
.
```

Figure 4.2: ACSL logical predicates

4.1.3 Typing

This section formalises the typing of ACSL terms and predicates. In this formalisation, we tried to stay as close as possible to the requirements informally described in the reference manual. The typing is very important, as it will guide the denotational semantics of ACSL terms and predicates. More precisely, their evaluation will be defined by recursion over the typing rules.

The typing rules shown in Figs. 4.3 to 4.5 state which terms are well typed under a given typing environment `te`, attributing them a type `ty` and a boolean `lr`, indicating whether the term is a valid left value. We use the notations `te ⊢`^{l}` t : ty` and `te ⊢`^{r}` t : ty` to state that term `t` is well typed with type `ty` as a left value or, respectively, a right value, as well as the notation `te ⊢(lr) t : ty` stating that `t` is a left or a right value, depending on the value of `lr`.

A value "lies in the C memory", according to the notion used in the reference manual, if it is a valid left value after these rules. A left value can be used as a right value in any context (rule `term_typed_lval`) or under the `Tat` operator to access a previous, labelled state (rule `term_typed_at`). Additionally, its address can be taken (rule `term_typed_addrof`). Examples of left values are occurrences of program variables and pointer dereferenciations. An access to an element of an array or to the field of a structure also is a left value if the base array or structure is a left value. Notice that in addition to pointers we also allow to dereference terms of an array type, as long as the array lies in the C memory, i.e. is a left value (rule `term_typed_deref`). This is formalised using the following predicate.

```
Inductive Is_Pointer_To cty: bool → type → Type :=
| Ptr_ptr  : Is_Pointer_To cty false (Tpointer cty)
| Ptr_arr z: Is_Pointer_To cty true (Tarray cty z)
| Ptr_larr : Is_Pointer_To cty true (Larray cty).
```

Typing environments are finite maps from identifiers to types which contain all the logical variables defined in the current scope (rule `term_typed_var`). They are enriched at local definitions or quantifications (rule `term_typed_let`). The typing environment does not contain information about C program variables. Indeed, occurrences of program variables and labels are not checked for being declared in the current skope by the typing rules (rules `term_typed_progvar` and `term_typed_at`). This design choice concerning the difference in treatment of logical and program variables is motivated by the different semantics of logical variables with respect to program variables. The former denote some value kept in a logical, type-safe environment that is enriched by local **\let** blocks and **\forall** quantifications, whereas the latter denote a memory access, which can fail in some circumstances even if the variable is declared. The evaluation function of ACSL terms which specifies the semantics of ACSL (Section 4.1.4) would thus have to handle the failure case even if had a typing environment for program variables and labels at its disposal. On the contrary not having to consider such a

```
Inductive term_typed (te: tenv) : term → bool → type → Type :=
| term_typed_lval t ty:
    te ⊢ˡ  t : ty
  ---------------
    te ⊢ʳ  t : ty

| term_typed_at l t ty:
      te ⊢ˡ  t : ty
  ----------------------
    te ⊢ʳ  Tat l t : ty

| term_typed_addrof t cty:
          te ⊢ˡ  t : cty
  -------------------------------------
    te ⊢ʳ  Tunop Uaddr t : Tpointer cty

| term_typed_deref t cty lr ty:
    te ⊢(lr) t : ty  ·  Is_Pointer_To cty lr ty
  -----------------------------------------------
          te ⊢ˡ  Tunop Uderef t : cty

| term_typed_progvar i cty:
        te.[i] = None
  -----------------------------
    te ⊢ˡ  Tprgvar i cty : cty

| term_typed_var i ty:
    te.[i] = Some ty
  --------------------
    te ⊢ʳ  Tvar i : ty

| term_typed_let i t1 ty1 t2 ty2:
    te ⊢ʳ  t1 : ty1  ·  te.[i <- ty1] ⊢ʳ  t2 : ty2
  ------------------------------------------------
              te ⊢ʳ  Tlet i t1 t2 : ty2

| term_typed_promote t ty1 ty2:
     te ⊢ʳ  t : ty1  ·  subtype*  ty1 ty2
  ---------------------------------------
            te ⊢ʳ  t : ty2

| term_typed_cast t ty1 ty2:
     te ⊢ʳ  t : ty1  ·  explicit_cast ty1 ty2
  --------------------------------------------
            te ⊢ʳ  Tcast ty2 t : ty2

| term_typed_cast_pointer t lr1 ty1 lr2 ty2:
    Is_Pointer_Type lr1 ty1  ·  Is_Pointer_Type lr2 ty2 ·
                  te ⊢(lr1) t : ty1
  -------------------------------------------------------
            te ⊢(lr2) Tcast ty2 t : ty2
```

Figure 4.3: ACSL typing rules for terms

```
| term_typed_binop_integer op t1 t2:
    te ⊢ʳ  t1 : Linteger · te ⊢ʳ  t2 : Linteger
    ------------------------------------------ op ∈ [ Bplus; Bminus; Bmult; Bdiv;
                                                      Bmod; Bbwand; Bbwor; Bbwxor ] →
     te ⊢ʳ  Tbinop op t1 t2 : Linteger

| term_typed_binop_real op t1 t2:
    te ⊢ʳ  t1 : Lreal · te ⊢ʳ  t2 : Lreal
    ----------------------------------- op ∈ [ Bplus; Bminus; Bmult; Bdiv ] →
     te ⊢ʳ  Tbinop op t1 t2 : Lreal

| term_typed_binop_boolean op t1 t2:
    te ⊢ʳ  t1 : Lboolean ·  te ⊢ʳ  t2 : Lboolean
    ----------------------------------------------                op ∈ [ Band; Bor ] →
      te ⊢ʳ  Tbinop op t1 t2 : Lboolean

| term_typed_binop_pointer op t1 t2 cty lr ty:
    te ⊢(lr) t1 : ty · Is_Pointer_To cty lr ty ·
            te ⊢ʳ  t2 : Linteger
    ------------------------------------------------          op ∈ [ Bplus; Bminus ] →
        te ⊢ʳ  Tbinop op t1 t2 : Tpointer cty

| term_typed_unop_integer op t:
       te ⊢ʳ  t : Linteger
    ------------------------------          op ∈ [ Uplus; Uminus; Ubw_compl ] →
     te ⊢ʳ  Tunop op t : Linteger

| term_typed_unop_real op t:
       te ⊢ʳ  t : Lreal
    ---------------------------                       op ∈ [ Uplus; Uminus ] →
     te ⊢ʳ  Tunop op t : Lreal

| term_typed_unop_boolean op t:
       te ⊢ʳ  t : Lboolean
    -----------------------------                          op ∈ [ Uneg ] →
     te ⊢ʳ  Tunop op t : Lboolean

| term_typed_relop op t1 t2 ty:
     te ⊢ʳ  t1 : ty · te ⊢ʳ  t2 : ty
    -------------------------------- ty ∈ [ Lboolean; Linteger; Lreal ] →
     te ⊢ʳ  Tcomp op t1 t2 : Lboolean

| term_typed_relop_pointer op t1 t2 cty1 cty2:
    te ⊢ʳ  t1 : Tpointer cty1  · te ⊢ʳ  t2 : Tpointer cty2
    -------------------------------------------------------
          te ⊢ʳ  Tcomp op t1 t2 : Lboolean

| term_typed_condition t0 t1 t2 ty:
    te ⊢ʳ  t0 : Lboolean  · te ⊢ʳ  t1 : ty  · te ⊢ʳ  t2 : ty
    ---------------------------------------------------------
             te ⊢ʳ  Tcondition t0 t1 t2 : ty
```

Figure 4.4: ACSL typing rules (continued)

```
| term_typed_array_access_l ta ti cty lr ty:
    te ⊢(lr) ta : ty · Is_Pointer_To cty lr ty ·
            te ⊢ʳ  ti : Linteger
  ----------------------------------------------
      te ⊢ˡ  Tarray_access ta ti : cty

| term_typed_array_access_r ta ti ty:
    te ⊢ʳ  ta : Larray ty  ·  te ⊢ʳ  ti : Linteger
  --------------------------------------------------
          te ⊢ʳ  Tarray_access ta ti : ty

| term_typed_field t si fld fi lr cty:
        te ⊢(lr) t : Tstruct si fld ·
    unrolled_field_type fi si fld = Some cty
  ----------------------------------------------
      te ⊢(lr) Tfield_access t fi : cty

| term_typed_array_update ta ti tx ty:
    te ⊢ʳ  ta : Larray ty  ·  te ⊢ʳ  ti : Linteger  ·
            te ⊢ʳ  tx : ty
  ---------------------------------------------------
      te ⊢ʳ  Tarray_update ta ti tx : Larray ty

| term_typed_struct_update ts fi tx si fld cty:
    te ⊢ʳ  ts : Tstruct si fld  ·  te ⊢ʳ  tx : cty  ·
      unrolled_field_type fi si fld = Some cty
  ---------------------------------------------------
    te ⊢ʳ  Tstruct_update ts fi tx : Tstruct si fld

| term_typed_empty_array ty tl:
          te ⊢ʳ  tl : Linteger
  --------------------------------------
    te ⊢ʳ  Tempty_array ty tl : Larray ty

| term_typed_empty_struct si fld:
                    ∅
  ------------------------------------------------
    te ⊢ʳ  Tempty_struct si fld : Tstruct si fld

| term_typed_integer z :
                ∅
  -------------------------------
    te ⊢ʳ  Tintconst z : Linteger

| term_typed_real r :
              ∅
  ----------------------------
    te ⊢ʳ  Trealconst r : Lreal

| term_typed_boolean b:
                  ∅
  -----------------------------------
    te ⊢ʳ  Tbooleanconst b : Lboolean

| term_typed_null :
              ∅
  -----------------------------
    te ⊢ʳ  Tnull : Tpointer Tvoid

where "te ⊢( lr ) t : ty" := (@term_typed te t lr ty)
and "te ⊢ʳ  t : ty" := (@term_typed te t false ty)
and "te ⊢ˡ  t : ty" := (@term_typed te t true ty).
```

Figure 4.5: ACSL typing rules for terms

```
Function subtype (ty1 ty2:type) {struct ty1} : Prop :=
  match ty1, ty2 with
    | Linteger,      (Lreal | Lboolean)
    | Tcint _ _,     Linteger
    | Tcfloat _,     Lreal       ⇒ True
    | Tcarray cty _, Larray ty  ⇒ cty = ty :> type
    | Larray ty1,    Larray ty2 ⇒ subtype ty1 ty2
    | _,_ ⇒ False
  end.

Function explicit_cast (ty1 ty2: type) {struct ty1} : Prop :=
  match ty1, ty2 with
    | Linteger, (Tcint _ _| Tcfloat _)
    | Lreal,    (Linteger | Tcint _ _| Tcfloat _)
    | Lboolean, (Linteger | Tcint _ _)
                              ⇒ True
    | Larray ty1, Tcarray ty2 _
    | Larray ty1, Larray ty2   ⇒ explicit_cast ty1 ty2
    | ty1,ty2                  ⇒ ty1 = ty2
  end.
```

Figure 4.6: ACSL casts

typing environment simplifies the specification of the semantics of ACSL expressions in Clight programs.

In ACSL logical expressions we want arithmetic operations to have exact semantics. In the logical expression `i + *p` the value of the C variable `i` and the value loaded from memory `*p` need to be promoted to mathematical integers or reals. The rules for arithmetic operations and comparisons (all shown in Fig. 4.4) therefore require the operands to have mathematical types. Promotions are formalised as coercive subtyping: a term with type `ty` can be typed `ty'` if `ty` is a subtype of `ty'` (rule `term_typed_promote`). The subtype relation (Fig. 4.6) formalises the subtyping as described by the informal specification: machine integrals are subtypes of **integer** which itself is subtype of **real** but also of **boolean**. Floating point types are subtypes of **real**. Furthermore subtyping recursively applies to elements inside arrays. Notice that subtyping is considered up to transitivity; `subtype` therefore does not need to contain the transitive cases. Also notice that there is no implicit promotion between arrays and pointers. We just allow pointer arithmetic and dereferences for left-valued arrays, as explained above.

Similarly the predicate `explicit_cast` relates any two types `ty1` and `ty2` such that a term of type `ty1` can be explicitly cast to type `ty2` (rule `term_typed_cast`). Also this relation is recursive to allow casts between array types if the element types are castable to each other. Note that there can be a sequence of implicit promotions before an explicit cast, so the relation does not contain all the possible cases. Also note that any type can of course be cast to itself, which justifies the base case of the relation. Casts between structure types, which are supposed to be applied recursively based on the order of their fields, are currently not supported. Also, we omit support for casts between integers and pointers, for reasons explained in Section 4.1.4.

All these casts are intended to convert right values and correspond to actual transformations of the representation of the value. A second class of casts are pointer casts (rule `term_typed_cast_pointer`) which are neutral in representation. In addition to casts between two pointers of different types, in ACSL we want to allow casting pointers to aggregate types and back if the array or structure lies in C memory, i.e. is a left value. A term of type `ty1`, that is a left value or not depending on `lr1`, may thus be cast to type `ty2` such that

the resulting term is a left or right value depending on `lr2`, if both pairs respect the following predicate.

```
Definition Is_Pointer_Type lr ty :=
  match lr, ty with
    | false, Tpointer _
    | true, Larray (Tctype _)
    | true, Tarray _ _
    | true, Tstruct _ _ ⇒ True
    | _,_ ⇒ False
  end.
```

A last set of rules concern access to structure fields and array elements. As anticipated above arrays and structures are values in ACSL and an access `a[i]` or `s.f` act as accessing the required element of the aggregate value. But if the original array `a` or structure `s` is a left value then so is `a[i]`, respectively `s.f`. For arrays, in addition to the straightforward rule `term_typed_array_access_r` we thus also have the extra rule `term_typed_array_access_l` for the latter case because `a[i]` is well typed also if `a` is a pointer. Concerning structures, rule `term_typed_field` handles both cases, stating that the type of a field access is computed by the following function.

```
Definition unrolled_field_type fi si fld : option ctype :=
  field_type fi (unroll_composite_fields si (Tstruct si fld) fld).
```

The remaining rules are straightforward, as well as the typing rules for ACSL predicates shown in Fig. 4.7.

Typing Ambiguities

A weakness of these typing rules is that they are ambiguous: the same term can be typed by different typing derivations. This is essentially due to the flexibility introduced by implicit promotions, but also it is not determined by the rules when to apply the `term_typed_lval` rule to get a right value from a left value.

Examples are (with `i` integer and `a` array):

1. `i1 + i2` implicit conversion of `i1` and `i2` to real and then real addition or direct integer addition

2. `(int) i1` implicit conversion to boolean before cast from `boolean` to `int` or direct cast from `integer` to `int`

3. `a.[i]` array access on the left value of `a` or lval to get the right value of `a` and the array access on the logical array

This issue is particularly problematic if the different derivations lead to different semantical evaluations, which is the case in example 2. So it is up to the typing algorithm to "do the right thing", that is to respect certain conventions like not applying unnecessary promotions and applying the `lval`-rule as late as possible. An interesting property that remains to be proved is that the typing rules are non ambiguous up to the same height of derivations.

Typing Algorithm

Given an ACSL expression in abstract syntax possibly constructed by a Frama-C plugin we need to make sure it is well typed according to the present typing rules. We thus define a Coq function `type_term` that given an ACSL term `t` and a typing environment for the logical variables `te` returns a type `ty` and a derivation of the typing rules proving that `t` is well-typed

```
Inductive pred_typed (te:tenv) : pred → Type :=
| pred_typed_false :
            ∅
  -----------------
     te ⊢ Pfalse

| pred_typed_true :
            ∅
  -----------------
     te ⊢ Ptrue

| pred_typed_not p:
        te ⊢ p
  --------------
     te ⊢ Pnot p

| pred_typed_relop op t1 t2 ty:
     te ⊢ʳ  t1 : ty  ·  te ⊢ʳ  t2 : ty
  ----------------------------------        ty ∈ [ Lboolean; Linteger; Lreal ] →
        te ⊢ Pcomp op t1 t2

| pred_typed_relop_pointer op t1 t2 cty1 cty2:
     te ⊢ʳ  t1 : Tpointer cty1 ·  te ⊢ʳ  t2 : Tpointer cty2
  ---------------------------------------------------------
                  te ⊢ Pcomp op t1 t2

| pred_typed_comb op p1 p2:
     te ⊢ p1  ·  te ⊢ p2
  ----------------------                          op ∈ [ Band; Bor; Bimply ] →
     te ⊢ Pcomb op p1 p2

| pred_typed_forall i ty p:
       te.[i <- ty] ⊢ p
  ----------------------
     te ⊢ Pforall i ty p

| pred_typed_let i t ty p:
     te ⊢ʳ  t : ty ·  te.[i <- ty] ⊢ p
  ------------------------------------
          te ⊢ Plet i t p

where "e ⊢ x" := (pred_typed e x).
```

Figure 4.7: ACSL typing rules for predicates

```
Program Fixpoint type_term te t
 : option { lr : bool & { ty : type & te ⊢(lr) t : ty } } :=
 match t with
 | Tbinop op t1 t2 ⇒
     let? (lr1,(ty1,tty1)) := type_term te t1 in
     let? (lr2,(ty2,tty2)) := type_term te t2 in
       if op ∈ [ Bplus; Bminus; Bmult; Bdiv; Bmod; Bbwand; Bbwor ] then
         if ty1 ≤ Linteger & ty2 ≤ Linteger then
           right Linteger
         else if op ∈ [ Bplus; Bminus; Bmult; Bdiv ]
                                       & ty1 ≤ Lreal & ty2 ≤ Lreal then
           right Lreal
         else if op ∈ [ Bplus; Bminus ] & ty2 ≤ Linteger then
           let? (cty, _) := get_pointer_type lr1 ty1 in
             right (Tpointer cty)
         else error
       else if op ∈ [ Band; Bor ] & ty1 ≤ Lboolean & ty2 ≤ Lboolean then
         right Lboolean
       else error
 | Tcondition t0 t1 t2 ⇒
     let? (lr0,(ty0, tty0)) := type_term_ t0 in
       if ty0 ≤ Lboolean then
         let? (_,(ty1, tty1)) := type_term_ t1 in
         let? (_,(ty2, tty2)) := type_term_ t2 in
         let? (ty, _, _) := unify tty1 tty2 in
           right ty
       else error
 | Tcast ty' t ⇒
     let? (lr,(ty, tty)) := type_term_ t in
     let? (lr',_) := explicitly_castable tty ty' in
       return (lr' & (ty' & _))
 | ...
```

Figure 4.8: The typing algorithm (excerpt)

with type `ty`. Additionally this function has to decide whether `t` is a left value. Its definition, by recursion on the structure of terms, is too long to be shown here. Fig. 4.8 shows only some examples of how the **Program** mechanism automatically generates typing derivations guided by the specified output type.

We use the notations

```
Notation left x := (Some (existT _ true (existT _ x _))).
Notation right x := (Some (existT _ false (existT _ x _))).
```

That is `right ty` creates an existential triple, such that the third component of type `te ⊢^r t : ty` must be inferred by Coq; similarly for `left ty`. These notations allows us specifying only how a given term should be typed with Coq automatically constructing the appropriate typing derivation. In most cases Coq would be able to infer the type automatically, too. For the above considerations it is however better to keep the control.

The case of arithmetic operations suggests that subtyping must be decidable: we use the infix notation `ty1 ≤ ty2` for a function that returns `true` if `subtype* ty1 ty2` is valid. Fig. 4.9 shows the way we define that function. By the use of tactics we define a first function `get_subtype_scalar` which returns a proof of `subtype* ty1 ty2` if `eauto` can find it automatically, which is the case if `ty1` and `ty2` are scalar types, `None` otherwise. This function is called from `is_subtype`, a boolean function that is proved correct separately.

```
Definition get_subtype_scalar ty1 ty2 : option (subtype*  ty1 ty2).
Proof.
  destruct ty1 as [| | | |[]], ty2 as [| | | |[]]; eauto 8 || right.
Defined.

Fixpoint is_subtype ty1 ty2 : bool :=
  if ty1 == ty2 then
    true
  else if get_subtype_scalar ty1 ty2 then
    true
  else
    match ty2,ty1 with
      | Larray ty2, (Tarray ty1 _| Larray ty1) ⇒ is_subtype ty1 ty2
      | _,_ ⇒ false
    end.

Lemma is_subtype_correct ty1 ty2:
  is_subtype ty1 ty2 = true → subtype*  ty1 ty2.

Remark is_subtype_complete ty1 ty2:
  subtype*  ty1 ty2 → is_subtype ty1 ty2 = true.
```

Figure 4.9: Decidable Subtyping

Besides being correct, `is_subtype` should also be *transitively complete*, i.e. if `ty1` is a subtype of `ty2`, then it should return `true`. This is important because it would lead to hard-to-debug typing errors if there were some `tya`, subtype of some `tyb`, itself subtype of some `tyc`, such that `is_subtype tya tyc = false`. Notice that we would have had to prove transitive completeness even if we had wanted to include the transitive cases directly inside the relation `subtype`.

A second case where subtyping comes into play is the ternary condition `t ? t1 : t2`, but also in comparisons `<`, `<=`, `==`, etc. Here we need to *unify* the types of `t1` and `t2` by finding their *least upper bound* in the subtyping hierarchy. As shown in Fig. 4.10 an upper bound of `ty1` and `ty2` is a type `ty'` such that both `ty1` and `ty2` are subtypes. Just as for the decisional function for subtyping, we define a first function to find an such an upper bound of two types by the use of tactics. Setting `ty'` to an existential variable, the `eauto` tactic tries to find two appropriate subtyping derivations. As a side effect, the existential variable is instantiated to the actual upper bound. Since `eauto` performs a breadth-first search the found upper bound happens to be minimal, i.e. the *least upper bound* (cf. `get_ub_minimal`). In order to handle array types, we then define the main function `get_ub`, which if `get_ub_scalar` fails tries to unify the two given types as arrays. Notice that here we need to duplicate some code to satisfy Coq's structural-decrease condition. The following, more direct definition is invalid:

```
Fail Program Fixpoint get_ub ty1 ty2 : option (ub ty1 ty2) :=
  match get_ub_scalar ty1 ty2, ty2, ty1 return _ with
    | Some x, _,_ ⇒ Some x
    | _, (Larray ty2 | Tarray ty2 _), (Larray ty1 | Tcarray ty1 _) ⇒
          let? ty := get_ub ty1 ty2 in
            return Larray (dfst3 ty)
    | _,_,_ ⇒ error
  end.
```

In the case where `ty1` is a `Tarray ty1'` and `ty2` is a `Tarray ty2'`, the function would call itself recursively with the arguments `ty1'` and `ty2'` which both have the type `ctype` and

```
Notation ub ty1 ty2 := { ty' : type & subtype*  ty1 ty' & subtype*  ty2 ty'}.

Definition get_ub_scalar ty1 ty2 : option (ub ty1 ty2).
Proof.
  destruct (ty1 == ty2); eauto.
  destruct ty1 as [| | | |[]], ty2 as [| | | |[]]; eauto 12 || right.
Defined.

Program Fixpoint get_ubc ty1 (ty2: ctype) : option (ub ty1 ty2) :=
  match get_ub_scalar ty1 ty2, ty2, ty1 with
    | Some x, _,_ ⇒ Some x
    | _, Tarray ty2 _, (Larray ty1 | Tcarray ty1 _) ⇒
          let? ty := get_ubc ty1 ty2 in
            return Larray (dfst3 ty)
    | _,_,_ ⇒ error
  end.

Program Fixpoint get_ub ty1 ty2 : option (ub ty1 ty2) :=
  match get_ub_scalar ty1 ty2, ty2, ty1 with
    | Some x, _,_ ⇒ Some x
    | _, Larray ty2, (Larray ty1 | Tcarray ty1 _) ⇒
          let? ty := get_ub ty1 ty2 in
            return Larray (dfst3 ty)
    | _, Tarray ty2 _, (Larray ty1 | Tcarray ty1 _) ⇒
          let? ty := get_ubc ty1 ty2 in
            return Larray (dfst3 ty)
    | _,_,_ ⇒ error
  end.

Program Definition unify te t1 lr1 ty1 t2 lr2 ty2
    (tty1: te ⊢(lr1) t1 : ty1) (tty2: te ⊢(lr2) t2 : ty2)
  : option { ty' : type & (te ⊢ʳ  t1 : ty') & (te ⊢ʳ  t2 : ty') } :=
  let? (ty, _, _) := get_ub ty1 ty2 in
    return ty.

Remark get_ub_complete ty2 ty1: ub ty1 ty2 → get_ub ty1 ty2 ≠ None.

Remark get_ub_minimal ty2 ty1 x: get_ub ty1 ty2 = Some x →
  ∀ty', subtype*  ty1 ty' → subtype*  ty2 ty' → subtype*  (dfst3 x) ty'.
```

Figure 4.10: Unification

thus are implicitly coerced to `type`. Considering the coercions the full recursive call would be `get_ub (Tctype ty1') (Tctype ty2')` where none of the arguments is structurally smaller of the original arguments `ty1` and `ty2`. To avoid this we need the second function `get_ubc` which does the same as `get_ub` but specifically for `ctype`s.

As for `is_subtype` we prove that `get_ub` is complete, but also that the returned upper bound is minimal. This ensures that the typing algorithm does not introduce unnecessary implicit promotions when calling `unify` to handle ternary conditions or comparisons.

Similarly implicit promotions may be necessary to type explicit casts. The term `(ty2) t` is well typed if `t` has type `ty1` which is a subtype of some `ty'` such that there is an explicit cast from `ty'` to `ty2`:

```
Notation indirect_explicit_cast ty1 ty2 :=
  { ty' : type & subtype*  ty1 ty' & explicit_cast ty' ty2 }.
```

We have to be able to infer this `ty'`, so we define a helper function similar to `get_ub`

```
Definition get_explicit_cast ty1 ty2
   : option (indirect_explicit_cast ty1 ty2).
```

Its definition is similar to the one of `get_ub` and similarly we prove that it is complete and minimal. We use this function in the following way, additionally checking if it is a pointer cast:

```
Program Definition explicitly_castable te t lr ty1 ty2
    (tty: te ⊢(lr) t : ty1)
   : option { lr' : bool & te ⊢(lr') Tcast ty2 t : ty2 } :=
  match get_pointer_cast ty1 lr ty2, get_explicit_cast ty1 ty2 with
    | Some lr,_ ⇒ return (lr & _)
    | None, Some _ ⇒ return (false & _)
    | None, None ⇒ error
  end.
```

This concludes the presentation of the typing algorithm for ACSL terms. ACSL predicates are typed in a similar way:

```
Program Fixpoint type_pred te p : option (te ⊢ p) :=
  match p with
    ...
  end.
```

4.1.4 Semantics of Terms and Predicates

ACSL semantics is informally specified in the reference manual. Some important semantical guidelines for formal semantics are however given in this document. First, all the functions used in logical terms shall be total. Even naturally partial functions, like arithmetic division, shall never be undefined. Every function must return an output value for every input, even if the output can be unspecified for some input. These functions are called underspecified. Typically, the only thing one knows about an unspecified value is that it is equal to itself. For instance the ACSL predicate $x/0 = x/0$ is valid for every x, but cannot say anything about $5/0 = 4/0$. Other examples for partial functions are operations involving memory, like pointer dereference or pointer comparison.

Underspecified total functions in Coq

The semantics of ACSL logic expressions is supposed to be based on mathematical first-order logic. A natural way to formally specify the semantics of terms is by mapping them to the first-order fragment of the logic of Coq. To this purpose we want to map arithmetic operations and functions and predicates concerning the memory with their Coq implementations in `ZArith` and Compcert. But in Coq there is no such thing as an underspecified function. A Coq function with the type `Z → Z → Z`, like for instance `ZOdiv`, returns a well specified integer for every combination of arguments. If this function implements some mathematical operation that is not defined for some particular input, like division, then this means that it returns just some default value for such an input. In the case of the division, the Coq function `Zdiv` returns `0` for every `0` divisor. Thus the Coq proposition `ZOdiv 4 0 = ZOdiv 5 0` is valid. This cannot be the semantics of the division operator in ACSL.

Similarly many functions implementing operations on the memory model are not total functions. Their result type is often an option type to model failure of this operation. For the sake of example consider a Coq function `mem_load_int : mem → pointer → option int` which given the current state and a pointer returns the integer read in the pointed memory cell, if the pointer is valid and

we have the permission for a read access in that cell. This may be the semantics of a pointer dereference in the case of an int pointer, but we need a total function that returns some value even if we cannot read one using the given pointer. To this purpose we could define

```
Definition mem_load_int_complete m p :=
  match mem_load_int m p with
  | Some i ⇒ i
  | None ⇒ Int.zero
  end.
```

But just as for `ZOdiv` we would have that `mem_load_int_complete m p1` = `mem_load int_complete m p2` for each `p1`, `p2` invalid in `m` and we don't want this for ACSL.

Instead, to get as close as possible to the concept of underspecified functions, we use the Coq mechanisms of *subset types* and *opaque definitions*. A subset type `{ x:A | P x }`, denotes the subset of elements of the type `A` that satisfy the predicate `P`. It is used to specify properties about the result of a function inside its type. The ``` ` ``` notation is available for the projection from such a subset type to the full type of elements `A`. On the other hand, opaque definitions are mostly used for proofs of theorems, where the actual implementation of the type does not matter. In Coq an opaquely defined reference is not reducible and cannot be proved equal to its definition. With these ingredients we can define the following:

```
Definition mem_load_int_spec m p :
   { i : int | forall i', mem_load_int m p = Some i' →  i = i' }.
Proof.
  exists (match mem_load_int m p with
          | Some i ⇒ i
          | None ⇒ Int.zero
         end).
  destruct (mem_load_int m p); inversion 1; auto.
Save.

Definition mem_load_int_complete m p := ` (mem_load_int_spec m p).
```

Having terminated the definition of `mem_load_int_spec` with **Save**, its definition is opaque and everything we will ever know about its result is the specification in its type. Hence, the obtained function has exactly the property we want: if `mem_load_int m p` returns some well defined value, then `mem_load_int_complete m p` is provably equal to that value. On the other hand, if `mem_load_int m p` is `None` then we cannot say anything about the result. In particular we cannot prove it equal to `Int.zero`.

We can generalise this approach to all partial functions whose return type is inhabited. Using type classes we can declare the needed default value once and for all for any needed type.

```
Class Inhabited A := Inhabited_witness : A.

Program Definition unoption A {d: Inhabited A} (o: option A)
   : { a : A | forall a', o = Some a' → a = a' } :=
  match o return _ with
    | Some a ⇒ a
    | None ⇒ d
  end.
```

The **Program** mechanism simplifies the definition by finding the required proof automatically. Figure 4.11 shows the definition of a family of `complete` operators, which given a partial

```
Section Complete.

  Context
    {A: Type}
    {B: forall (a:A), Type}
    {C: forall (a:A) (b: B a), Type}
    {d1: forall (a:A), Inhabited (B a)}
    {d2: forall (a:A) (b: B a), Inhabited (C a b)}
    (f1: forall (a:A), option (B a))
    (f2: forall (a:A) (b: B a), option (C a b))
    (a:A) (b: B a)
  .
  Definition compl1 : { x : B a | ∀x', f1 a = Some x' → x = x' }.
  Proof unoption (f1 a).

  Definition compl2 : { x : C a b | ∀x', f2 a b = Some x' → x = x' }.
  Proof unoption (f2 a b).

  Definition complete1 := `compl1.
  Definition complete2 := `compl2.

End Complete.
```

Figure 4.11: Complete operators for functions of arities 1 and 2

function of an appropriate arity and an inhabited return type, return a new, total function that behaves like the original function in its domain of definition.

Using one of these operators we can simplify the definition of our complete load function for integers, after having declared an instance of an inhabited type:

```
Instance default_int : Inhabited int := Int.zero.

Definition mem_load_int_complete := complete2 mem_load_int.
```

Denotational semantics

We are now ready to formally specify the semantics of ACSL terms and predicates, that is their denotation in the logic of Coq. As ACSL terms are multi-sorted, we start by giving the semantics of C types and ACSL types as Coq types (Fig. 4.12). The semantics of a type establish the semantical representations of a value of that type. It should precisely define the range of values that an term of a given type can assume.

For instance a term of type `signed int32` should assume no other values than integers of the range $[-2^{31}...2^{31} - 1]$ and term of type `char`, integers of the range $[0..255]$. We thus use bounded integers as type for machine integers to precisely capture the limits in their size. The bounds of the integer they can hold are calculated in function of a given `intsize` and `signedness`. For floats we use their axiomatisation of Compcert. Pointers are represented by the `address` type, which is a pair of a block and an offset, just like in Compcert. As in Compcert a valid pointer can never have its block equal to zero, we decide that such an `address` with `adr_block = 0` represents a pointer that has been casted from an integer, e.g. the null pointer is represented by as `mk_address 0 Int.zero`. Values of type array are essentially mappings from integers to elements, but we also keep the size indicated at creation of the array. Structures and unions are mappings from field names to elements. Precisely their semantical

```
Definition bZ sz sg := {z : Z | In_range sz sg z}.

Definition single := { f : float | f = Float.singleoffloat f }.

Definition bfloat sz :=
  match sz with
    | F32 ⇒ single
    | F64 ⇒ float
  end.

Record address := mk_address {
  adr_block : block;
  adr_ofs : int
}.

Record array T := {
  array_length : Z;
  array_elements : Z → T
}.

Fixpoint den_ctype cty : Type :=
  match cty with
    | Tint sz sg ⇒ bZ sz sg
    | Tfloat sz ⇒ bfloat sz
    | Tpointer x ⇒ address
    | Tvoid ⇒ unit
    | Tarray cty _ ⇒ array (den_ctype cty)
    | Tfunction _ _ ⇒ address
    | Tstruct _ fld ⇒ ∀fi, den_field_type fi fld
    | Tunion _ fld ⇒ ∀fi, den_field_type fi fld
    | Tcomp_ptr _ ⇒ address
  end

with den_field_type (fi: ident) (fld: fieldlist) : Type :=
  match fld with
    | Fnil ⇒ unit
    | Fcons fi' t fld' ⇒ if ident_eq fi fi' then den_ctype t
                         else den_field_type fi fld'
  end.

Fixpoint den_type ty : Type :=
  match ty with
    | Linteger ⇒ Z
    | Lreal ⇒ R
    | Lboolean ⇒ bool
    | Larray ty ⇒ array (den_type ty)
    | Tctype cty ⇒ den_ctype cty
  end.
```

Figure 4.12: Denotational semantics of ACSL types

representation is given by mutually recursive functions, because the type of C types is mutually inductive.

We take advantage of this definition of denotations of types and of the typing rules for terms we previously defined to directly state their denotational semantics as Coq values of the correct type. If a term is a left value, then its denotation is a memory address, and if it's a right value, then its denotation has the type that is the denotation of the type of the term:

```
Definition den_lrtype lr ty := if lr then address else den_type ty.
```

The denotational semantics are then defined as a fixed point over the well-typing relation:

```
Fixpoint evalterm te (G_e: Env te) t lr ty
   (tty: term_typed te t lr ty) : den_lrtype lr ty :=
   match tty in term_typed _ t lr ty return den_lrtype lr ty with
     | ...
   end.
```

Inside the match the return type `den_lrtype lr ty` simplifies to whatever type is required as a result for the logic term in question and we can concisely express its denotation without worrying about typing issues in most cases. The whole definition of `evalterm` is shown in Figs. 4.13 and 4.14. It depends on a given evaluation environment `G_e` which assigns a value to each logic variable declared in the typing environment `te` and which is updated in case of a local definition (`\let x = t1; t2`). Additionally, the semantics is defined under the current memory state `m`, the set of previous, labelled memory states `lm` and the global and current local C environments `cge` and `ce` which contain the memory allocations of the global and local C variables, respectively.

These environments and the memory model are the same used in the specification of the standard Clight semantics. We just add the finite map from labels to memory states `lm` for the semantics of logical expressions referring to previous states. When evaluating a post-condition `lm` will contain a binding for the label `\old`. Inside the body of a function it is possible to enrich `lm` with user-defined labels which can then be referred to in any assertion or loop annotation inside its scope.

In the definition of the semantics we use the **Program** environment to automatically resolve the contradictions, indicated by a `!`, due to cases of inner matches that can be excluded by typing.

Most of the functions referred to in these semantics are taken from the Coq standard library or obvious variations. Notably exceptions are the functions interfacing with the Compcert memory model `accs_progvar` and `logic_load`. The former looks up the given identifier in the local and then in the global C evaluation environment and checks if the given type coincides with the one registered for that identifier:

```
Definition accs_progvar cge ce i cty : address :=
   match ce.[i], Genv.find_symbol cge i with
     | Some (b, cty'), _ ⇒
         if cty' == cty then return mk_address b Int.zero
         else error
     | None, Some b ⇒
         let? cty' := type_of_global cge b in
           if cty' == cty then return mk_address b Int.zero
           else error
     | None, None ⇒ error
   end.
```

It is, appropriately completed to a total function, the semantics of a program variable occurrence `term_typed_progvar` which, being a left value, is just a memory address. Note that also a pointer dereference just evaluates to the pointer's address as a left value. The actual right value is then read from memory when needed (typing rule `term_typed_lval`).

The definition of `logic_load` is a bit more involved. Given a memory `m`, an address `a` and a type `ty`, which might be an aggregate type, it should return the value of type `den_type ty` that resides in `m` at address `a`. It should refer to Compcert's `Csem.load_value_of_type`

```
Variable cge: genv.
Variable ce: env.
Variable m: mem.
Variable lm : lmem.

Program Fixpoint evalterm te (G_e: Env te) t lr ty
     (tty: term_typed te t lr ty): den_lrtype lr ty :=
   match tty in term_typed _ t lr ty return den_lrtype lr ty with
      | term_typed_integer z ⇒ z
      | term_typed_real r ⇒ IrealR r
      | term_typed_boolean b ⇒ b
      | term_typed_null ⇒ null
      | term_typed_var i ty x ⇒ G_e _ _ x
      | term_typed_progvar i cty _ ⇒ complete4 accs_progvar cge ce i cty
      | term_typed_binop_integer op t1 t2 tty1 tty2 _ ⇒ match op with
         | Bplus ⇒ Zplus
         | Bminus ⇒ Zminus
         | Bmult ⇒ Zmult
         | Bdiv ⇒ complete2 Zdiv_round
         | Bmod ⇒ complete2 Zmod_round
         | Bbwand ⇒ Zand
         | Bbwor ⇒ Zor
         | Bbwxor ⇒ Zxor
         | _ ⇒ !
        end (evalterm G_e tty1) (evalterm G_e tty2)
      | term_typed_binop_real op t1 t2 tty1 tty2 _ ⇒
        match op with
         | Bplus ⇒ Rplus
         | Bminus ⇒ Rminus
         | Bmult ⇒ Rmult
         | Bdiv ⇒ Rdiv
         | _ ⇒ !
        end (evalterm G_e tty1) (evalterm G_e tty2)
      | term_typed_binop_boolean op t1 t2 tty1 tty2 _ ⇒
        match op with
         | Band ⇒ andb
         | Bor ⇒ orb
         | _ ⇒ !
        end (evalterm G_e tty1) (evalterm G_e tty2)
      | term_typed_binop_pointer op t1 t2 cty lr ty tty1 _ tty2 _ ⇒
        match op with
         | Bplus ⇒ fun a z, address_shift_array a cty z
         | Bminus ⇒ fun a z, address_shift_array a cty (-z)
         | _ ⇒ !
        end (casT (evalterm G_e tty1)) (evalterm G_e tty2)
      | term_typed_relop op t1 t2 ty tty1 tty2 _ ⇒ den_relop op
        match ty with
         | Linteger ⇒ Zcompare
         | Lreal ⇒ Rcompare
         | Lboolean ⇒ Bcompare
         | _ ⇒ !
        end (evalterm G_e tty1) (evalterm G_e tty2)
      | term_typed_relop_pointer op t1 t2 cty1 cty2 tty1 tty2 ⇒
         complete4 address_compare m op
           (evalterm G_e tty1) (evalterm G_e tty2)
```

Figure 4.13: Denotational semantics of ACSL terms

```
| term_typed_unop_integer op t tty1 _ ⇒
  match op with
    | Uplus ⇒ id
    | Uminus ⇒ Zopp
    | Ubw_compl ⇒ Znot
    | _ ⇒ !
  end (evalterm G_e tty1)
| term_typed_unop_real op t tty1 _ ⇒
  match op with
    | Uplus ⇒ id
    | Uminus ⇒ Ropp
    | _ ⇒ !
  end (evalterm G_e tty1)
| term_typed_unop_boolean op t tty1 _ ⇒
  match op with
    | Uneg ⇒ negb
    | _ ⇒ !
  end (evalterm G_e tty1)
| term_typed_condition _ _ _ _ tty0 tty1 tty2 ⇒
    if (evalterm G_e tty0) then (evalterm G_e tty1) else (evalterm G_e tty2)
| term_typed_let i _ _ _ _ tty1 tty2 ⇒
    evalterm (add i (evalterm G_e tty1) G_e) tty2
| term_typed_cast t ty1 ty2 tty ec ⇒
    den_explicit_cast ec (evalterm G_e tty)
| term_typed_cast_pointer t lr1 ty1 lr2 ty2 x1 x2 x ⇒ casT (evalterm G_e x)
| term_typed_promote _ _ ty' tty ic ⇒
    den_trans_conversion den_implicit_cast ic (evalterm G_e tty)
| term_typed_deref t cty lr ty x _ ⇒ casT (evalterm G_e x)
| term_typed_lval t ty x ⇒ logic_load ty (evalterm G_e x) m _
| term_typed_at l t ty x ⇒ logic_load ty (evalterm G_e x)
     (complete2 access_label lm l) _
| term_typed_addrof t cty x ⇒ evalterm G_e x: _
| term_typed_field t ti fld fi lr cty x x0 ⇒
    (if lr return den_lrtype lr (Tstruct ti fld) → den_lrtype lr cty
      then fun pt, address_shift_field pt fi fld
      else fun str, cast (T:=id) (str fi) )
    (evalterm G_e x)
| term_typed_array_access_l ta ti cty lr ty x0 _ x1 ⇒
    address_shift_array (casT (evalterm G_e x0)) cty (evalterm G_e x1)
| term_typed_array_access_r ta ti ty x0 x1 ⇒
    (evalterm G_e x0).(array_elements) (evalterm G_e x1)
| term_typed_array_update ta ti tx ty x x0 x1 ⇒
    array_update (evalterm G_e x) (evalterm G_e x0) (evalterm G_e x1)
| term_typed_struct_update ts fi tx si fld cty x x0 x1 ⇒
    @struct_update fld (evalterm G_e x) fi (cast (T:=id) (evalterm G_e x0))
| term_typed_empty_array ty tl x ⇒
    {| array_length := evalterm G_e x; array_elements := unspecified |}
| term_typed_empty_struct si fld ⇒ unspecified
d.
```

Figure 4.14: Denotational semantics of ACSL terms (continued)

to handle basic types and proceed recursively in case of aggregate types. An important point though is that `load_value_of_type` returns a value of type `val` defined as follows:

```
Inductive val: Type :=
  | Vundef: val
  | Vint: int →  val
  | Vfloat: float →  val
  | Vptr: block →  int →  val.
```

That is a sum type that represents any kind of basic value. The result of a load is thus potentially in no relation with the requested type. As we want the above described constraints to hold about values denoting ACSL terms, we need to check them on such low-level values returned by `load_value_of_type`. We therefore define a function from these untyped basic values, to dependently typed values of the desired type:

```
Definition type_val cty v : option (den_ctype cty) :=
  match cty,v with
    | Tint sz sg, Vint i      ⇒ bZ_of_int sz sg i
    | Tfloat _,   Vfloat f   ⇒ return f
    | Tpointer _, Vint i      ⇒ return mk_address 0 i
    | Tpointer _, Vptr b ofs ⇒
        if b == 0 then error else return mk_address b ofs
    | _,_ ⇒ error
  end.
```

where `bZ_of_int` is a partial function returning the integer that is represented by a given machine integer if it respects the bounds. `type_val` will be used in the following everywhere we need to relate Compcert output values with well-typed logical values.

The formal semantics of a logical memory access is given in Fig. 4.15. We first define a `csimple_load` which reads one basic value from memory and is used inside `clogic_load` which collects aggregate values from memory. To finally define `logic_load`, which is supposed to handle any ACSL type, we need to restrict them to types actually being able to reside in memory, that is C types or logic arrays of C types. This is the purpose of the predicate `Valid_left_type`, which is true for types of terms well-typed as left values:

Remark `term_ltyped_valid te t ty:` $te \vdash^l t : ty \rightarrow$ `Valid_left_type ty.`

This concludes the specification of denotational semantics of ACSL terms. The semantics of ACSL predicates are specified in Fig. 4.16. It gives the denotation of an ACSL predicate as a Coq predicate referring to the semantics of terms when needed. For instance the semantics of an universal quantification in ACSL over a given identifier is a universal quantification in Coq over a value of the given type which is assigned to the identifier in the evaluation environment for the sub term.

To be integrated into Clight whose semantics are given for the untyped abstract syntax we say that an untyped ACSL predicate is valid for a given logical typing environment `te` and an appropriate logical evaluation environment `G_e`, if it is well-typed under `te` and it evaluates to a valid Coq predicate `G_e`.

```
Definition valid_acsl te (G_e: Env te) p :=
  ∃pty : pred_typed te p, evalpred cge ce m lm G_e pty.

Definition valid_closed_acsl := valid_acsl empty.
```

Recall that the semantics of predicates are defined under the global C environment `cge`, a current local C environment `ce`, a current memory state `m` and the stack of previous, labelled

```
Definition csimple_load cty m b ofs :=
  let? v:= Csem.load_value_of_type cty m b ofs in
    type_val cty v.

Fixpoint clogic_load cty b ofs m : den_ctype cty :=
  match cty with
    | (Tint _ _ | Tfloat _ | Tpointer _ ) as cty ⇒
        complete4 csimple_load cty m b ofs
    | Tvoid ⇒ tt
    | Tarray cty _ ⇒
        {|array_length :=
            Zdiv (Mem.high_bound m b – Int.unsigned ofs) (sizeof cty);
          array_elements i :=
            clogic_load cty b (ofs_shift_array ofs cty i) m|}
    | Tfunction _ _ ⇒ mk_address b ofs
    | Tstruct _ fld ⇒ fun fi, clogic_field_load fi fld b
                                  (ofs_shift_field ofs fi fld) m
    | Tunion _ fld ⇒ fun fi, clogic_field_load fi fld b ofs m
    | Tcomp_ptr _ ⇒ complete4 csimple_load (Tpointer Tvoid) m b ofs
  end

with clogic_field_load fi fld b ofs m: (den_field_type fi fld) :=
  match fld return (den_field_type fi fld) with
    | Fnil ⇒ tt
    | Fcons fi' t fld' ⇒
        if ident_eq fi fi' as x
         return if x then den_ctype t else den_field_type fi fld'
         then clogic_load t b ofs m
         else clogic_field_load fi fld' b ofs m
end.

Program Definition logic_load ty a m (_: Valid_left_type ty)
  : den_type ty :=
  let b := a.(adr_block) in let ofs := a.(adr_ofs) in
  match ty with
    | Tctype cty ⇒ clogic_load cty b ofs m
    | Larray (Tctype cty) ⇒ clogic_load (Tarray cty 0) b ofs m
    | _ ⇒ !
  end.
```

Figure 4.15: Logical load

memory states `lm`, so the full type of `valid_closed_acsl` is

$$\mid \texttt{genv} \rightarrow \texttt{Clight.env} \rightarrow \texttt{mem} \rightarrow \texttt{lmem} \rightarrow \texttt{pred} \rightarrow \textbf{Prop}$$

4.2 Formalisation of Annotated Clight

The purpose of this section is to formalise the abstract syntax and semantics of ACSL-annotated Clight. It will be the front end of our certified tool chain as annotated Clight will be directly generated by a Frama-C plugin. Clight is a simplified C with only pure expressions and used to be the front end of the certified part of the Compcert compiler before the introduction of Compcert C.

```
Program
Fixpoint evalpred te (G_e: Env te) p (pty: pred_typed te p) :=
  match pty return _ with
    | pred_typed_true ⇒ True
    | pred_typed_false ⇒ False
    | pred_typed_not _ pty ⇒ not (evalpred G_e pty)
    | pred_typed_relop op t1 t2 ty tty1 tty2 _ ⇒
        (match ty with
          | Linteger ⇒ den_relop_Z op
          | Lreal ⇒ den_relop_R op
          | Lboolean ⇒ den_relop_prop op bool_lt
          | _ ⇒ !
        end) (evalterm G_e tty1) (evalterm G_e tty2)
    | pred_typed_relop_pointer op t1 t2 cty1 cty2 tty1 tty2 ⇒
        complete4 address_compare m op
          (evalterm G_e tty1) (evalterm G_e tty2) = true
    | pred_typed_comb op p1 p2 pty1 pty2 _ ⇒
      match op with Band ⇒ and | Bor ⇒ or | Bimply ⇒ impl end
      (evalpred G_e pty1) (evalpred G_e pty2)
    | pred_typed_forall i ty p pty ⇒
        ∀x : den_type ty, evalpred (add i x G_e) pty
    | pred_typed_let i t ty p tty pty ⇒ evalpred
      (add i (evalterm G_e tty) G_e) pty
  end.
```

Figure 4.16: Denotational semantics of ACSL predicates

4.2.1 Abstract Syntax

We extend the abstract syntax of Clight statements with assertions and loop invariants. Clight function definitions are extended with function contracts. The resulting abstract syntax is shown in Fig. 4.17. The only changes with respect to Clight are `Sassert`, which is new, and the looping statements `Swhile`, `Sdowhile` and `Sfor'`, where we add a predicate for a loop invariant and an optional list of logic terms representing the loop assigns clause, that is a list of left values that may be modified inside the loop.

Function contracts are represented by a pre- and a post-condition predicate and the names of the `\old` label and the `\result` variable to be referred to in the post-condition. Additionally, a function contract can have an assigns clause, similarly to loop invariants.

4.2.2 Semantics

The semantics of annotated Clight is given in a blocking big-step style. During execution, every encountered annotation must be valid in the current state, otherwise the execution blocks. The concerned rules are shown in Figs. 4.18 and 4.19. The only rule that allows executing an assertion `exec_Sassert` requires the asserted predicate to be valid. Similarly, the loop invariants need to be valid at every loop iteration. The loop assigns clause is currently not considered in the semantics. Its purpose is to strengthen the loop invariant for the WP calculus as explained in the next chapter. It would however be better to clearly specify the semantics of the loop assigns clause.

Intuitively the given list of left values should determine the set of memory locations that are allowed to be modified by an iteration of the loop. The difficulty comes from the fact that the evaluation of the given terms could change between two iterations. Consider for instance a loop assigns clause containing `p` and `*p`. As `p` may be modified, `*p` denotes a different memory location at each iteration. So the set of memory locations to be modified by the loop should be the

```
Inductive statement : Type :=
  | Sassert : pred →  statement
  | Sskip : statement
  | Sassign : expr →  expr →  statement
  | Sset : ident →  expr →  statement
  | Scall: option ident →  expr →  list expr →  statement
  | Ssequence : statement →  statement →  statement
  | Sifthenelse : expr →  statement →  statement →  statement
  | Swhile : pred →  option (list term) →  expr →  statement
      →  statement
  | Sdowhile : pred →  expr →  statement →  statement
  | Sfor': pred →  expr →  statement →  statement →  statement
  | Sbreak : statement
  | Scontinue : statement
  | Sreturn : option expr →  statement
  | Sswitch : expr →  labeled_statements →  statement
  | Slabel : label →  statement →  statement
  | Sgoto : label →  statement
with labeled_statements : Type :=
  | LSdefault: statement →  labeled_statements
  | LScase: int →  statement →  labeled_statements
        →  labeled_statements.

Record function : Type := mkfunction {
  fn_return: ctype;
  fn_params: list (ident * ctype);
  fn_vars: list (ident * ctype);
  fn_temps: list (ident * ctype);
  fn_old: label;
  fn_pre: pred;
  fn_post_result_var : ident;
  fn_post: pred;
  fn_assigns: option (list term);
  fn_body: statement
}.
```

Figure 4.17: Abstract syntax of annotated Clight statements

evaluation of the list of left values before the first loop iteration. This is not trivial to express in the semantics of statements as at a generic loop iteration, the memory state before the first iteration is not available anymore.

The simplest solution is probably to consider a loop assigns clause as syntactic sugar for an enriched loop invariant referring to a label that is an implicitly introduced in front of the whole loop. This could be achieved by adding a semantical rule for **while** loops with an assigns clause, that declares the label and integrates the clause into the loop invariant using some built-in predicate `Ppreserve`:

```
| exec_Swhile_start: forall e le m lm inv assigns a s v,
    exec_stmt e le m lm.[l_pre <- m]
      (Swhile (Pcomb Band inv (Ppreserve l_pre assigns)) None a s)
            t le1 m1 out →
    exec_stmt e le m lm (Swhile inv (Some assigns) a s)
            t le1 m1 out
```

If all the other rules for **while** loops require that the assigns clause is `None`, this would lead to the desired result of storing the current state before the first loop iteration.

```
Inductive exec_stmt: env →  temp_env →  mem →  lmem →  statement
 →  trace →  temp_env →  mem →  outcome →  Prop :=
| ...
| exec_Sassert p e le m lm:
    valid_closed_acsl e m lm p →
    exec_stmt e le m lm (Sassert p) E0 le m Out_normal
| exec_Swhile_false e le m lm inv assigns a s v:
    valid_closed_acsl e m lm inv →
    eval_expr e le m a v →
    bool_val v (typeof a) = Some false →
    exec_stmt e le m lm (Swhile inv assigns a s)
              E0 le m Out_normal
| exec_Swhile_stop e le m lm inv assigns a v s t le' m' out' out:
    valid_closed_acsl e m lm inv →
    eval_expr e le m a v →
    bool_val v (typeof a) = Some true →
    exec_stmt e le m lm s t le' m' out' →
    out_break_or_return out' out →
    exec_stmt e le m lm (Swhile inv assigns a s)
              t le' m' out
| exec_Swhile_loop e le m lm inv assigns a s v t1 le1 m1 out1
  t2 le2 m2 out:
    valid_closed_acsl e m lm inv →
    eval_expr e le m a v →
    bool_val v (typeof a) = Some true →
    exec_stmt e le m lm s t1 le1 m1 out1 →
    out_normal_or_continue out1 →
    exec_stmt e le1 m1 lm (Swhile inv assigns a s) t2 le2 m2 out →
    exec_stmt e le m lm (Swhile inv assigns a s)
              (t1 ** t2) le2 m2 out
| exec_Slabel e le m lm l s t le' m' out:
    exec_stmt e le m lm.[l <- m] s t le' m' out →
    exec_stmt e le m lm (Slabel l s)
              t le' m' out
```

Figure 4.18: Big-step semantics of annotated Clight statements (assertions, loops and labels)

For the execution of a function call to be successful, the pre-condition must be valid in the pre-state, that is before the local variables are allocated on the stack, and the post-condition must be valid in the post-state, that is after the local variables are freed. Note that the formal parameters are considered as logic variables in the function contract. This avoids the need to specify that they should refer to their values interpreted in the pre-state as in the reference manual. On the contrary inside the function's body the formal parameters are considered program variables to their full effect.

With these semantics if a program succeeds to execute without blocking then it is guaranteed to respect all of its annotations and notably its function contracts.

Another change with respect to the semantics of Clight is the additional parameter of type `lmem`, that is a finite map from identifiers to memory states. When executing a labelled statement, the current memory state is added to the map with the name of the label (rule `exec_Slabel`). Annotations under the label can refer to that memory state using this name.

4.2.3   Annotated C programs in the Compcert Chain

A first simple but important property we prove about our annotated Clight is that if we erase all annotations from a program to obtain a standard Clight program, then the semantics are preserved.

```
| exec_Scall_some:
   forall e le m lm id a al tyargs tyres vf vargs f t m' vres,
     classify_fun (typeof a) = fun_case_f tyargs tyres →
     eval_expr e le m a vf →
     eval_exprlist e le m al tyargs vargs →
     Genv.find_funct cge vf = Some f →
     type_of_fundef f = Tfunction tyargs tyres →
     eval_funcall m f vargs t m' vres →
     exec_stmt e le m lm (Scall (Some id) a al)
                 t (PTree.set id vres le) m' Out_normal

with eval_funcall: mem →  fundef →  list val
  →  trace →  mem →  val →  Prop :=
| eval_funcall_internal:
    forall le m f vargs G_args t e m1 m2 m3 out vres m4 G_post,
      logic_bind_parameters f.(fn_params) vargs G_args →
      valid_acsl empty_env m [] G_args f.(fn_pre) →
      alloc_variables empty_env m (f.(fn_params) ++ f.(fn_vars)) e m1 →
      Coqlib.list_norepet (var_names f.(fn_params) ++ var_names f.(fn_vars)) →
      bind_parameters e m1 f.(fn_params) vargs m2 →
      exec_stmt e (PTree.empty val) m2 [].[f.(fn_old) <- m] f.(fn_body)
                 t le m3 out →
      outcome_result_value out f.(fn_return) vres →
      Mem.free_list m3 (blocks_of_env e) = Some m4 →
      logic_bind_result G_args f.(fn_post_result_var) f.(fn_return) vres G_post→
      valid_acsl empty_env m4 [].[f.(fn_old) <- m] G_post  f.(fn_post) →
      eval_funcall m (Internal f) vargs t m4 vres
```

Figure 4.19: Big-step semantics of annotated Clight statements (function calls)

Erasure of annotations is defined by an obvious recursion over statements. The only particularity is that we also erase labels: a statement node `Slabel l s` simply becomes `s`. This is necessary because the standard Clight semantics don't handle labels as usually their only purpose is to act as entry points for `goto` jumps which are not supported by big-step semantics.

Theorem 10 *Preservation of semantics ClightACSL to Clight*

```
forall p tr i,
  bigstep_program_terminates p tr i →
  Clight.bigstep_program_terminates (erase_program p) tr i.
```

Proof. This is the theorem `erase_program_terminates` of our Coq development. The proof proceeds by induction over the derivation of the semantics of statements and function calls.

This theorem chains up with the preservation properties of the Compcert compiler and guarantees that if we compile an annotated Clight program after having erased the annotations, then the compiled assembler program has the same semantics and therefore respects the annotations.

4.3 Conclusions

We have formalised the ACSL logic language as informally specified in the reference manual and we integrated it in form of code annotations into the Clight language and its big-step operational semantics to obtain formal semantics for ACSL annotated normalised C programs as produced by the FramaC parser.

As requested by the informal specification, we cover array types with logical elements, which are currently missing in the Frama-C implementation. Their formalisation require the duplication of the array type but thanks to its integration into the subtyping system, this does not lead to too many difficulties.

We contribute a concise formalisation of formal parameters inside function contracts, which were not well specified.

Chapter 5

Development of a Certified Verifier for C+ACSL

The purpose of this chapter is to define a certified compilation from ACSL annotated C programs to Whycert programs. Section 5.1 instantiates Whycert by a particular logical context suitable for compilation of annotated Clight: it defines the set of "objects" on which C programs operate, notably machine integers and memory heap. Section 5.2 presents the compilation of basic entities that are variables and labels shared between the compilation of terms and expressions. Section 5.3 presents the compilation of terms and predicates and Section 5.4 presents the compilation of expressions. Section 5.5 presents the compilation of statements and functions, then states and proves the main result of this chapter, i.e. the soundness of the verification condition generation for annotated Clight programs. Section 5.6 explains how we extract a Frama-C plugin independent from Coq and presents some representative examples. Section 5.7 discusses about some choices made and difficulties encountered during this development.

5.1 A Whycert Logical Context for Annotated C Programs

Whycert as defined in Chapter 3 is very generic. It depends on a logical context containing the abstract types and abstract symbols which are the atoms of the language. We thus begin by instantiating Whycert to the target language of our compilation of C.

5.1.1 Types and Symbols

As described in Section 3.2, Whycert types are second order types depending on a user defined set of first order types `utype`:

```
Variable utype : Type.

Inductive type :=
| Tuser :> utype → type
| Tarrow : type → type → type
| Tprop.

Infix "-->" := Tarrow.
```

For our compilation of C programs we instantiate `utype` as follows

```
Inductive integer_unop := ZOneg | ZObwneg.
Inductive integer_binop :=
  ZOadd | ZOsub | ZOmul | ZOdiv | ZOmod | ZObwand | ZObwor | ZObwxor.
Inductive comparison : Type :=
  Ceq | Cne | Clt | Cle | Cgt | Cge.
Inductive real_unop := ROneg.
Inductive real_binop := ROadd | ROsub | ROmul | ROdiv.
Inductive boolean_unop := BOnot.
Inductive boolean_binop := BOand | BOor.
Inductive comparable_type := CompInteger | CompReal | CompFloat sz.

Inductive conversion : atype → atype → Type :=
| conv_integer_real: conversion Linteger Lreal
| conv_integer_int sz sg: conversion Linteger (Tint sz sg)
| conv_integer_float sz: conversion Linteger (Tfloat sz)
| conv_integer_boolean: conversion Linteger Lboolean
| conv_boolean_integer: conversion Lboolean Linteger
| conv_real_integer: conversion Lreal Linteger
| conv_real_float sz: conversion Lreal (Tfloat sz)
| conv_int_integer sz sg: conversion (Tint sz sg) Linteger
| conv_float_real sz: conversion (Tfloat sz) Lreal
| conv_float_integer sz: conversion (Tfloat sz) Linteger
| conv_float_float sz1 sz2: conversion (Tfloat sz1) (Tfloat sz2)
| conv_pointer_pointer cty1 cty2: conversion (Tpointer cty1) (Tpointer cty2)
| conv_carray_array cty l: conversion (Tarray cty l) (Larray cty)
| conv_array_array aty1 aty2: clos_trans_1n _ conversion aty1 aty2 →
    conversion (Larray aty1) (Larray aty2)
.
```

Figure 5.1: Types used inside symbols

```
Notation atype := ClightACSL.type.

Inductive utype :=
| Tatype :> atype → utype
| Tmem
| Tmap
| Tintlist
| Tident.
```

That is types used in logical symbols can be any ACSL type, including C types, with the addition of types for the C memory, C temporary variable mappings, lists of integers and identifiers. The constructors `Tatype` and `Tuser` are declared as coercions so that we can omit them and write an ACSL type in any context a Why type is expected.

In the same way it is generic over types, Whycert is generic over a set of abstract symbols. The symbols needed for the compilation of C programs are shown in Fig. 5.2 with auxiliary constants in Fig. 5.1. They include propositional operators, constants, integer arithmetic and symbols for the memory model.

Notice that some symbols like `SymEq` or `SymLoad` depend on a type. They can be used like polymorphic symbols but as Whycert doesn't have any abstract syntax for type variables they can be used only instantiated. Unlike with fully polymorphic symbols, we can specify over which subclass of types such a symbol is polymorphic, i.e. `SymLoad` is polymorphic over C-types and `SymCmp` over comparable types.

```
Inductive sym : type → Type :=
(* propositional operators *)
| SymTrue: sym Tprop
| SymNot: sym (Tprop --> Tprop)
| SymOr: sym (Tprop --> Tprop --> Tprop)
| SymEq ty: sym (ty --> ty --> Tprop)
| SymIfThenElse ty: sym (Lboolean --> ty --> ty --> ty)
(* constants *)
| SymConstFloat sz (fl: float): sym (Tfloat sz)
| SymConstInteger (z: Z): sym Linteger
| SymConstReal (r: real): sym Lreal
| SymConstBool (b: bool) : sym Lboolean
| SymConsttt : sym Tunit
| SymConstNull : sym (Tpointer Tvoid)
(* arithmetic operators *)
| SymZUop zuop: sym (Linteger --> Linteger)
| SymZBop zbop: sym (Linteger --> Linteger --> Linteger)
| SymRUop ruop: sym (Lreal --> Lreal)
| SymRBop rbop: sym (Lreal --> Lreal --> Lreal)
| SymBUop buop: sym (Lboolean --> Lboolean)
| SymBBop bbop: sym (Lboolean --> Lboolean --> Lboolean)
| SymRel compty : comparison → sym (compty --> compty --> Tprop)
| SymCmp compty : comparison → sym (compty --> compty --> Lboolean)
(* memory model *)
| SymRelPtr t1 t2: comparison → sym (Tmem --> Tpointer t1 --> Tpointer t2 --> Tprop)
| SymNotNull cty: sym (Tpointer cty --> Tprop)
| SymCGlobal idt cty: sym (Tpointer cty)
| SymSizeOf cty: sym Linteger
| SymPtrArith cty : sym (Tpointer cty --> Linteger --> Tpointer cty)
| SymCarrayOffset cty l: sym (Tpointer (Tarray cty l) --> Linteger --> Tpointer cty)
| SymField si fld cty: ident → sym (Tpointer (Tstruct si fld) --> Tpointer cty)
| SymLoad cty: sym (Tmem --> Tpointer cty --> cty)
| SymLoadArray cty: sym (Tmem --> Tpointer cty --> Larray cty)
| SymStore cty: sym (Tmem --> Tpointer cty --> cty --> Tmem)
| SymFree cty: sym (Tmem --> Tpointer cty --> Tmem)
| SymNextBlock: sym (Tmem --> Linteger)
| SymBlock cty: sym (Tpointer cty --> Linteger)
| SymValidLoad cty: sym (Tmem --> Tpointer cty --> Tprop)
| SymValidStore cty: sym (Tmem --> Tpointer cty --> Tprop)
| SymMemPreservePermissions: sym (Tmem --> Tmem --> Tprop)
| SymMemPreserveContents: sym (Tintlist --> Tmem --> Tmem --> Tprop)
(* lists of integers, used for loop assigns clauses *)
| SymIntlistNil: sym Tintlist
| SymIntlistCons: sym (Linteger --> Tintlist --> Tintlist)
| SymIntlistIn: sym (Linteger --> Tintlist --> Tprop)
(* mappings of C temporary variables *)
| SymMapEmpty: sym Tmap
| SymMapGet mty: sym (Tident --> Tmap --> mty)
| SymMapValidGet mty: sym (Tident --> Tmap --> Tprop)
| SymMapSet mty: sym (Tident --> Tmap --> mty --> Tmap)
| SymMapPreserve: sym (Tident --> Tmap --> Tmap --> Tprop)
(* misc *)
| SymConversion aty1 aty2:> conversion aty1 aty2 → sym (aty1 --> aty2)
| SymArrayAccess aty: sym (Larray aty --> Linteger --> aty)
| SymArrayEmpty aty: sym (Linteger --> Larray aty)
| SymArrayUpdate aty: sym (Larray aty --> Linteger --> aty --> Larray aty)
| SymStructAccess si fld cty: ident → sym (Tstruct si fld --> cty)
| SymStructEmpty si fld: sym (Tstruct si fld)
| SymStructUpdate si fld cty: ident → sym (Tstruct si fld --> cty --> Tstruct si fld)
| SymDefaultValue mty : sym mty
| SymIdent idt : sym (Tident)
.
```

Figure 5.2: Symbols

In Coq subtyping is formalised with coercions. An important subtype is `mtype`, that is C types of values that can be written to memory:

```
Inductive mtype :=
| Mint (sz:intsize) (sg:signedness)
| Mfloat (sz:floatsize)
| Mpointer (cty: ctype) .

Coercion ctype_of_mtype mty : ctype :=
  match mty with
    | Mint sz sg ⇒ Tint sz sg
    | Mfloat sz ⇒ Tfloat sz
    | Mpointer cty ⇒ Tpointer cty
  end.
```

Being as coercion we can usually omit `ctype_of_mtype` and directly use a `mtype` in every context a `ctype` is expected. In the following we will use `mty` and `cty` variables to bind values of type `mtype` and `ctype`, respectively.

```
Implicit Type mty : mtype.
Implicit Type cty : ctype.
```

5.1.2 Interpretation of Types and Symbols

The verification condition generator has been proved correct for any interpretation of these abstract types. It is however required to provide one, as the semantics of the language depends on the semantics of the types and symbols. The semantics of Whycert types are expressed in terms of Coq types:

```
Fixpoint denutype uty : Type :=
  match uty with
    | Tatype aty ⇒ ClightACSLSemantics.den_type aty
    | Tmap ⇒ Clight.temp_env
    | Tmem ⇒ Mem.mem
    | Tintlist ⇒ list Z
    | Tident ⇒ ident
  end.
```

Here `Clight.temp_env`, `Mem.mem` and `ident` are Compcert types respectively for temporary variable mappings, memory states and identifiers.

In the following we will consider `dentype` instantiated with `denutype`.

```
Notation dentype := (dentype denutype).
```

The semantics of symbols are given as Coq terms whose type depend on the symbol's type. Figure 5.3 shows only some examples of how symbols are mapped to Coq terms in a natural way thanks to the dependent typing. The semantics of the remaining symbols will be shown when needed in the following sections. Notice here the use of our `complete` operators to give semantics to logical symbols that naturally wouldn't be defined for all inputs. Concerning the memory model, whereas `clogic_load` is defined in order to recursively load aggregate types (Fig. 4.15), we define `simple_store` as a simple operation only for basic types, referring to the implementation of Compcert's front end formalisation `Csem.store_value_of_type`. This requires to make a basic `val` of our dependently typed values, that is the converse of what does `type_val` (defined on page 76):

```
Definition simple_store cty m a (x: den_ctype cty) :=
  Csem.store_value_of_type cty m a.(adr_block) a.(adr_ofs) (val_untype x).
Definition m_valid_access m cty b ofs pm :=
  Mem.range_perm m b ofs (ofs + sizeof cty) pm ∧
  (alignof cty | ofs) ∧ ofs + sizeof cty < Int.modulus.
Definition m_valid_access_p m cty pt :=
  m_valid_access m cty pt.(adr_block) (Int.unsigned pt.(adr_ofs)).
Definition m_preserve_permissions m m' :=
  ∀ cty pt pm, m_valid_access_p m cty pt pm → m_valid_access_p m' cty pt pm.
Definition m_valid_load cty m pt :=
  ∃x, csimple_loadp cty pt m = Some x.

Definition densym ty (s: sym ty) : dentype ty :=
  match s in sym ty return dentype ty with
    | SymNot ⇒ not
    | SymOr ⇒ or
    | SymEq ty ⇒ eq
    | SymZBop zbop ⇒ match zbop with
                        | ZOadd ⇒ Zplus
                        | ZOsub ⇒ Zminus
                        | ZOmul ⇒ Zmult
                        | ZOdiv ⇒ complete2 Zdiv_round
                        | ZOmod ⇒ complete2 Zmod_round
                        | ZObwand ⇒ Zand
                        | ZObwor ⇒ Zor
                        | ZObwxor ⇒ Zxor
                      end
    | SymLoad cty ⇒ fun m a, clogic_load cty a m
    | SymStore cty ⇒ fun m a x, complete4 simple_store cty m a x
    | SymValidLoad cty ⇒ m_valid_load cty
    | SymValidStore cty ⇒ m_valid_access_p m cty pt Writable
    | SymMemPreservePermissions ⇒ m_preserve_permissions
    | ...
  end.
```

Figure 5.3: Interpretation of Logical Symbols

```
Definition val_untype cty :=
  match cty return den_ctype cty → val with
    | Tint sz sg ⇒ fun bz, Vint (Int.repr (Z_of_bZ bz))
    | Tfloat sz ⇒ Vfloat
    | Tpointer ty ⇒ fun a,
        if a.(adr_block) == 0 then
          Vint a.(adr_ofs)
        else
          Vptr a.(adr_block) a.(adr_ofs)
    | _ ⇒ fun _, Vundef
  end.
```

for simplicity we chose this to be a total function that returns `Vundef` for non basic values. We have the following two properties to relate this to `type_val`:

```
Lemma type_val_untype mty (x: den_ctype mty) :
  type_val mty (val_untype x) = Some x.

Lemma untype_val_type cty v x:
  type_val cty v = Some x → val_untype x = v.
```

Notice that the first one is valid only for `mty : mtype`.

The interpretation of the symbols concerning permissions `SymValidStore` and `SymMemPreservePermissions` directly interface with the Compcert memory model, which provides a detailed formalisation of permissions. We just add some information about validity of the offset. On the other hand, for `SymValidLoad` we chose a stronger interpretation which doesn't only state that loading from at given address is permitted but also guarantees that the given address is correctly initialised. However, this predicate is currently only useful for basic types, support for aggregate types should be implemented in the future.

5.1.3 Axioms

The sense of giving the semantics for types and symbols is twofold. First, they influence the semantics of the whole language, which must be preserved by the compilation from C + ACSL using these symbols. But second, they justify the axiomatisation we define to accompany the proof obligations sent to external provers in order to allow them to reason about the symbols that are completely abstract for them. More precisely we distinguish two classes of symbols: symbols that are built-in the external provers, like the propositional combinators and the arithmetical constants and operators, and symbols that are abstract for them. The first class is more critical because we need to trust the fact that the external provers interpret the symbols the same way. For the second class we can state and prove inside Coq that the "axioms" that are sent to the provers are effectively valid.

As an example for such an axiom consider the following property:

```
Lemma load_store_eq mty m a (x: den_ctype mty):
  m_valid_store m mty a →
  clogic_load mty a (simple_store m a x) = x.
```

This is proven in Coq and ensures that if we have the right to store a value of a given memory type at a given address, then we can read back the stored value at that address. This lemma can be expressed in abstract syntax as a Whycert term:

```
Definition axiom_load_store_eq mty :=
  Tforall (Tforall (Tforall (
    let m := Tvar HI2 in let a := Tvar HI1 in let x := Tvar HI0 in
  Timply
    (Tsymapp (SymValidStore mty) m a)
    ((Tsymapp (SymEq mty)
      (Tsymapp (SymLoad mty) (Tsymapp (SymStore mty) m a x) a)
      x)))))).
```

This term can be extracted out of Coq and sent to external provers to be used as an axiom, as we have the proof that it is valid:

```
Goal forall S mty, evalterm (axiom_load_store_eq mty) [] [] S.
Proof.  intro. exact load_store_eq.  Qed.
```

Notice that this axiom is valid for any `mty`. To be sent to external provers it needs to be instantiated to a closed term once for each type the prover will need to reason about, which is at

most all the types appearing in the program.

Given any list of such polymorphic axioms `axioms_mty` of type `list (mtype → closed_prop)`, any list of axioms depending on a `ctype` `axioms_cty` of type `list (ctype → closed_prop)` and any list of non polymorphic axioms `axioms0` of type `list closed_prop` we can compute the list of needed, instantiated axioms based on the list of types appearing in the program.

```
Definition with_memtype A cty (f:mtype → A → A) : A → A :=
  match get_memtype cty with Some mty ⇒ f mty | None ⇒ id end.

Definition inst_axioms cty :=
  ( List.fold_left (fun l ax, (ax cty)::l) axioms_cty;
    with_memtype cty
      (fun mty, List.fold_left (fun l ax, (ax mty)::l) axioms_mty)).

Definition needed_axioms ctypes :=
  List.fold_left (flip inst_axioms) ctypes axioms0.
```

With this mechanism we can add as many axioms as necessary for the provers to prove the generated proof obligations - it is perfectly safe, as long as we can prove in Coq that the axioms' interpretations are valid, that is that:

```
Definition valid_list (l: list (prop 0 [])) :=
  ∀S, List.Forall (fun p, evalterm p [] [] S) l.

Lemma needed_axioms_valid ctl : valid_list (needed_axioms ctl).
```

The axioms needed to process the first examples are introduced as needed in Section 5.6.3.

5.1.4 References and Exceptions

Another generic parameter of our intermediate language is the set of global references. To model the C memory heap we simply chose a reference of type `Tmem` and a reference of type `Tmap` for the current function's local temporary variables.

```
Inductive reference :=  Rmem_ | Rtmp_.

Definition type_reference r : type :=
  match r with
    | Rmem_ ⇒ Some Tmem
    | Rtmp_ ⇒ Some Tmap
  end.

Definition ref : type →  Type := phantomtype type_reference.

Program Definition Rmem : ref Tmem := Rmem_.
Program Definition Rtmp : ref Tmap := Rtmp_.
```

The `phantomtype` operator produces a type with the annotations defined by the given function. That is the above definition is more convenient to be used as an ordered type but otherwise equivalent to the following:

```
Inductive reference : type →  Type :=
| Rmem: ref Rmem
| Rtmp: ref Tmap.
```

```
Inductive par : list type → type → Type :=
| PAR_degrade_int sz sg: par [Linteger] (Tint sz sg)
| PAR_mem_alloc cty: par [] (Tpointer cty).

Definition Pbounds {L} E sz sg (t: term E Linteger) : prop L E :=
  Pand (Tsymapp (SymRelZ Cle) (Tsym (SymConstInteger (min_int sz sg))) t)
    (Tsymapp (SymRelZ Cle) t (Tsym (SymConstInteger (max_int sz sg)))).

Definition get_parspec P A (pa: par P A) : parspec P A :=
  match pa with
    | PAR_degrade_int sz sg ⇒ {| par_contract := {|
          pre := Pbounds sz sg (Tvar HI0);
          post := Peq (Tvar HI1)
            (Tsymapp (conv_int_integer sz sg) (Tvar HI0)) |};
        par_effects := {} |}
    | PAR_mem_alloc cty ⇒ {| par_contract := {|
          pre := Ptrue; post := post_alloc cty |};
        par_effects := { (&Rmem) } |}
  end.

Definition den_par P A (pa: par P A) :=
  match pa in par P A return env P → state → state * dentype A with
    | PAR_degrade_int sz sg ⇒ fun G S, (S, bZ_of_Z sz sg (accslidx HI0 G))
    | PAR_mem_alloc cty ⇒ fun G S,
        let (m, b) := Mem.alloc (cmem S) 0 (sizeof cty) in (update S Rmem m, b)
  end.

Lemma den_par_correct : forall P A (pa: par P A),
 ∀G S, evalterm pa.(pre) G [] S →
  let (S', a) := den_par pa G S in
    evalterm pa.(post) (a::G) [S] S' ∧  assigns S pa.(par_effects) S'.
```

Figure 5.4: Whycert Parameters

Notice that using a global reference for the current local temporary variables is an encoding for local references. If we had local references in Whycert, we would use one inside each function to keep its local temporaries. Instead we decide that every function uses the shared global reference, taking care of storing the contents of the reference in a local Whycert variable before a call of another function (see Section 5.5.1).

As we will use them quite often, we also define the following shortcuts:

```
Definition cmem (S: state) := S Tmem Rmem.
Definition ctmp (S: state) := S Tmap Rtmp.
```

As exceptions we chose the ones we need to encode the C control flow:

```
Inductive exn : type → Type :=
| ExcBreak : exn Tunit
| ExcContinue : exn Tunit
| ExcReturn ty : exn ty.
```

`ExcBreak` and `ExcContinue` don't need to carry a value, so its type is `Tunit`. `ExcReturn` carries a value of any type, which may also be `Tunit`.

5.1.5 Abstract Program Parameters

A last generic feature of Whycert is abstract parameters. By declaring a set of parameters along with their specification, including pre- and post-condition and effects, and their interpretation, we can use them inside expressions applying them to some arguments. The WP calculus then simply replaces them with their specification. For the moment, we don't make a very extensive use of parameters, we only use them for integer casts and memory allocations (Fig. 5.4). In the future they could however be used to model external functions like system calls. The parameter `PAR_degrade_int` takes as argument a logical integer and returns a machine integer of the given size. More precisely it is a family of parameters, one for each `intsize` and `signedness`. Its specification says that if the argument respects the bounds of the machine integer type, then the result is, promoted to a logical integer, equal to the argument. In Why3 concrete syntax this abstract parameter (instantiated for signed 32-bit integers) would be declared as

```
degrade_int_32_signed (i: integer) :
{ -2^31 ≤ i < 2^31 } int32 { result = conv_int32_integer i }
```

Notice that this parameter is similar to the logical conversion symbol `conv_integer_int`. Using the abstract parameter has the advantage of producing convenient proof obligations.

The other family of parameters `PAR_mem_alloc`, one for each C type, modifies the memory and returns a pointer to the newly allocated block. Its post-condition contains all the properties of Compcert's memory model we need to reason about the new block as explained in Section 5.6.3. In any case, Coq forces us to prove that the specification matches the implementation of the symbol. For `PAR_mem_alloc` we directly use the function of Compcert's memory model and to prove the specified properties we use the axiomatisation Compcert's memory model.

5.2 Compilation of Variables and Labels

To compile annotated Clight to Whycert we need to provide support for the leaves of the syntax tree, notably variables and labels. We have three different types of variables: global program variables, local function variables and, inside annotations, also logical variables.

Occurrences of global variables are simply mapped to symbols `SymCGlobal` which carry the identifier and the type of the variable. This symbol represents the pointer to the memory block that will be allocated for that global variable. Notice that at time of compilation we do not assume that the Clight program is well typed so the compilation may fail on typing errors. For a global variable the compilation checks that the global environment of the C program `cge` contains a binding for that variable and that type. This global environment will be the same at execution time but can be computed statically and we suppose to know it during the compilation.

For the different kinds of local objects, namely local C variables, logical ACSL variables and labels, we use mappings to De-Bruijn indices, respectively `cvarmap`, `lvarmap` and `labmap`. Such mappings are constructed at the beginning of the compilation of C functions and are lifted as necessary while descending into the syntax tree. Recall that these De-Bruijn indices are typed. In particular, indices corresponding to local C variables have a pointer type, whereas indices for logical ACSL variables have the type of the value they represent in the current logical environment. These maps are constructed at the beginning of the compilation of a function, which simulates allocation and deallocation of local variables and formal parameters.

Relating C and Whycert environments   To prepare the proof of soundness of the compilation we define relations between C environments and the generated Whycert environments. These re-

lations are hypotheses for the preservation of semantics by the compilation of ACSL expressions, C expressions and statements and are established by the compilation of functions.

Each of the above variable mappings relates several semantical objects of the source and the target of the compilation. These relations require some additional properties that need to be invariant during the compilation.

For C variables we need to be sure that the variable mapping contains the same entries as the current C environment and that the corresponding Whycert environment contains the correct pointer:

```
Definition rel_Gce (ce: Clight.env) E (vm: cvarmap E) (G : env E):=
  forall i,
    match ce.[i], vm i with
      | Some (b, cty), Some (cty'&v) ⇒ cty = cty' ∧  b ≠ 0 ∧
          accsvar v G = {| adr_block := b; adr_ofs := Int.zero |}
      | None, None ⇒ True
      | _, _ ⇒ False
    end.
```

Similarly for logical variables we require that the current logical environment is mapped to the corresponding Whycert environment:

```
Definition rel_Gs E te (lvm : lvarmap E te)
  (G : env E) (G_e : Env te) :=
    forall (i : ident) (ty : atype) (e : (te.[i] = Some ty)),
      accsvar (lvm i ty e) G = G_e i ty e.
```

This relation can be written more concisely than `rel_Gce` by taking advantage of the typing environment for logical environments.

At last we require that the mapping for labels contains all and only the labels for that currently exists a labelled memory state:

```
Definition rel_lm_S L (SS: states L) (lm: lmem) (lmp: labmap L) :=
  forall idt,
    match lm.[idt], lmp idt with
      | Some m, Some l ⇒ At l SS Rmem = m
      | None, None ⇒ True
      | _, _ ⇒ False
    end.
```

These relations are hypotheses for the preservation of semantics by the compilation of ACSL expressions, C expressions and statements and are established by the compilation of functions.

5.3   Compilation of Logical Expressions

Compilation of ACSL terms and predicates is defined by recursion over the respective typing rules. Recall that an ACSL term can be typed either as a left or as a right value. In the former case the term evaluates to a memory address from which the actual value can be loaded. In the compilation of terms we take over this idea: for a left-value input the type of the resulting Whycert term is a pointer type `Tpointer cty`, where `cty` is the type of the input term. Also recall that in some cases the typing rules allow logical arrays as left values. Such a term represents the address of the first element of the array, which is necessarily a C type. For convenience we define the argument of `Tpointer` as follows:

```
Fixpoint pt_type aty :=
  match aty with
    | Larray aty ⇒ pt_type aty
    | Tctype cty ⇒ cty
    | _ ⇒ Tvoid
  end.
```

The function compiling an ACSL term takes as arguments a mapping for logical variables `lvm` with respect a given Whycert typing environment `E` and a ACSL typing environment `te`, a mapping for C variables `vm`, a well typed ACSL term `t` of type `aty` which is either a left-value or a right-value depending on `lr` and its typing derivation `tty`. The return type of a compiled term of type `aty` is then

```
Definition lr_type lr aty : atype :=
  if lr then Tpointer (pt_type aty) else aty.

Fixpoint comp_acsl_term E te (lvm: lvarmap E te) (vm: cvarmap E)
  t lr aty (tty: term_typed te t lr aty)
  : option (term L E (lr_type lr aty)) :=
  match tty with
    ...
  end.
```

A part of the compilation is shown in Fig. 5.5. Notice the way the De-Bruijn mappings are lifted when the compilation descends under a binder (case `term_typed_let`). Also notice that in several cases (`deref`, `addrof` and conversions between arrays and pointers) the compilation just returns the recursively computed term without adding anything, whereas in other cases (`cast_ptr_arr`, `promote_arr_arr`) some "identitary" pointer cast is needed to fix the typing of the generated terms. The definition of `pr_type` tries to minimise such casts. We could have avoided pointer casts altogether by using a common pointer type without parameter, but we think the type parameter could be helpful for SMT provers to automatically prove goals involving pointers by giving useful hints to which axiom to apply.

We prove that the semantics is preserved by this compilation. More precisely, we prove that

Theorem 11 *a compiled term evaluates, according to the semantics of Whycert, to the value to which evaluates the original term according to the semantics of ACSL terms.*

A simple cast is needed to state the equality.

```
Definition cast_lrtype lr aty:
  den_lrtype lr aty → dentype (lr_type lr aty) :=
  match lr with false ⇒ id | true ⇒ id end.

Lemma comp_acsl_term_correct:
 forall te t lr aty (tty: term_typed te t lr aty) E
   (lvm: lvarmap E te) (vm: cvarmap E) G (G_e: Env te),
     rel_lm_S SS lm lmp → rel_Gs lvm G G_e →  rel_Gce ce vm G →
       forall wt, comp_acsl_term cge lmp lvm vm tty = Some wt →
       evalterm wt G SS S = cast_lrtype
         (ClightACSLSemantics.evalterm cge ce (cmem S) lm G_e tty).
```

Proof. by induction on the typing derivation and application of the hypotheses in the base cases.

ACSL predicates are compiled similarly, by recursion on the typing rules and by referring to the compilation of terms where necessary.

```
Program Definition comp_prog_var E (vm: cvarmap E) i cty
 : option (term L E (Tpointer cty)) :=
  match vm i, Genv.find_symbol cge i with
    | Some (cty' & v), _ ⇒ if cty' == cty then return (Tvar v) else error
    | None, Some b ⇒
        let? cty' := type_of_global cge b in
          if cty' == cty then return (Tsym (SymCGlobal i cty)) else error
    | None, None ⇒ error
  end.

Program Fixpoint comp_acsl_term E te (lvm: lvarmap E te) (vm: cvarmap E)
  t lr aty (tty: term_typed te t lr aty)
  : option (term L E (lr_type lr aty)) :=
  match tty with
    | term_typed_integer z ⇒ return Tsym (SymConstInteger z)
    | term_typed_var i aty x ⇒ return Tvar (lvm i aty x)
    | term_typed_progvar i cty x ⇒ (comp_prog_var vm i cty)
    | term_typed_binop_integer op t1 t2 tty1 tty2 _ ⇒
        let? wt1 := comp_acsl_term lvm vm tty1 in
        let? wt2 := comp_acsl_term lvm vm tty2 in
          return Tsymapp
            (SymZBop (match op with Bplus ⇒ ZOadd | Bminus ⇒ ZOsub | ... end))
            wt1 wt2
    | term_typed_let i _ _ _ _ tty1 tty2 ⇒
        let? wt1 := comp_acsl_term lvm vm tty1 in
        let? wt2 := comp_acsl_term (varmap_add lvm i) (pvarmap_lift vm) tty2 in
          return Tlet wt1 wt2
    | term_typed_cast_arr_ptr t cty x ⇒ comp_acsl_term lvm vm x
    | term_typed_cast_ptr_arr t cty1 cty2 x ⇒
        let? wt := comp_acsl_term lvm vm x in
          return Tcnv (conv_pointer_pointer _ _) wt
    | term_typed_promote_ptr_arr t cty x ⇒ comp_acsl_term lvm vm x
    | term_typed_promote_arr_arr t cty z lr x ⇒
        let? wt := comp_acsl_term lvm vm x in return
          if lr then
            Tcnv (conv_pointer_pointer _ _) wt
           else
            Tcnv (conv_carray_array cty z) wt
    | term_typed_deref t aty x ⇒ comp_acsl_term lvm vm x
    | term_typed_lval t aty x ⇒
        let? wt := comp_acsl_term lvm vm x in fun _ wt, return
          match aty with
            | Larray (Tctype cty) ⇒ Tsymapp (SymLoadArray cty) (Tderef Rmem) wt
            | Tctype cty ⇒ Tsymapp (SymLoad cty) (Tderef Rmem) wt
            | _ ⇒ !
          end
    | term_typed_addrof t cty x ⇒ comp_acsl_term lvm vm x
    | term_typed_array_access ta ti lr ty x0 x1 ⇒
        let? wta := comp_acsl_term lvm vm x0 in
        let? wti := comp_acsl_term lvm vm x1 in
          if lr then
            return Tsymapp (SymPtrArith _) wta wti
          else
            return Tsymapp (SymArrayAccess ty) wta wti
    | ...
  end.
```

Figure 5.5: Compilation of ACSL terms

Theorem 12 *A compiled predicate is logically equivalent, according to the semantics of Whycert, to the original predicate according to the semantics of ACSL predicates.*

```
Program Fixpoint comp_acsl_prop E te (lvm: lvarmap E te)
  (vm: cvarmap E) p (pty: pred_typed te p): option (prop L E) :=
  match pty with
   | ...
  end.

Lemma comp_acsl_prop_correct:
 forall te t (pty: pred_typed te t) E
  (lvm: lvarmap E te) (vm: cvarmap E) G (G_e: Env te):
    rel_lm_S SS lm lmp → rel_Gs lvm G G_e →  rel_Gce ce vm G →
     forall wt, comp_acsl_prop cge lmp lvm vm pty = Some wt →
       (evalterm wt G SS S ↔
         ClightACSLSemantics.evalpred cge ce cm lm G_e pty).
```

5.4 Compilation of Expressions

Compilation of Clight expressions is more involved than the compilation of logical expressions, as expressions may not execute safely, e.g. arithmetic overflow or invalid memory access may occur. These errors have to be excluded by appropriate assertions inserted into the compiled Whycert expressions. More precisely, given a Clight expression we want to generate a Whycert expression, such that if the latter executes to a value without blocking then the former can be evaluated to the same value without any errors. We thus need to rule out invalid memory accesses, arithmetic operations on invalid values and anything else that may go wrong at execution time. In addition we want to avoid arithmetic overflows, even if they wouldn't lead to execution errors, for the reason we detail below. [1]

5.4.1 Integer Arithmetic

C integer arithmetic has a semantics modulo the maximum representable integer, 2^{32} in Compcert. This means that an operation, whose result wouldn't be representable as a machine integer, does not fail but silently return the result modulo 2^{32}. The following examples leads to such *overflows* (with `MAX_INT` $= 2^{31} - 1$):

```
int a = MAX_INT;
int b = a + 1;
unsigned c = 1;
unsigned d = c - 2;
```

Here `b` and `d` do not contain the result of the logical operations $a + 1$ and $c - 2$, respectively.

The situation is complicated by the fact that the semantics of integer operations depend on the types of the operands. More precisely, a combination of the types of the operands define how they are interpreted: if one of them has the type **`unsigned int`** (32-bits) then both are interpreted as unsigned. Consider the following example where the semantics is not as expected:

---

1. To be precise, only for the Compcert semantics, overflows do not lead to errors. The ANSI/ISO norm still considers overflowing arithmetic operations as "undefined" except for operands of type `unsigned int`

```
Definition classify_int_type sz sg :=
  match sz,sg with I32, Unsigned ⇒ Unsigned | _, _ ⇒ Signed end.
Definition or_unsigned sg1 sg2 :=
  match sg1, sg2 with Signed, Signed ⇒ Signed | _,_ ⇒ Unsigned end.
Definition classify_int_type2 sz1 sg1 sz2 sg2 :=
  or_unsigned (classify_int_type sz1 sg1) (classify_int_type sz2 sg2).
Definition P0rel r E sz sg t : prop E :=
  Tsymapp (SymRelZ r) (Tsym (SymConstInteger 0))
    (Tsymapp (conv_int_integer sz sg) t).
Definition Pvalid_unsigned2 E sz1 sz2 sg1 sg2
    (t1: term E (Tint sz1 sg1)) (t2: term E (Tint sz2 sg2))  :=
  match classify_int_type sz1 sg1, classify_int_type sz2 sg2 with
    | Unsigned, Unsigned ⇒ Ptrue
    | Unsigned, Signed ⇒ P0rel Cle t2
    | Signed, Unsigned ⇒ P0rel Cle t1
    | Signed, Signed ⇒ Ptrue
  end.
Definition Pdiv_ne0 E (zbopc: integer_binop + comparison)
   sz sg (t: term E (Tint sz sg)) :=
  match zbopc with
    | inl ZOdiv | inl ZOmod ⇒ P0rel Cne t
    | _ ⇒ Ptrue
  end.
Definition TZBopOrCmp {E} zbopc (t1 t2: term0 E Linteger) :=
  match zbopc with
    | inl x ⇒ Tsymapp (SymZBop x) t1 t2
    | inr x ⇒ Tsymapp (SymIfThenElse _)
        (Tsymapp (SymCmp CompInteger x) t1 t2)
        (Tsym (SymConstInteger 1)) (Tsym (SymConstInteger 0))
  end.

Definition Ezbop op sz sg E sz1 sz2 sg1 sg2 (e1: expr _ (Tint sz1 sg1))
    (e2: expr _ (Tint sz2 sg2)) : expr E (Tint sz sg) :=
  Elet e1 (Elet e2
    (Eassert (Pand (Pvalid_unsigned2 (Tvar HI0) (Tvar HI1))
      (Pdiv_ne0 op (Tvar HI0)))
      (Ecallpar (PAR_degrade_int sz sg)
        [TZBopOrCmp op (Tsymapp (conv_int_integer sz1 sg1) (Tvar HI1))
          (Tsymapp (conv_int_integer sz2 sg2) (Tvar HI0))])))).
```

Figure 5.6: Compilation Scheme for arithmetic expressions

```
signed int a = −4, b = 2;
unsigned int c = 2;
int r1 = a / b, r2 = a / c;
puts (r1 == r2 ? "true" : "false");
```

The execution will print out "false", because in the second division `a / c`, the occurrence of `a` is interpreted as unsigned and evaluates to $2^{32} − 4$ which is then divided by 2.

This example also shows the criticality of the typing of C expressions. The current formalisations of Clight and CompcertC consider expressions as annotated with their types. This assumes that when constructing the AST from concrete syntax the correct types are inserted starting from the types of constants and variables respecting some typing environment. As the semantics of C expressions depend on their typing, the same importance should be attributed to both of their formal specifications.

Because of these subtleties in the evaluation of C arithmetic expressions, in a context of critical software, overflows are usually considered as errors. Thus a typical design choice of static analysers is to detect and signal possible overflows and we make the same choice in this development. More precisely we define the absence of overflow as the following property: the evaluation of an arithmetic expression must give the same result as if it was evaluated assuming exact arithmetic on Z.

We rule out overflows by inserting appropriate assertions into the compiled expressions. The compilation scheme for arithmetic expressions is shown in Fig. 5.6. Given an operator `op` and two compiled expressions `e1` and `e2` of integer types with size `sz1`, `sz2` and signedness `sg1`, `sg2`, respectively, the function `Ezbop` returns an expression that simulates the operation ensuring the absence of overflows. If one of the operands is an **unsigned int** then it checks that the other one is not less than zero. In case of a division it also checks that the divisor is not equal to zero. The operands are then promoted to logical integers and the operator to its mathematical counterpart. In order to convert the result back to a machine integer the parameter `PAR_degrade_int` checks that it respects the bounds of the return type. Notice that this return type needs to be specified: `Ezbop` returns an expression of type `Tint sz sg` for any given `sz` and `sg`. There is some flexibility in choosing them, in particular we don't want to force the return type to be equal to the type of the C expression to be compiled, e.g.

```
signed int a = MAX_INT;
unsigned int b = a + a;
unsigned int c = (a + a) - b;
```

In Compcert, there is no overflow in the example and even if the type of `a + a` is **signed int**, when it is used in contexts that expect an **unsigned int** we want to allow it to assume values of the whole range and avoid generating too restrictive proof obligations. C expressions are therefore compiled to Whycert expressions with information about the expected type.

The function `comp_zbop` (Fig. 5.7) compiles a binary integer Clight expression to a Whycert expression and is part of a generic compilation of C expressions `comp_expr` which it assumes to be recursively defined. These functions accepts the additional argument `hint` of type `option mtype` as an optional suggestion for the return type of corresponding Whycert expression, e.g. the type of the variable the original C expression is assigned to. `comp_zbop` adopts the suggestion if given for the case of binary integer expressions and computes a suggestion for the recursive compilation of sub-expressions.

5.4.2 Equivalence Relation and Compilation

In order to prepare the compilation of C to Whycert we start by relating a subset of Clight expressions with Whycert expressions that have the same semantics. Any C expression will then be compiled to a related Whycert expression. The relation, defined by the two mutually recursive inductives `rel_expr` and `rel_lexpr` is shown in Figs. 5.8 and 5.9. The latter relates a Clight l-value expression `e` with a Whycert expression of type `Tpointer (typeof e)`, whereas the former relates a Clight r-value expression `e` with a memory type `mty` and a Whycert expression of type `mty`, which is in general independent from the type of `e`.

Notice that, for the reasons above, `Ebinop zbopc ce1 ce2 (Tint sz' sg')` is in relation with the Whycert expression `Ezbop zbopc sz sg e1 e2` for any `sz` and `sg`. The remaining cases of C expressions that we treat so far are simpler to compile than arithmetic expressions. Notice however that in some cases subtle conditions are imposed to the equivalence, for instance implicit typing of the variable `mty` ensures that the Clight expression for a temporary variable `Etempvar i cty` has a counterpart in Whycert only if the type of the variable `cty`

```
Definition get_int_type cty : { x : option (intsize * signedness)
   | ∀sz sg, x = Some (sz, sg) ↔ cty = Tint sz sg } := ...

Definition comp_zbop E {vm: cvarmap E} (hint:option mtype) zbop e1 e2
  : option {mty : mtype & expr L E mty} :=
 let? (sz1, sg1) := get_int_type (typeof e1) in
 let? (sz2, sg2) := get_int_type (typeof e2) in
 let sgr := classify_int_type2 sz1 sg1 sz2 sg2 in
 let hint' := Some (Mint I32 sgr) in
   match comp_expr hint' e1, comp_expr hint' e2 with
     | Some (Mint sz1' sg1' & e1), Some (Mint sz2' sg2' & e2) ⇒
         if classify_int_type2 sz1' sg1' sz2' sg2' == sgr then
           match hint with
             | Some (Mint sz sg) ⇒ return (& Ezbop zbop sz sg e1 e2)
             | _ ⇒ return (& Ezbop zbop I32 sgr e1 e2)
           end
         else error
     | _,_ ⇒ error
   end.
```

Figure 5.7: Compilation of binary integer operations

is actually an `mtype` (see rule `relexpr_tempvar`). This is because we want to reuse the type of temporary variable environments as defined in Compcert, i.e. dictionaries from identifiers to basic values `val`. We apply our operators `type_val` and `val_untype` to interface these untyped `val` with our well-typed world:

```
Program Definition get_tmpvar mty idt (le:temp_env) : dentype mty :=
  complete0 (let? v := le.[idt] in type_val mty v) .
Definition validget_tmpvar mty idt (le: temp_env) :=
  le.[idt] = Some (val_untype (get_tmpvar mty idt le)).
Program Definition set_tmpvar mty idt (le:temp_env)(x:dentype mty) :=
  le.[idt <- val_untype x ].
```

These will be the interpretations of the symbols `SymMapGet`, `SymMapValidGet` and `SymMapSet` justifying the equivalence relation and the axioms.

   We prove that this equivalence relation is correct, that is that every two related expressions have the same semantics:

```
Lemma rel_expr_correct:
  forall L E (vm: cvarmap E) e mty (we: expr L E mty) (G : env E) x,
    e =∼=ʳ  we → rel_Gce vm G → sempure G SS S we x →
      eval_expr cge ce cle cm e (val_untype x).

Lemma rel_lexpr_correct:
  forall L E (vm: cvarmap E) e (we: expr L E (typeof e)) G x,
    e =∼=ˡ  we → rel_Gce vm G → sempure G SS S we a →
      eval_lvalue cge ce cle cm e a.(adr_block) a.(adr_ofs)
      ∧  a.(adr_block) ≠ 0.
```

Proof. by mutual induction on `rel_expr` and `rel_lexpr` and repeated inversions of `sem`. The proofs of the sub cases largely depend on the interpretations attributed to the abstract symbols involved in the corresponding Whycert expressions. For instance as shown in Figure 5.3 the family of symbols `SymZBop` for binary integer operations are interpreted as their mathematical counterpart. The asserted conditions that ensure the absence of overflows allow for each operation

```
Definition Eptrarith L E cty (e1: exp (Tpointer cty)) sz sg (e2: exp (Tint sz sg))
  : expr L E _ :=
  Elet e1 (Elet e2 (Eassert ((TnotNullBlock (Tvar HI1))) (Eterm (Tsymapp
    (SymPtrArith cty) (Tvar HI1) (Tsymapp (SymPromoteInteger sz sg) (Tvar HI0)))))).

Definition Eetempvar {E} idt mty : expr E _ :=
  (Eassert (Tsymapp (SymMapValidGet mty) (Tsym (SymIdent idt)) (Tderef (Rtmp)))
    (Eterm (Tsymapp (SymMapGet mty) (Tsym (SymIdent idt)) (Tderef (Rtmp))))).

Inductive rel_expr E (vm: cvarmap E): Clight.expr → forall mty, expr L E mty → Prop :=
| relexpr_constint sz sg i:
                              ∅
 ----------------------------------------------------------------------
    Econst_int i (Tint sz sg) =~=ʳ  Ecallpar (PAR_degrade_int sz sg)
                                     [Tsym (SymConstInteger (sg i))]
| relexpr_constfloat sz f:
                              ∅
 ---------------------------------------------------------------------
   Econst_float f (Tfloat sz) =~=ʳ  Eterm (Tsym (SymConstFloat sz f))

| relexpr_tempvar idt mty:
                     ∅
 --------------------------------------------
   Etempvar idt mty =~=ʳ  Eetempvar idt mty

| relexpr_addrof e we:
               e =~=ˡ  we
 ---------------------------------------------
    Eaddrof e (Tpointer (typeof e)) =~=ʳ  we

| relexpr_arith_zbop (zbopc: integer_binop_or_cmp) sz' sg' ce1 ce2 sz1 sz2 sg1 sg2
    sz1' sz2' sg1' sg2' (e1: exp (Mint sz1 sg1)) (e2: exp (Mint sz2 sg2)) sz sg:
                          ce1 =~=ʳ  e1 ·  ce2 =~=ʳ  e2  ·
           typeof ce1 = Tint sz1' sg1'  ·  typeof ce2 = Tint sz2' sg2'  ·
      classify_int_type2 sz1 sg1 sz2 sg2 = classify_int_type2 sz1' sg1' sz2' sg2'
 ------------------------------------------------------------------------------------
    Ebinop zbopc ce1 ce2 (Tint sz' sg') =~=ʳ  Ezbop zbopc sz sg e1 e2

| relexpr_ptr_arith ce1 ce2 cty sz sg sz' sg'
   (e1: exp (Mpointer cty)) (e2: exp (Mint sz sg)):
                 ce1 =~=ʳ  e1  ·  ce2 =~=ʳ  e2  ·
    typeof ce1 = Tpointer cty   ·  typeof ce2 = Tint sz' sg'
 --------------------------------------------------------------
    Ebinop Oadd ce1 ce2 (Tpointer cty) =~=ʳ  Eptrarith e1 e2

| relexpr_lvalue e we mty (Heq:typeof e = mty):
                     e =~=ˡ  we
 ------------------------------------------------------
    e =~=ʳ  Eload (cast_eq Heq (T:=(Tpointer;exp)) we)

| relexpr_lvalue_arr e we cty l (Heq: typeof e = Tarray cty l) :
                     e =~=ˡ  we
 -------------------------------------------------------------------------------
    e =~=ʳ  Elet (cast_eq Heq (T:=(Tpointer;exp)) we)
            (Eterm (Tsymapp (conv_pointer_pointer (Tarray cty l) cty) (Tvar HI0)))

where "e =~=ʳ  we" := (rel_expr (E:=_) _ e  we)
```

Figure 5.8: Equivalence relation between C and Whycert expressions (right values)

```
with rel_lexpr E (vm: cvarmap E):
  forall e: Clight.expr, expr L E (Tpointer (typeof e)) → Prop :=
| rellexpr_var idt cty v:
           vm idt = Some (cty & v)
  ----------------------------------------
    Evar idt cty =∼=ˡ  Eterm (Tvar v)

| rellexpr_gvar i b gv:
                      vm i = None        ·
    Genv.find_symbol cge i = Some b  ·  Genv.find_var_info cge b = Some gv
  ----------------------------------------------------------------------
    Evar i gv.(gvar_info) =∼=ˡ  Eterm (Tsym (SymCGlobal i gv.(gvar_info)))

| rellexpr_deref e cty (we: exp (Mpointer cty)) :
                  e =∼=ʳ  we
  ------------------------------------------------------------------
    Ederef e cty =∼=ˡ  Elet we (Eassert ((TnotNullBlock (Tvar HI0)))
                            (Eterm (Tvar HI0)))

| rellexpr_field e si fi fld we cty (Heq: typeof e = Tstruct si fld):
                  e =∼=ˡ  we   ·   unrolled_field_type fi si fld = Some cty
  --------------------------------------------------------------------------
    Efield e fi cty =∼=ˡ  Elet (cast_eq Heq (T:=(Tpointer;exp)) we)
                            (Eterm (Tsymapp (SymField si fld cty fi) (Tvar HI0)))

  where "e =∼=ˡ  we" := (@rel_lexpr _ _ e we).
```

Figure 5.9: Equivalence relation between C and Whycert expressions (left values)

to prove the equivalence with its modulo-semantics.

This intermediate result guarantees us that the inserted assertions establish the sufficient conditions for the evaluation of the given C expression to succeed.

To actually compile C code we define a function that given a Clight expressions returns a Whycert expression that is in relation with it. This compilation function also needs to decide whether to compile the given Clight expression as a left value or as a right value. It therefore returns the following sum type:

```
Definition left_or_right_expr E e :=
  (expr L E (Tpointer (typeof e))) + {mty : mtype & expr L E mty }.
```

Obviously we can turn a left value into a right value by loading it from memory, while the converse is not true.

```
Program Definition get_right E e (x:option (left_or_right_expr E e))
    : option { mty : mtype & expr L E mty } :=
  match x with
    | Some (inr r) ⇒ return r
    | Some (inl x) ⇒
        let? (mty,_) := get_memtype (typeof e) in
          return (mty & Eload (cast (T:=(Tpointer;exp)) x))
    | None ⇒ error
  end.
```

```
Program Fixpoint comp_lrexpr hint E {vm: cvarmap E} (e: Clight.expr)
  : option (left_or_right_expr E e) :=
  match e return option (left_or_right_expr E e) with
    | Econst_int i cty ⇒ let? (sz, sg) := get_int_type in
        right (Ecallpar (PAR_degrade_int sz sg) [Tsym (SymConstInteger(sg i))])
    | Econst_float f cty ⇒ let? sz := get_float_type cty in
        right (Eterm (Tsym (SymConstFloat sz f)))
    | Evar idt cty ⇒
        match vm idt return _ with
          | Some (cty' & vr) ⇒ if cty == cty'  then left (Eterm (Tvar vr))
                                 else error
          | None ⇒ let? b := Genv.find_symbol cge idt in
                   let? gv := Genv.find_var_info cge b in
                     if gv.(gvar_info) == cty then
                       left (Eterm (Tsym (SymCGlobal idt cty)))
                     else error
        end
    | Etempvar idt cty ⇒ let? mty:= get_memtype cty in right(Eetempvar idt mty)
    | Ederef e cty ⇒ let? (cty', we) := get_right (comp_lrexpr None e) in
        if cty' == Mpointer cty then
          left (Elet (cast we)
                (Eassert ((TnotNullBlock (Tvar HI0))) (Eterm (Tvar HI0))))
        else error
    | Eaddrof e cty ⇒ if cty == Mpointer (typeof e) then
        let? we := get_left (comp_lrexpr None e) in
          right we
        else error
    | Eunop x x0 x1 ⇒ error
    | Ebinop bop e1 e2 (Tint sz sg) ⇒
        let? zbop := get_zbop bop in
        let? we := comp_zbop comp_lrexpr hint zbop e1 e2 in
          right' we
    | Ebinop bop e1 e2 (Tpointer cty) ⇒
        match bop return _ with Oadd ⇒
          let? we := comp_ptr_arith comp_lrexpr e1 e2 cty in
            right' we
          | _ ⇒ error end
    | Ebinop bop e1 e2 _ ⇒ error
    | Efield e fi cty ⇒
        let? we := get_left (comp_lrexpr None e) in
        match (typeof e) with Tstruct si fld ⇒
        let? cty' := unrolled_field_type fi si fld in
          if cty == cty' then
            left (Elet (cast (T:=(Tpointer;exp)) we)
              (Eterm (Tsymapp (SymField si fld cty fi) (Tvar HI0))))
          else error
          | _ ⇒ error
        end
    | Clight.Ecast x x0 ⇒ error
    | Econdition x x0 x1 x2 ⇒ error
    | Esizeof x x0 ⇒ error
  end.

Definition comp_expr (hint: option mtype) E {vl: cvarmap E} e :=
  get_right (comp_lrexpr hint e).

Definition comp_lexpr E {vl: cvarmap E} (e: Clight.expr) :=
  get_left (comp_lrexpr None e).
```

Figure 5.10: Compilation of Clight expressions

```
Definition get_left E e (x :option (left_or_right_expr E e)) :
  option (expr L E (Tpointer (typeof e))) :=
  match x with
    | Some (inl l) ⇒ return l
    | _ ⇒ error
  end.
```

The full definition of the compilation of Clight expressions is shown in Fig. 5.10. Besides the treatment of binary integer operations, most cases are easy and only checking of some type constraints is required.

To complete the proof of soundness of the compilation of C expressions, we prove that source and the target expressions are in our equivalence relation:

```
Definition rel_lrexpr E {vl} e (lre: left_or_right_expr E e) :=
  match lre with
    | inl we ⇒ e =∼=ˡ  we
    | inr (mty & we) ⇒ e =∼=ʳ  we
  end.

Theorem complrexpr_correct: forall e E (vl : cvarmap E)
  (lre : left_or_right_expr E e) hint,
  comp_lrexpr hint e = Some lre → rel_lrexpr e lre

Corollary compexpr_correct: forall e E (vl : cvarmap E)
  mty (we : expr L E mty) hint,
  comp_expr hint e = Some (mty & we) → e =∼=ʳ  we.

Corollary complexpr_correct: forall e E (vl : cvarmap E)
  (we : expr L E (Tpointer (typeof e))),
  comp_lexpr e = Some we → e =∼=ˡ  we.
```

Along with `rel_expr_correct` and `rel_lexpr_correct` which prove that equivalence relation is correct, this implies that the compilation of expressions preserves the semantics.

Obviously the presented compilation of C expressions remains incomplete. One could add a missing case by completing the function `comp_lrexpr` and fill the new hole in proof of `complrexpr_correct`.

5.5   Compilation of Statements and Functions

The intention of the compilation is to produce a Whycert program with the same structure and the same control flow as the C program: the Whycert program must simulate the C program. The Whycert language provides for everything necessary: conditionals, loops, exceptions to encode **break** and **return** and functions, that are mutually recursive as in C. Notable exceptions are **goto** jumps which may introduce unstructured control flow.

5.5.1   Compilation Schemes

We start by defining the compilation schemes for the statements. They are given as Coq functions whose arguments are to be provided by the actual compilation of the sub-expressions or sub-statements.

The easiest one, the Clight **skip**, is mapped to a constant of type **void**:

```
Definition Esskip {E} : expr E Tvoid := Eterm (Tsym SymConsttt).
```

Not much more difficult but with two arguments is the compilation of C sequences `s1; s2` which has its direct counterpart in Whycert:

```
Definition Esseq E ty1 ty2 (e1: expr E ty1) (e2: expr E ty2) :=
  Eseq e1 e2.
```

To simulate the setting of a temporary Clight variable we assign an updated mapping to the reference for temporary variables. A Whycert assignment returns the assigned value and thus has the type of the assigned expression. As we want all compiled expressions to have type **void**, we use the `Eignore` operator to ignore the returned value:

```
Definition Eignore E ty (we: exp ty) : expr E Tvoid :=
  Eseq we Esskip.

Definition Esset E mty idt (e: exp mty) : expr E Tvoid :=
  Eignore (Elet e (Eassign Rtmp (Tsymapp (SymMapSet mty)
    (Tsym (SymIdent idt)) (Tderef (Rtmp)) (Tvar HI0)))).
```

The equivalent of `Esset` is

```
(let v0 = e in tmp := mapset idt !tmp v0); skip
```

Similarly a C assignment becomes a Whycert assignment with an updated memory. However, here we need to make sure that the simulated C assignment does not fail at execution time, e.g. because of missing permissions. This is ensured by the abstract predicate `SymValidStore`. Also, as the C semantics provides for the assigned expressions to be casted to the type of the pointer, which is the type that will determine size of the memory chunk to be written, we want to make sure that this cast is neutral, i.e. that is does not change the value to be stored. We thus require the type of the assignment and the type of the assigned expression to be in the relation `neutral_cast`, which determines an expression that performs the necessary conversion, as formalised in `Encast`.

```
Inductive neutral_cast : mtype → mtype → Type :=
| nc_ii sz1 sg1 sz2 sg2: neutral_cast (Mint sz1 sg1) (Mint sz2 sg2)
| nc_pp cty: neutral_cast (Mpointer cty) (Mpointer cty).

Definition Encast_int {E} sz1 sg1 sz2 sg2 t : expr E _ :=
  Ecallpar (PAR_degrade_int sz2 sg2)
    [Tsymapp conv_int_integer sz1 sg1 t].

Definition Encast E mty1 mty2 (C: neutral_cast mty1 mty2) we :=
  (Elet we
    match C in neutral_cast mty1 mty2 with
      | nc_ii sz1 sg1 sz2 sg2 ⇒ (Encast_int sz2 sg2 (Tvar HI0))
      | nc_pp cty ⇒ (Eterm ((Tvar HI0)))
    end).

Definition Esassign E mty1 mty2 (C: neutral_cast mty1 mty2)
  (e1: exp (Tpointer mty2)) (e2: exp mty1) :=
  Eignore (Elet e1 (Elet (Encast C e2)
    (Eassert (Tsymapp (SymValidStore _) (Tderef Rmem) (Tvar HI1))
      (Eassign Rmem (Tsymapp (SymStore mty2) (Tderef Rmem)
                     (Tvar HI1) (Tvar HI0)))))).
```

which is equivalent in Why3 concrete syntax to

```
let v1 = e1 in let v0 = cast c e2 in
assert { valid_store !mem v1 };
mem := store mty2 !mem v1 v0
```

To compile C conditionals we need to convert the guard expression, which can be of any memory type, to a predicate stating that it is different from zero.

```
Definition Tbool E mty: ter mty → term E Tprop :=
  match mty return (ter mty) → ter Tprop with
    | Mint sz sg ⇒ fun t, Tsymapp (SymRel CompInteger Cne)
                              (Tsymapp (conv_int_integer sz sg) t)
                              (Tsymapp (SymConstInteger Z0))
    | Mfloat sz ⇒ fun t, Tsymapp (SymRel (CompFloat _) Cne)
                              (Tsymapp (conv_float_float _ _) t)
                              (Tsymapp (SymConstFloat Float.zero))
    | Mpointer cty ⇒ fun t, Tsymapp (SymNotNull _) t
  end.

Definition Esifthenelse E mty (e: exp mty) ty (e1 e2: exp ty) :=
  Elet e (Eif (Tbool (Tvar HI0)) e1 e2).
```

The same predicate is used for the guard of **while** loops. Here we also need to simulate the handling of the **break** and **continue** statements, which is encoded using exceptions. Conforming to the semantics of the C control, the **continue** exception is caught inside the loop, such that the next cycle can go on, whereas the **break** exception is caught outside the loop, in fact interrupting the loop. For reasons explained below, the compilation of **while** loops is split into a core part around which is added a label.

```
Definition Esbreak := Eraise ExcBreak (Tsym SymConsttt).

Definition Escontinue := Eraise ExcContinue (Tsym SymConsttt).

Definition Eswhil L E t_inv mty (e_cnd: exp mty) (e: exp Tunit) :=
  Etry
    (Eloop t_inv
      (Esifthenelse e_cnd (Etry e ExcContinue Esskip) Esbreak))
    ExcBreak Esskip.

Definition Eswhile L E t_inv mty (e_cnd: exp mty) (e: exp Tunit):=
  Elab (Eswhil t_inv e_cnd e).
```

The last exception involved in the compilation of C statements is used to encode a **return**. In the case the C function returns some value, there is an implicit cast to the function's return type, just like in an assignment, so we make use of the neutral cast operator `Encast`.

```
Definition Esreturn_some E mty1 mty2 {ty} (e: exp mty1)
  (C: neutral_cast mty1 mty2) : expr E ty :=
  Elet (Encast C e) (Eraise (ExcReturn mty2) (Tvar HI0)).

Definition Esreturn_none {E ty} : expr E ty :=
  Eraise (ExcReturn Tunit) (Tsym SymConsttt).
```

This exception is caught at the end of the body of the function to properly return the result. In addition the compilation of the body must simulate the memory allocations for parameters and local variables, the initialisation of the newly allocated memory blocks that correspond to formal parameters and, at the end, the release of memory. Here we need to construct the required vari-

able maps and establish all the hypotheses described at the beginning of this chapter (Section 5.2) which we made use of to prove the soundness of the compilation of logic expression and C expressions, namely `rel_Gce` and `rel_Gs`.

The following example illustrates the compilation scheme for C functions. Given is the following function with two parameters and two local variables:

```
void cf(int a, int* b){
  float c;
  int d[5];
  ...
}
```

For clarity, the result is shown in Why3 concrete syntax but using our actual symbols and parameters.

```
let f (a : C_int) (b : C_pointer_int) =
  { (* compilation of cf.(pre) *) }
  rtmp := SymMapEmpty;
  let m_a = PAR_mem_alloc_int () in
  let m_b = PAR_mem_alloc_pointer_int () in
  let m_c = PAR_mem_alloc_float () in
  let m_d = PAR_mem_alloc_array_int_5 () in
  rmem := SymStore_pointer_int (SymStore_int !rmem m_a a) m_b b;
  let result = try (* comp. of cf.(body) *) with Exc_return x → x in
  rmem := SymFree_int (SymFree_pointer_int !rmem m_b) m_a;
  result
  { (* compilation of cf.(post) *) }
```

Notice that the compilation of `cf.(pre)` and `cf.(post)` can refer to `a` and `b` and the compilation of cf.(body) can refer to `m_a`, `m_b`, `m_c` and `m_d`. It could also refer to `a` and `b`, but it doesn't. All these variable names are only for sake of explanation: the actual compilation uses well-typed De-Bruijn indices for which we must formally specify the typing environment they are well typed in.

We start by defining the signature of the compiled function. It is deduced directly from the signature of the C function to be compiled `cf`:

```
Definition mkt (x : ident * ctype) : type := snd x.
Definition mapt := List.map mkt.
Definition fsignature cf :=
  {| returns := cf.(fn_return); receives := mapt cf.(fn_params) |}.
```

`mapt cf.(fn_params)` is thus the typing environment for the contract and the body of the Whycert function obtained by a compilation of the Clight function `cf`. It contains an entry for each formal parameter. In function contracts, an occurrence of a formal parameter with the identifier `id` will thus be compiled to a De-Bruijn index that refers to the position at which `id` occurs in the list of formal parameters `cf.(fn_params)`.

On the contrary, inside the body of a C function occurrences of formal parameters are treated like program variables and just like for local variables we allocate a memory block for them. For this we iterate of the list of variables and for each of them call the parameter `PAR_mem_alloc` which modifies the memory reference `Rmem` and returns the newly allocated pointer. Every result is kept in a local **let** as illustrated by the example. Inside the nesting of **let** blocks, the allocated pointers are thus visible in reverse order: the innermost **let** block holds the last allocated pointer. Given a generic list of variables `el` of type `list (ident * ctype)` and a generic outer typing environment `E`, we define the typing environment inside the nesting of

let blocks `mk_El el E` as follows:

```
Definition mkpt (x : ident * ctype) : type := Tpointer (snd x).
Definition mappt := List.map mkpt.
Infix "+<+" := List.rev_append (at level 60, right associativity).
Definition mk_El el E := mappt el +<+ E.
```

Given a Clight function `cf`, its body is thus compiled as a Whycert expression with the following typing environment:

```
Definition body_E cf :=
  mk_El (cf.(fn_params) ++ cf.(fn_vars)) (mapt cf.(fn_params)).
```

In the example this would be the following list:

```
[ C_pointer_array_int_5; C_pointer_float; C_pointer_pointer_int;
  C_pointer_int; C_int; C_pointer_int ]
```

The design choice of dependently typed expressions forces us to be this precise but it guarantees that every generated De-Bruijn index is correct by construction.

Fig. 5.11 shows the definition of the helper functions needed for the compilation of function bodies. As explained above, we need to construct a nesting of let blocks, one for each local variable to be allocated (including formal parameters). This is the role of `alloc_vars` which, given an expression `e`, recursively constructs the blocks around `e`. Notice that for a list of variables `el`, `e` has type `expr (mk_El el E) A`, i.e. it is any expression that can refer to the allocated pointers that are bound by the let blocks.

The first thing we need to do with these pointers is binding the formal parameters. For each parameter we need to pick, using two De-Bruijn indices, its value and the pointer, that we just allocated for it, to store the former into the latter. Considering the list of parameters split into two parts, a reversed prefix `pelr` and a suffix `pel`, these indices refer to a typing environment defined by `prefinal_E pelr pel vel`, where `vel` is the list of local variables. The functions `geti1` and `geti2` respectively compute these two indices, as illustrated by the following graphic, where the red, big box represents the portion of typing environment corresponding to `vel`, the blue, medium box represents the portions of typing environment corresponding to `pel` and the green, smallest box represents the portions of typing environment corresponding to `pelr`:
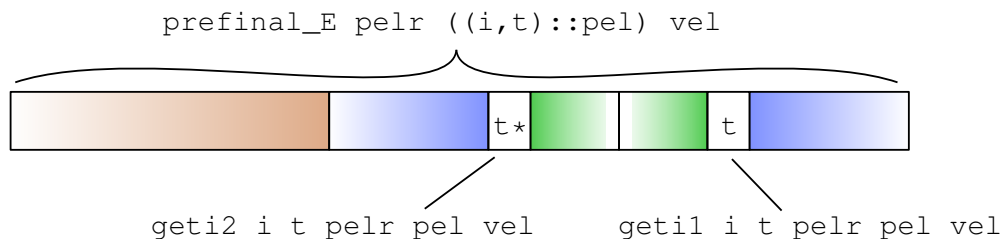
The function `bind_pars` traverses the list of parameters and, starting from the current memory, stores each `geti1` at the pointer `geti2` to obtain a memory in which every parameter is bound to its memory block.

Eventually the function `free_vars` traverses the list of parameters and local variables to construct a sequence of memory updates representing the release of all allocated blocks. The respective De-Bruijn indices are computed by `geti3` which given a list of variables and a reversed

```
Fixpoint alloc_vars el E A : expr (mk_El el E) A → expr E A :=
   match el with
     | [] ⇒ id
     | (idt,cty)::l' ⇒ fun e, Elet (Ecallpar (PAR_mem_alloc cty) [])
                                 (alloc_vars e)
   end.

Infix "@" := List.map (at level 15, right associativity) : list_scope.

Definition prefinal_E pelr pel vel :=
  mk_El (pelr +<+ pel ++ vel) (mkt @ pelr +<+ mkt @ pel).

Definition final_E := prefinal_E [].

Definition geti1 idt cty pelr pel vel
  : var cty (prefinal_E pelr ((idt, cty) :: pel) vel) := ...

Definition geti2 idt cty pelr pel vel
  : var (Tpointer cty) (prefinal_E pelr ((idt,cty)::pel) vel) := ...

Fixpoint mk_bind_pars {pel vel} pelr
  : term (prefinal_E pelr pel vel) Tmem → term (prefinal_E pelr pel vel) Tmem:=
  match pel with
    | nil ⇒ id
    | (idt, cty)::pel' ⇒ fun t, mk_bind_pars ((idt, cty) :: pelr)
          (Tsymapp (SymStore cty) t (Tvar (geti2 idt cty pelr pel' vel))
            (Tvar (geti1 idt cty pelr pel' vel)))
  end.

Definition bind_pars {pel vel} : term (final_E pel vel) Tmem :=
  mk_bind_pars [] (Tderef Rmem).

Definition geti3 cty elr idt el E
  : var (Tpointer cty) (mk_El (elr +<+ (idt, cty) :: el) E) := ...

Fixpoint mk_free_vars {el E E'} elr: term (E' ++ (mk_El (elr +<+ el) E)) Tmem:=
  match el with
    | nil ⇒ Tderef Rmem
    | (idt, cty)::el' ⇒ Tsymapp (SymFree cty) (mk_free_vars ((idt, cty)::elr))
                            (Tvar (lift_idx E' (geti3 cty elr idt el' E)))
  end.

Definition free_vars {A pel vel} : term (A::(final_E pel vel)) Tmem :=
  mk_free_vars (E':=[A]) [].

Definition Ebody cf (e: expr (body_E cf) cf.(fn_return)) :=
  (Eseq (Eassign Rtmp (Tsym SymMapEmpty))
    (alloc_vars
      (Eseq (Eassign Rmem bind_pars)
        (Elet (Etry e (ExcReturn cf.(fn_return)) (Eterm (Tvar HI0)))
          (Eseq (Eassign Rmem free_vars) (Eterm (Tvar HI0))))))).
```

Figure 5.11: Compilation of function bodies

prefix, returns the index of the pointer to the allocated block at that position, similarly to `geti2` but without discriminating between formal parameters and local variables, as all of them need to be freed.

The implementations of `geti1`, `geti2` and `geti3` are not shown in the figure as they are quite tedious and not very interesting, quite the contrary: as in general the types of the variables and parameters are all different from each other, there should be only one way to construct an index that respects the required type.[2]

This concludes the compilation scheme for function bodies, `Ebody` (Fig. 5.11), which given the C function in question `cf` and the compiled body of the function of type `expr (body_E cf) cf.(fn_return)`, i.e. able to refer to all the allocated pointers and whose type is the return type of `cf`. Notice that most of the statements are compiled to expressions of type `Tvoid`. The only statement compiled to a non-void expression is **return** `e` which is thus necessary and the end of a non-void function.

At last, to simulate a call to such a function, we must construct a list of terms for the arguments because Whycert supports only pure terms as function arguments. As the global reference `Rtmp` for the current temporary variables is shared and will be overwritten by the called function, we must also store its current contents in a local **let** and then restore it after the function call, possibly updated with the result of the function. For instance, the compilation of `tmp1 = f(a, b, c)` looks like this:

```
let ttmp = !rtmp in
let result =
  let t1 = (* compilation of a *) in
  let t2 = (* compilation of b *) in
  let t3 = (* compilation of c *) in
    f (t1, t2, t3)
in rtmp := SymMapSet "tmp1" ttmp result; ()
```

To be able to define a convenient compilation scheme like for all the other statements, we use a supplementary data structure for the list of compiled argument expressions `exprlist` (see Fig. 5.12). It is essentially a heterogeneous list of expressions (heterogeneous with respect to types of the expressions), but where the typing environment of each expression contains the types of all the previous ones. This is to simplify the construction of the nesting of let-blocks as illustrated by the example. Another dependent data structure `set_result cty oid` indicates whether the result of the function of type `cty` should be set to an identifier `Some id` or not (`None`). In the first case, `cty` is constrained to be a memory type. `Escall` formally defines the compilation scheme for function calls, given a C function `cf`, a heterogeneous list of expressions whose types coincide with the types of the parameters of `cf`, a `set_result` to indicate whether to store the result respecting the return type of `cf` and the De-Bruijn index corresponding to `cf` in the list of functions of the program.

5.5.2   Compilation function and Proof Approach

Given all these compilation snippets we can define a structurally recursive function mapping Clight statements to Whycert expressions, say `comp_stmt` of type `statement → option Why.expr`. As we saw with the compilation of logical terms and Clight expressions, a next step is to prove that this function preserves the semantics of a given statement. Roughly, we need to prove that for every statement `s` that can be compiled to

---

2. This statement is probably not provable in Coq itself and for this development to be fully certified we however prove certain properties about it, but a huge amount of confidence could be given to this kind of index calculation even without such a proof.

```
Inductive exprlist E : list type → Type :=
| el_nil : exprlist E []
| el_cons ty P: expr E ty →  exprlist (ty::E) P → exprlist E (ty::P).

Fixpoint fold_exprlist E P (l: exprlist E P) A
  : expr (P +<+ E) A → expr E A :=
  match l with
    | el_nil ⇒ id
    | el_cons ty _ e _ l' ⇒ fun e', Elet e (fold_exprlist l' e')
  end.

Inductive set_result : ctype → option ident → Type :=
| set_result_yes mty idt: set_result mty (Some idt)
| set_result_no cty: set_result cty None.

Definition option_mapset E cty oidt (X: set_result cty oidt)
  : term L E cty → term L E Tmap → term L E Tmap :=
  match X  with
    | set_result_yes mty idt ⇒ fun t tm,
                Tsymapp (SymMapSet mty) (Tsym (SymIdent idt)) tm t
    | set_result_no omty ⇒ const id
  end.

Fixpoint mk_arglist P E : hlist (term0 (P +<+ E)) P :=
  match P with
    | [] ⇒ []%hlist
    | A::P' ⇒ (Tvar (lift_rev_idx P' HI0) :: mk_arglist P' (A :: E))%hlist
  end.

Definition Escall E cf (l: exprlist (Tmap :: E) (body_P cf))
  res (sres: set_result cf.(fn_return) res) (fi: lidx cf cfl)
  : expr  E Tunit:=
  Elet (Eterm (Tderef Rtmp))
    (Elet
      (fold_exprlist l (Ecall (mk_fidx fi) (mk_arglist (body_P cf) _)))
      (Eignore (Eassign Rtmp (option_mapset sres (Tvar HI0) (Tvar HI1))))))).
```

Figure 5.12: Compilation scheme of function calls

some Why expression `e`, if `e` executes without blocking, which can be verified thanks to the WP calculus defined in Chapter 3, then so does `s`:

```
forall s e, comp_stmt s = Some e →
    (G/SS/S/ e ⟹ S'/o →  exec_stmt ce cle cm clm s t cle' cm' co) ∧
    (G/SS/S/ e ⟹ ∞ →  execinf_stmt ce cle cm clm s t)
```

where we should not worry about the free variables, neither the non-terminating case at the moment. ⟹ (denoting `Why.sem`) and `exec_stmt` are defined by the inference rules shown in Figs. 3.8 and 4.18, respectively.

In the previous proofs an induction over the syntactic structure produced for each syntactic case the inductive hypotheses that were needed to prove the correct evaluation in this case. Such proofs by structural induction were possible because the evaluation of expressions and logical terms is structural, i.e. the evaluation of any expression or term depends only on the evaluation of sub-expressions, respectively sub-terms. This is clearly not the case for the execution of C statements: the execution of a loop depends on the execution of the same loop if the side conditions are valid and the execution of a function call depends on the execution of the body of the given

function. In these cases an induction over the structure of the syntax of statements does not lead to sufficient inductive hypothesis.

This is not really surprising as we need hypotheses about the execution, the semantics of a program, not about its structure. Recall that the semantics of our languages are inductively defined in a big-step style. A next attempt is thus a proof by induction over the structure of derivations of the operational semantics. As shown in [36] and [9] with this approach it is rather easy to prove the *forward simulation* of a compilation, which in our case would be stated as

```
forall s e, comp_stmt s = Some e →
  exec_stmt ce cle cm clm s t cle' cm' co →  G/SS/S/ e ⟹ S'/o
```

Such a proof of forward simulation can be made easy thanks to an induction over the semantic rules of the source language, in our case `exec_stmt`, replaces, for each rule, `s` by the concrete statement occurring in the conclusion of the rule. Consequently the hypothesis `comp_stmt s = Some e` simplifies according to the definition of `comp_stmt` similarly to what happens during a structural induction. Additionally the appropriate inductive hypotheses about the execution of sub-statements of `s` are provided.

However, that's not our desired result. By proving the verification conditions we obtain a hypothesis about the execution of the target language and what we want to obtain is a conclusion about the execution of the source language, so what we need to prove is the reverse implication, i.e. *reverse forward simulation*. Moreover, forward simulation does not necessarily hold as for simplicity or any other reason the compilation may need to introduce assertions that are stronger than actually required by the semantics. One important example is the encoding of loop-assigns clauses which are currently not considered in the semantics (see Section 4.2.2).

But reverse forward simulation cannot not be proved directly by induction over the semantic rules of the target language, in our case `Why.sem`, as this replaces, for each rule, `e` by the expression occurring in the conclusion of the rule leaving `s` and thus `comp_stmt s` unchanged. More importantly, the inductive hypotheses are about the execution of sub-expressions of `e` which do not generally correspond to compilations of sub-expressions of `s`.

There are several approaches to work around this problem, some of which will be discussed in Section 5.7.1. We chose the following one which will be detailed below: Roughly we inductively define the sub-language of Whycert expressions that are target of the compilation of Clight statements `gexpr`. We then inductively define the terminating semantics of this sub-language `gensem` with an (almost) exact one-to-one correspondence with the semantics of Clight statements: for each rule of the latter there is exactly one rule of the former, with an exception for the **while** loops. The compilation now produces expressions of `gexpr` and the soundness proof is split into two stages: one showing that `Why.sem` includes `gensem` and another one showing that `gensem` includes `exec_stmt`. The latter can now make profit of a convenient induction scheme as the two inductive predicates have the same structure and we can focus on the important details that make that the Whycert program actually simulates the Clight program. On the other hand, the former should be obvious as it's a matter of an equivalence between semantics of a language and its own sub-language.

```
forall e, Why.sem G SS S e S' o →  gensem G SS S e S' o
```

However this can still not be proved by a direct induction over `Why.sem` as this suffers from the same problem as above, i.e. the inductive hypotheses would not be the right ones. Instead we inductively define the absence of semantics of a Whycert expression in a given state `Why.nsem` and prove that `Why.sem`, `Why.seminf` and `Why.nsem` form a partition for any expression and state, i.e. they are mutually exclusive and collectively exhaustive. By determinism of the semantics and principle of excluded middle our goal is equivalent to

```
Inductive gexpr {cfl : list function} {L E}
   : forall {ty}, expr (mk_F cfl) L E ty → Type :=
| geassert p: gexpr (Esassert p)
| geskip : gexpr Esskip
| geassign mty1 mty2 (C : neutral_cast mty1 mty2)
  (e1 : exp (Mpointer mty2))
  (e2 : exp mty1) :
    pure e1 → pure e2 →
    gexpr (Esassign C e1 e2)
| geset mty idt (e: exp mty):
    pure e →
    gexpr (Esset idt e)
| gecall cf (el: exprlist (Tuser Tmap :: E) _) res
  (sres: set_result cf.(fn_return) res) (fi:cfidx cf cfl):
    gexpr (Escall el sres fi)
| geseq ty1 (e1: exp ty1) ty2 (e2: exp ty2):
    gexpr e1 → gexpr e2 → gexpr (Eseq e1 e2)
| geifthenelse mty (e0: exp mty) ty1 (e1 e2: exp ty1):
    pure e0 →
    gexpr (E:=_) e1 → gexpr (E:=_) e2 →
    gexpr (Esifthenelse e0 e1 e2)
| gereturn_none {ty}:
    gexpr (Esreturn_none (ty:=ty))
| gereturn_some mty1 mty2 {ty} e (C: neutral_cast mty1 mty2):
    pure e →
    gexpr (Esreturn_some (ty:=ty) e C)
| gebreak {ty}:
    gexpr (Esbreak (A:=ty))
| gecontinue {ty}:
    gexpr (Escontinue (A:=ty))
| gewhile (t_inv: ter Tprop) mty (e_cnd: exp mty) (e: exp Tunit):
    pure e_cnd →
    gexpr (L:=_) (E:=_) e →
    gexpr (Eswhil t_inv e_cnd e)
| gelab ty (e: exp ty):
    gexpr (L:=_) e → gexpr (Elab e)
| gelab' ty (e: exp ty):
    gexpr (L:=_) e → gexpr (Elab e).
```

Figure 5.13: Sub-language of Whycert expressions used in the compilation of C statements

```
forall e, ¬Why.nsem G SS S e →  ¬(∃S' o, gensem G SS S e S' o) →
    Why.seminf G SS S e
```

which can be proved by co-induction and case analysis over `e`.

The complete proof scheme is illustrated in the diagram in Figure 5.16.

The definition of the sub-language `gexpr` is shown in Figure 5.13. It as given as a dependently typed inductive with one case for each currently handled C statement plus one, `gelab'`, for the label around `while` loops needed to encode the `loop assigns` clause. The definition makes use of another sub-language, `pure`, of Whycert expressions that do not have side effects. The compilation of Clight expressions generates pure Whycert expressions and we define the following helper functions that directly return the appropriate proof:

```
Fixpoint alloc_state (el: list (ident * ctype)) (S: state) :=
  match el with
    | [] ⇒ S
    | (idt, cty)::el ⇒ alloc_state el (update_alloc S cty)
  end.
Fixpoint alloc_env (el: list (ident * ctype)) E (G:env E) S
  : env (mk_El el E) :=
  match el with
    | [] ⇒ G
    | (idt, cty)::el ⇒
        alloc_env el (mem_alloc_pt (cmem S) cty::G) (update_alloc S cty)
    end.
Definition optresult_set_tmpvar (cty:ctype) res (sres: set_result cty res) te :
  dentype cty → temp_env :=
  match sres with
    | set_result_yes mty idt ⇒ set_tmpvar idt te
    | set_result_no omty ⇒ const te
  end.
Inductive genout (cty: ctype): outcome cty → dentype cty → Prop :=
| genout_normal x : genout (Outval x) x
| genout_exn x : genout (Outexn ExcReturn x) x.


Inductive gensem L E (G: env E) (SS: states L) S
  : forall ty (e: expr L E ty), gexpr e → state → outcome ty → Prop :=
| gensem_assert p: evalprop p G SS S → gensem G SS S (geassert p) S outvoid
| gensem_skip : gensem G SS S (ty:=Tunit) geskip S outvoid
| gensem_assign mty1 mty2 (C : neutral_cast mty1 mty2)
  (e1 : exp (Mpointer mty2)) (pe1: pure e1)
  (e2 : exp mty1) (pe2: pure e2) x1 x2:
    sempure G SS S e1 x1 → sempure_ncast (x1::G) SS S C e2 x2 →
    m_valid_access_p (cmem S) mty2 x1 Writable →
    gensem G SS S (geassign C pe1 pe2)
      (update_mem S (complete4 simple_storep _ (cmem S) x1 x2)) outvoid
| gensem_set mty idt (e: exp mty) (pe: pure e)  x:
    sempure G SS S e x →
    gensem G SS S (geset idt pe)
      (update S (Rtmp) (set_tmpvar idt (ctmp S) x)) outvoid
| gensem_call cf (el: exprlist F L (Tuser Tmap :: E) _) G_args res
  (sres: set_result cf.(fn_return) res) (fi:cfidx cf) S' o x:
    let le := ctmp S in
    sempurelist (le::G) SS S el G_args →
    evalprop fi.(ppre) G_args [] S →
    let S1 := update S (Rtmp) []%imap in
    let S2 := alloc_state (cf.(fn_params) ++ cf.(fn_vars)) S1 in
    let G_body := alloc_env (cf.(fn_params) ++ cf.(fn_vars)) G_args S1 in
    let S3 := update_mem S2 (evalterm bind_pars G_body [S] S2) in
    gensem (L:=1) (E:=body_E cf) G_body [S] S3 fi.(pbody) S' o →
    genout o x →
    let S'1 := update_mem S' (evalterm free_vars (x::G_body) [S] S') in
    evalprop fi.(ppost) (x::G_args) [S] S'1 →
    let S'2 := update S'1 (Rtmp) (optresult_set_tmpvar sres le x) in
    gensem G SS S (gecall el sres fi) S'2 outvoid
| gensem_seq1 ty1 e1 (e1: gexpr e1) ty2 e2 (e2: gexpr e2) S' x S'' o:
    gensem G SS S e1 S' (Outval x) → gensem G SS S' e2 S'' o →
    gensem G SS S (geseq e1 e2) S'' o
| gensem_seq2 ty1 e1(e1: gexpr e1) ty2 e2(e2: gexpr e2) S' tyx (ex: exn tyx) x:
    gensem G SS S e1 S' (Outexn ex x) →
    gensem G SS S (geseq e1 e2) S' (Outexn ex x)
```

Figure 5.14: Whycert Specialised semantics

```
| gensem_ifthenelse mty (e0: exp mty) (pe0: pure e0) x ty1 (e1 e2: exp ty1)
      (e1: gexpr e1) (e2: gexpr e2) S' b  o:
    sempure_bool G SS S e0 x b →
    gensem (E:=_) (x::G) SS S (gexpr_bool b e1 e2) S' o →
    gensem G SS S (geifthenelse pe0 e1 e2) S' o
| gensem_return_none ty:
    gensem G SS S gereturn_none S (Outexn (A:=ty) ExcReturn void)
| gensem_return_some mty1 mty2 ty e (pe: pure e) (C: neutral_cast mty1 mty2) x:
    sempure_ncast G SS S C e x →
    gensem G SS S (gereturn_some C pe) S (Outexn (A:=ty) ExcReturn x)
| gensem_break ty:
    gensem G SS S gebreak S (Outexn (A:=ty) ExcBreak void)
| gensem_continue ty:
    gensem G SS S gecontinue S (Outexn (A:=ty) ExcContinue void)
| gensem_while_false (t_inv: ter Tprop) mty (e_cnd: exp mty)
      (pe_cnd: pure e_cnd) x (e: exp Tunit) (e: gexpr e):
    evalterm t_inv G (SS) S → sempure_bool G (SS) S e_cnd x false →
    gensem G SS S (gewhile t_inv pe_cnd e) S outvoid
| gensem_while_stop  (t_inv: ter Tprop) mty (e_cnd: exp mty) x
      (e: exp Tunit) (pe_cnd: pure e_cnd) (e: gexpr e) S' o o':
    evalterm t_inv G (SS) S → sempure_bool G (SS) S e_cnd x true →
    gensem (L:=_) (E:=_) (x::G) (SS) S e S' o →
    outcome_break_or_return o o' →
    gensem G SS S (gewhile t_inv pe_cnd e) S' o'
| gensem_while_loop (t_inv: ter Tprop) mty (e_cnd: exp mty) x
      (e: exp Tunit) (pe_cnd: pure e_cnd) (e: gexpr e) S' o S'' o':
    evalterm t_inv G (SS) S → sempure_bool G (SS) S e_cnd x true →
    gensem (L:=_) (E:=_) (x::G) (SS) S e S' o →
    outcome_normal_or_continue o →
    gensem G SS S' (gewhile t_inv pe_cnd e) S'' o' →
    gensem G SS S (gewhile t_inv pe_cnd e) S'' o'
| gensem_label ty (e: exp ty) (e: gexpr e) S' o:
    gensem (L:=_) G (S::SS) S e S' o →
    gensem G SS S (gelab e) S' o
| gensem_label' ty (e: exp ty) (e: gexpr e) S' o:
    gensem (L:=_) G (S::SS) S e S' o →
    gensem G SS S (gelab' e) S' o
.
```

Figure 5.15: Whycert Specialised semantics

```
Program Definition compp_expr hint E {vl:cvarmap E} (e:Clight.expr)
  :  option {mty : mtype & { e : expr L E mty | pure e } } :=
  let? e ::= comp_expr hint e in return (& dsnd e).

Program Definition compp_lexpr E {vl: cvarmap E } (e: Clight.expr)
  :  option { e : expr L E (Tpointer (typeof e)) | pure e } :=
  let? e ::= comp_lexpr e in return e.
```

The definition of the compilation function is then easy (Fig. 5.17): we simply have to recurse over the structure of statements compiling all the sub-terms and sub-expressions if possible to construct such a `gexpr`. Given a mapping for label names `lm`, a mapping for variable names `vm`, a boolean `brk` indicating whether the expression is allowed to raise a **break** or a **continue**, the return type of the currently compiled function `rtt`, a C statement `cs` and a type `ty`, the function `comp_stmt` returns a value of type `{ e : expr F L E ty & gexpr e }` or fails. Recall that statements can only be compiled to expressions of type `Tvoid`, so we use the

```
Definition PpreservePermissions {L E} : prop (succ L) E :=
  Tsymapp SymMemPreservePermissions (Tat B0 Rmem) (Tderef Rmem).

Fixpoint comp_stmt L E (lm: labmap L) {vm: cvarmap E} brk {rtt} cs ty
   : option { e : expr F L E ty & gexpr e } :=
  match cs with
    | Sassert p ⇒ ⇒()
        let? wt := comp_pred lm vm0 vm p in
          return geassert wt
    | Sskip ⇒ ⇒() return geskip
    | Sassign e1 e2 ⇒ ⇒()
        let? (mty1,e1) := compp_memtype_lexpr e1 in
        let? (mty2, e2) := compp_expr (Some mty1) e2 in
        let? C := ' (get_cast mty2 mty1) in
          return (geassign C (dsnd e1) (dsnd e2))
    | Sset idt e ⇒ ⇒()
        let? (mty, e) := compp_expr None e in
          return geset idt (dsnd e)
    | Ssequence cs1 cs2 ⇒ let? e1 := comp_stmt lm brk cs1 Tunit in
                          let? e2 := comp_stmt lm brk cs2 ty in
                            return geseq (dsnd e1) (dsnd e2)
    | Sifthenelse e cs1 cs2 ⇒
        let? (mty, e) := compp_expr None e in
        let? e1 := comp_stmt lm brk cs1 ty in
        let? e2 := comp_stmt lm brk cs2 ty in
          return geifthenelse (dsnd e) (dsnd e1) (dsnd e2)
    | Swhile inv tl e s ⇒ ⇒()
        let? wt := comp_pred (lm_lift lm) vm0 vm inv in
        let? wt_ass := comp_assigns_clause (lm_lift lm) vm0 vm tl
                                     (Tat B0 Rmem) (Tderef Rmem) in
        let w_inv := Pand wt (Pand PpreservePermissions wt_ass) in
        let? (mty, e) := compp_expr None e in
        let? e1 := comp_stmt (lm_lift lm) true s Tunit  in
          return gelab' (gewhile w_inv (dsnd e) (dsnd e1))
    | Sbreak ⇒ if brk then return gebreak else error
    | Scontinue ⇒ if brk then return gecontinue else error
    | Sreturn e ⇒
        let? omty := get_omemtype rtt in
        match e,'omty with
          | None,None ⇒ return gereturn_none
          | Some e,Some mty ⇒
              let? (mty', e) := compp_expr (Some mty) e in
              let? C := ' (get_cast mty' mty) in
                return gereturn_some C (dsnd e)
          | _, _ ⇒ error
        end
    | Slabel l s ⇒
        let? e := comp_stmt (lm_add lm l) brk s ty in
          return gelab (dsnd e)
    | Scall res e el ⇒ ⇒()
        let? (cf, f) := get_fun e in
        let? sres := get_set_result cf.(fn_return) res in
        let? el := comp_expr_list (body_P cf) el in
          return gecall el sres f
    | Sdowhile _ _ _
    | Sfor' _ _ _ _
    | Sswitch _ _
    | Sgoto _ ⇒ error
  end.
```
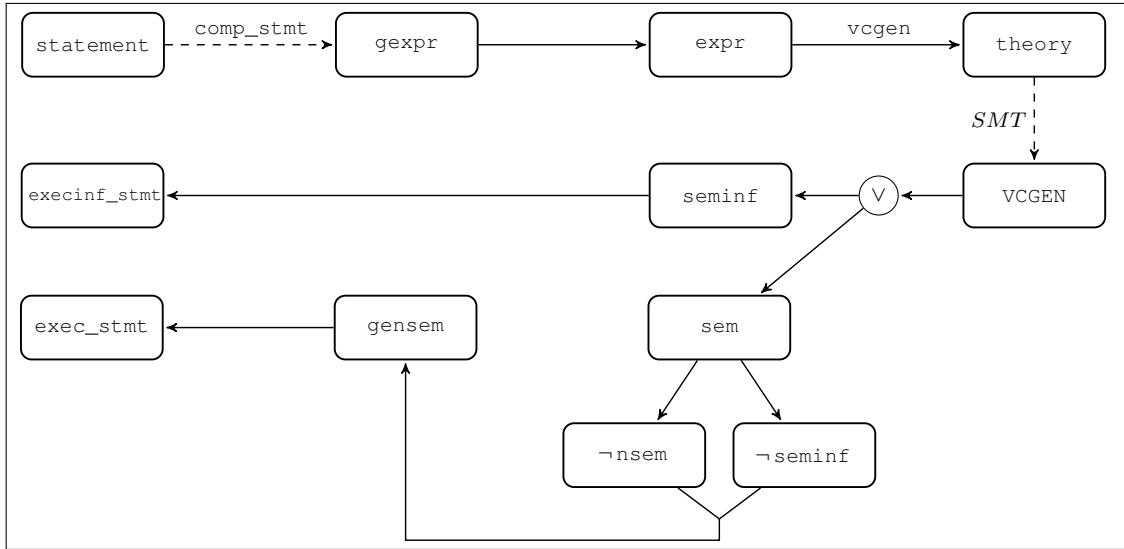
Figure 5.17: Compilation of statements

Figure 5.16: Proof Schema

notation $\Rightarrow$`() e` to return `e` if the given `ty` is `Tvoid` and to fail otherwise.

In the case of **while** loops, the invariant is enriched with information about permissions and with an encoding of the **loop assigns** clause relating the current memory (`Tderef Rmem`) with the memory at the enclosing label (`Tat B0 Rmem`), i.e. outside the loop. The compiled invariant is thus stronger than the original one and also stronger than needed to prove the preservation of semantics as the **assigns** clause is currently not considered by the semantics of annotated Clight. This strengthening is required to make the generated proof obligations practical, as explained in Section 5.6.3.

We currently don't handle **do**`{}`**while**`()` and **for**`(,,)` loops as well as well-structured Clight **switch** clauses, even if these cases could be easily encoded in Whycert. On the other hand, **goto** jumps are excluded as they would require a different kind of WP calculus.

With the ability to compile statements we can finally compile C functions and whole programs (Fig. 5.18). For convenience, we define a data structure for the compiled C functions `prefunc` to be generated by `precomp_func` which compiles the body of a given function along with its pre- and post-condition using appropriately initialised variable mappings for the formal parameters and local variables. Notice that, similarly to **while** loops, the function contract is strengthened according to its `assigns` clause. Furthermore we strengthen the pre-conditions with information about the global program variables (see Sections 5.6.3 and 5.6.3).

Whole programs are compiled by compiling the list of functions and mapping them to their final form (`mkcfunc`) where the compiled body is embedded into an expression that simulates the allocation and deallocation of the local variables (`Ebody`). We however check certain conditions required by the semantics of whole programs, such as the existence and correct type of the main function and the possibility to construct the initial memory according to initialisers of the global variables. In addition to the compiled program, `comp_prog` returns also the pre-condition of the main function, such that it can be added to the list of verification conditions to be proved valid in the initial memory. This finally allows us to define the verification condition generator for a whole annotated Clight program `vcgen`: given the list of types appearing in the program `ctypes` it returns the list of needed axioms (cf. Section 5.1.3) and the list of concrete verification conditions (cf. Section 3.5.1) enriched by the pre-condition of the main function in the initial state (cf. Section 5.6.3).

```
Record prefunc cf := {
  ppre : prop 0 (body_P cf); ppost : prop 1 (cf.(fn_return) :: body_P cf);
  pbodye : expr F 1 (body_E cf) cf.(fn_return); pbody : gexpr pbodye
}.
Definition precomp_func cf : option (prefunc cfl cf) :=
  if Coqlib.list_norepet_dec ident_eq
      (var_names cf.(fn_params) ++ var_names cf.(fn_vars)) then
    let vm := contract_vm cf.(fn_params) in
    let? pre := comp_pred lm0 vm pvm0 cf.(fn_pre) in
    let vm1 := varmap_add vm cf.(fn_post_result_var) in
    let lm1 := lm_add lm0 cf.(fn_old) in
    let? post := comp_pred lm1 vm1 pvm0 cf.(fn_post) in
    let? post_ass := comp_assigns_clause lm1 vm1 pvm0 cf.(fn_assigns)
                    (Tat B0 Rmem) (Tderef Rmem) in
    let? e := comp_stmt lm1 (vm:=body_vm _ (cf.(fn_params) ++ cf.(fn_vars)))
              (rtt:=cf.(fn_return)) false cf.(fn_body) cf.(fn_return) in
      return {|
        ppre  := Pand (Pvalid_globvars prog_vars (Tderef Rmem)) (pre);
        ppost := Pand post (Pand PpreservePermissions post_ass);
        pbody := dsnd e |}
  else error.

Definition mkcfunc cf (pf: prefunc cf) : func F (fsignature cf) :=
   {|pre := pf.(ppre); post := pf.(ppost); body := Ebody pf.(pbodye) |}.

Definition comp_prog cp : option (prog psignature * closed_prop) :=
  if list_norepet_dec (prog_funct_names cp ++ prog_var_names cp) then
    let? (mainf, fi) := fm.[cp.(prog_main)] in
    let? initm := Genv.init_mem cp in
    let? fl := precomp_funclist cp.(prog_vars) cge fm cp.(prog_funct) in
      if type_of_function mainf == Tfunction Tnil (Tint I32 Signed) then
        return (Indexes.xmap mkcfunc fl, abstrv (accslidx fi fl).(ppre))
      else error
  else error.

Definition vcgen ctypes cp : option (vc_list * vc_list) :=
  let? (p,pre_main) := comp_prog cp in
    return (needed_axioms ctypes,
      Pimply (Pinit_globvars cp) pre_main :: vcgen p).
```

Figure 5.18: Compilation of C functions

5.5.3   Soundness Proof

The outlined approach allows performing the proof of semantics preservation as anticipated above. Several details which complicate the actual proof are explained in this section.

The first part to be proved is that the specialised semantics `gensem`, shown in Figures 5.14 and 5.15, is simulated by the original Whycert semantics `sem`:

```
Theorem gensem_reverse forward:
  forall L E G SS S ty (e: expr L E ty) (ge: gexpr e) S' o,
    sem G SS S e S' o → gensem G SS S ge S' o.
```

By determinism of the semantics and principle of excluded middle this is equivalent to

```
Lemma ngensem_nnsem_seminf:
  forall L E G SS S ty (e : expr L E ty) (ge : gexpr e),
    ¬(∃S' o, gensem G SS S ge S' o) → ¬nsem G SS S e
      → seminf G SS S e.
```

This proof is done by co-induction towards `seminf` and case analysis of `ge`. Similarly to the proof of soundness of the WP calculus in Whycert, for each case it proceeds by analysis of the execution of sub-expressions and evaluation of the side-conditions: by excluded middle any condition is either true or false, any sub-expression either executes or does not, so depending on the case we try either to contradict one of the hypotheses or to co-inductively, i.e. using the co-inductive hypothesis, construct an infinite execution according to `seminf`. But due to important restrictions inside a co-inductive proof, this can be rather tricky, especially when an additional lemma is needed to proceed. Consider the case of a function call where we have to prove that

```
¬(∃S' o, gensem G SS S (gecall el sres fi) S' o) →
¬nsem G SS S (Escall el sres fi) →
  seminf G SS S (Escall el sres fi)
```

Supposing that the list of arguments `el` can be evaluated correctly, the goal reduces to

```
sempurelist G' SS S el G_args →
  seminf G' SS S
    (fold_exprlist el (Ecall (mk_fidx fi) (mk_arglist (body_P cf))))
```

As shown in Section 5.5.1, `fold_exprlist` is defined by recursion over `el` to build nested `Elet` blocks around the given expression (`Ecall ...`). To proceed here, we may want to generalise the expression `Ecall (mk_fidx fi)` into any function `mk` returning an expression from a list of terms and define the following lemma

```
Lemma seminf_arglist L P E ty ps G SS S G_args el
  (mk : hlist (term 0 (P +<+ E)) P → expr L (P +<+ E) ty):
  sempurelist G SS S el G_args →
  seminf (G_args +<+ G) SS S (mk ps) →
  seminf G SS S (fold_exprlist el (mk ps)).
```

that is if an `exprlist` `el` $= [e_0..e_n]$ evaluates to a list of values `G_args` and if the generalised expression `mk ps` diverges, then the `let`-expression `let v0 = e0 in .. let vn = en in mk ps` diverges. This has a straightforward proof by induction over `sempurelist`. By applying this lemma the goal reduces to

```
seminf (G_args +<+ G') SS S
  (Ecall (mk_fidx fi) (mk_arglist (body_P cf)))
```

and the proof can be completed as described above. However, it is not accepted by Coq's type-checker which finds that the co-recursive definition of the completed proof is ill-formed. The proof term looks roughly like this:

```
cofix CO L E G SS S ty (e : expr L E ty) (ge : gexpr e) :=
  match ge with
    | gecall ... ⇒
        ... (seminf_arglist (...) (... (CO ...)))
    | _ ⇒ ...
  emd
```

The problem is that the second argument of `seminf_arglist` is constructed using the co-inductive hypothesis `CO`, so the proof term of `seminf_arglist` must be transparent and respect the guard restrictions of co-inductive proofs, which is not the case as nested inductions are not allowed. A next attempt would be to prove this lemma by co-induction towards `seminf` and case analysis over `sempurelist`, as nested co-inductions are allowed. This proof is as easy as the direct induction of the first attempt, but again it prevents `seminf_arglist` from being used in our main goal. This time the problem is that the critical second argument is a recursive argument of the co-fixpoint, what is not allowed. It must stay outside the recursion forcing us to generalise all the variables appearing in it. The final attempt for this proof is through the following additional lemma

```
Lemma seminf_arglist' L E' ty SS S G' (e : expr L E' ty):
  seminf G' SS S e →
  forall E P ps el G_args G
    (mk : hlist (term 0 (P +<+ E)) P → expr L (P +<+ E) ty),
    sempurelist G SS S el G_args →
    (P +<+ E & mk ps) = (E' & e) →
    (P +<+ E & G_args +<+ G) = (E' & G') →
    seminf G SS S (fold_exprlist el (mk ps)).
```

which can be proved by co-induction without including its first hypothesis into the recursion and is therefore safe to be used in our main proof.

To conclude, this proof is much less routine than expected and much more involved than it morally deserves.

In the second part we need to prove that the compilation actually preserves the semantics:

```
Theorem comp_stmt_correct_fin:
  forall cge cfl L E ty (x : expr (mk_F cfl) L E ty) (e : gexpr x)
    (fm: funmap cfl) (lmp : labmap L) (vm : cvarmap E)
    (rtt : returntype) (brk : bool) (s : statement),
    comp_stmt cge fm lmp (vm:=vm) brk (rtt:=rtt) s ty = Some (x&e) →
    forall pp prog_vars G SS S S' o lm ce,
      rel_fm cge fm pp →
      rel_lm_S SS lm lmp →
      rel_Gce ce vm G →
      gensem cge pp G SS S e S' o →
      ∃t o', exec_stmt cge ce (ctmp S) (cmem S) lm s
                t (ctmp S') (cmem S') o'.
```

Here the three additional hypotheses relate run-time elements between the specialised Whycert semantics and the Clight semantics:

`rel_fm cge fm pp` relates functions in the `funmap` `fm` with those in the Clight global environment `cge` and their compiled counterpart in the Whycert program `pp`

`rel_lm_S SS lm lmp` relates labels in the `labmap` `lm` with their entries in the Clight labelled memory `lm` and the corresponding one in the Whycert stack of previous states

`rel_Gce ce vm G` relates local Clight variables in the `varmap` `vm` with their entries in the local Clight environment `ce` and the corresponding one in the Whycert execution environment `G`

As explained above, this cannot be proved by induction over the structure of statements as this would lack information about the execution of loops and function calls. Rather the proof should be by induction over the specialised semantics rules `gensem`, followed by a case analysis of the structure of statements. For each rule, this should exclude all the cases except one, as we specialised the semantics such that there would be one rule for each rule of the operational

semantics of Clight. However there is one exception: recall that we have to introduce a label around the compilation of **while** loops in order to encode the `loop assigns` clause: for appropriately chosen `t_inv`, `e_cnd` and `e`, such a loop is compiled to

```
(Elab
  (Etry
    (Eloop t_inv
      (Esifthenelse e_cnd (Etry e ExcContinue Esskip) Esbreak))
    ExcBreak Esskip))
```

Obviously this label should be added on top of the stack only once before the first iteration of the loop, so this cannot be simulated by a rule that loops itself. Instead, as already shown in Section 5.5.1, the compilation of loops is given as a core part which actually loops with an external label around. Just as the Clight semantics the specialised Whycert semantics `gensem` provides for three rules to execute this looping core expression plus an additional rule for the label. This extra rule forces us to generalise the goal with respect to the compilation function. We define an auxiliary relation between statements and Whycert expressions which are compiled one towards the other, either directly or with an additional label around:

```
Inductive rel_stmt E vm brk rtt s ty:
  forall L (lmp: labmap L) (x: expr L E ty), gexpr x → Prop :=
| rel_stmt_1 L (lmp: labmap L) (x: expr L E ty) (e: gexpr x):
    comp_stmt cge fm lmp (vm:=vm) brk (rtt:=rtt) s ty = Some (x&e) →
    rel_stmt E vm brk rtt s ty L lmp x e
| rel_stmt_2 L (lmp: labmap L) (x: expr (succ L) E ty) (e: gexpr x):
    comp_stmt cge fm lmp (vm:=vm) brk (rtt:=rtt) s ty =
      Some (Elab x & gelab' e) →
    rel_stmt E vm brk rtt s ty (succ L) (lm_lift lmp) x e.
```

Then our proof goes through the following lemma, where an induction over `gensem` followed by a case analysis of `rel_stmt` and `s` issues exactly the required inductive hypotheses for each case.

```
Lemma rel_stmt_correct L E G SS S ty x (e: gexpr x) S' o:
  gensem cge pp G SS S e S' o →
  forall lmp vm brk rtt s,
    rel_stmt cfl cge fm E vm brk rtt s ty L lmp x e →
  forall lm ce, rel_lm_S SS lm lmp → rel_Gce ce vm G →
    exists t o',
      exec_stmt cge ce (ctmp S)(cmem S) lm s t (ctmp S')(cmem S') o'
      ∧ rel_outcome brk rtt o' o ∧ PresS S S'.
```

However this approach introduces a technical complication: for each rule (17 cases), Coq must perform one inversion (2 cases), which is already quite costly, followed by a destruct (16 cases). This leads to 544 sub-goals, in each of which Coq must simplify an equation like

```
comp_stmt cge fm lmp brk (Sxxx ...) ty = Some (Exxx .. & gexx ..)
```

That is computing the left-hand side expression, identifying the sub-cases and performing an injection or discrimination. All this can be done automatically using tactics but it is quite time consuming. In order to separate these computations from the proofs of the sub-goals, which need to be done manually, we define a generic induction scheme. This can be done automatically, thanks to our self-defined tactic `admit as hypothesis`. It works like `admit`, except that the admitted sub-goal is not stored as an axiom, but as a section hypothesis. On closing the current section, all the admitted hypotheses then become formal parameters to the lemma in question.

Consider the following example:

```
Section S.
  Variables A B C : Prop.
  Lemma or_ind2 : A ∨ B → C.
    destruct 1; admit as hypothesis.
  Save.
End S.
```

The type of `or_ind2` is then

```
forall A B C : Prop, (A → C) → (B → C) → A ∨ B → C
```

This allows arbitrary generation and simplification of sub-goals which are then included in the induction principle.

In our concrete case, the principle looks like this:

```
Variable P :
  forall L E ty (G: env E) (SS: states L) (S: state)
    (x: expr F L E ty) (e: gexpr x),
      state → outcome ty → labmap L → cvarmap E → bool →
        returntype → statement → Prop.

Theorem gensem_rel_stmt_ind L E G SS S ty x e S' o:
  gensem cge pp G SS S e S' o →
  forall lmp vm brk rtt s,
    rel_stmt E vm brk rtt s ty L lmp x e →
    P L E G SS S ty x e S' o lmp vm brk rtt s.
```

It is proved as described generating the 17 desired sub-goals but it takes about 70 minutes on a 3.2GHz CPU consuming up to 2GB memory. For convenience, to avoid rebuilding and rechecking this principle at every minor change, its type can be textually copied and assumed as an axiom which should be updated only when the semantics changes.

The proof of `rel_stmt_correct` then consists in an application of this principle followed by the proofs of the various sub-goals, most of which are trivial after application of `rel_expr_correct`, `rel_lexpr_correct` and `comp_acsl_prop_correct` described above. Only the case of the function call is more involved, as we have to show that the compiled program correctly simulates the allocation and deallocation of local variables and formal parameters establishing the hypotheses `rel_lm_S SS lm lmp` and `rel_Gce ce vm G` where a quite tedious reasoning involving De-Bruijn indices is required.

This concludes the soundness proof as it implies safe whole program semantics under the condition that the verification conditions and that the pre-condition of the main function are valid:

```
Theorem comp_prog_correct pg pre_main m:
  comp_prog cp = Some (pg, pre_main) →
  Genv.init_mem cp = Some m →
  VCGEN pg → evalterm pre_main G0 SS0 (mkstate m) →
  (∃t i, bigstep_program_terminates cp t i)
  ∨ (∃t, bigstep_program_diverges cp t).

Theorem vcgen_finally_correct ct ax gl:
  vcgen cp ct = Some (ax, gl) →
  (valid_list ax → valid_list gl) →
  (∃t i, bigstep_program_terminates cp t i)
   ∨ (∃t, bigstep_program_diverges cp t).
```

```
Definition comp_utype ut :=
  match ut with
    | Tmem | Tmap | Tident ⇒ make_ty (mkstring_of_utype ut"")
    | Tintlist ⇒ stdlib_ty "list" "List" "list" [ty_int]
    | Tatype t ⇒ match t with
                  | Tctype x ⇒ make_ty (mkstring_of_Csyntax'type x"")
                  | Linteger ⇒ ty_int
                  | Lreal ⇒ ty_real
                  | Lboolean ⇒ stdlib_ty "bool" "Bool" "bool" []
                  | Larray _ ⇒ make_ty (mkstring_of_ClightACSL'type t"")
                 end
  end.
```

Figure 5.19: Generation of Why3 Logical Types

This result guarantees that the initial Clight program respects its annotations if the generated proof obligations are valid.

5.6 Extraction and Experimentations

5.6.1 Generation of Why3 Theories

Using the Why3 Back-end described in Section 3.5.2 we can directly generate Why3 theories containing the axioms and goals computed by `vcgen`. The back-end is parametrised by functions to compile types and symbols to Why3 types and symbols, either by reusing symbols from the Why3 standard library or by generating new, abstract symbols. Our Why3 interface provides for functions accessing the standard library and creating new symbols:

```
Parameter stdlib_ls : ocamlstring → ocamlstring → ocamlstring →
  forall {ar vl}, lsymbol ar vl.
Parameter stdlib_ty : ocamlstring → ocamlstring → ocamlstring →
  list ty → ty.

Parameter make_lsymbol : ocamlstring → forall ar vl, lsymbol ar vl.
Parameter make_ty: ocamlstring → ty.
```

The latter two require a name for the new symbol or type, we thus need to define functions that assign distinct strings to each constructor of several inductive types. In order to avoid doing this manually we define a small Coq plugin providing a Coq command `Generate StringOf` that, given an inductive type, tries to generate a possibly recursive function assigning to each constructor a string containing its name and a sub-string based on its arguments. For the types of the arguments it tries to reuse previously defined `stringof` functions or to recursively define them.

For instance the command

```
Generate StringOf Csyntax.type.
```

defines the functions `mkstring_of_floatsize`, `mkstring_of_signedness`, `mkstring_of_intsize`, `mkstring_of_typelist`, `mkstring_of_fieldlist` and `mkstring_of_Csyntax'type`, if `mkstring_of_Z` and `mkstring_of_ident` are already present in the current environment. The latter may be pretty printed in a more appropriate way than just by their constructors.

```
Fixpoint mk_tc ty: (termlist(ty_args ty) → term(ty_value ty)) → termcons ty :=
  match ty with
    | Tarrow ty ty' ⇒ fun f t, mk_tc (ty:=ty') (fun l, f (termlist_cons t l))
    | _ ⇒ fun f, f termlist_nil
  end.
Definition std {ty} s1 s2 s3: termcons ty := mk_tc (t_app (stdlib_ls s1 s2 s3)).
Definition new {ty} s: termcons ty := mk_tc (t_app (make_lsymbol s)).

Definition comp_sym ty (s: sym ty) : termcons ty :=
  match s in sym ty return termcons ty with
    | SymTrue ⇒ t_true
    | SymNot ⇒ t_not
    | SymOr ⇒ t_or
    | SymEq ty ⇒ t_equ
    | SymConstFloat fl ⇒ new (ocamlstring_of_float fl)
    | SymConstInteger i ⇒ t_int_const i
    | SymConstReal r ⇒ t_real_const r
    | SymIfThenElse ty ⇒ fun t0, t_if (t_equ t0 (t_bool_const true))
    | SymRel compty x ⇒ comp_rel compty x
    | SymCmp compty x ⇒ fun t1 t2, t_bool_of_t_prop (comp_rel compty x t1 t2)
    | SymConstBool b ⇒ t_bool_const b
    | SymZUop zuop ⇒ match zuop with
                        | ZOneg ⇒ std "int" "Int" "prefix -"
                        | ZObwneg ⇒ new "bitwise_neg"
                     end
    | SymZBop zbop ⇒ match zbop with
                        | ZOadd ⇒ std "int" "Int" "infix +"
                        | ZOsub ⇒ std "int" "Int" "infix -"
                        | ZOmul ⇒ std "int" "Int" "infix *"
                        | ZOdiv ⇒ std "int" "ComputerDivision" "div"
                        | ZOmod ⇒ std "int" "ComputerDivision" "mod"
                        | ZObwand ⇒ new "bitwise_and"
                        | ZObwor ⇒ new "bitwise_or"
                        | ZObwxor ⇒ new "bitwise_xor"
                     end
    | SymRUop o ⇒ match o with ROneg ⇒ std "real" "Real" "prefix -" end
    | SymRBop rbop ⇒ std "real" "Real" match rbop with
                                          | ROadd ⇒ "infix +"
                                          | ROsub ⇒ "infix -"
                                          | ROmul ⇒ "infix *"
                                          | ROdiv ⇒ "infix /"
                                       end
    | SymBUop buop ⇒ match buop with BOnot ⇒ boolls "notb" end
    | SymBBop bbop ⇒ boolls match bbop with
                              | BOand ⇒ "andb"
                              | BOor ⇒ "orb"
                           end
    | SymConversion conv_integer_real ⇒ std "real" "FromInt" "from_int"
    | SymConversion conv_real_integer ⇒ std "real" "Truncate" "truncate"
    | SymConversion c ⇒ new (mkstring_of_conversion c"")
    | SymIntlistNil ⇒ std "list" "List" "Nil"
    | SymIntlistCons ⇒ std "list" "List" "Cons"
    | SymIntlistIn ⇒ std "list" "Mem" "mem"
    | s ⇒ new (mkstring_of_sym s"")
  end.

Definition ref_name ty (r: ref ty) : ocamlstring :=
  match `r with Rmem_ ⇒ "m" | Rlocaltmp_ ⇒ "tmpmap" end.
```

Figure 5.20: Generation of Why3 Logical Symbols

We use this mechanism to generate functions printing types and symbols. See Figures 5.19 and 5.20 for how we map elements of our Whycert instantiation to Why3 types and symbols. Given this mapping and a small function `ref_name` assigning names to the references, we can instantiate the Why3 theory generation back-end and define the verification condition generator for C programs to Why3 theories:

```
Definition cvcgen3 ctypes cp : IOx theory :=
  let? (ax, gl) := Whygen.vcgen ctypes cp in
    comptheory comp_sym ref_name ax gl.
```

By chaining up the theorems `vcgen_finally_correct` and `comptheory_correct` we obtain the following

```
Theorem cvcgen3_correct ct th z:
  ioget (cvcgen3 cp ct) z = Some th →
  theory_valid th →
  (∃t i, bigstep_program_terminates cp t i)
  ∨ (∃t, bigstep_program_diverges cp t).
```

To be precise, this only holds if it is possible to construct a `den_local_lsymbol` such that the hypotheses in Figure 3.20 hold. Morally, this is possible if `comp_sym` is injective with respect to the newly generated logical symbols and we currently omit this proof.

5.6.2 Architecture of the Tool Chain

The verification condition generator, along with the compilation to Why3 theories, is then extracted to OCaml code. This results into a certified OCaml library for generation of Why3 verification conditions for annotated Clight programs. The latter can be produced starting from a Frama-C AST and the Frama-C platform provides a plug-in mechanism in order to interface with its API for parsing and analysing of ACSL annotated C source files. Similarly, the Why3 platform accepts plug-ins generating Why3 theories in order to process them further or send them to external provers – either interactively using why3ide or with the command-line interface. We thus define two plug-ins, one for Frama-C to generate the verification conditions for a given C source file and one for Why3 to execute Frama-C and then construct a theory for the resulting conditions. The latter are sent from the Frama-C process to the Why3 process by marshalling them through a UNIX pipe. This architecture is shown in Figure 5.21.

As explained above, these plug-ins are entirely proved in Coq. However, the interfaces with the input and output, i.e. the transformations corresponding to the arrows in the diagram entering and leaving the plug-in boxes, are not certified as they cannot be defined in Coq. The soundness of the whole tool chain thus depends on the correct implementation of theses interfaces – as well as obviously on the correct implementation of Frama-C and Why3.

On the input site, the Frama-C AST has to be transformed into a Clight/ACSL AST making use of the information retrieved by the analysis of the source file performed by Frama-C. The two ASTs are slightly different. For instance, declarations of local variables inside nested blocks have to be moved to the top of the function. Also, occurrences of formal parameters in function contracts have to be recognised in order to treat them as logical variables instead of program variables. A less linear transformation is to extract the code annotations attached to a given statement and to make an `assert` statement out of them. In general this AST transformation is partial as some features are not yet implemented and others, as termination annotations, **goto**-jumps, volatile variables, etc., are not supported.

On the output site, no AST compilations are necessary as we formalised the abstract syntax of Why3 terms directly in Coq. However, these terms need to be included into a fresh Why3 theory

Figure 5.21: Diagram of the Frama-C/Why3 Plugin

providing run-time support for accessing the Why3 standard library.

5.6.3 Examples

The purpose of the following toy examples is to introduce the basic axioms in the logical context. Unless stated otherwise, the generated verification conditions are proved automatically by Alt-Ergo, CVC3 and Z3.

Toy Example 1: Simple Assert

The first, simplest example aims to test the condition generated for just an assertion

```
int main() {
  //@ assert 42+2 == 44;
  return 0;
}
```

When parsing this code Frama-C transforms it slightly, so the plug-in prints it after these transformations:

```
int main(void)
{
  int __retres;
  /*@ assert 42+2 ≡ 44; */ ;
  __retres = 0;
  return (__retres);
}
```

An auxiliary variable is introduced in order to simplify the **return** statement. When building the Clight AST from this Frama-C AST, we recognise the generated variables and compile them to Clight temporary variables which are provided exactly for this purpose. Our plugin then prints out the Clight AST, omitting the annotations:

```
int main(void)
  {
    {{/* assert ... */;
    /*skip*/;}
    {$2 = 0;
    return $2;}}
  }
```

On this small program we already generate 5 proof obligations.[3] The first is the expected one:

```
goal Goal_2 : (42 + 2) = 44
```

Its name is due from the fact that it comes from the verification condition generated for the first function, given that `Goal_1` is always the name of the pre-condition of the main function, which is trivial in this example and is eliminated by splitting. The following two obligations can be summarized as

```
−2147483648 ≤ 0 ≤ 2147483647
```

It results from the conversion from `0` as a logical integer to a bounded 32-bit integer and ensures that this conversion is possible without loss of information.

The next one is a bit more interesting, as the first proof obligation involving abstract symbols:

```
goal Goal_2 :
  SymMapValidGet_Mint_I32_Signed SymIdent___retres
    (SymMapSet_Mint_I32_Signed SymIdent___retres SymMapEmpty i)
```

Notice the type name `Mint_I32_Signed` inside the name of the symbols, which were polymorphic in Coq but became instantiated by the Why3 back-end. The proof obligation results from the attempt to access the temporary variables environment in the argument of **return**. This environment, initially `SymMapEmpty`, is updated in the previous statement with the binding of type `int32` from `SymIdent___retres` to `i`. We need to prove that this updated environment has a valid binding of type `int32` for `SymIdent___retres`. That's not hard but the external provers need an axiom for that:

---

3. Strictly speaking there is always exactly one goal per function plus one, but we will consider the total number of goals after an initial split of conjunctions into parts

```
axiom Axiom_13 :
 forall i:ident.
   forall x:tint_I32_Signed.
    forall t:tmpmap.
     SymMapValidGet_Mint_I32_Signed i
        (SymMapSet_Mint_I32_Signed i t x)
```

This axiom, whose name depends on its current position in the list of axioms and may vary, results from the following entry in `axioms_mty` (cf. Section 5.1.3):

```
Definition axiom_map_set_validget mty : closed_prop :=
  Tforall (Tforall (Tforall
    (Tsymapp (SymMapValidGet mty) (Tvar HI2)
      (Tsymapp (SymMapSet mty) (Tvar HI2) (Tvar HI0) (Tvar HI1)))))).

Definition axioms_mty :=
  [ axiom_map_set_validget; ... ]
```

For the interpretation of `SymMapValidGet` and `SymMapSet` given above (page 98), the evaluation of the axiom results in the type of the following lemma, which is proved in Coq hence validating the use of the axiom.

```
Lemma validget_tmpvar_1 mty idt le (x: den_ctype mty) :
  validget_tmpvar mty idt (set_tmpvar idt le x).
```

The last obligation comes from the post-condition of the main function. It is automatically generated (`PpreservePermissions`) in order to relate the permissions of memory in the pre-state with the one in the post-state.

```
goal Goal_2 : SymMemPreservePermissions m m
```

As there isn't any memory update in this example the two memories are the same and this can be proved with the reflexivity axiom, trivially justified by the interpretation of `SymMemPreservePermissions` (cf. Fig. 5.3).

```
axiom Axiom_15 : forall m1:memory. SymMemPreservePermissions m1 m1
```

Toy Example 2: A global variable

The next example aims to test our tool in presence of assignments on global variables in several functions:

```
int x;

void forty_two() {
  x = 42;
  //@ assert x+2 == 44;
  return;
}

int main() {
  forty_two();
  x = 40;
  return 0;
}
```

On this example we generate 14 proof obligations: 5 for the function `forty_two`, 8 for `main` and one for the pre-condition of `main`. For the assignment `x = 42` to be safe, `x` must have write permissions for its type `int32` and `42` must be small enough to fit into this type. The latter is similar as for `0` in the previous example, whereas the former is new:

```
SymValidStore_Tint_I32_Signed m SymCGlobal_x_Tint_I32_Signed
```

This is ensured by the pre-condition `Pvalid_globvars` automatically added to every function of the program as shown in Figure 5.18:

```
Definition Pvalid_globvar L E (g: ident * globvar ctype): prop L E :=
  let (gid, gv) := g in
  if Genv.init_data_list_size gv.(gvar_init) == sizeof gv.(gvar_info)
      & gv.(gvar_volatile) == false & gv.(gvar_readonly) == false
      & Z_lt_dec (sizeof gv.(gvar_info)) Int.modulus then
    Tsymapp (SymValidStore gv.(gvar_info)) (Tderef Rmem)
      (Tsymapp (SymCGlobal gid gv.(gvar_info)))
  else
    Ptrue.

Definition Pvalid_globvars {L E} :=
  List.fold_right (fun g, Pand (Pvalid_globvar L E g)) Ptrue.
```

For every global variable that respects certain criteria we add the predicate `SymValidStore` to the preconditions of all the functions. To be sure to have safe whole program semantics, we must prove that the pre-condition of `main` is valid in the initial state, which is guaranteed by the following lemma.

```
Lemma init_globvars_valid cp m:
  Genv.init_mem cp = Some m →
  evalterm (Pvalid_globvars cp.(prog_vars)) [] [] (mkstate m).
```

The last two proof obligations for the function `forty_two` concern the memory obtained by performing the store:

```
goal Goal_2 :
  (conv_int_integer_I32_Signed
    (SymLoad_Tint_I32_Signed
      (SymStore_Tint_I32_Signed m SymCGlobal_x_Tint_I32_Signed x)
      SymCGlobal_x_Tint_I32_Signed) + 2) = 44

goal Goal_2 :
  SymMemPreservePermissions m
    (SymStore_Tint_I32_Signed m SymCGlobal_x_Tint_I32_Signed x)
```

The former is generated for the assertion `x+2 == 44` and can be solved using the load-store axiom, already shown in Section 5.1.3. On the other hand, just like in the previous example, the latter is generated for the automatic post-condition, except that here the memory has been modified, so we need the following axiom, justified by the Compcert memory model.

```
axiom Axiom_16 :
  forall m1:memory.
   forall p:tpointer_Tint_I32_Signed.
    forall i:tint_I32_Signed.
      SymValidStore_Tint_I32_Signed m1 p →
        SymMemPreservePermissions m1 (SymStore_Tint_I32_Signed m1 p i)
```

Most of the proof obligations generated for the main function are similar to the ones already explained. The only interesting ones are the first and the fourth:

```
axiom H: SymValidStore_Tint_I32_Signed m SymCGlobal_x_Tint_I32_Signed
goal Goal_3: SymValidStore_Tint_I32_Signed m
               SymCGlobal_x_Tint_I32_Signed
axiom H2: SymMemPreservePermissions m m1
goal Goal_3: SymValidStore_Tint_I32_Signed m1
               SymCGlobal_x_Tint_I32_Signed
```

The former comes from the automatic pre-condition of `forty_two` and can be proved with the automatic pre-condition of `main` (`H`) while the latter comes from the assignment `x=40` showing the importance of the automatic post-condition of `main` (`H2`).

Toy example 3: Function with Formal Parameter, Allocation

In this example we try to use formal parameters and pre- and post-conditions about them but we also for the first time try to actually load values from memory:

```
/*@ requires x == 42;
  @ ensures x == 42;
  @*/
void forty_two(int x) {
  x += 2;
  //@ assert x==44;
  return;
}

int main() {
  forty_two(42);
  return 0;
}
```

This example also illustrates the ACSL semantics of formal parameters in function contracts, which are considered as logical variables independent from memory. Thus the given post-condition should be trivially valid and not `x == 44` (cf. Section 4.2.2).

For this example our tool generates 16 proof obligations, 9 for `forty_two` and 7 for `main`. Most of them are already explained, but the first one is new:

```
goal Goal_2 :
   SymValidLoad_Tint_I32_Signed (SymStore_Tint_I32_Signed m1 p i) p
```

It is generated for the very first statement `x += 2` ensuring that `x` is loadable and correctly initialised. That's not a problem as the current memory has been updated with an allocation for each formal parameter, each of which has been bound to the corresponding value (cf. Fig. 5.11). An axiom allows the external provers to succeed:

```
axiom Axiom_18 :
  forall m1:memory.
   forall p:tpointer_Tint_I32_Signed.
    forall i:tint_I32_Signed.
      SymValidStore_Tint_I32_Signed m1 p →
       SymValidLoad_Tint_I32_Signed(SymStore_Tint_I32_Signed m1 p i) p
```

Five more obligations are generated for this line: two trivial overflow checks for the expression `2`, two overflow checks for the expression `x + 2`, solvable thanks to the pre-condition, and a per-

mission check to store into `x`:

```
axiom H1 : SymValidStore_Tint_I32_Signed m1 p
goal Goal_2 :
  SymValidStore_Tint_I32_Signed (SymStore_Tint_I32_Signed m1 p i) p
```

Here, `m1` is the memory resulting of the allocation for the formal parameter `x` which ensures that the allocated pointer is writable. However, we need to be sure that this still holds after the initialisation, i.e. that the permissions are preserved by a store operation. This can be proved thanks to the last axiom introduced for the previous example and the following one, providing the specification of `SymMemPreservePermissions` to the external provers:

```
axiom Axiom_26 :
  forall m1:memory.
   forall m2:memory.
    SymMemPreservePermissions m1 m2 →
     (forall p:tpointer_Tint_I32_Signed.
       SymValidStore_Tint_I32_Signed m1 p →
         SymValidStore_Tint_I32_Signed m2 p)
```

The next two obligations concern the assertion `x==44` and the post-condition `x == 42`. As explained above they are not contradictory. The first proof obligation is about loading a value from memory that results from storing a value at the same position, while the second one is a trivial consequence of the pre-condition.

```
axiom H : conv_int_integer_I32_Signed i = 42
goal Goal_2 : conv_int_integer_I32_Signed i = 42
```

The last proof obligation again ensures that the permissions are preserved by the function, but this time allocation and deallocation actually modify the permissions in the Compcert memory. However, it is guaranteed that `alloc` and `free` modify the permission of the allocated block only, leaving the permissions of all the other blocks unchanged. If thus a given piece of code that preserves the permissions is preceded by the allocation of a pointer and followed by the deallocation of that same pointer, then the resulting piece of code still preserves the permissions according to the definition of `m_preserve_permissions`. This can be formalised with the following lemma:

```
Lemma m_alloc_free_preserves_permissions cty b m1 m2 m3:
  sizeof cty < Int.modulus →
  Mem.alloc m1 0 (sizeof cty) = (m2, b) →
  m_preserve_permissions m2 m3 →
  m_preserve_permissions m1 (complete3 Mem.free m3 b 0 (sizeof cty)).
```

This property is included into the post-conditions of `PAR_mem_alloc`, so that proof obligation has the required information in its context:

```
axiom H4 :
  forall m2:memory.
   SymMemPreservePermissions m1 m2 →
    SymMemPreservePermissions m (SymFree_Tint_I32_Signed m2 p)

goal Goal_2 :
  SymMemPreservePermissions m (SymFree_Tint_I32_Signed
    (SymStore_Tint_I32_Signed (SymStore_Tint_I32_Signed m1 p i) p i2)
    p)
```

This concludes the proof obligations for `forty_two`. This function is then called from the main function, which has thus to make sure the pre-condition holds for the given argument.

Toy Example 4: Local variable and initialisation

In this example we show a proof obligation that cannot be proved because of missing initialisation of a local variable.

```
int main() {
  int x;
  //@ assert x == x;
  x = x;
  x = 42;
  //@ assert x+2 == 44;
  return 0;
}
```

For this function we generate 11 proof obligations. The first two are interesting:

```
goal Goal_2 :
  conv_int_integer_I32_Signed (SymLoad_Tint_I32_Signed m1 p) =
   conv_int_integer_I32_Signed (SymLoad_Tint_I32_Signed m1 p)

goal Goal_2 : SymValidLoad_Tint_I32_Signed m1 p
```

They are generated for the first assertion and the first assignment, respectively. While the former is easy, nothing permits to prove the latter. The point is that, as requested by the ACSL specification, the load operation is modelled as a complete function, i.e. always returning a result. This result is obviously equal to itself, even if nothing else is specified about the function for a given argument. To ensure that the corresponding load operation in Compcert succeeds we need to prove the `SymValidLoad` predicate, which is only possible if the given pointer has been correctly initialised, which is not the case for fresh local variables. Notice that with most compilers, including Compcert, the execution of this program would still not crash. That's why Jessie does not generate an assertion for this assignment and proves this program safe. However, its semantics according to Compcert is undefined, so we must not be able to prove it safe with our tool.

Toy Example 5: Two local variables, assigns clause

```
/*@ ensures \result == 44;
  @ assigns \nothing;
  @*/
int main() {
  int x;
  int y;
  x = 42;
  y = 2;
  return x+y;
}
```

The new feature of this example is the use of multiple variables and the **assigns** clause. The difficulty here is to prove properties about `x` in a memory where `y` has been modified, for instance `ValidLoad`:

```
goal Goal_2 :
  SymValidLoad_Tint_I32_Signed
    (SymStore_Tint_I32_Signed(SymStore_Tint_I32_Signed m2 p i)p1 i1) p
```

Here a new predicate comes into play: `SymMemPreserveContents`. Given a list of blocks, it relates any two memory states with the same contents except possibly in the blocks that are specified in the list. The list of blocks is intended as a first approximation for a generic set of memory locations specified by a set of terms including intervals and aggregates as in ACSL. Its interpretation is given as `fun l, m_preserve_contents (except_list l)` pragmatically defined as follows:

```
Definition except_list l pt := ¬ pt.(adr_block) ∈ l.
Definition m_preserve_valid_load (P: address → Prop) m m' :=
  forall mty pt, P pt → m_valid_load m mty pt
    → m_valid_load m' mty pt.
Definition m_preserve_load (P: address → Prop) m m' :=
  forall mty pt, P pt → m_valid_load m mty pt
    → clogic_loadp mty pt m' = clogic_loadp mty pt m.
Definition m_preserve_contents (P: address → Prop) m m' :=
  m_preserve_valid_load P m m' ∧  m_preserve_load P m m'.
```

It is easy to see that the following axiom is valid

```
axiom Axiom_15 :
  forall m1:memory.
   forall p:tpointer_Tint_I32_Signed.
    forall i:tint_I32_Signed.
      SymValidStore_Tint_I32_Signed m1 p →
        SymMemPreserveContents
          (Cons (SymBlock_Tint_I32_Signed p) (Nil:list int)) m1
          (SymStore_Tint_I32_Signed m1 p i)
```

This, together with two axioms for the specification of the predicate, allows CVC3, but not Altergo, neither Z3, to prove the above and all the other proof obligations, except the one concerning the `assigns` clause, whose encoding is shown in Figure 5.22. Basically, it asserts `SymMemPreserveContents` with the list of blocks compiled from the given list terms in the clause. In the example, we require **assigns \nothing**, so the list is empty.

In spite of several axioms provided in order to reason about this predicate, including reflexivity, transitivity and monotony, no prover is currently able to solve the concerned proof obligation

```
goal Goal_2 :
  SymMemPreserveContents (Nil:list int) m
   (SymFree_Tint_I32_Signed
     (SymFree_Tint_I32_Signed
       (SymStore_Tint_I32_Signed
         (SymStore_Tint_I32_Signed m2 p i) p1 i1)
      p1) p)
```

Because of the symmetric allocation and deallocation of local variables, the proof obligation concerning the preserving of permissions is easily proved thanks to the property of `PAR_mem_alloc` described above.

Toy Example 6: Pointers

The following example is automatically proved

```
Fixpoint comp_assigns_list L (lm : labmap L) E te lvm vm tl :=
  match tl with
    | [] ⇒ return Tsym SymIntlistNil
    | t::tl ⇒
        let? (cty, t') := comp_left_term lm lvm vm t in
        let? l := comp_assigns_list lm lvm vm tl in
          return Tsymapp SymIntlistCons (Tsymapp (SymBlock cty) t') l
  end.

Definition comp_assigns_clause L (lm : labmap L) E te lvm vm otl m1 m2:=
  match otl with
    | Some tl ⇒
        let? l := comp_assigns_list lm lvm vm tl in
          return Tsymapp SymMemPreserveContents l m1 m2
    | None ⇒ return Ptrue
  end.
```

Figure 5.22: Encoding of assigns-clauses

```
int *r;

int main () {
  int i;
  r = &i;
  *r = 42;
  //@ assert 0 < i;
  return i;
}
```

There is nothing new to add with respect to the previous examples. Since our memory model is already low-level, that is variables as `i` above lie in memory with some address. In other words storing into a variable is not different from storing in memory at a given address.

A slightly more realistic example: Integer Square Root

The example shown in Figure 5.23 is typically taught to students when teaching Hoare logic. However this is a real implementation taking into account the size of machine integers. It illustrates the use of loop invariants and loop assigns clauses.

On this example we generate 78 proof obligations. All are proved automatically, except a few similar to the ones explained in Section 5.6.3.

5.7 Discussions

5.7.1 About the approach for proving soundness of compilation

A huge part of this work consisted in developing the proof of semantics preservation of the compilation of statements. As observed in other works, the proof of *backward simulation* is always difficult and *reverse forward simulation*, as required in this work, is even stronger.

In Bertot [9] this proof is approached by defining a decompilation relation between the source and the target language. Such a predicate relates an expression of the source language $\mathcal{L}_S$ with an equivalent expression of the target language $\mathcal{L}_T$. To be useful, it must contain an entry for every expression e_T of $\mathcal{L}_T$ that is the result of a compilation for any expression of $\mathcal{L}_S$, but also for every

```
/*@ requires 0 <= x <= 32768 * 32768 - 1;
  @ assigns \nothing;
  @ ensures \result >= 0 &&
  @         \result * \result <= x &&
  @         x < (\result + 1) * (\result + 1);
  @*/
int isqrt(int x) {
  int count = 0, sum = 1;
  /*@ loop invariant 0 <= count <= 32767 &&
    @                 x >= count * count &&
    @                 sum == (count+1)*(count+1);
    @ loop assigns count, sum;
    @ // loop variant  x - count;
    @*/
  while (sum <= x) sum += 2 * ++count + 1;
  return count;
}

//@ ensures \result == 4;
int test () {
  int r;
  r = 0 + isqrt(17);
  //@ assert r < 4 ==> \false;
  //@ assert r > 4 ==> \false;
  return r;
}

int main () {
  return 0;
}
```

Figure 5.23: Example: Integer Square Root

sub-expression of e_T, for which it is not always obvious, if not impossible, to find a counterpart in $\mathcal{L}_S$.

Instead we adopted the approach described in Section 5.5.2 which consists in splitting the proof `sem -> exec_stmt`, into `sem -> gensem` and `gensem -> exec_stmt` but which however poses some difficulties. To begin it is necessary to define specialised semantics for Whycert `gensem`, with one case for each rule of `Clight.exec_stmt`, and an inductive predicate for the absence of Whycert semantics `nsem`. The former predicate is quite long to write, nevertheless it can be seen as a series of lemmas and justified by making the proof easier. The latter is also long, though completely mechanical to define and one could investigate if such a predicate – together with the predicate for diverging semantics – can be generated automatically starting from the inductive definition of big-step terminating semantics.

Both parts of this splitting are however not trivial. Concerning `gensem -> exec_stmt`, because of the special case of the label in front of the loop, a simple decompilation relation can not be completely avoided and contributes to the exorbitant computation time to generate the induction scheme `gensem_rel_stmt_ind`. On the other hand, the proof of `sem -> gensem` by co-induction is certainly simpler than by decompilation but still not as straightforward as it should be. While the forward simulation proof `gensem -> sem` took 5 minutes, the reverse simulation required a case by case reasoning for the execution of each sub-expression and was complicated by the technical issues about co-induction described.

5.7.2   Evolution of Compcert

This work refers to the version 1.9 of Compcert. More recent versions add support for several features of the C programming language, notably using structures and unions as r-values, e.g. in assignments and function arguments. Our ACSL formalisation natively supports aggregate values and the compilation towards Whycert naturally encodes them as logical values. However the design choice made in formalising the semantics of composite r-values in Compcert is not compatible with our formalisation and if we tried to naively add support for them to our compilation to Whycert we would break several invariants. In particular, according to Compcert's new semantics composite types are not actually loaded from memory by value but the new access mode "by copy" is introduced. This access mode is more similar to "by reference" than to "by value", as for such a type a dereference simply evaluates to its address in memory. This formalisation is certainly closest to the *implementation* of the semantics, but we believe that it does not correspond to the expected *logical meaning*. Furthermore it prevents the support for returning composites by value from a function.

Instead we would suggest a formalisation where the notion of *value* in the front-end languages (at least CompcertC and Clight) is extended by aggregate values and that a later compilation pass appropriately replaces by-value accesses of composites by explicit copies. In any case support for structures and unions as r-values should not be attempted until this issue has been discussed.

Chapter 6

Conclusions and Perspectives

We have developed a VC generator for ACSL annotated Clight programs. Thanks to the Coq extraction it is a standalone tool able to call external theorem provers to discharge generated proof obligations. It has a relatively small trusted code base:

- the Frama-C front-end that parses C source code and produces a Clight/ACSL abstract syntax tree
- the Why3 back-end that translates formulas to provers
- the provers themselves
- the Coq kernel and the extraction mechanism

In particular the set of axioms sent to provers enabling them to reason about the memory is proved correct in Coq with respect to the Compcert semantics.

We conclude by some discussions and perspectives.

6.1 ACSL coverage

An important missing feature in the logic language with respect to its informal specification is the possibility to define new logic functions and predicates, which should not be too difficult to add. Also we currently do not support any built-in ACSL predicates like `\valid`, `\fresh`, `\separated`, etc. The semantics of these predicates with respect to the Compcert memory model would have to be discussed. In particular, it seems that `\valid` is not strong enough to ensure a successful load operation on the given pointer.

Concerning code annotations several ACSL annotation features are missing for the moment. Some, like statement contracts or arbitrary invariants, may be encoded in the proposed solution. An ACSL feature that is not supported at all is loop variants to express termination. To implement this in our tool chain we would need support for this in the verification condition generator first. A whole category of programs we decide not to handle is programs with jump statements. This decision is expressed in the choice of big step semantics for annotated C programs. This is motivated not only by difficulties in formally define and prove a WP calculus on unstructured programs. It is even difficult to formalise the semantics of loop invariants and loop assigns clauses for arbitrary unstructured loops in presence of pointers.

6.2 About the Soundness Proof

In Section 5.7.1 we have discussed in detail the technical difficulties we had to perform the proof of soundness of compilation from annotated Clight to Whycert.

Given the difficulties and the amount of time needed to perform the soundness proof of the compilation one may wonder whether it would have been easier to avoid this semantics preser-

vation proof altogether by proving the soundness of a WP calculus directly with respect to the semantics of Clight. Such a WP calculus could even be defined directly on annotated Clight without going through an intermediate language like Whycert. At least, this would incite focusing on the real issues of the C language.

6.2.1   Difficulties with Dependent Types

A design choice in this development was the use of dependent types to encode typing constrains directly inside the intermediate languages. In particular dependently types De-Bruijn indices ensure that every variable in a term a correctly bound. This obviously facilitates the definition of functions deconstructing such terms, like the weakest precondition calculus, as they don't have to handle typing errors, but complicates the definition of functions constructing such terms, like the compilation from Clight, as they must ensure all the constrains. Whereas forcing the developer to write well-typed functions is probably a good thing, the resulting terms can be quite hard to reason about. In particular case analysis or inductions over dependently typed inductives require to generalise all the terms they depend on to free variables, which often forces to meticulously cut the proof into lemmas. While most of the time there is a solution, this may take some time to find. Most importantly it blocks the proof search of the really interesting results.

To conclude, we believe that dependent types should be used but wisely. For instance, it was an error in the compilation of Clight functions to simulate the allocation of local variables by a series of nested `let`-blocks, which forced us to handle that complex typing environment. Instead, we should have applied an abstract program parameter allocating all the necessary variables and storing them in an abstract mapping from identifiers to pointers with the properties specified in the post-condition of the parameter.

6.2.2   Alternative Memory Models

All these difficulties took unexpectedly long to resolve and prevented us from inspecting the real issues of the C language. We notably would have wanted to investigate about different memory models in the generated proof obligations. The presented solution adopts a slight abstraction of the Compcert memory model and is therefore the easiest to prove. The generated proof obligations are however as expected too hard to handle by the automatic provers. A next step would be to separate at least the variables which are never accessed by their address and treat them as Hoare variables, as it is done in the Jessie plug-in. Such a change would introduce a more complex relation between the Clight state and the Whycert state. With some effort the proof could be make more generic with respect to this relation by identifying the lemmas that interface with the properties of the relation.

6.3   Future Works

As an alternative to introducing separation at the level of the compilation from Clight to Whycert, i.e. switching memory model, it could be investigated whether, while keeping the same memory model, the generated proof obligations can be made more convenient to discharge by automatically adding additional hypotheses. For instance a simple analysis could compute an approximation of the set of memory locations modified between every two program points. With this information at its disposal, an external prover would then only have to prove that a given location is not in such a set when it needs to compare values loaded from memory. The correctness of this additional information could always be guaranteed by Coq. The proposed solution can thus be considered a good base to explore alternative but safe axiomatisations, that would be more convenient for the external provers in discharging proof obligations.

In order to increase the confidence into the Frama-C platform, the use of ACSL annotated CompcertC as front-end language should be considered. The performed normalisations could then be proved in Coq. Also it would be interesting to generate certified proof obligations ensuring that a given program is not ambiguous.

Bibliography

[1] J.-R. Abrial. *The B-Book, assigning programs to meaning*. Cambridge University Press, 1996.

[2] A. W. Appel. Foundational proof-carrying code. In *LICS*, pages 247–256. IEEE Computer Society, 2001.

[3] M. Barnett, R. DeLine, B. Jacobs, B.-Y. E. Chang, and K. R. M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387, 2005.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

[5] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*, pages 301–317. Springer, 2006.

[6] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. *ACM Trans. Program. Lang. Syst.*, 31(5), 2009.

[7] G. Barthe, B. Grégoire, and M. Pavlova. Preservation of proof obligations from java to the java virtual machine. In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 83–99. Springer, 2008.

[8] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009. http://frama-c.cea.fr/acsl.html.

[9] Y. Bertot. A certified Compiler for an Imperative Language. Technical Report RR-3488, INRIA, Sept. 1998.

[10] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.

[11] F. Besson, D. Cachera, T. P. Jensen, and D. Pichardie. Certified static analysis by abstract interpretation. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *FOSAD*, volume 5705 of *Lecture Notes in Computer Science*, pages 223–257. Springer, 2009.

[12] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *Proceedings of Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, 2006.

[13] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.

[14] S. Böhme, M. Moskal, W. Schulte, and B. Wolff. HOL-Boogie - an interactive prover-backend for the verifying C compiler. *Journal of Automated Reasoning*, 44(1-2):111–144, 2010.

[15] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 2004.

[16] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[17] A. Charguéraud. Characteristic formulae for the verification of imperative programs. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming (ICFP)*, pages 418–430, Tokyo, Japan, September 2011. ACM.

[18] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011. http://adam.chlipala.net/cpdt/.

[19] A. J. Chlipala, J. G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In G. Hutton and A. P. Tolmach, editors, *ICFP*, pages 79–90, Edinburgh, Scotland, 2009. ACM.

[20] D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *CASSIS*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.

[21] P. Cuoq, J. Signoles, P. Baudin, R. Bonichon, G. Canet, L. Correnson, B. Monate, V. Prevosto, and A. Puccetti. Experience report: Ocaml for an industrial-strength static analysis framework. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming (2009)*, pages 281–286, 2009.

[22] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC: Contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, pages 429–430. IEEE Comp. Soc. Press, 2009.

[23] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, Dec. 1998. See also http://research.compaq.com/SRC/esc/.

[24] J.-C. Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.

[25] J.-C. Filliâtre and C. Marché. Multi-Prover Verification of C Programs. In *Sixth International Conference on Formal Engineering Methods*, pages 15–29. Springer, 2004.

[26] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.

[27] P. Herms. Certification of a chain for deductive program verification. In Y. Bertot, editor, *2nd Coq Workshop, satellite of ITP'10*, 2010.

[28] P. Herms, C. Marché, and B. Monate. A certified multi-prover verification condition generator. In R. Joshi, P. Müller, and A. Podelski, editors, *VSTTE*, Lecture Notes in Computer Science. Springer, 2012.

[29] P. V. Homeier and D. F. Martin. A mechanically verified verification condition generator. *The Computer Journal*, 38(2):131–141, July 1995.

[30] G. Huet, G. Kahn, and C. Paulin-Mohring. *The Coq Proof Assistant - A tutorial - Version 8.0*, Apr. 2004.

[31] M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[32] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. *Commun. ACM*, 53(6):107–115, June 2010.

[33] J. Lagarias. The $3x + 1$ problem and its generalizations. *American Mathematical Monthly*, pages 3–23, Jan. 1985.

[34] D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In B. Aichernig and B. Beckert, editors, *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005), 5–9 September 2005, Koblenz, Germany*, pages 2–11, 2005.

[35] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Conference Record of the 33rd Symposium on Principles of Programming Languages*, Charleston, South Carolina, Jan. 2006. ACM Press.

[36] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[37] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207:284–304, February 2009.

[38] P. Letouzey. A new extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer, 2003.

[39] P. Letouzey. *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq*. Thèse de doctorat, Université Paris-Sud, July 2004.

[40] C. Marché and C. Paulin-Mohring. Reasoning about Java programs with aliasing and frame conditions. In J. Hurd and T. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 179–194. Springer, Aug. 2005.

[41] C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004. `http://krakatoa.lri.fr`.

[42] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *19th Conference on Automated Deduction*, Lecture Notes in Computer Science. Springer, 2003.

[43] B. Meyer. *Eiffel: The Language*. Prentice Hall, Hemel Hempstead, 1992.

[44] Y. Moy and C. Marché. *Jessie Plugin Tutorial,* Beryllium *version*. INRIA, 2009. `http://www.frama-c.cea.fr/jessie.html`.

[45] Y. Moy and C. Marché. *The Jessie plugin for Deduction Verification in Frama-C — Tutorial and Reference Manual*. INRIA & LRI, 2011. `http://krakatoa.lri.fr/`.

[46] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *Proceedings of ICFP'08*, 2008.

[47] G. C. Necula. Proof-carrying code. In P. Lee, F. Henglein, and N. D. Jones, editors, *POPL*, pages 106–119. ACM Press, 1997.

[48] G. Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1981. `http://www.cs.washington.edu/education/courses/cse599f/06sp/papers/NelsonThesis.pdf`.

[49] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[50] M. Norrish. *C Formalised in HOL*. PhD thesis, University of Cambridge, Nov. 1998.

[51] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752, Saratoga Springs, NY, June 1992. Springer.

[52] C. Parent-Vigouroux. Verifying programs in the calculus of inductive constructions. *Formal Aspects of Computing*, 9:484–517, 1997.

[53] D. Pariente and E. Ledinot. Formal verification of industrial C code using Frama-C: a case study. In B. Beckert and C. Marché, editors, *Formal Verification of Object-Oriented Software, Papers Presented at the International Conference*, Karlsruhe Reports in Informatics, pages 205–219, Paris, France, June 2010. `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000019083`.

[54] F. Randimbivololona, J. Souyris, P. Baudin, A. Pacalet, J. Raguideau, and D. Schoen. Applying formal proof techniques to avionics software: A pragmatic approach. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1798–1815. Springer, 1999.

[55] N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[56] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.3*, 2010. `http://coq.inria.fr`.

Remerciements

Je tiens à remercier tout d'abord mes deux rapporteurs Xavier Leroy et Gilles Barthe pour leur intérêt dans mon travail exprimé dans leurs nombreux commentaires et suggestions. Merci aussi aux examinateurs pour leurs questions intéressants, en particulier au président du jury Roberto di Cosmo merci pour son engagement.

Merci ensuite à mes directeurs de thèse Claude Marché et Benjamin Monate pour l'encadrement exceptionnel dont ils m'ont fait bénéficier à toute étape et de tout point de vue, y compris scientifique, rédactionnel et financier. Ils ont un grand mérite au succès de cette thèse et c'était un vrai plaisir de travailler avec eux.

Merci beaucoup aussi à mes deux équipes, le LSL coté CEA et Proval coté Inria, pour leur accueil et pour la belle collaboration. Coté sportif, merci en particulier à Virgile et Richard pour les nombreuses courses à pied et à Jean-Christophe et Bernard pour avoir rendu possible le foot du mercredi pendant toutes ces trois années.

Un grand merci aux assistantes Frédérique et Régine pour leur support amical et très professionnel, même des fois à la dernière minute.

Merci enfin, il va sans dire, à ma famille et ma copine pour leur soutien et pour toujours être là pour moi. Sans eux, rien ne serait possible.